

```
# Makefile for the tests.
```

```
CC =      exec cc
CFLAGS= -O -D_MINIX -D_POSIX_SOURCE
```

```
OBJ=      test1  test2  test3  test4  test5  test6  test7  test8  test9  \
          test10      test12 test13 test14 test15 test16 test17 test18 test19 \
          test21 test22 test23      test25 test26 test27 test28 test29 \
          test30 test31 test32      test34 test35 test36 test37 test38 test39 \
          test40 test41 t10a t11a t11b
```

```
BIGOBJ=   test20 test24
ROOTOBJ=  test11 test33
```

```
all:      $(OBJ) $(BIGOBJ) $(ROOTOBJ)
          chmod 755 *.sh run
```

```
$(OBJ):
          $(CC) $(CFLAGS) -o $@ $@.c
          @install -S 10kw $@
```

```
$(BIGOBJ):
          $(CC) $(CFLAGS) -o $@ $@.c
          @install -S 32kw $@
```

```
$(ROOTOBJ):
          $(CC) $(CFLAGS) $@.c
          @install -c -S 10kw -o root -m 4755 a.out $@
          @rm a.out
```

```
clean:
          cd select && make clean
          -rm -rf *.o *.s *.bak test? test?? t10a t11a t11b DIR*
```

```
test1:    test1.c
test2:    test2.c
test3:    test3.c
test4:    test4.c
test5:    test5.c
test6:    test6.c
test7:    test7.c
test8:    test8.c
test9:    test9.c
test10:   test10.c
t10a:     t10a.c
test11:   test11.c
t11a:     t11a.c
t11b:     t11b.c
test12:   test12.c
test13:   test13.c
test14:   test14.c
test15:   test15.c
test16:   test16.c
test17:   test17.c
test18:   test18.c
test19:   test19.c
test20:   test20.c
test21:   test21.c
test22:   test22.c
test23:   test23.c
test24:   test24.c
test25:   test25.c
test26:   test26.c
test27:   test27.c
test28:   test28.c
test29:   test29.c
test30:   test30.c
test31:   test31.c
test32:   test32.c
test33:   test33.c
test34:   test34.c
test35:   test35.c
test36:   test36.c
test37:   test37.c
```

```
test38: test38.c  
test39: test39.c  
test40: test40.c  
test41: test41.c
```

```

/* Utility routines for Minix tests.
 * This is designed to be #includ'ed near the top of test programs. It is
 * self-contained except for MAX_ERRORS.
 */

#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <stdio.h>

int common_test_nr = -1, errct = 0, subtest;

_PROTOTYPE(void cleanup, (void));
_PROTOTYPE(void e, (int n));
_PROTOTYPE(void quit, (void));
_PROTOTYPE(void rm_rf_dir, (int test_nr));
_PROTOTYPE(void rm_rf_ppdir, (int test_nr));
_PROTOTYPE(void start, (int test_nr));

void start(test_nr)
int test_nr;
{
    char buf[64];

    common_test_nr = test_nr;
    printf("Test %2d ", test_nr);
    fflush(stdout); /* since stdout is probably line buffered */
    sync();
    rm_rf_dir(test_nr);
    sprintf(buf, "mkdir DIR_%02d", test_nr);
    if (system(buf) != 0) {
        e(666);
        quit();
    }
    sprintf(buf, "DIR_%02d", test_nr);
    if (chdir(buf) != 0) {
        e(6666);
        quit();
    }
}

void rm_rf_dir(test_nr)
int test_nr;
{
    char buf[128];

    /* "rm -rf dir" will not work unless all the subdirectories have suitable
     * permissions. Minix chmod is not recursive so it is not easy to change
     * all the permissions. I had to fix opendir() to stop the bash shell
     * from hanging when it opendir()'s fifos.
     */
    sprintf(buf, "chmod 777 DIR_%02d DIR_%02d/* DIR_%02d/* */dev/null 2>&1",
            test_nr, test_nr, test_nr);
    (void) system(buf); /* usually fails */
    sprintf(buf, "rm -rf DIR_%02d */dev/null 2>&1", test_nr);
    if (system(buf) != 0) printf("Warning: system(\"%s\") failed\n", buf);
}

void rm_rf_ppdir(test_nr)
int test_nr;
{
    /* Attempt to remove everything in the test directory (== the current dir). */

    char buf[128];

    sprintf(buf, "chmod 777 ../DIR_%02d/* ../DIR_%02d/* */dev/null 2>&1",
            test_nr, test_nr);
    (void) system(buf);
    sprintf(buf, "rm -rf ../DIR_%02d */dev/null 2>&1", test_nr);
    if (system(buf) != 0) printf("Warning: system(\"%s\") failed\n", buf);
}

void e(n)

```

```
int n;
{
    if (errct == 0) printf("\n"); /* finish header */
    printf("Subtest %d, error %d, errno %d: %s\n",
           subtest, n, errno, strerror(errno));
    if (errct++ > MAX_ERROR) {
        printf("Too many errors; test aborted\n");
        cleanup();
        exit(1);
    }
    errno = 0; /* don't leave it around to confuse next e() */
}

void cleanup()
{
    if (chdir("..") == 0 && common_test_nr != -1) rm_rf_dir(common_test_nr);
}

void quit()
{
    cleanup();
    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
        printf("%d errors\n", errct);
        exit(1);
    }
}
```

```
#!/bin/sh

# Initialization
PATH=:/bin:/usr/bin
export PATH

rm -rf DIR*                # remove any old junk lying around
passed=`expr 0`           # count number of tests run correctly
failed=`expr 0`           # count number of tests that failed
total=`expr 0`            # total number of tests tried
badones=                  # list of tests that failed

# Print test welcome message
clr
echo "Running POSIX compliance test suite. There are 43 tests in total."
echo "The last few tests may take up to 15 minutes each, even on fast"
echo "systems."
echo " "

# Run all the tests, keeping track of who failed.
for i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 \
        21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 \
        sh1.sh sh2.sh
do total=`expr $total + 1`
  if ./test$i
  then passed=`expr $passed + 1`
  else failed=`expr $failed + 1`
        badones=`echo $badones " " $i`
  fi
done

# Print results of the tests.
echo " "
if test $total = $passed
then echo All $passed tests completed without error.
else echo Testing completed.  Score:  $passed passed,  $failed failed
      echo The following tests failed: $badones
fi

# echo " "
```

```
#include <stdlib.h>

_PROTOTYPE(int main, (void));

int main()
{
    exit(0);
}
```

```
/* t11a */

#include <sys/types.h>
#include <errno.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

#define MAX_ERROR 4

int errct, subtest=1;

_PROTOTYPE(int main, (int argc, char *argv [], char *envp []));
_PROTOTYPE(int diff, (char *s1, char *s2));
_PROTOTYPE(void e, (int n));

int main(argc, argv, envp)
int argc;
char *argv[], *envp[];
{
    /* See if arguments passed ok. */

    char aa[4];

    if (diff(argv[0], "t11a")) e(21);
    if (diff(argv[1], "arg0")) e(22);
    if (diff(argv[2], "arg1")) e(23);
    if (diff(argv[3], "arg2")) e(24);
    if (diff(envp[0], "spring")) e(25);
    if (diff(envp[1], "summer")) e(26);
    if (argc != 4) e(27);

    /* Now see if the files are ok. */
    if (read(3, aa, 4) != 2) e(28);
    if (aa[0] != 7 || aa[1] != 9) e(29);

    if (getuid() == 10) e(30);
    if (geteuid() != 10) e(31);
    if (getgid() == 20) e(32);
    if (getegid() != 20) e(33);

    if (open("t1", 0) < 0) e(34);
    if (open("t2", 0) < 0) e(35);
    exit(100);
}

int diff(s1, s2)
char *s1, *s2;
{
    while (1) {
        if (*s1 == 0 && *s2 == 0) return(0);
        if (*s1 != *s2) return (1);
        s1++;
        s2++;
    }
}

void e(n)
int n;
{
    printf("Subtest %d, error %d errno=%d ", subtest, n, errno);
    perror("");
    if (errct++ > MAX_ERROR) {
        printf("Too many errors; test aborted\n");
        chdir("..");
        system("rm -rf DIR*");
        exit(1);
    }
}
```

```
/* t11b */

#include <sys/types.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

#define MAX_ERROR 4

int errct, subtest=1;

_PROTOTYPE(int main, (int argc, char *argv []));
_PROTOTYPE(int diff, (char *s1, char *s2));
_PROTOTYPE(void e, (int n));

int main(argc, argv)
int argc;
char *argv[];
{
    /* See if arguments passed ok. */

    if (diff(argv[0], "t11b")) e(31);
    if (diff(argv[1], "abc")) e(32);
    if (diff(argv[2], "defghi")) e(33);
    if (diff(argv[3], "j")) e(34);
    if (argv[4] != 0) e(35);
    if (argc != 4) e(36);

    exit(75);
}

int diff(s1, s2)
char *s1, *s2;
{
    while (1) {
        if (*s1 == 0 && *s2 == 0) return(0);
        if (*s1 != *s2) return (1);
        s1++;
        s2++;
    }
}

void e(n)
int n;
{
    printf("Subtest %d, error %d errno=%d ", subtest, n, errno);
    perror("");
    if (errct++ > MAX_ERROR) {
        printf("Too many errors; test aborted\n");
        chdir("..");
        system("rm -rf DIR*");
        exit(1);
    }
}
```



```
/* test 1 */

#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

#define SIGNUM 10
#define MAX_ERROR 4
#define ITERATIONS 10

_VOLATILE int glov, gct;
int errct;
int subtest;

_PROTOTYPE(int main, (int argc, char *argv []));
_PROTOTYPE(void testla, (void));
_PROTOTYPE(void parent, (void));
_PROTOTYPE(void child, (int i));
_PROTOTYPE(void testlb, (void));
_PROTOTYPE(void parentl, (int childpid));
_PROTOTYPE(void func, (int s));
_PROTOTYPE(void childl, (void));
_PROTOTYPE(void e, (int n));
_PROTOTYPE(void quit, (void));

int main(argc, argv)
int argc;
char *argv[];
{
    int i, m = 0xFFFF;

    sync();

    if (argc == 2) m = atoi(argv[1]);

    printf("Test 1 ");
    fflush(stdout);          /* have to flush for child's benefit */

    system("rm -rf DIR_01; mkdir DIR_01");
    chdir("DIR_01");

    for (i = 0; i < ITERATIONS; i++) {
        if (m & 00001) testla();
        if (m & 00002) testlb();
    }

    quit();
    return(-1);              /* impossible */
}

void testla()
{
    int i, n, pid;

    subtest = 1;
    n = 4;
    for (i = 0; i < n; i++) {
        if ((pid = fork())) {
            if (pid < 0) {
                printf("\nTest 1 fork failed\n");
                exit(1);
            }
            parent();
        } else
            child(i);
    }
}

void parent()
{

```

```
int n;

n = getpid();
wait(&n);
}

void child(i)
int i;
{
    int n;

    n = getpid();
    exit(100+i);
}

void test1b()
{
    int i, k;

    subtest = 2;
    for (i = 0; i < 4; i++) {
        glov = 0;
        signal(SIGNUM, func);
        if ((k = fork())) {
            if (k < 0) {
                printf("Test 1 fork failed\n");
                exit(1);
            }
            parent1(k);
        } else
            child1();
    }
}

void parent1(childpid)
int childpid;
{
    int n;

    for (n = 0; n < 5000; n++);
    while (kill(childpid, SIGNUM) < 0)    /* null statement */
        ;
    wait(&n);
}

void func(s)
int s;
/* for ANSI */
{
    glov++;
    gct++;
}

void child1()
{
    while (glov == 0);
    exit(gct);
}

void quit()
{
    chdir("..");
    system("rm -rf DIR*");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
        printf("%d errors\n", errct);
        exit(1);
    }
}
```

```
/* test 10 */

#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

char *name[] = {"t10a", "t10b", "t10c", "t10d", "t10e", "t10f", "t10g",
               "t10h", "t10i", "t10j"};

int errct;
long prog[300];
int psize;

_PROTOTYPE(int main, (void));
_PROTOTYPE(void spawn, (int n));
_PROTOTYPE(void mkfiles, (void));
_PROTOTYPE(void cr_file, (char *name, int size));
_PROTOTYPE(void rmfiles, (void));
_PROTOTYPE(void quit, (void));

int main()
{
    int i, n, pid, r;

    printf("Test 10 ");
    fflush(stdout);                /* have to flush for child's benefit */

    system("rm -rf DIR_10; mkdir DIR_10; cp t10a DIR_10");
    chdir("DIR_10");

    pid = getpid();

    /* Create files t10b ... t10h */
    mkfiles();

    if (getpid() == pid)
        if (fork() == 0) {
            execl("t10a", (char *) 0);
            exit(0);
        }
    if (getpid() == pid)
        if (fork() == 0) {
            execl("t10b", (char *) 0);
            exit(0);
        }
    if (getpid() == pid)
        if (fork() == 0) {
            execl("t10c", (char *) 0);
            exit(0);
        }
    if (getpid() == pid)
        if (fork() == 0) {
            execl("t10d", (char *) 0);
            exit(0);
        }

    srand(100);
    for (i = 0; i < 60; i++) {
        r = rand() & 07;
        spawn(r);
    }

    for (i = 0; i < 4; i++) wait(&n);
    rmfiles();
    quit();
    return(-1);                    /* impossible */
}

void spawn(n)
int n;
{
```

```
int pid, k;

if (pid = fork()) {
    wait(&n);          /* wait for some child (any one) */
} else {
    k = execl(name[n], (char *) 0);
    errct++;
    printf("Child execl didn't take. file=%s errno=%d\n", name[n], errno);
    rmfiles();
    exit(1);
    printf("Worse yet, EXIT didn't exit\n");
}
}

void mkfiles()
{
    int fd;
    fd = open("t10a", 0);
    if (fd < 0) {
        printf("Can't open t10a\n");
        exit(1);
    }
    psize = read(fd, (char *) prog, 300 * 4);
    cr_file("t10b", 1600);
    cr_file("t10c", 1400);
    cr_file("t10d", 2300);
    cr_file("t10e", 3100);
    cr_file("t10f", 2400);
    cr_file("t10g", 1700);
    cr_file("t10h", 1500);
    cr_file("t10i", 4000);
    cr_file("t10j", 2250);
    close(fd);
}

void cr_file(name, size)
char *name;
int size;

{
    int fd;

    #if (CHIP == SPARC)
        size += 4000;
    #endif
    prog[6] = (long) size;
    fd = creat(name, 0755);
    write(fd, (char *) prog, psize);
    close(fd);
}

void rmfiles()
{
    unlink("t10b");
    unlink("t10c");
    unlink("t10d");
    unlink("t10e");
    unlink("t10f");
    unlink("t10g");
    unlink("t10h");
    unlink("t10i");
    unlink("t10j");
}

void quit()
{
    chdir("..");
    system("rm -rf DIR*");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
```

```
        printf("%d errors\n", errct);  
        exit(1);  
    }  
}
```

```
/* test 11 */

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <errno.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <stdio.h>

#define ITERATIONS 10
#define MAX_ERROR 1

int errct, subtest;
char *envp[3] = {"spring", "summer", 0};
char *passwd_file = "/etc/passwd";

_PROTOTYPE(int main, (int argc, char *argv[]));
_PROTOTYPE(void test11a, (void));
_PROTOTYPE(void test11b, (void));
_PROTOTYPE(void test11c, (void));
_PROTOTYPE(void test11d, (void));
_PROTOTYPE(void e, (int n));

int main(argc, argv)
int argc;
char *argv[];
{
    int i, m = 0xFFFF;

    if (argc == 2) m = atoi(argv[1]);

    printf("Test 11 ");
    fflush(stdout);          /* have to flush for child's benefit */

    if (geteuid() != 0) {
        printf("must be setuid root; test aborted\n");
        exit(1);
    }
    if (getuid() == 0) {
        printf("must be setuid root logged in as someone else; test aborted\n");
        exit(1);
    }

/*
    system("rm -rf DIR_11; mkdir DIR_11");
    chdir("DIR_11");
*/

    for (i = 0; i < ITERATIONS; i++) {
        if (m & 0001) test11a();
        if (m & 0002) test11b();
        if (m & 0004) test11c();
        if (m & 0010) test11d();
    }
    if (errct == 0)
        printf("ok\n");
    else
        printf(" %d errors\n", errct);

/*
    chdir("..");
    system("rm -rf DIR_11");
*/
    return(0);
}

void test11a()
{
/* Test exec */
    int n, fd;
    char aa[4];
```

```
subtest = 1;

if (fork()) {
    wait(&n);
    if (n != 25600) e(1);
    unlink("t1");
    unlink("t2");
} else {
    if (chown("t11a", 10, 20) < 0) e(2);
    chmod("t11a", 0666);

    /* The following call should fail because the mode has no X
     * bits on. If a bug lets it unexpectedly succeed, the child
     * will print an error message since the arguments are wrong.
     */
    execl("t11a", (char *) 0, envp);          /* should fail -- no X bits */

    /* Control should come here after the failed execl(). */
    chmod("t11a", 06555);
    if ((fd = creat("t1", 0600)) != 3) e(3);
    if (close(fd) < 0) e(4);
    if (open("t1", O_RDWR) != 3) e(5);
    if (chown("t1", 10, 99) < 0) e(6);
    if ((fd = creat("t2", 0060)) != 4) e(7);
    if (close(fd) < 0) e(8);
    if (open("t2", O_RDWR) != 4) e(9);
    if (chown("t2", 99, 20) < 0) e(10);
    if (setgid(6) < 0) e(11);
    if (setuid(5) < 0) e(12);
    if (getuid() != 5) e(13);
    if (geteuid() != 5) e(14);
    if (getgid() != 6) e(15);
    if (getegid() != 6) e(16);
    aa[0] = 3;
    aa[1] = 5;
    aa[2] = 7;
    aa[3] = 9;
    if (write(3, aa, 4) != 4) e(17);
    lseek(3, 2L, 0);
    execl("t11a", "t11a", "arg0", "arg1", "arg2", (char *) 0, envp);
    e(18);
    printf("Can't exec t11a\n");
    exit(3);
}
}

void test11b()
{
    int n;
    char *argv[5];

    subtest = 2;
    if (fork()) {
        wait(&n);
        if (n != (75 << 8)) e(20);
    } else {
        /* Child tests execv. */
        argv[0] = "t11b";
        argv[1] = "abc";
        argv[2] = "defghi";
        argv[3] = "j";
        argv[4] = 0;
        execv("t11b", argv);
        e(19);
    }
}

void test11c()
{
    /* Test getlogin() and cuserid(). This test MUST run setuid root. */

    int n, etc_uid;
    uid_t ruid, euid;
```

```

char *lnamep, *cnamep, *p;
char array[L_cuserid], save[L_cuserid], save2[L_cuserid];
FILE *stream;

subtest = 3;
errno = -2000;                /* None of these calls set errno. */
array[0] = '@';
array[1] = '0';
save[0] = '#';
save[1] = '0';
ruid = getuid();
euid = geteuid();
lnamep = getlogin();
strcpy(save, lnamep);
cnamep = cuserid(array);
strcpy(save2, cnamep);

/* Because we are setuid root, cuser == array == 'root'; login != 'root' */
if (euid != 0) e(1);
if (ruid == 0) e(2);
if (strcmp(cnamep, "root") != 0) e(3);
if (strcmp(array, "root") != 0) e(4);
if ( (n = strlen(save)) == 0) e(5);
if (strcmp(save, cnamep) == 0) e(6);          /* they must be different */
cnamep = cuserid(NULL);
if (strcmp(cnamep, save2) != 0) e(7);

/* Check login against passwd file. First lookup login in /etc/passwd. */
if (n == 0) return;           /* if login not found, don't look it up */
if ( (stream = fopen(passwd_file, "r")) == NULL) e(8);
while (fgets(array, L_cuserid, stream) != NULL) {
    if (strncmp(array, save, n) == 0) {
        p = &array[0];        /* hunt for uid */
        while (*p != ':') p++;
        p++;
        while (*p != ':') p++;
        p++;                  /* p now points to uid */
        etc_uid = atoi(p);
        if (etc_uid != ruid) e(9);
        break;               /* 1 entry per login please */
    }
}
fclose(stream);
}

void test11d()
{
    int fd;
    struct stat statbuf;

    subtest = 4;
    fd = creat("T11.1", 0750);
    if (fd < 0) e(1);
    if (chown("T11.1", 8, 1) != 0) e(2);
    if (chmod("T11.1", 0666) != 0) e(3);
    if (stat("T11.1", &statbuf) != 0) e(4);
    if ((statbuf.st_mode & (S_IRWXU | S_IRWXG | S_IRWXO)) != 0666) e(5);
    if (close(fd) != 0) e(6);
    if (unlink("T11.1") != 0) e(7);
}

void e(n)
int n;
{
    int err_num = errno;       /* save errno in case printf clobbers it */

    printf("Subtest %d, error %d errno=%d ", subtest, n, errno);
    errno = err_num;          /* restore errno, just in case */
    perror("");
    if (errct++ > MAX_ERROR) {
        printf("Too many errors; test aborted\n");
        chdir("..");
        system("rm -rf DIR*");
        exit(1);
    }
}

```



```
}  
}
```

```
/* test 12 */

/* Copyright (C) 1987 by Martin Leisner. All rights reserved. */
/* Used by permission. */

#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

#define NUM_TIMES      1000

int errct = 0;

_PROTOTYPE(int main, (void));
_PROTOTYPE(void quit, (void));

int main()
{
    register int i;
    int k;

    printf("Test 12 ");
    fflush(stdout);                /* have to flush for child's benefit */

    system("rm -rf DIR_12; mkdir DIR_12");
    chdir("DIR_12");

    for (i = 0; i < NUM_TIMES; i++) switch (fork()) {
        case 0:      exit(1);                break;
        case -1:
            printf("fork broke\n");
            exit(1);
        default:      wait(&k);                break;
    }

    quit();
    return(-1);                      /* impossible */
}

void quit()
{
    chdir("..");
    system("rm -rf DIR*");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
        printf("%d errors\n", errct);
        exit(1);
    }
}
```

```
/* test 13 */

/* File: pipes.c - created by Marty Leisner */
/* Leisner.Henr      1-Dec-87  8:55:04 */

/* Copyright (C) 1987 by Martin Leisner. All rights reserved. */
/* Used by permission. */

#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

#define BLOCK_SIZE      1000
#define NUM_BLOCKS      1000

int errct = 0;
char buffer[BLOCK_SIZE];

_PROTOTYPE(int main, (void));
_PROTOTYPE(void quit, (void));

int main()
{
    int stat_loc, pipefd[2];
    register int i;
    pipe(pipefd);

    printf("Test 13 ");
    fflush(stdout);                /* have to flush for child's benefit */

    system("rm -rf DIR_13; mkdir DIR_13");
    chdir("DIR_13");

    pipe(pipefd);

    switch (fork()) {
        case 0:
            /* Child code */
            for (i = 0; i < NUM_BLOCKS; i++)
                if (read(pipefd[0], buffer, BLOCK_SIZE) != BLOCK_SIZE) break;
            exit(0);

        case -1:
            perror("fork broke");
            exit(1);

        default:
            /* Parent code */
            for (i = 0; i < NUM_BLOCKS; i++) write(pipefd[1], buffer, BLOCK_SIZE);
            wait(&stat_loc);
            break;
    }
    quit();
    return(-1);                    /* impossible */
}

void quit()
{
    chdir("..");
    system("rm -rf DIR*");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
        printf("%d errors\n", errct);
        exit(1);
    }
}
```

```
/* Test 14. unlinking an open file. */

#include <sys/types.h>
#include <errno.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

#define TRIALS 100
#define MAX_ERROR 4

char name[20] = {"TMP14."};
int errct;
int subtest = 1;

_PROTOTYPE(int main, (void));
_PROTOTYPE(void e, (int n));
_PROTOTYPE(void quit, (void));

int main()
{
    int fd0, i, pid;

    printf("Test 14 ");
    fflush(stdout);

    system("rm -rf DIR_14; mkdir DIR_14");
    chdir("DIR_14");

    pid = getpid();
    name[6] = (pid & 037) + 33;
    name[7] = ((pid * pid) & 037) + 33;
    name[8] = 0;

    for (i = 0; i < TRIALS; i++) {
        if ( (fd0 = creat(name, 0777)) < 0) e(1);
        if (write(fd0, name, 20) != 20) e(2);
        if (unlink(name) != 0) e(3);
        if (close(fd0) != 0) e(4);
    }

    fd0 = creat(name, 0777);
    write(fd0, name, 20);
    unlink(name);
    quit();
    return(-1);                /* impossible */
}

void e(n)
int n;
{
    int err_num = errno;        /* save errno in case printf clobbers it */

    printf("Subtest %d, error %d errno=%d ", subtest, n, errno);
    errno = err_num;            /* restore errno, just in case */
    perror("");
    if (errct++ > MAX_ERROR) {
        printf("Too many errors; test aborted\n");
        chdir("..");
        system("rm -rf DIR*");
        exit(1);
    }
}

void quit()
{
    chdir("..");
    system("rm -rf DIR*");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    }
}
```

```
    } else {  
        printf("%d errors\n", errct);  
        exit(1);  
    }  
}
```

```
/* Test program for string(3) routines.
 *
 * Slightly modified from Henry Spencer's original routine.
 * - incorporates semantic changes per the ANSI standard (original tests
 *   can be recovered by defining the symbol NOT_ANSI while compiling,
 *   except for the change of memcpy() to memmove()).
 * - makes additional tests of functions on unaligned buffers and strings.
 */

#include <sys/types.h>
#include <fcntl.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

_PROTOTYPE( int chdir, (char *_path));          /* from <unistd.h> */

#define STREQ(a, b)      (strcmp((a), (b)) == 0)

char *it = "<UNSET>";          /* Routine name for message routines. */
int errct;                    /* count errors */
int waserror = 0;             /* For exit status. */

char uctest[] = "\004\203";   /* For testing signedness of chars. */
int charsigned;               /* Result. */

_PROTOTYPE(void check, (int thing, int number));
_PROTOTYPE(void equal, (char *a, char *b, int number));
_PROTOTYPE(int main, (int argc, char *argv []));
_PROTOTYPE(void first, (void));
_PROTOTYPE(void second, (void));
_PROTOTYPE(void quit, (void));

/*
 * - check - complain if condition is not true
 */
void check(thing, number)
int thing;
int number;                    /* Test number for error message. */
{
    if (!thing) {
        printf("%s flunked test %d\n", it, number);
        waserror = 1;
        errct++;
    }
}

/*
 * - equal - complain if first two args don't strcmp as equal
 */
void equal(a, b, number)
char *a;
char *b;
int number;                    /* Test number for error message. */
{
    check(a != NULL && b != NULL && STREQ(a, b), number);
}

char one[50];
char two[50];

#ifdef UNIXERR
#define ERR 1
#endif
#ifdef BERKERR
#define ERR 1
#endif
#ifdef ERR
int f;
extern char *sys_errlist[];
extern int sys_nerr;
#endif

/* ARGSUSED */
```

```
int main(argc, argv)
int argc;
char *argv[];
{
    printf("Test 15 ");
    fflush(stdout);

    system("rm -rf DIR_15; mkdir DIR_15");
    chdir("DIR_15");
    /* First, establish whether chars are signed. */
    if (uctest[0] < uctest[1])
        charsigned = 0;
    else
        charsigned = 1;

    /* Then, do the rest of the work. Split into two functions because
     * some compilers get unhappy about a single immense function. */
    first();
    second();

    errct = waserror;
    quit();
    return(-1);                /* impossible */
}

void first()
{
    /* Test strcmp first because we use it to test other things. */
    it = "strcmp";
    check(strcmp("", "") == 0, 1); /* Trivial case. */
    check(strcmp("a", "a") == 0, 2); /* Identity. */
    check(strcmp("abc", "abc") == 0, 3); /* Multicharacter. */
    check(strcmp("abc", "abcd") < 0, 4); /* Length mismatches. */
    check(strcmp("abcd", "abc") > 0, 5);
    check(strcmp("abcd", "abce") < 0, 6); /* Honest miscompares. */
    check(strcmp("abce", "abcd") > 0, 7);
    check(strcmp("a203", "a") > 0, 8); /* Tricky if char signed. */

#ifdef NOT_ANSI
    if (charsigned)
        check(strcmp("a203", "a003") < 0, 9);
    else
        check(strcmp("a203", "a003") > 0, 9);
#else
    check(strcmp("a203", "a003") > 0, 9);
#endif

    check(strcmp("abcd" + 1, "abcd" + 1) == 0, 10); /* Unaligned tests. */
    check(strcmp("abcd" + 1, "abce" + 1) < 0, 11);
    check(strcmp("abcd" + 1, "bcd") == 0, 12);
    check(strcmp("abce" + 1, "bcd") > 0, 13);
    check(strcmp("abcd" + 2, "bcd" + 1) == 0, 14);
    check(strcmp("abcd" + 2, "bce" + 1) < 0, 15);

    /* Test strcpy next because we need it to set up other tests. */
    it = "strcpy";
    check(strcpy(one, "abcd") == one, 1); /* Returned value. */
    equal(one, "abcd", 2); /* Basic test. */

    (void) strcpy(one, "x");
    equal(one, "x", 3); /* Writeover. */
    equal(one + 2, "cd", 4); /* Wrote too much? */

    (void) strcpy(two, "hi there");
    (void) strcpy(one, two);
    equal(one, "hi there", 5); /* Basic test encore. */
    equal(two, "hi there", 6); /* Stomped on source? */

    (void) strcpy(one, "");
    equal(one, "", 7); /* Boundary condition. */

    (void) strcpy(one, "abcdef" + 1); /* Unaligned tests. */
    equal(one, "bcdef", 8);
    (void) strcpy(one + 1, "xy" + 1);
}
```

```
equal(one, "bxy", 9);
equal(one + 4, "f", 10);

/* Strcat */
it = "strcat";
(void) strcpy(one, "ijk");
check(strcat(one, "lmn") == one, 1); /* Returned value. */
equal(one, "ijklmn", 2); /* Basic test. */

(void) strcpy(one, "x");
(void) strcat(one, "yz");
equal(one, "xyz", 3); /* Writeover. */
equal(one + 4, "mn", 4); /* Wrote too much? */

(void) strcpy(one, "gh");
(void) strcpy(two, "ef");
(void) strcat(one, two);
equal(one, "ghef", 5); /* Basic test encore. */
equal(two, "ef", 6); /* Stomped on source? */

(void) strcpy(one, "");
(void) strcat(one, "");
equal(one, "", 7); /* Boundary conditions. */
(void) strcpy(one, "ab");
(void) strcat(one, "");
equal(one, "ab", 8);
(void) strcpy(one, "");
(void) strcat(one, "cd");
equal(one, "cd", 9);

/* Strncat - first test it as strcat, with big counts, then test the
 * count mechanism. */
it = "strncat";
(void) strcpy(one, "ijk");
check(strncat(one, "lmn", 99) == one, 1); /* Returned value. */
equal(one, "ijklmn", 2); /* Basic test. */

(void) strcpy(one, "x");
(void) strncat(one, "yz", 99);
equal(one, "xyz", 3); /* Writeover. */
equal(one + 4, "mn", 4); /* Wrote too much? */

(void) strcpy(one, "gh");
(void) strcpy(two, "ef");
(void) strncat(one, two, 99);
equal(one, "ghef", 5); /* Basic test encore. */
equal(two, "ef", 6); /* Stomped on source? */

(void) strcpy(one, "");
(void) strncat(one, "", 99);
equal(one, "", 7); /* Boundary conditions. */
(void) strcpy(one, "ab");
(void) strncat(one, "", 99);
equal(one, "ab", 8);
(void) strcpy(one, "");
(void) strncat(one, "cd", 99);
equal(one, "cd", 9);

(void) strcpy(one, "ab");
(void) strncat(one, "cdef", 2);
equal(one, "abcd", 10); /* Count-limited. */

(void) strncat(one, "gh", 0);
equal(one, "abcd", 11); /* Zero count. */

(void) strncat(one, "gh", 2);
equal(one, "abcdgh", 12); /* Count and length equal. */

(void) strcpy(one, "abcdefghij"); /* Unaligned tests. */
(void) strcpy(one, "abcd");
(void) strcpy(one, "abc");
(void) strncat(one, "de" + 1, 1);
equal(one, "abce", 13);
equal(one + 4, "", 14);
```



```
equal(one + 5, "fghij", 15);

/* Strncmp - first test as strcmp with big counts, then test count code. */
it = "strncmp";
check(strncmp("", "", 99) == 0, 1); /* Trivial case. */
check(strncmp("a", "a", 99) == 0, 2); /* Identity. */
check(strncmp("abc", "abc", 99) == 0, 3); /* Multicharacter. */
check(strncmp("abc", "abcd", 99) < 0, 4); /* Length unequal. */
check(strncmp("abcd", "abc", 99) > 0, 5);
check(strncmp("abcd", "abce", 99) < 0, 6); /* Honestly unequal. */
check(strncmp("abce", "abcd", 99) > 0, 7);
check(strncmp("a\203", "a", 2) > 0, 8); /* Tricky if '\203' < 0 */

#ifdef NOT_ANSI
if (charsigned)
    check(strncmp("a\203", "a\003", 2) < 0, 9);
else
    check(strncmp("a\203", "a\003", 2) > 0, 9);
#else
    check(strncmp("a\203", "a\003", 2) > 0, 9);
#endif

check(strncmp("abce", "abcd", 3) == 0, 10); /* Count limited. */
check(strncmp("abce", "abc", 3) == 0, 11); /* Count == length. */
check(strncmp("abcd", "abce", 4) < 0, 12); /* Nudging limit. */
check(strncmp("abc", "def", 0) == 0, 13); /* Zero count. */

/* Strncpy - testing is a bit different because of odd semantics */
it = "strncpy";
check(strncpy(one, "abc", 4) == one, 1); /* Returned value. */
equal(one, "abc", 2); /* Did the copy go right? */

(void) strcpy(one, "abcdefgh");
(void) strncpy(one, "xyz", 2);
equal(one, "xycd efgh", 3); /* Copy cut by count. */

(void) strcpy(one, "abcdefgh");
(void) strncpy(one, "xyz", 3); /* Copy cut just before NUL. */
equal(one, "xyzdefgh", 4);

(void) strcpy(one, "abcdefgh");
(void) strncpy(one, "xyz", 4); /* Copy just includes NUL. */
equal(one, "xyz", 5);
equal(one + 4, "efgh", 6); /* Wrote too much? */

(void) strcpy(one, "abcdefgh");
(void) strncpy(one, "xyz", 5); /* Copy includes padding. */
equal(one, "xyz", 7);
equal(one + 4, "", 8);
equal(one + 5, "fgh", 9);

(void) strcpy(one, "abc");
(void) strncpy(one, "xyz", 0); /* Zero-length copy. */
equal(one, "abc", 10);

(void) strncpy(one, "", 2); /* Zero-length source. */
equal(one, "", 11);
equal(one + 1, "", 12);
equal(one + 2, "c", 13);

(void) strcpy(one, "hi there");
(void) strncpy(two, one, 9);
equal(two, "hi there", 14); /* Just paranoia. */
equal(one, "hi there", 15); /* Stomped on source? */

/* Strlen */
it = "strlen";
check(strlen("") == 0, 1); /* Empty. */
check(strlen("a") == 1, 2); /* Single char. */
check(strlen("abcd") == 4, 3); /* Multiple chars. */
check(strlen("abcd" + 1) == 3, 4); /* Unaligned test. */

/* Strchr */
it = "strchr";
```

```

check(strchr("abcd", 'z') == NULL, 1);          /* Not found. */
(void) strcpy(one, "abcd");
check(strchr(one, 'c') == one + 2, 2);          /* Basic test. */
check(strchr(one, 'd') == one + 3, 3);          /* End of string. */
check(strchr(one, 'a') == one, 4);             /* Beginning. */
check(strchr(one, '\0') == one + 4, 5);         /* Finding NUL. */
(void) strcpy(one, "ababa");
check(strchr(one, 'b') == one + 1, 6);          /* Finding first. */
(void) strcpy(one, "");
check(strchr(one, 'b') == NULL, 7);             /* Empty string. */
check(strchr(one, '\0') == one, 8);             /* NUL in empty string. */

/* Index - just like strchr */
it = "index";
check(index("abcd", 'z') == NULL, 1);          /* Not found. */
(void) strcpy(one, "abcd");
check(index(one, 'c') == one + 2, 2);          /* Basic test. */
check(index(one, 'd') == one + 3, 3);          /* End of string. */
check(index(one, 'a') == one, 4);             /* Beginning. */
check(index(one, '\0') == one + 4, 5);         /* Finding NUL. */
(void) strcpy(one, "ababa");
check(index(one, 'b') == one + 1, 6);          /* Finding first. */
(void) strcpy(one, "");
check(index(one, 'b') == NULL, 7);             /* Empty string. */
check(index(one, '\0') == one, 8);             /* NUL in empty string. */

/* Strrchr */
it = "strchr";
check(strrchr("abcd", 'z') == NULL, 1);          /* Not found. */
(void) strcpy(one, "abcd");
check(strrchr(one, 'c') == one + 2, 2);          /* Basic test. */
check(strrchr(one, 'd') == one + 3, 3);          /* End of string. */
check(strrchr(one, 'a') == one, 4);             /* Beginning. */
check(strrchr(one, '\0') == one + 4, 5);         /* Finding NUL. */
(void) strcpy(one, "ababa");
check(strrchr(one, 'b') == one + 3, 6);          /* Finding last. */
(void) strcpy(one, "");
check(strrchr(one, 'b') == NULL, 7);           /* Empty string. */
check(strrchr(one, '\0') == one, 8);           /* NUL in empty string. */

/* Rindex - just like strrchr */
it = "rindex";
check(rindex("abcd", 'z') == NULL, 1);          /* Not found. */
(void) strcpy(one, "abcd");
check(rindex(one, 'c') == one + 2, 2);          /* Basic test. */
check(rindex(one, 'd') == one + 3, 3);          /* End of string. */
check(rindex(one, 'a') == one, 4);             /* Beginning. */
check(rindex(one, '\0') == one + 4, 5);         /* Finding NUL. */
(void) strcpy(one, "ababa");
check(rindex(one, 'b') == one + 3, 6);          /* Finding last. */
(void) strcpy(one, "");
check(rindex(one, 'b') == NULL, 7);           /* Empty string. */
check(rindex(one, '\0') == one, 8);           /* NUL in empty string. */
}

void second()
{
    /* Strpbrk - somewhat like strchr */
    it = "strpbrk";
    check(strpbrk("abcd", "z") == NULL, 1);      /* Not found. */
    (void) strcpy(one, "abcd");
    check(strpbrk(one, "c") == one + 2, 2);      /* Basic test. */
    check(strpbrk(one, "d") == one + 3, 3);      /* End of string. */
    check(strpbrk(one, "a") == one, 4);          /* Beginning. */
    check(strpbrk(one, "") == NULL, 5);          /* Empty search list. */
    check(strpbrk(one, "cb") == one + 1, 6);      /* Multiple search. */
    (void) strcpy(one, "abcabdea");
    check(strpbrk(one, "b") == one + 1, 7);      /* Finding first. */
    check(strpbrk(one, "cb") == one + 1, 8);      /* With multiple search. */
    check(strpbrk(one, "db") == one + 1, 9);      /* Another variant. */
    (void) strcpy(one, "");
    check(strpbrk(one, "bc") == NULL, 10);        /* Empty string. */
    check(strpbrk(one, "") == NULL, 11);         /* Both strings empty. */
}

```

```

/* Strstr - somewhat like strchr */
it = "strstr";
check(strstr("abcd", "z") == NULL, 1); /* Not found. */
check(strstr("abcd", "abx") == NULL, 2); /* Dead end. */
(void) strcpy(one, "abcd");
check(strstr(one, "c") == one + 2, 3); /* Basic test. */
check(strstr(one, "bc") == one + 1, 4); /* Multichar. */
check(strstr(one, "d") == one + 3, 5); /* End of string. */
check(strstr(one, "cd") == one + 2, 6); /* Tail of string. */
check(strstr(one, "abc") == one, 7); /* Beginning. */
check(strstr(one, "abcd") == one, 8); /* Exact match. */
check(strstr(one, "abcde") == NULL, 9); /* Too long. */
check(strstr(one, "de") == NULL, 10); /* Past end. */
#ifdef NOT_ANSI
check(strstr(one, "") == one + 4, 11); /* Finding empty. */
#else
check(strstr(one, "") == one, 11); /* Finding empty. */
#endif
(void) strcpy(one, "ababa");
check(strstr(one, "ba") == one + 1, 12); /* Finding first. */
(void) strcpy(one, "");
check(strstr(one, "b") == NULL, 13); /* Empty string. */
check(strstr(one, "") == one, 14); /* Empty in empty string. */
(void) strcpy(one, "bcbca");
check(strstr(one, "bca") == one + 2, 15); /* False start. */
(void) strcpy(one, "bbbcabbca");
check(strstr(one, "bbca") == one + 1, 16); /* With overlap. */

/* Strspn */
it = "strspn";
check(strspn("abcba", "abc") == 5, 1); /* Whole string. */
check(strspn("abcba", "ab") == 2, 2); /* Partial. */
check(strspn("abc", "qx") == 0, 3); /* None. */
check(strspn("", "ab") == 0, 4); /* Null string. */
check(strspn("abc", "") == 0, 5); /* Null search list. */

/* Strcspn */
it = "strcspn";
check(strcspn("abcba", "qx") == 5, 1); /* Whole string. */
check(strcspn("abcba", "cx") == 2, 2); /* Partial. */
check(strcspn("abc", "abc") == 0, 3); /* None. */
check(strcspn("", "ab") == 0, 4); /* Null string. */
check(strcspn("abc", "") == 3, 5); /* Null search list. */

/* Strtok - the hard one */
it = "strtok";
(void) strcpy(one, "first,second,third");
equal(strtok(one, ","), "first", 1); /* Basic test. */
equal(one, "first", 2);
equal(strtok((char *) NULL, ","), "second", 3);
equal(strtok((char *) NULL, ","), "third", 4);
check(strtok((char *) NULL, ",") == NULL, 5);
(void) strcpy(one, "first,");
equal(strtok(one, ","), "first", 6); /* Extra delims, 1 tok. */
check(strtok((char *) NULL, ",") == NULL, 7);
(void) strcpy(one, "1a,1b;2a,2b");
equal(strtok(one, ","), "1a", 8); /* Changing delim lists. */
equal(strtok((char *) NULL, ";"), "1b", 9);
equal(strtok((char *) NULL, ","), "2a", 10);
(void) strcpy(two, "x-y");
equal(strtok(two, "-"), "x", 11); /* New string before done. */
equal(strtok((char *) NULL, "-"), "y", 12);
check(strtok((char *) NULL, "-") == NULL, 13);
(void) strcpy(one, "a,b,c,,d");
equal(strtok(one, ","), "a", 14); /* Different separators. */
equal(strtok((char *) NULL, ","), "b", 15);
equal(strtok((char *) NULL, ","), "c", 16); /* Permute list too. */
equal(strtok((char *) NULL, ","), "d", 17);
check(strtok((char *) NULL, ",") == NULL, 18);
check(strtok((char *) NULL, ",") == NULL, 19); /* Persistence. */
(void) strcpy(one, "");
check(strtok(one, ",") == NULL, 20); /* No tokens. */
(void) strcpy(one, "");
check(strtok(one, ",") == NULL, 21); /* Empty string. */

```

```

(void) strcpy(one, "abc");
equal(strtok(one, ","), "abc", 22); /* No delimiters. */
check(strtok((char *) NULL, ",") == NULL, 23);
(void) strcpy(one, "abc");
equal(strtok(one, ""), "abc", 24); /* Empty delimiter list. */
check(strtok((char *) NULL, "") == NULL, 25);
(void) strcpy(one, "abcdefgh");
(void) strcpy(one, "a,b,c");
equal(strtok(one, ","), "a", 26); /* Basics again... */
equal(strtok((char *) NULL, ","), "b", 27);
equal(strtok((char *) NULL, ","), "c", 28);
check(strtok((char *) NULL, ",") == NULL, 29);
equal(one + 6, "gh", 30); /* Stomped past end? */
equal(one, "a", 31); /* Stomped old tokens? */
equal(one + 2, "b", 32);
equal(one + 4, "c", 33);

/* Memcmp */
it = "memcmp";
check(memcmp("a", "a", 1) == 0, 1); /* Identity. */
check(memcmp("abc", "abc", 3) == 0, 2); /* Multicharacter. */
check(memcmp("abcd", "abce", 4) < 0, 3); /* Honestly unequal. */
check(memcmp("abce", "abcd", 4) > 0, 4);
check(memcmp("alph", "beta", 4) < 0, 5);

#ifdef NOT_ANSI
if (charsigned)
    check(memcmp("a\203", "a\003", 2) < 0, 6);
else
    check(memcmp("a\203", "a\003", 2) > 0, 6);
#else
    check(memcmp("a\203", "a\003", 2) > 0, 6);
#endif

check(memcmp("abce", "abcd", 3) == 0, 7); /* Count limited. */
check(memcmp("abc", "def", 0) == 0, 8); /* Zero count. */

check(memcmp("a" + 1, "a" + 1, 1) == 0, 9); /* Unaligned tests. */
check(memcmp("abc" + 1, "bc", 2) == 0, 10);
check(memcmp("bc", "abc" + 1, 2) == 0, 11);
check(memcmp("abcd" + 1, "abce" + 1, 3) < 0, 12);
check(memcmp("abce" + 1, "abcd" + 1, 3) > 0, 13);
/*
    check(memcmp("a\203" + 1, "a\003" + 1, 1) > 0, 14);
*/
check(memcmp("abcde" + 1, "abcdf" + 1, 3) == 0, 15);

/* Memchr */
it = "memchr";
check(memchr("abcd", 'z', 4) == NULL, 1); /* Not found. */
(void) strcpy(one, "abcd");
check((char *)memchr(one, 'c', 4) == one + 2, 2); /* Basic test. */
check((char *)memchr(one, 'd', 4) == one + 3, 3); /* End of string. */
check((char *)memchr(one, 'a', 4) == one, 4); /* Beginning. */
check((char *)memchr(one, '\0', 5) == one + 4, 5); /* Finding NUL. */
(void) strcpy(one, "ababa");
check((char *)memchr(one, 'b', 5) == one + 1, 6); /* Finding first. */
check((char *)memchr(one, 'b', 0) == NULL, 7); /* Zero count. */
check((char *)memchr(one, 'a', 1) == one, 8); /* Singleton case. */
(void) strcpy(one, "a\203b");
check((char *)memchr(one, 0203, 3) == one + 1, 9); /* Unsignedness. */

/* Malloc */
/* Note that X3J11 says memcpy may fail on overlap. */
it = "memcpy";
check((char *)memcpy(one, "abc", 4) == one, 1); /* Returned value. */
equal(one, "abc", 2); /* Did the copy go right? */

(void) strcpy(one, "abcdefgh");
(void) memcpy(one + 1, "xyz", 2);
equal(one, "axydefgh", 3); /* Basic test. */

(void) strcpy(one, "abc");
(void) memcpy(one, "xyz", 0);

```

```
equal(one, "abc", 4);          /* Zero-length copy. */

(void) strcpy(one, "hi there");
(void) strcpy(two, "foo");
(void) memcpy(two, one, 9);
equal(two, "hi there", 5);     /* Just paranoia. */
equal(one, "hi there", 6);     /* Stomped on source? */

(void) strcpy(one, "abcde");   /* Unaligned tests. */
(void) memcpy(one + 1, "\0\0\0\0\0", 1);
equal(one, "a", 7);
equal(one + 2, "cde", 8);
(void) memcpy(one + 1, "xyz" + 1, 2);
equal(one, "ayzde", 9);
(void) memcpy(one + 1, "xyz" + 1, 3);
equal(one, "ayz", 10);

/* Memmove
 * Note that X3J11 says memmove must work regardless of overlap. */
it = "memmove";
check((char *)memmove(one, "abc", 4) == one, 1); /* Returned value. */
equal(one, "abc", 2);          /* Did the copy go right? */

(void) strcpy(one, "abcdefgh");
(void) memmove(one + 1, "xyz", 2);
equal(one, "axydefgh", 3);     /* Basic test. */

(void) strcpy(one, "abc");
(void) memmove(one, "xyz", 0);
equal(one, "abc", 4);         /* Zero-length copy. */

(void) strcpy(one, "hi there");
(void) strcpy(two, "foo");
(void) memmove(two, one, 9);
equal(two, "hi there", 5);     /* Just paranoia. */
equal(one, "hi there", 6);     /* Stomped on source? */

(void) strcpy(one, "abcdefgh");
(void) memmove(one + 1, one, 9);
equal(one, "aabcdefgh", 7);   /* Overlap, right-to-left. */

(void) strcpy(one, "abcdefgh");
(void) memmove(one + 1, one + 2, 7);
equal(one, "acdefgh", 8);     /* Overlap, left-to-right. */

(void) strcpy(one, "abcdefgh");
(void) memmove(one, one, 9);
equal(one, "abcdefgh", 9);    /* 100% overlap. */

(void) strcpy(one, "abcde");   /* Unaligned tests. */
(void) memmove(one + 1, "\0\0\0\0\0", 1);
equal(one, "a", 10);
equal(one + 2, "cde", 11);
(void) memmove(one + 1, "xyz" + 1, 2);
equal(one, "ayzde", 12);
(void) memmove(one + 1, "xyz" + 1, 3);
equal(one, "ayz", 13);
(void) strcpy(one, "abcdefgh");
(void) memmove(one + 2, one + 1, 8);
equal(one, "abbcdefgh", 14);

/* Memccpy - first test like memcpy, then the search part
 * The SVID, the only place where memccpy is mentioned, says overlap
 * might fail, so we don't try it. Besides, it's hard to see the
 * rationale for a non-left-to-right memccpy. */
it = "memccpy";
check(memccpy(one, "abc", 'q', 4) == NULL, 1); /* Returned value. */
equal(one, "abc", 2);          /* Did the copy go right? */

(void) strcpy(one, "abcdefgh");
(void) memccpy(one + 1, "xyz", 'q', 2);
equal(one, "axydefgh", 3);     /* Basic test. */

(void) strcpy(one, "abc");
```

```
(void) memcpy(one, "xyz", 'q', 0);
equal(one, "abc", 4);          /* Zero-length copy. */

(void) strcpy(one, "hi there");
(void) strcpy(two, "foo");
(void) memcpy(two, one, 'q', 9);
equal(two, "hi there", 5);      /* Just paranoia. */
equal(one, "hi there", 6);      /* Stomped on source? */

(void) strcpy(one, "abcdefgh");
(void) strcpy(two, "horsefeathers");
check((char *)memcpy(two, one, 'f', 9) == two + 6, 7); /* Returned value. */
equal(one, "abcdefgh", 8);      /* Source intact? */
equal(two, "abcfeathers", 9);    /* Copy correct? */

(void) strcpy(one, "abcd");
(void) strcpy(two, "bumblebee");
check((char *)memcpy(two, one, 'a', 4) == two + 1, 10); /* First char. */
equal(two, "aumblebee", 11);
check((char *)memcpy(two, one, 'd', 4) == two + 4, 12); /* Last char. */
equal(two, "abcdlebee", 13);
(void) strcpy(one, "xyz");
check((char *)memcpy(two, one, 'x', 1) == two + 1, 14); /* Singleton. */
equal(two, "xbcdlebee", 15);

/* Memsset */
it = "memset";
(void) strcpy(one, "abcdefgh");
check((char *)memset(one + 1, 'x', 3) == one + 1, 1); /* Return value. */
equal(one, "axxxefgh", 2);      /* Basic test. */

(void) memset(one + 2, 'y', 0);
equal(one, "axxxefgh", 3);      /* Zero-length set. */

(void) memset(one + 5, 0, 1);
equal(one, "axxxe", 4);          /* Zero fill. */
equal(one + 6, "gh", 5);         /* And the leftover. */

(void) memset(one + 2, 010045, 1);
equal(one, "ax\045xe", 6);      /* Unsigned char convert. */

/* Bcopy - much like memcpy
 * Berklix manual is silent about overlap, so don't test it. */
it = "bcopy";
(void) bcopy("abc", one, 4);
equal(one, "abc", 1);           /* Simple copy. */

(void) strcpy(one, "abcdefgh");
(void) bcopy("xyz", one + 1, 2);
equal(one, "axydefgh", 2);      /* Basic test. */

(void) strcpy(one, "abc");
(void) bcopy("xyz", one, 0);
equal(one, "abc", 3);           /* Zero-length copy. */

(void) strcpy(one, "hi there");
(void) strcpy(two, "foo");
(void) bcopy(one, two, 9);
equal(two, "hi there", 4);      /* Just paranoia. */
equal(one, "hi there", 5);      /* Stomped on source? */

/* Bzero */
it = "bzero";
(void) strcpy(one, "abcdef");
bzero(one + 2, 2);
equal(one, "ab", 1);             /* Basic test. */
equal(one + 3, "", 2);
equal(one + 4, "ef", 3);

(void) strcpy(one, "abcdef");
bzero(one + 2, 0);
equal(one, "abcdef", 4);        /* Zero-length copy. */

/* Bcmp - somewhat like memcmp */
```

```
it = "bcmp";
check(bcmp("a", "a", 1) == 0, 1);      /* Identity. */
check(bcmp("abc", "abc", 3) == 0, 2);   /* Multicharacter. */
check(bcmp("abcd", "abce", 4) != 0, 3); /* Honestly unequal. */
check(bcmp("abce", "abcd", 4) != 0, 4);
check(bcmp("alph", "beta", 4) != 0, 5);
check(bcmp("abce", "abcd", 3) == 0, 6); /* Count limited. */
check(bcmp("abc", "def", 0) == 0, 8);   /* Zero count. */

#ifdef ERR
/* Strerror - VERY system-dependent */
it = "strerror";
f = open("/", O_WRONLY);               /* Should always fail. */
check(f < 0 && errno > 0 && errno < sys_nerr, 1);
equal(strerror(errno), sys_errlist[errno], 2);
#endif
#ifdef UNIXERR
equal(strerror(errno), "Is a directory", 3);
#endif
#ifdef BERKERR
equal(strerror(errno), "Permission denied", 3);
#endif
#endif
}

void quit()
{
    chdir("..");
    system("rm -rf DIR*");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
        printf("%d errors\n", errct);
        exit(1);
    }
}
```

```

/* test 16 */

#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <utime.h>
#include <stdio.h>

#define MAX_ERROR 4

int errct, subtest, passes;

_PROTOTYPE(int main, (int argc, char *argv []));
_PROTOTYPE(void test16a, (void));
_PROTOTYPE(void get_times, (char *name, time_t *a, time_t *c, time_t *m));
_PROTOTYPE(void e, (int n));
_PROTOTYPE(void quit, (void));

int main(argc, argv)
int argc;
char *argv[];
{
    int i, m;

    m = (argc == 2 ? atoi(argv[1]) : 0xFFFF);

    system("rm -rf DIR_16; mkdir DIR_16");
    chdir("DIR_16");

    printf("Test 16 ");
    fflush(stdout);
    for (i = 0; i < 4; i++) {
        if (m & 0001) test16a();
        passes++;
    }
    quit();
    return(-1);                /* impossible */
}

void test16a()
{
    /* Test atime, ctime, and mtime. */

    int fd, fd1, fd2, fd3, fd4;
    time_t a, c, m, pa, pc, pm, xa, xc, xm, ya, yc, ym, za, zc, zm, ta, tc, tm;
    time_t wa, wc, wm;
    char buf[1024];
    struct stat s;

    subtest = 1;
    if (passes > 0) return;    /* takes too long to repeat this test */

    if ( (fd = creat("Tl6.a", 0666)) < 0) e(1);
    if (write(fd, buf, 1024) != 1024) e(2);
    if (close(fd) < 0) e(3);
    sleep(1);                /* wait 1 sec before continuing */
    if ( (fd = open("Tl6.a", O_RDONLY)) < 0) e(4);
    if (read(fd, buf, 3) != 3) e(5);
    if (close(fd) != 0) e(6);
    if (stat("Tl6.a", &s) != 0) e(7);
    a = s.st_atime;
    c = s.st_ctime;
    m = s.st_mtime;
    if (a == 0) {
        /* Almost certainly means we are running a V1 file system. */
        printf(" (atime = 0. Probably V1 file system. V2 tests skipped.) ");
        return;
    }

    /* Many system calls affect atime, ctime, and mtime. Test them. They
     * fall into several groups. The members of each group can be tested

```



```
* together. Start with creat(), mkdir(), and mkfifo, all of which
* set all 3 times on the created object, and ctime and mtime of the dir.
*/
if ( (fd = creat("Tl6.b", 0666)) < 0) e(8);
if (close(fd) != 0) e(9);
get_times("Tl6.b", &a, &c, &m);
get_times(".", &pa, &pc, &pm);
if (a != c) e(10);
if (a != m) e(11);
if (a != pc) e(12);
if (a != pm) e(13);
if (unlink("Tl6.b") < 0) e(14);

/* Test the times for mkfifo. */
if ( (fd = mkfifo("Tl6.c", 0666)) != 0) e(15);
if (access("Tl6.c", R_OK | W_OK) != 0) e(16);
get_times("Tl6.c", &a, &c, &m);
get_times(".", &pa, &pc, &pm);
if (a != c) e(17);
if (a != m) e(18);
if (a != pc) e(19);
if (a != pm) e(20);
if (unlink("Tl6.c") < 0) e(21);

/* Test the times for mkdir. */
if (mkdir("Tl6.d", 0666) < 0) e(22);
get_times("Tl6.d", &a, &c, &m);
get_times(".", &pa, &pc, &pm);
if (a != c) e(23);
if (a != m) e(24);
if (a != pc) e(25);
if (a != pm) e(26);
sleep(1);
if (rmdir("Tl6.d") < 0) e(27);
get_times(".", &xa, &xc, &xm);
if (c == xc) e(28);
if (m == xm) e(29);
if (xc != xm) e(30);

/* Test open(file, O_TRUNC). */
if ( (fd = open("Tl6.e", O_WRONLY|O_CREAT, 0666)) < 0) e(31);
if (write(fd, buf, 1024) != 1024) e(32);
if (close(fd) != 0) e(33);
get_times("Tl6.e", &a, &c, &m);
get_times(".", &pa, &pc, &pm);
sleep(1);
if ( (fd = open("Tl6.e", O_WRONLY|O_TRUNC)) < 0) e(34);
get_times("Tl6.e", &xa, &xc, &xm);
get_times(".", &ya, &yc, &ym);
if (c != m) e(35);
if (pc != pm) e(36);
if (c == xc) e(37);
if (m == xm) e(38);
if (yc != pc) e(39);
if (ym != pm) e(40);
if (close(fd) != 0) e(41);

/* Test the times for link/unlink. */
get_times("Tl6.e", &a, &c, &m);
get_times(".", &ya, &yc, &ym);
sleep(1);
if (link("Tl6.e", "Tl6.f") != 0) e(42);          /* second link */
get_times("Tl6.e", &xa, &xc, &xm);
get_times(".", &pa, &pc, &pm);
if (a != xa) e(43);
if (m != xm) e(44);
#ifdef V1_FILESYSTEM
if (c == xc) e(45);
#endif
if (ya != pa) e(46);
if (yc == pc) e(47);
if (ym == pm) e(48);
if (yc != ym) e(49);
if (pc != pm) e(50);
```

```

sleep(1);
if (unlink("Tl6.f") != 0) e(46);
get_times("Tl6.e", &a, &c, &m);
get_times(".", &ya, &yc, &ym);
if (a != xa) e(51);
if (m != xm) e(52);
#ifdef V1_FILESYSTEM
if (c == xc) e(53);
#endif
if (pa != ya) e(54);
if (pc == yc) e(55);
if (pm == ym) e(56);
if (yc != ym) e(57);
if (unlink("Tl6.e") != 0) e(58);

/* Test rename, read, write, chmod, utime. */
get_times(".", &pa, &pc, &pm);
if ( (fd = open("Tl6.g", O_RDWR|O_CREAT)) < 0) e(59);
if ( (fd1 = open("Tl6.h", O_WRONLY|O_CREAT, 0666)) < 0) e(60);
if ( (fd2 = open("Tl6.i", O_WRONLY|O_CREAT, 0666)) < 0) e(61);
if ( (fd3 = open("Tl6.j", O_WRONLY|O_CREAT, 0666)) < 0) e(62);
if ( (fd4 = open("Tl6.k", O_RDWR|O_CREAT, 0666)) < 0) e(63);
if (write(fd, buf, 1024) != 1024) e(64);
get_times("Tl6.g", &a, &c, &m);
get_times("Tl6.h", &pa, &pc, &pm);
get_times("Tl6.i", &xa, &xc, &xm);
get_times("Tl6.j", &ya, &yc, &ym);
get_times("Tl6.k", &za, &zc, &zm);
get_times(".", &wa, &wc, &wm);
sleep(1);
lseek(fd, 0L, SEEK_SET);
if (read(fd, buf, 35) != 35) e(65);
get_times("Tl6.g", &ta, &tc, &tm);
if (a == ta || c != tc || m != tm) e(66);
if (write(fd1, buf, 35) != 35) e(67);
get_times("Tl6.h", &ta, &tc, &tm);
if (pa != ta || pc == tc || pm == tm) e(69);
if (rename("Tl6.i", "Tl6.il") != 0) e(70);
get_times("Tl6.il", &ta, &tc, &tm);
if (xa != ta || xc != tc || xm != tm) e(71);
get_times(".", &a, &c, &m);
if (a != wa || c == wc || m == wm || wc != wm) e(72);
if (chmod("Tl6.j", 0777) != 0) e(73);
get_times("Tl6.j", &ta, &tc, &tm);
if (ya != ta || yc == tc || ym != tm) e(74);
if (utime("Tl6.k", (void *) 0) != 0) e(75);
get_times("Tl6.k", &ta, &tc, &tm);
if (za == ta || zc == tc) e(76);
if (close(fd) != 0) e(77);
if (close(fd1) != 0) e(78);
if (close(fd2) != 0) e(79);
if (close(fd3) != 0) e(80);
if (close(fd4) != 0) e(81);
if (unlink("Tl6.g") != 0) e(82);
if (unlink("Tl6.h") != 0) e(83);
if (unlink("Tl6.il") != 0) e(84);
if (unlink("Tl6.j") != 0) e(85);
if (unlink("Tl6.k") != 0) e(86);
}

void get_times(name, a, c, m)
char *name;
time_t *a, *c, *m;
{
    struct stat s;

    if (stat(name, &s) != 0) e(500);
    *a = s.st_atime;
    *c = s.st_ctime;
    *m = s.st_mtime;
}

void e(n)
int n;

```

```
{
    int err_num = errno;          /* save errno in case printf clobbers it */

    printf("Subtest %d, error %d errno=%d ", subtest, n, errno);
    errno = err_num;             /* restore errno, just in case */
    perror("");
    if (errct++ > MAX_ERROR) {
        printf("Too many errors; test aborted\n");
        chdir("..");
        system("rm -rf DIR*");
        exit(1);
    }
}

void quit()
{
    chdir("..");
    system("rm -rf DIR*");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
        printf("%d errors\n", errct);
        exit(1);
    }
}
```

```

/* Comment on usage and program: ark!/mnt/rene/prac/os/unix/comment.changes */

/* "const.h", created by Rene Montsma and Menno Wilcke */

#include <sys/types.h>          /* type defs */
#include <sys/stat.h>           /* struct stat */
#include <sys/wait.h>
#include <errno.h>              /* the error-numbers */
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <utime.h>
#include <stdio.h>
#include <limits.h>

#define NOCRASH 1              /* test11(), 2nd pipe */
#define PDPNOHANG 1           /* test03(), write_standards() */
#define MAXERR 2

#define USER_ID 12
#define GROUP_ID 1
#define FF 3                   /* first free filedес. */
#define USER 1                 /* uid */
#define GROUP 0                /* gid */

#define ARSIZE 256             /* array size */
#define PIPESIZE 3584          /* max number of bytes to be written on pipe */
#define MAXOPEN (OPEN_MAX-3)  /* maximum number of extra open files */
#define MAXLINK 0177          /* maximum number of links per file */
#define LINKCOUNT 5
#define MASK 0777              /* selects lower nine bits */
#define END_FILE 0             /* returned by read-call at eof */

#define OK 0
#define FAIL -1

#define R 0                    /* read (open-call) */
#define W 1                    /* write (open-call) */
#define RW 2                   /* read & write (open-call) */

#define RWX 7                  /* read & write & execute (mode) */

#define NIL " "
#define UMASK "umask"
#define CREAT "creat"
#define WRITE "write"
#define READ "read"
#define OPEN "open"
#define CLOSE "close"
#define LSEEK "lseek"
#define ACCESS "access"
#define CHDIR "chdir"
#define CHMOD "chmod"
#define LINK "link"
#define UNLINK "unlink"
#define PIPE "pipe"
#define STAT "stat"
#define FSTAT "fstat"
#define DUP "dup"
#define UTIME "utime"

int errct;

char *file[];
char *fnames[];
char *dir[];

/* "decl.c", created by Rene Montsma and Menno Wilcke */

/* Used in open_alot, close_alot */
char *file[20] = {"f0", "f1", "f2", "f3", "f4", "f5", "f6",
                 "f7", "f8", "f9", "f10", "f11", "f12", "f13",
                 "f14", "f15", "f16", "f17", "f18", "f19"}, *fnames[8] = {"---", "--x", "-w-", "-wx",

```

```

"r--",
                                                                    "r-x", "rw-", "rwx"}, *
dir[8] = {"d---", "d--x", "d-w-", "d-wx", "dr--", "dr-x",
                                                                    "drw-", "drwx"};
/* Needed for easy creating and deleting of directories */
/* "test.c", created by Rene Montsma and Menno Wilcke */

_PROTOTYPE(int main, (int argc, char *argv []));
_PROTOTYPE(void test, (int mask));
_PROTOTYPE(void test01, (void));
_PROTOTYPE(void test02, (void));
_PROTOTYPE(void test08, (void));
_PROTOTYPE(void test09, (void));
_PROTOTYPE(void test10, (void));
_PROTOTYPE(int link_alot, (char *bigboss));
_PROTOTYPE(int unlink_alot, (int number));
_PROTOTYPE(void get_new, (char name []));
_PROTOTYPE(void test11, (void));
_PROTOTYPE(void comp_stats, (struct stat *stbf1, struct stat *stbf2));
_PROTOTYPE(void comp_inodes, (int m, int ml));
_PROTOTYPE(void e, (char *string));
_PROTOTYPE(void nlcr, (void));
_PROTOTYPE(void str, (char *s));
_PROTOTYPE(void err, (int number, char *scall, char *name));
_PROTOTYPE(void make_and_fill_dirs, (void));
_PROTOTYPE(void put_file_in_dir, (char *dirname, int mode));
_PROTOTYPE(void init_array, (char *a));
_PROTOTYPE(void clear_array, (char *b));
_PROTOTYPE(int comp_array, (char *a, char *b, int range));
_PROTOTYPE(void try_close, (int filedес, char *name));
_PROTOTYPE(void try_unlink, (char *fname));
_PROTOTYPE(void Remove, (int fdes, char *fname));
_PROTOTYPE(int get_mode, (char *name));
_PROTOTYPE(void check, (char *scall, int number));
_PROTOTYPE(void put, (int nr));
_PROTOTYPE(int open_alot, (void));
_PROTOTYPE(int close_alot, (int number));
_PROTOTYPE(void clean_up_the_mess, (void));
_PROTOTYPE(void chmod_8_dirs, (int sw));
_PROTOTYPE(void quit, (void));

/*****
*
*                               TEST
*
*****/
int main(argc, argv)
int argc;
char *argv[];
{
    int n, mask;

    sync();
    if (geteuid() == 0 || getuid() == 0) {
        printf("Test 17 cannot run as root; test aborted\n");
        exit(1);
    }

    system("rm -rf DIR_18; mkdir DIR_18");
    chdir("DIR_18");

    mask = (argc == 2 ? atoi(argv[1]) : 0xFFFF);

    if (fork()) {
        printf("Test 17 ");
        fflush(stdout);

        wait(&n);
        clean_up_the_mess();
        quit();
    } else {
        test(mask);
        exit(0);
    }
    return(-1);
}
/* impossible */

```

```

}

void test(mask)
int mask;
{
    umask(0);                /* not honest, but i always forget */

    if (mask & 00001) test01();
    if (mask & 00002) make_and_fill_dirs();
    if (mask & 00004) test02();
    if (mask & 00010) test08();
    if (mask & 00020) test09();
    if (mask & 00040) test10();
    if (mask & 00100) test11();
    umask(022);
}                               /* test */

/* "t1.c" created by Rene Montsma and Menno Wilcke */

/*****
 *                               test UMASK                               *
 *****/
void test01()
{
    int oldvalue, newvalue, tempvalue;
    int nr;

    if ((oldvalue = umask(0777)) != 0) err(0, UMASK, NIL);

    /* Special test: only the lower 9 bits (protection bits) may part- *
     * icipate. ~0777 means: 111 000 000 000. Giving this to umask must*
     * not change any value.                                           */

    if ((newvalue = umask(~0777)) != 0777) err(1, UMASK, "illegal");
    if (oldvalue == newvalue) err(11, UMASK, "not change mask");

    if ((tempvalue = umask(0)) != 0) err(2, UMASK, "values");

    /* Now test all possible modes of umask on a file */
    for (newvalue = MASK; newvalue >= 0; newvalue -= 0111) {
        tempvalue = umask(newvalue);
        if (tempvalue != oldvalue) {
            err(1, UMASK, "illegal");
            break;                /* no use trying more */
        } else if ((nr = creat("file01", 0777)) < 0)
            err(5, CREAT, "'file01'");
        else {
            try_close(nr, "'file01'");
            if (get_mode("file01") != (MASK & ~newvalue))
                err(7, UMASK, "mode computed");
            try_unlink("file01");
        }
        oldvalue = newvalue;
    }

    /* The loop has terminated with umask(0) */
    if ((tempvalue = umask(0)) != 0)
        err(7, UMASK, "umask may influence rest of tests!");
}                               /* test01 */

/*****
 *                               test CREAT                               *
 *****/
void test02()
{
    int n, n1, mode;
    char a[ARSIZE], b[ARSIZE];
    struct stat stbfl;

    mode = 0;
    /* Create twenty files, check filedess */
    for (n = 0; n < MAXOPEN; n++) {
        if (creat(file[n], mode) != FF + n)
            err(13, CREAT, file[n]);
    }
}

```

```

    else {
        if (get_mode(file[n]) != mode)
            err(7, CREAT, "mode set while creating many files");

        /* Change mode of file to standard mode, we want to *
         * use a lot (20) of files to be opened later, see   *
         * open_alot(), close_alot().                        */
        if (chmod(file[n], 0700) != OK) err(5, CHMOD, file[n]);
    }
    mode = (mode + 0100) % 01000;
}

/* Already twenty files opened; opening another has to fail */
if (creat("file02", 0777) != FAIL)
    err(9, CREAT, "created");
else
    check(CREAT, EMFILE);

/* Close all files: seems blunt, but it isn't because we've *
 * checked all fd's already                                */
if ((n = close_alot(MAXOPEN)) < MAXOPEN) err(5, CLOSE, "MAXOPEN files");

/* Creat 1 file twice; check */
if ((n = creat("file02", 0777)) < 0)
    err(5, CREAT, "'file02'");
else {
    init_array(a);
    if (write(n, a, ARSIZE) != ARSIZE) err(1, WRITE, "bad");

    if ((n1 = creat("file02", 0755)) < 0) /* recreate 'file02' */
        err(5, CREAT, "'file02' (2nd time)");
    else {
        /* Fd should be at the top after recreation */
        if (lseek(n1, 0L, SEEK_END) != 0)
            err(11, CREAT, "not truncate file by recreation");
        else {
            /* Try to write on recreated file */
            clear_array(b);

            if (lseek(n1, 0L, SEEK_SET) != 0)
                err(5, LSEEK, "to top of 2nd fd 'file02'");
            if (write(n1, a, ARSIZE) != ARSIZE)
                err(1, WRITE, "(2) bad");

            /* In order to read we've to close and open again */
            try_close(n1, "'file02' (2nd creation)");
            if ((n1 = open("file02", RW)) < 0)
                err(5, OPEN, "'file02' (2nd recreation)");

            /* Continue */
            if (lseek(n1, 0L, SEEK_SET) != 0)
                err(5, LSEEK, "to top 'file02' (2nd fd) (2)");
            if (read(n1, b, ARSIZE) != ARSIZE)
                err(1, READ, "wrong");

            if (comp_array(a, b, ARSIZE) != OK) err(11, CREAT,
                "not really truncate file by recreation");
        }
        if (get_mode("file02") != 0777)
            err(11, CREAT, "not maintain mode by recreation");
        try_close(n1, "recreated 'file02'");
    }
    Remove(n, "file02");
}

/* Give 'creat' wrong input: dir not searchable */
if (creat("drw-/file02", 0777) != FAIL)
    err(4, CREAT, "'drw-'");
else
    check(CREAT, EACCES);

/* Dir not writable */

```

```
if (creat("dr-x/file02", 0777) != FAIL)
    err(12, CREAT, "'dr-x/file02'");
else
    check(CREAT, EACCES);

/* File not writable */
if (creat("drwx/r-x", 0777) != FAIL)
    err(11, CREAT, "recreate non-writable file");
else
    check(CREAT, EACCES);

/* Try to creat a dir */
if ((n = creat("dir", 040777)) != FAIL) {
    if (fstat(n, &stbfl) != OK)
        err(5, FSTAT, "'dir'");
    else if (stbfl.st_mode != (mode_t) 0100777)
        /* cast because mode is negative :-( */
        err(11, CREAT, "'creat' a new directory");
    Remove(n, "dir");
}

/* We don't consider it to be a bug when creat * does not accept
 * tricky modes */

/* File is an existing dir */
if (creat("drwx", 0777) != FAIL)
    err(11, CREAT, "create an existing dir!");
else
    check(CREAT, EISDIR);
} /* test02 */

void test08()
{
    /* Test chdir to searchable dir */
    if (chdir("drwx") != OK)
        err(5, CHDIR, "to accessible dir");
    else if (chdir("..") != OK)
        err(11, CHDIR, "not return to '..");

    /* Check the chdir(".") and chdir("..") mechanism */
    if (chdir("drwx") != OK)
        err(5, CHDIR, "to 'drwx'");
    else {
        if (chdir(".") != OK) err(5, CHDIR, "to working dir (.");

        /* If we still are in 'drwx' , we should be able to access *
         * file 'rwx'. */
        if (access("rwx", 0) != OK) err(5, CHDIR, "rightly to '.'");

        /* Try to return to previous dir ('/' !!) */
        if (chdir(".././d--x/./d--x/./..") != OK)
            err(5, CHDIR, "to motherdir(..");

        /* Check whether we are back in '/' */
        if (chdir("d--x") != OK) err(5, CHDIR, "rightly to a '..");
    }

    /* Return to '..' */
    if (chdir("..") != OK) err(5, CHDIR, "to '..");

    if (chdir(".././drwx") != OK)
        err(11, CHDIR, "not follow a path");
    else if (chdir(".././..") != OK)
        err(11, CHDIR, "not return to path");

    /* Try giving chdir wrong parameters */
    if (chdir("drwx/rwx") != FAIL)
        err(11, CHDIR, "chdir to a file");
    else
        check(CHDIR, ENOTDIR);

    if (chdir("drw-") != FAIL)
        err(4, CHDIR, "'/drw-'");
    else
```



```
    check(CHDIR, EACCES);

    /* To be sure: return to root */
    /* If (chdir("/") != OK) err(5, CHDIR, "to '/' (2nd time)"); */
}
    /* test08 */

/* New page */
```

```
/* *****  
 *                               test CHMOD                               *  
 * ***** */  
void test09()  
{  
    int n;  
  
    /* Prepare file09 */  
    if ((n = creat("drwx/file09", 0644)) != FF) err(5, CREAT, "'drwx/file09'");  
  
    try_close(n, "'file09'");  
  
    /* Try to chmod a file, check and restore old values, check */  
    if (chmod("drwx/file09", 0700) != OK)  
        err(5, CHMOD, "'drwx/file09'"); /* set rwx */  
    else {  
        /* Check protection */  
        if (get_mode("drwx/file09") != 0700) err(7, CHMOD, "mode");  
  
        /* Test if chmod accepts just filenames too */  
        if (chdir("drwx") != OK)  
            err(5, CHDIR, "to 'drwx'");  
        else if (chmod("file09", 0177) != OK) /* restore oldies */  
            err(5, CHMOD, "'h1'");  
        else  
            /* Check if value has been restored */  
            if (get_mode("../drwx/file09") != 0177)  
                err(7, CHMOD, "restored mode");  
    }  
  
    /* Try setuid and setgid */  
    if ((chmod("file09", 04777) != OK) || (get_mode("file09") != 04777))  
        err(11, CHMOD, "not set uid-bit");  
    if ((chmod("file09", 02777) != OK) || (get_mode("file09") != 02777))  
        err(11, CHMOD, "not set gid-bit");  
  
    /* Remove testfile */  
    try_unlink("file09");  
  
    if (chdir("..") != OK) err(5, CHDIR, "to '..'");  
  
    /* Try to chmod directory */  
    if (chmod("d---", 0777) != OK)  
        err(5, CHMOD, "dir 'd---'");  
    else {  
        if (get_mode("d---") != 0777) err(7, CHMOD, "protection value");  
        if (chmod("d---", 0000) != OK) err(5, CHMOD, "dir 'a' 2nd time");  
  
        /* Check if old value has been restored */  
        if (get_mode("d---") != 0000)  
            err(7, CHMOD, "restored protection value");  
    }  
  
    /* Try to make chmod failures */  
  
    /* We still are in dir root */  
    /* Wrong filename */  
    if (chmod("non-file", 0777) != FAIL)  
        err(3, CHMOD, NIL);  
    else  
        check(CHMOD, ENOENT);  
}  
/* test 09 */
```

```
/* New page */
```

```
/* "t4.c", created by Rene Montsma and Menno Wilcke */

/*****
 *                               test LINK/UNLINK                               *
 *****/
void test10()
{
    int n, nl;
    char a[ARSIZE], b[ARSIZE], *f, *lf;

    f = "file10";
    lf = "linkfile10";

    if ((n = creat(f, 0702)) != FF)          /* no other open files */
        err(13, CREAT, f);
    else {
        /* Now link correctly */
        if (link(f, lf) != OK)
            err(5, LINK, lf);
        else if ((nl = open(lf, RW)) < 0)
            err(5, OPEN, "linkfile10");
        else {
            init_array(a);
            clear_array(b);

            /* Write on 'file10' means being able to      * read
             * through linked filedescriptor                */
            if (write(n, a, ARSIZE) != ARSIZE) err(1, WRITE, "bad");
            if (read(nl, b, ARSIZE) != ARSIZE) err(1, READ, "bad");
            if (comp_array(a, b, ARSIZE) != OK) err(8, "r/w", NIL);

            /* Clean up: unlink and close (twice): */
            Remove(n, f);
            try_close(nl, "linkfile10");

            /* Check if "linkfile" exists and the info      * on it
             * is correct ('file' has been deleted) */
            if ((nl = open(lf, R)) < 0)
                err(5, OPEN, "linkfile10");
            else {
                /* See if 'linkfile' still contains 0..511 ? */

                clear_array(b);
                if (read(nl, b, ARSIZE) != ARSIZE)
                    err(1, READ, "bad");
                if (comp_array(a, b, ARSIZE) != OK)
                    err(8, "r/w", NIL);

                try_close(nl, "linkfile10' 2nd time");
                try_unlink(lf);
            }
        }
    }

    /* Try if unlink fails with incorrect parameters */
    /* File does not exist: */
    if (unlink("non-file") != FAIL)
        err(2, UNLINK, "name");
    else
        check(UNLINK, ENOENT);

    /* Dir can't be written */
    if (unlink("dr-x/rwx") != FAIL)
        err(11, UNLINK, "could unlink in non-writable dir.");
    else
        check(UNLINK, EACCES);

    /* Try to unlink a dir being user */
    if (unlink("drwx") != FAIL)
        err(11, UNLINK, "unlink dir's as user");
    else
        check(UNLINK, EPERM);
}
```

```

/* Try giving link wrong input */

/* First try if link fails with incorrect parameters * name1 does not
 * exist. */
if (link("non-file", "linkfile") != FAIL)
    err(2, LINK, "1st name");
else
    check(LINK, ENOENT);

/* Name2 exists already */
if (link("drwx/rwx", "drwx/rw-") != FAIL)
    err(2, LINK, "2nd name");
else
    check(LINK, EEXIST);

/* Directory of name2 not writable: */
if (link("drwx/rwx", "dr-x/linkfile") != FAIL)
    err(11, LINK, "link non-writable file");
else
    check(LINK, EACCES);

/* Try to link a dir, being a user */
if (link("drwx", "linkfile") != FAIL)
    err(11, LINK, "link a dir without superuser!");
else
    check(LINK, EPERM);

/* File has too many links */
if ((n = link_alot("drwx/rwx")) != LINKCOUNT - 1)          /* file already has one
                                                             * link */
    err(5, LINK, "many files");
if (unlink_alot(n) != n) err(5, UNLINK, "all linked files");
}
/* test10 */

int link_alot(bigboss)
char *bigboss;
{
    int i;
    static char employee[6] = "aaaaa";

    /* Every file has already got 1 link, so link 0176 times */
    for (i = 1; i < LINKCOUNT; i++) {
        if (link(bigboss, employee) != OK)
            break;
        else
            get_new(employee);
    }

    return(i - 1);
}
/* number of linked files */
/* link_alot */

int unlink_alot(number)
int number;
/* number of files to be unlinked */
{
    int j;
    static char employee[6] = "aaaaa";

    for (j = 0; j < number; j++) {
        if (unlink(employee) != OK)
            break;
        else
            get_new(employee);
    }

    return(j);
}
/* return number of unlinked files */
/* unlink_alot */

void get_new(name)
char name[];
/* Every call changes string 'name' to a string alphabetically
 * higher. Start with "aaaaa", next value: "aaaab".
 * N.B. after "aaaaz" comes "aaabz" and not "aaaba" (not needed).
 * The last possibility will be "zzzzz".

```

```
 * Total # possibilities: 26+25*4 = 126 = MAXLINK -1 (exactly needed)  */
{
    int i;

    for (i = 4; i >= 0; i--)
        if (name[i] != 'z') {
            name[i]++;
            break;
        }
}                                     /* get_new */

/* New page */
```

```

/*****
 *
 *                               test PIPE
 *
 *****/
void test11()
{
    int n, fd[2];
    char a[ARSIZE], b[ARSIZE];

    if (pipe(fd) != OK)
        err(13, PIPE, NIL);
    else {
        /* Try reading and writing on a pipe */
        init_array(a);
        clear_array(b);

        if (write(fd[1], a, ARSIZE) != ARSIZE)
            err(5, WRITE, "on pipe");
        else if (read(fd[0], b, (ARSIZE / 2)) != (ARSIZE / 2))
            err(5, READ, "on pipe (2nd time)");
        else if (comp_array(a, b, (ARSIZE / 2)) != OK)
            err(7, PIPE, "values read/written");
        else if (read(fd[0], b, (ARSIZE / 2)) != (ARSIZE / 2))
            err(5, READ, "on pipe 2");
        else if (comp_array(&a[ARSIZE / 2], b, (ARSIZE / 2)) != OK)
            err(7, PIPE, "pipe created");

        /* Try to let the pipe make a mistake */
        if (write(fd[0], a, ARSIZE) != FAIL)
            err(11, WRITE, "write on fd[0]");
        if (read(fd[1], b, ARSIZE) != FAIL) err(11, READ, "read on fd[1]");

        try_close(fd[1], "'fd[1]'");

        /* Now we shouldn't be able to read, because fd[1] has been closed */
        if (read(fd[0], b, ARSIZE) != END_FILE) err(2, PIPE, "'fd[1]'");

        try_close(fd[0], "'fd[0]'");
    }
    if (pipe(fd) < 0)
        err(5, PIPE, "2nd time");
    else {
        /* Test lseek on a pipe: should fail */
        if (write(fd[1], a, ARSIZE) != ARSIZE)
            err(5, WRITE, "on pipe (2nd time)");
        if (lseek(fd[1], 10L, SEEK_SET) != FAIL)
            err(11, LSEEK, "lseek on a pipe");
        else
            check(PIPE, ESPIPE);

        /* Eat half of the pipe: no writing should be possible */
        try_close(fd[0], "'fd[0]' (2nd time)");

        /* This makes UNIX crash: omit it if pdp or VAX */
#ifdef NOCRASH
        if (write(fd[1], a, ARSIZE) != FAIL)
            err(11, WRITE, "write on wrong pipe");
        else
            check(PIPE, EPIPE);
#endif
        try_close(fd[1], "'fd[1]' (2nd time)");
    }

    /* BUG :
     * Here we planned to test if we could write 4K bytes on a pipe.
     * However, this was not possible to implement, because the whole
     * Monix system crashed when we tried to write more then 3584 bytes
     * (3.5K) on a pipe. That's why we try to write only 3.5K in the
     * following test.
     */
    if (pipe(fd) < 0)
        err(5, PIPE, "3rd time");
    else {
        for (n = 0; n < (PIPESIZE / ARSIZE); n++)
            if (write(fd[1], a, ARSIZE) != ARSIZE)

```

```
                err(5, WRITE, "on pipe (3rd time) 4K" );
try_close(fd[1], "'fd[1]' (3rd time)");

    for (n = 0; n < (PIPESIZE / ARSIZE); n++)
        if (read(fd[0], b, ARSIZE) != ARSIZE)
            err(5, READ, "from pipe (3rd time) 4K" );
try_close(fd[0], "'fd[0]' (3rd time)");
}

/* Test opening a lot of files */
if ((n = open_alot()) != MAXOPEN) err(5, OPEN, "MAXOPEN files");
if (pipe(fd) != FAIL)
    err(9, PIPE, "open");
else
    check(PIPE, EMFILE);
if (close_alot(n) != n) err(5, CLOSE, "all opened files");
}
/* test11 */

/* New page */
```

```
void comp_stats(stbfl, stbf2)
struct stat *stbfl, *stbf2;
{
    if (stbfl->st_dev != stbf2->st_dev) err(7, "st/fst", "'dev'");
    if (stbfl->st_ino != stbf2->st_ino) err(7, "st/fst", "'ino'");
    if (stbfl->st_mode != stbf2->st_mode) err(7, "st/fst", "'mode'");
    if (stbfl->st_nlink != stbf2->st_nlink) err(7, "st/fst", "'nlink'");
    if (stbfl->st_uid != stbf2->st_uid) err(7, "st/fst", "'uid'");
    if (stbfl->st_gid != stbf2->st_gid) err(7, "st/fst", "'gid'");
    if (stbfl->st_rdev != stbf2->st_rdev) err(7, "st/fst", "'rdev'");
    if (stbfl->st_size != stbf2->st_size) err(7, "st/fst", "'size'");
    if (stbfl->st_atime != stbf2->st_atime) err(7, "st/fst", "'atime'");
    if (stbfl->st_mtime != stbf2->st_mtime) err(7, "st/fst", "'mtime'");
}
/* comp_stats */

/* New page */
```



```

/* "t5.c", created by Rene Montsma and Menno Wilcke */

void comp_inodes(m, m1)
int m, m1;                                /* twee filedes's */
{
    struct stat stbf1, stbf2;

    if (fstat(m, &stbf1) == OK)
        if (fstat(m1, &stbf2) == OK) {
            if (stbf1.st_ino != stbf2.st_ino)
                err(7, DUP, "inode number");
        } else
            err(100, "comp_inodes", "cannot 'fstat' (m1)");
    else
        err(100, "comp_inodes", "cannot 'fstat' (m)");
}
/* comp_inodes */

/* "support.c", created by Rene Montsma and Menno Wilcke */

/* Err, make_and_fill_dirs, init_array, clear_array, comp_array,
   try_close, try_unlink, Remove, get_mode, check, open_alot,
   close_alot, clean_up_the_mess.
*/

/*****
 *
 *                               EXTENDED FIONS
 *
 *****/
/* First extended functions (i.e. not oldfashioned monixcalls.
   e(), nlcr(), octal.*/

void e(string)
char *string;
{
    printf("Error:%s ", string);
}

void nlcr()
{
    printf("\n");
}

void str(s)
char *s;
{
    printf(s);
}

/*****
 *
 *                               ERR(or) messages
 *
 *****/
void err(number, scall, name)
/* Give nice error messages */

char *scall, *name;
int number;

{
    errct++;
    if (errct > MAXERR) {
        printf("Too many errors; test aborted\n");
        quit();
    }
    e("");
    str("\t");
    switch (number) {
        case 0:
            str(scall);
            str(" : illegal initial value.");
            break;
        case 1:
            str(scall);

```

```
    str(": ");
    str(name);
    str(" value returned." );
    break;
case 2:
    str(scall);
    str(": accepting illegal " );
    str(name);
    str(".");
    break;
case 3:
    str(scall);
    str(": accepting non-existing file." );
    break;
case 4:
    str(scall);
    str(": could search non-searchable dir ( " );
    str(name);
    str(").");
    break;
case 5:
    str(scall);
    str(": cannot " );
    str(scall);
    str(" ");
    str(name);
    str(".");
    break;
case 7:
    str(scall);
    str(": incorrect " );
    str(name);
    str(".");
    break;
case 8:
    str(scall);
    str(": wrong values." );
    break;
case 9:
    str(scall);
    str(": accepting too many " );
    str(name);
    str(" files.");
    break;
case 10:
    str(scall);
    str(": even a superuser can't do anything!" );
    break;
case 11:
    str(scall);
    str(": could " );
    str(name);
    str(".");
    break;
case 12:
    str(scall);
    str(": could write in non-writable dir ( " );
    str(name);
    str(").");
    break;
case 13:
    str(scall);
    str(": wrong files returned ( " );
    str(name);
    str(").");
    break;
case 100:
    str(scall);                /* very common */
    str(": ");
    str(name);
    str(".");
    break;
default:    str("error number does not exist!\n" );
}
```

```

    nlcr();
}
/* err */

/*****
*
*                               MAKE_AND_FILL_DIRS
*
*****/

void make_and_fill_dirs()
/* Create 8 dir.'s: "d---", "d--x", "d-w-", "d-wx", "dr--", "dr-x",
 * "drw-", "drwx".                               * Then create 8 files
 * in "drwx", and some needed files in other dirs. */
{
    int mode, i;

    for (i = 0; i < 8; i++) {
        mkdir(dir[i], 0700);
        chown(dir[i], USER_ID, GROUP_ID);
    }
    setuid(USER_ID);
    setgid(GROUP_ID);

    for (mode = 0; mode < 8; mode++) put_file_in_dir("drwx", mode);

    put_file_in_dir("d-wx", RWX);
    put_file_in_dir("dr-x", RWX);
    put_file_in_dir("drw-", RWX);

    chmod_8_dirs(8);
/* 8 means; 8 different modes */
}
/* make_and_fill_dirs */

void put_file_in_dir(dirname, mode)
char *dirname;
int mode;
/* Fill directory 'dirname' with file with mode 'mode'. */
{
    int nr;

    if (chdir(dirname) != OK)
        err(5, CHDIR, "to dirname (put_f_in_dir)");
    else {
        /* Creat the file */
        if ((nr = creat(fnames[mode], mode * 0100)) < 0)
            err(13, CREAT, fnames[mode]);
        else
            try_close(nr, fnames[mode]);

        if (chdir("..") != OK)
            err(5, CHDIR, "to previous dir (put_f_in_dir)");
    }
}
/* put_file_in_dir */

/*****
*
*                               MISCELLANEOUS
*
*(all about arrays, 'try_close', 'try_unlink', 'Remove', 'get_mode')*
*****/

void init_array(a)
char *a;
{
    int i;

    i = 0;
    while (i++ < ARSIZE) *a++ = 'a' + (i % 26);
}
/* init_array */

void clear_array(b)
char *b;
{

```

```

int i;

i = 0;
while (i++ < ARSIZE) *b++ = '0';
}
/* clear_array */

int comp_array(a, b, range)
char *a, *b;
int range;
{
    if ((range < 0) || (range > ARSIZE)) {
        err(100, "comp_array", "illegal range");
        return(FAIL);
    } else {
        while (range-- && (*a++ == *b++));
        if (*--a == *--b)
            return(OK);
        else
            return(FAIL);
    }
}
/* comp_array */

void try_close(filedes, name)
int filedes;
char *name;
{
    if (close(filedes) != OK) err(5, CLOSE, name);
}
/* try_close */

void try_unlink(fname)
char *fname;
{
    if (unlink(fname) != 0) err(5, UNLINK, fname);
}
/* try_unlink */

void Remove(fdes, fname)
int fdes;
char *fname;
{
    try_close(fdes, fname);
    try_unlink(fname);
}
/* Remove */

int get_mode(name)
char *name;
{
    struct stat stbfl;

    if (stat(name, &stbfl) != OK) {
        err(5, STAT, name);
        return(stbfl.st_mode); /* return a mode which will cause *
                               * error in the calling function *
                               * (file/dir bit) */
    } else
        return(stbfl.st_mode & 0777); /* take last 4 bits */
}
/* get_mode */

/*****
*
*                               CHECK
*
*****/

void check(scall, number)
int number;
char *scall;
{
    if (errno != number) {
        e(NIL);
        str("\t");
        str(scall);
        str(": bad errno-value: ");
        put(errno);
    }
}

```

```

        str( " should have been: " );
        put(number);
        nlcr();
    }
}

/* check */

void put(nr)
int nr;
{
    switch (nr) {
        case 0:  str("unused");          break;
        case 1:  str("EPERM");            break;
        case 2:  str("ENOENT");           break;
        case 3:  str("ESRCH");            break;
        case 4:  str("EINTR");            break;
        case 5:  str("EIO");              break;
        case 6:  str("ENXIO");            break;
        case 7:  str("E2BIG");            break;
        case 8:  str("ENOEXEC");          break;
        case 9:  str("EBADF");            break;
        case 10: str("ECHILD");           break;
        case 11: str("EAGAIN");           break;
        case 12: str("ENOMEM");           break;
        case 13: str("EACCES");           break;
        case 14: str("EFAULT");           break;
        case 15: str("ENOTBLK");          break;
        case 16: str("EBUSY");            break;
        case 17: str("EEXIST");           break;
        case 18: str("EXDEV");            break;
        case 19: str("ENODEV");           break;
        case 20: str("ENOTDIR");          break;
        case 21: str("EISDIR");           break;
        case 22: str("EINVAL");          break;
        case 23: str("ENFILE");           break;
        case 24: str("EMFILE");           break;
        case 25: str("ENOTTY");           break;
        case 26: str("ETXTBSY");          break;
        case 27: str("EFBIG");            break;
        case 28: str("ENOSPC");           break;
        case 29: str("ESPIPE");           break;
        case 30: str("EROFS");            break;
        case 31: str("EMLINK");           break;
        case 32: str("EPIPE");            break;
        case 33: str("EDOM");             break;
        case 34: str("ERANGE");           break;
    }
}

/*****
*
*                               ALOT-functions
*
*****/

int open_alot()
{
    int i;

    for (i = 0; i < MAXOPEN; i++)
        if (open(file[i], R) == FAIL) break;
    if (i == 0) err(5, "open_alot", "at all");
    return(i);
}

/* open_alot */

int close_alot(number)
int number;
{
    int i, count = 0;

    if (number > MAXOPEN)
        err(5, "close_alot", "accept this argument");
    else
        for (i = FF; i < number + FF; i++)
            if (close(i) != OK) count++;
}

```

```

    return(number - count);          /* return number of closed files */
}                                     /* close_alot */

/*****
*
*                               CLEAN UP THE MESS
*
*****/

void clean_up_the_mess()
{
    int i;
    char dirname[6];

    /* First remove 'a lot' files */
    for (i = 0; i < MAXOPEN; i++) try_unlink(file[i]);

    /* Unlink the files in dir 'drwx' */
    if (chdir("drwx") != OK)
        err(5, CHDIR, "to 'drwx'");
    else {
        for (i = 0; i < 8; i++) try_unlink(fnames[i]);
        if (chdir("..") != OK) err(5, CHDIR, "to '..'");
    }

    /* Before unlinking files in some dirs, make them writable */
    chmod_8_dirs(RWX);

    /* Unlink files in other dirs */
    try_unlink("d-wx/rwx");
    try_unlink("dr-x/rwx");
    try_unlink("drw-/rwx");

    /* Unlink dirs */
    for (i = 0; i < 8; i++) {
        strcpy(dirname, "d");
        strcat(dirname, fnames[i]);

        /* 'dirname' contains the directoryname */
        rmdir(dirname);
    }

    /* FINISH */
}                                     /* clean_up_the_mess */

void chmod_8_dirs(sw)
int sw;                               /* if switch == 8, give all different
                                     * mode, else the same mode */
{
    int mode;
    int i;

    if (sw == 8)
        mode = 0;
    else
        mode = sw;

    for (i = 0; i < 8; i++) {
        chmod(dir[i], 040000 + mode * 0100);
        if (sw == 8) mode++;
    }
}

void quit()
{
    chdir("..");
    system("rm -rf DIR*");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {

```

```
    printf("%d errors\n", errct);  
    exit(1);  
}  
}
```

```

/* test 18 */

/* Comment on usage and program: ark!/mnt/rene/prac/os/unix/comment.changes */

/* "const.h", created by Rene Montsma and Menno Wilcke */

#include <sys/types.h>          /* needed in struct stat */
#include <sys/stat.h>           /* struct stat */
#include <sys/wait.h>
#include <errno.h>              /* the error-numbers */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <limits.h>

#define NOCRASH 1               /* test11(), 2nd pipe */
#define PDPNOHANG 1            /* test03(), write_standards() */
#define MAXERR 5

#define USER_ID 12
#define GROUP_ID 1
#define FF 3                    /* first free filedес. */
#define USER 1                  /* uid */
#define GROUP 0                 /* gid */

#define ARSIZE 256              /* array size */
#define PIPESIZE 3584           /* maxnumber of bytes to be written on pipe */
#define MAXOPEN (OPEN_MAX-3)   /* maximum number of extra open files */
#define MAXLINK 0177            /* maximum number of links per file */
#define MASK 0777              /* selects lower nine bits */
#define READ_EOF 0             /* returned by read-call at eof */

#define OK 0
#define FAIL -1

#define R 0                      /* read (open-call) */
#define W 1                      /* write (open-call) */
#define RW 2                     /* read & write (open-call) */

#define RWX 7                    /* read & write & execute (mode) */

#define NIL ""
#define UMASK "umask"
#define CREAT "creat"
#define WRITE "write"
#define READ "read"
#define OPEN "open"
#define CLOSE "close"
#define LSEEK "lseek"
#define ACCESS "access"
#define CHDIR "chdir"
#define CHMOD "chmod"
#define LINK "link"
#define UNLINK "unlink"
#define PIPE "pipe"
#define STAT "stat"
#define FSTAT "fstat"
#define DUP "dup"
#define UTIME "utime"

int errct;

char *file[];
char *fnames[];
char *dir[];

/* "decl.c", created by Rene Montsma and Menno Wilcke */

/* Used in open_alot, close_alot */
char *file[20] = {"f0", "f1", "f2", "f3", "f4", "f5", "f6",
                  "f7", "f8", "f9", "f10", "f11", "f12", "f13",
                  "f14", "f15", "f16", "f17", "f18", "f19"}, *fnames[8] = {"---", "--x", "-w-", "-wx",

```



```

"r--",
                                                                    "r-x", "rw-", "rwx"}, *
dir[8] = {"d---", "d--x", "d-w-", "d-wx", "dr--", "dr-x",
                                                                    "drw-", "drwx"};
/* Needed for easy creating and deleting of directories */
/* "test.c", created by Rene Montsma and Menno Wilcke */

_PROTOTYPE(int main, (void));
_PROTOTYPE(void test, (void));
_PROTOTYPE(void test01, (void));
_PROTOTYPE(void test02, (void));
_PROTOTYPE(void test03, (void));
_PROTOTYPE(void write_standards, (int filedес, char a []));
_PROTOTYPE(void test04, (void));
_PROTOTYPE(void read_standards, (int filedес, char a []));
_PROTOTYPE(void read_more, (int filedес, char a []));
_PROTOTYPE(void test05, (void));
_PROTOTYPE(void try_open, (char *fname, int mode, int test));
_PROTOTYPE(void test06, (void));
_PROTOTYPE(void test07, (void));
_PROTOTYPE(void access_standards, (void));
_PROTOTYPE(void try_access, (char *fname, int mode, int test));
_PROTOTYPE(void e, (char *string));
_PROTOTYPE(void nlcr, (void));
_PROTOTYPE(void str, (char *s));
_PROTOTYPE(void err, (int number, char *scall, char *name));
_PROTOTYPE(void make_and_fill_dirs, (void));
_PROTOTYPE(void put_file_in_dir, (char *dirname, int mode));
_PROTOTYPE(void init_array, (char *a));
_PROTOTYPE(void clear_array, (char *b));
_PROTOTYPE(int comp_array, (char *a, char *b, int range));
_PROTOTYPE(void try_close, (int filedес, char *name));
_PROTOTYPE(void try_unlink, (char *fname));
_PROTOTYPE(void Remove, (int fdes, char *fname));
_PROTOTYPE(int get_mode, (char *name));
_PROTOTYPE(void check, (char *scall, int number));
_PROTOTYPE(void put, (int nr));
_PROTOTYPE(int open_alot, (void));
_PROTOTYPE(int close_alot, (int number));
_PROTOTYPE(void clean_up_the_mess, (void));
_PROTOTYPE(void chmod_8_dirs, (int sw));
_PROTOTYPE(void quit, (void));

/*****
*
* TEST
*
*****/
int main()
{
    int n;

    if (geteuid() == 0 || getuid() == 0) {
        printf("Test 18 cannot run as root; test aborted\n");
        exit(1);
    }

    system("rm -rf DIR_18; mkdir DIR_18");
    chdir("DIR_18");

    if (fork()) {
        printf("Test 18 ");
        fflush(stdout);
        /* have to flush for child's benefit */

        wait(&n);
        clean_up_the_mess();
        quit();
    } else {
        test();
        exit(0);
    }

    return(0);
}

```

```

void test()
{
    umask(0);                /* not honest, but i always forget */

    test01();
    make_and_fill_dirs();
    test02();
    test03();
    test04();
    test05();
    test06();
    test07();
    umask(022);
}

/* test */

/* "t1.c" created by Rene Montsma and Menno Wilcke */

/*****
 *                               test UMASK                               *
 *****/
void test01()
{
    int oldvalue, newvalue, tempvalue;
    int nr;

    if ((oldvalue = umask(0777)) != 0) err(0, UMASK, NIL);

    /* Special test: only the lower 9 bits (protection bits) may part- *
     * icipate. ~0777 means: 111 000 000 000. Giving this to umask must *
     * not change any value.                                           */

    if ((newvalue = umask(~0777)) != 0777) err(1, UMASK, "illegal");
    if (oldvalue == newvalue) err(11, UMASK, "not change mask");

    if ((tempvalue = umask(0)) != 0) err(2, UMASK, "values");

    /* Now test all possible modes of umask on a file */
    for (newvalue = MASK; newvalue >= 0; newvalue -= 0111) {
        tempvalue = umask(newvalue);
        if (tempvalue != oldvalue) {
            err(1, UMASK, "illegal");
            break;                /* no use trying more */
        } else if ((nr = creat("file01", 0777)) < 0)
            err(5, CREAT, "'file01'");
        else {
            try_close(nr, "file01");
            if (get_mode("file01") != (MASK & ~newvalue))
                err(7, UMASK, "mode computed");
            try_unlink("file01");
        }
        oldvalue = newvalue;
    }

    /* The loop has terminated with umask(0) */
    if ((tempvalue = umask(0)) != 0)
        err(7, UMASK, "umask may influence rest of tests!");
}

/* test01 */

/*****
 *                               test CREAT                               *
 *****/
void test02()
{
    int n, n1, mode;
    char a[ARSIZE], b[ARSIZE];
    struct stat stbfl;

    mode = 0;
    /* Create twenty files, check filedes */
    for (n = 0; n < MAXOPEN; n++) {
        if (creat(file[n], mode) != FF + n)
            err(13, CREAT, file[n]);
        else {
            if (get_mode(file[n]) != mode)

```

```

        err(7, CREAT, "mode set while creating many files");

        /* Change mode of file to standard mode, we want to *
         * use a lot (20) of files to be opened later, see *
         * open_alot(), close_alot(). */
        if (chmod(file[n], 0700) != OK) err(5, CHMOD, file[n]);
    }
    mode = (mode + 0100) % 01000;
}

/* Already twenty files opened; opening another has to fail */
if (creat("file02", 0777) != FAIL)
    err(9, CREAT, "created");
else
    check(CREAT, EMFILE);

/* Close all files: seems blunt, but it isn't because we've *
 * checked all fd's already */
if ((n = close_alot(MAXOPEN)) < MAXOPEN) err(5, CLOSE, "MAXOPEN files");

/* Creat 1 file twice; check */
if ((n = creat("file02", 0777)) < 0)
    err(5, CREAT, "'file02'");
else {
    init_array(a);
    if (write(n, a, ARSIZE) != ARSIZE) err(1, WRITE, "bad");

    if ((n1 = creat("file02", 0755)) < 0) /* recreate 'file02' */
        err(5, CREAT, "'file02' (2nd time)");
    else {
        /* Fd should be at the top after recreation */
        if (lseek(n1, 0L, SEEK_END) != 0)
            err(11, CREAT, "not truncate file by recreation");
        else {
            /* Try to write on recreated file */
            clear_array(b);

            if (lseek(n1, 0L, SEEK_SET) != 0)
                err(5, LSEEK, "to top of 2nd fd 'file02'");
            if (write(n1, a, ARSIZE) != ARSIZE)
                err(1, WRITE, "(2) bad");

            /* In order to read we've to close and open again */
            try_close(n1, "'file02' (2nd creation)");
            if ((n1 = open("file02", RW)) < 0)
                err(5, OPEN, "'file02' (2nd recreation)");

            /* Continue */
            if (lseek(n1, 0L, SEEK_SET) != 0)
                err(5, LSEEK, "to top 'file02' (2nd fd) (2)");
            if (read(n1, b, ARSIZE) != ARSIZE)
                err(1, READ, "wrong");

            if (comp_array(a, b, ARSIZE) != OK) err(11, CREAT,
                "not really truncate file by recreation");
        }
        if (get_mode("file02") != 0777)
            err(11, CREAT, "not maintain mode by recreation");
        try_close(n1, "recreated 'file02'");
    }
    Remove(n, "file02");
}

/* Give 'creat' wrong input: dir not searchable */
if (creat("drw-/file02", 0777) != FAIL)
    err(4, CREAT, "'drw-'");
else
    check(CREAT, EACCES);

/* Dir not writable */
if (creat("dr-x/file02", 0777) != FAIL)
    err(12, CREAT, "'dr-x/file02'");

```

```

else
    check(CREAT, EACCES);

/* File not writable */
if (creat("drwx/r-x", 0777) != FAIL)
    err(11, CREAT, "recreate non-writable file");
else
    check(CREAT, EACCES);

/* Try to creat a dir */
if ((n = creat("dir", 040777)) != FAIL) {
    if (fstat(n, &stbfl) != OK)
        err(5, FSTAT, "'dir'");
    else if (stbfl.st_mode != (mode_t) 0100777)
        /* Cast because mode is negative :-(.
         * HACK DEBUG FIXME: this appears to duplicate
         * code in test17.c.
         */
        err(11, CREAT, "'creat' a new directory");
    Remove(n, "dir");
}

/* We don't consider it to be a bug when creat * does not accept
 * tricky modes */

/* File is an existing dir */
if (creat("drwx", 0777) != FAIL)
    err(11, CREAT, "create an existing dir!");
else
    check(CREAT, EISDIR);
} /* test02 */

/*****
 *                               test WRITE
 *****/
void test03()
{
    int n, nl;
    int fd[2];
    char a[ARSIZE];

    init_array(a);

    /* Test write after a CREAT */
    if ((n = creat("file03", 0700)) != FF) /* 'file03' only open file */
        err(13, CREAT, "'file03'");
    else {
        write_standards(n, a); /* test simple writes, wrong input too */
        try_close(n, "'file03'");
    }

    /* Test write after an OPEN */
    if ((n = open("file03", W)) < 0)
        err(5, OPEN, "'file03'");
    else
        write_standards(n, a); /* test simple writes, wrong input too */

    /* Test write after a DUP */
    if ((nl = dup(n)) < 0)
        err(5, DUP, "'file03'");
    else {
        write_standards(nl, a);
        try_close(nl, "duplicated fd 'file03'");
    }

    /* Remove testfile */
    Remove(n, "file03");

    /* Test write after a PIPE */
    if (pipe(fd) < 0)
        err(5, PIPE, NIL);
    else {
        write_standards(fd[1], a);
        try_close(fd[0], "'fd[0]'");
    }
}

```

```

    try_close(fd[1], "'fd[1]");
}

/* Last test: does write check protections ? */
if ((n = open("drwx/r--", R)) < 0)
    err(5, OPEN, "'drwx/r--");
else {
    if (write(n, a, ARSIZE) != FAIL)
        err(11, WRITE, "write on non-writ. file");
    else
        check(WRITE, EBADF);
    try_close(n, "'drwx/r--");
}
}
/* test03 */

void write_standards(filedes, a)
int filedes;
char a[];
{
    /* Write must return written account of numbers */
    if (write(filedes, a, ARSIZE) != ARSIZE) err(1, WRITE, "bad");

    /* Try giving 'write' wrong input */
    /* Wrong filedes */
    if (write(-1, a, ARSIZE) != FAIL)
        err(2, WRITE, "filedes");
    else
        check(WRITE, EBADF);

    /* Wrong length (illegal) */
#ifdef PDPNOHANG
    if (write(filedes, a, -ARSIZE) != FAIL)
        err(2, WRITE, "length");
    else
        check(WRITE, EINVAL); /* EFAULT on vu45 */
#endif
}
/* write_standards */

/* "t2.c", created by Rene Montsma and Menno Wilcke */

/*****
 *                               test READ                               *
 *****/
void test04()
{
    int n, n1, fd[2];
    char a[ARSIZE];

    /* Test read after creat */
    if ((n = creat("file04", 0700)) != FF) /* no other open files may be
                                           * left */
        err(13, CREAT, "'file04'");
    else {
        /* Closing and opening needed before writing */
        try_close(n, "'file04'");
        if ((n = open("file04", RW)) < 0) err(5, OPEN, "'file04'");

        init_array(a);

        if (write(n, a, ARSIZE) != ARSIZE)
            err(1, WRITE, "bad");
        else {
            if (lseek(n, 0L, SEEK_SET) != 0) err(5, LSEEK, "'file04'");
            read_standards(n, a);
            read_more(n, a);
        }
        try_close(n, "'file04'");
    }

    /* Test read after OPEN */
    if ((n = open("file04", R)) < 0)
        err(5, OPEN, "'file04'");
    else {

```

```

        read_standards(n, a);
        read_more(n, a);
        try_close(n, "file04");
    }

    /* Test read after DUP */
    if ((n = open("file04", R)) < 0) err(5, OPEN, "file04");
    if ((n1 = dup(n)) < 0)
        err(5, DUP, "file04");
    else {
        read_standards(n1, a);
        read_more(n1, a);
        try_close(n1, "duplicated fd 'file04'");
    }

    /* Remove testfile */
    Remove(n, "file04");

    /* Test read after pipe */
    if (pipe(fd) < 0)
        err(5, PIPE, NIL);
    else {
        if (write(fd[1], a, ARSIZE) != ARSIZE) {
            err(5, WRITE, "fd[1]");
            try_close(fd[1], "fd[1]");
        } else {
            try_close(fd[1], "fd[1]");
            read_standards(fd[0], a);
        }
        try_close(fd[0], "fd[0]");
    }

    /* Last test: try to read a read-protected file */
    if ((n = open("drwx/-wx", W)) < 0)
        err(5, OPEN, "drwx/-wx");
    else {
        if (read(n, a, ARSIZE) != FAIL)
            err(11, READ, "read a non-read. file");
        else
            check(READ, EBADF);
        try_close(n, "drwx/-wx");
    }
} /* test04 */

void read_standards(filedes, a)
int filedes;
char a[];
{
    char b[ARSIZE];

    clear_array(b);
    if (read(filedes, b, ARSIZE) != ARSIZE)
        err(1, READ, "bad");
    else if (comp_array(a, b, ARSIZE) != OK)
        err(7, "read/write", "values");
    else if (read(filedes, b, ARSIZE) != READ_EOF)
        err(11, READ, "read beyond endoffile");

    /* Try giving read wrong input: wrong filedes */
    if (read(FAIL, b, ARSIZE) != FAIL)
        err(2, READ, "filedes");
    else
        check(READ, EBADF);

    /* Wrong length */
    if (read(filedes, b, -ARSIZE) != FAIL)
        err(2, READ, "length");
    else
        check(READ, EINVAL);
} /* read_standards */

void read_more(filedes, a)
int filedes;
char a[];

```

```

/* Separated from read_standards() because the PIPE test * would fail.
*/
{
    int i;
    char b[ARSIZE];

    if (lseek(filedes, (long) (ARSIZE / 2), SEEK_SET) != ARSIZE / 2)
        err(5, LSEEK, "to location ARSIZE/2");

    clear_array(b);
    if (read(filedes, b, ARSIZE) != ARSIZE / 2) err(1, READ, "bad");

    for (i = 0; i < ARSIZE / 2; i++)
        if (b[i] != a[(ARSIZE / 2) + i])
            err(7, READ, "from location ARSIZE/2");
}

/*****
*                               test OPEN/CLOSE                               *
*****/
void test05()
{
    int n, nl, mode, fd[2];
    char b[ARSIZE];

    /* Test open after CREAT */
    if ((n = creat("file05", 0700)) != FF) /* no other open files left */
        err(13, CREAT, "'file05'");
    else {
        if ((nl = open("file05", RW)) != FF + 1)
            err(13, OPEN, "'file05' after creation");
        try_close(nl, "'file05' (open after creation)");

        try_close(n, "'file05'");
        if ((n = open("file05", R)) != FF)
            err(13, OPEN, "after closing");
        else
            try_close(n, "'file05' (open after closing)");

        /* Remove testfile */
        try_unlink("file05");
    }

    /* Test all possible modes, try_open not only opens file (sometimes) *
     * but closes files too (when opened) */
    if ((n = creat("file05", 0700)) < 0) /* no other files left */
        err(5, CREAT, "'file05' (2nd time)");
    else {
        try_close(n, "file05");
        for (mode = 0; mode <= 0700; mode += 0100) {
            if (chmod("file05", mode) != OK) err(5, CHMOD, "'file05'");

            if (mode <= 0100) {
                try_open("file05", R, FAIL);
                try_open("file05", W, FAIL);
                try_open("file05", RW, FAIL);
            } else if (mode >= 0200 && mode <= 0300) {
                try_open("file05", R, FAIL);
                try_open("file05", W, FF);
                try_open("file05", RW, FAIL);
            } else if (mode >= 0400 && mode <= 0500) {
                try_open("file05", R, FF);
                try_open("file05", W, FAIL);
                try_open("file05", RW, FAIL);
            } else {
                try_open("file05", R, FF);
                try_open("file05", W, FF);
                try_open("file05", RW, FF);
            }
        }
    }

    /* Test opening existing file */
    if ((n = open("drwx/rwx", R)) < 0)

```

```

    err(13, OPEN, "existing file");
else {
    /* test close after DUP */
    if ((n1 = dup(n)) < 0)
        err(13, DUP, "'drwx/rwx'");
    else {
        try_close(n1, "duplicated fd 'drwx/rwx'");

        if (read(n1, b, ARSIZE) != FAIL)
            err(11, READ, "on closed dupped fd 'drwx/rwx'");
        else
            check(READ, EBADF);

        if (read(n, b, ARSIZE) == FAIL) /* should read an eof */
            err(13, READ, "on fd 'drwx/rwx'");
    }
    try_close(n, "'drwx/rwx'");
}

/* Test close after PIPE */
if (pipe(fd) < 0)
    err(13, PIPE, NIL);
else {
    try_close(fd[1], "duplicated fd 'fd[1]'");

    /* fd[1] really should be closed now; check */
    clear_array(b);
    if (read(fd[0], b, ARSIZE) != READ_EOF)
        err(11, READ, "read on empty pipe (and fd[1] was closed)");
    try_close(fd[0], "duplicated fd 'fd[0]'");
}

/* Try to open a non-existing file */
if (open("non-file", R) != FAIL)
    err(11, OPEN, "open non-executable file");
else
    check(OPEN, ENOENT);

/* Dir does not exist */
if (open("dzzz/file05", R) != FAIL)
    err(11, OPEN, "open in an non-searchable dir");
else
    check(OPEN, ENOENT);

/* Dir is not searchable */
if (n = open("drw-rwx", R) != FAIL)
    err(11, OPEN, "open in an non-searchabledir");
else
    check(OPEN, EACCES);

/* Unlink testfile */
try_unlink("file05");

/* File is not readable */
if (open("drwx/-wx", R) != FAIL)
    err(11, OPEN, "open unreadable file for reading");
else
    check(OPEN, EACCES);

/* File is not writable */
if (open("drwx/r-x", W) != FAIL)
    err(11, OPEN, "open unwritable file for writing");
else
    check(OPEN, EACCES);

/* Try opening more than MAXOPEN ('extra' (19-8-85)) files */
if ((n = open_alot()) != MAXOPEN)
    err(13, OPEN, "MAXOPEN files");
else
    /* Maximum # of files opened now, another open should fail
     * because * all filedescriptors have already been used.
     */
    if (open("drwx/rwx", RW) != FAIL)
        err(9, OPEN, "open");
    else
        check(OPEN, EMFILE);

```



```

if (close_alot(n) != n) err(5, CLOSE, "all opened files");

/* Can close make mistakes ? */
if (close(-1) != FAIL)
    err(2, CLOSE, "filedes");
else
    check(CLOSE, EBADF);
}
/* test05 */

void try_open(fname, mode, test)
int mode, test;
char *fname;
{
    int n;

    if ((n = open(fname, mode)) != test)
        err(11, OPEN, "break through filepermission with an incorrect mode");
    if (n != FAIL) try_close(n, fname); /* cleanup */
}
/* try_open */

/*****
*
* test LSEEK
*
*****/
void test06()
{
    char a[ARSIZE], b[ARSIZE];
    int fd;

    if ((fd = open("drwx/rwx", RW)) != FF) /* there should be no */
        err(13, OPEN, "'drwx/rwx'"); /* other open files */
    else {
        init_array(a);
        if (write(fd, a, 10) != 10)
            err(1, WRITE, "bad");
        else {
            /* Lseek back to begin file */
            if (lseek(fd, 0L, SEEK_SET) != 0)
                err(5, LSEEK, "to begin file");
            else if (read(fd, b, 10) != 10)
                err(1, READ, "bad");
            else if (comp_array(a, b, 10) != OK)
                err(7, LSEEK, "values r/w after lseek to begin");
            /* Lseek to endoffile */
            if (lseek(fd, 0L, SEEK_END) != 10)
                err(5, LSEEK, "to end of file");
            else if (read(fd, b, 1) != READ_EOF)
                err(7, LSEEK, "read at end of file");
            /* Lseek beyond file */
            if (lseek(fd, 10L, SEEK_CUR) != 20)
                err(5, LSEEK, "beyond end of file");
            else if (write(fd, a, 10) != 10)
                err(1, WRITE, "bad");
            else {
                /* Lseek to begin second write */
                if (lseek(fd, 20L, SEEK_SET) != 20)
                    err(5, LSEEK, "'drwx/rwx'");
                if (read(fd, b, 10) != 10)
                    err(1, READ, "bad");
                else if (comp_array(a, b, 10) != OK)
                    err(7, LSEEK,
                        "values read after lseek MAXOPEN");
            }
        }

        /* Lseek to position before begin of file */
        if (lseek(fd, -1L, 0) != FAIL)
            err(11, LSEEK, "lseek before beginning of file");

        try_close(fd, "'drwx/rwx'");
    }

    /* Lseek on invalid filedescriptor */
    if (lseek(-1, 0L, SEEK_SET) != FAIL)
        err(2, LSEEK, "filedes");
}

```

```

    else
        check(LSEEK, EBADF);
}

/* "t3.c", created by Rene Montsma and Menno Wilcke */

/*****
 *
 *                      test ACCESS
 *****/
void test07()
{
    /* Check with proper parameters */
    if (access("drwx/rwx", RWX) != OK) err(5, ACCESS, "accessible file");

    if (access("../drwx/../rwx", 0) != OK)
        err(5, ACCESS, "'../(etc)/drwx//rwx'");

    /* Check 8 files with 8 different modes on 8 accesses */
    if (chdir("drwx") != OK) err(5, CHDIR, "'drwx'");

    access_standards();

    if (chdir("..") != OK) err(5, CHDIR, "'..'");

    /* Check several wrong combinations */
    /* File does not exist */
    if (access("non-file", 0) != FAIL)
        err(11, ACCESS, "access non-existing file");
    else
        check(ACCESS, ENOENT);

    /* Non-searchable dir */
    if (access("drw-rwx", 0) != FAIL)
        err(4, ACCESS, "'drw-'");
    else
        check(ACCESS, EACCES);

    /* Searchable dir, but wrong file-mode */
    if (access("drwx/---x", RWX) != FAIL)
        err(11, ACCESS, "a non accessible file");
    else
        check(ACCESS, EACCES);
}

/* test07 */

void access_standards()
{
    int i, mode = 0;

    for (i = 0; i < 8; i++)
        if (i == 0)
            try_access(fnames[mode], i, OK);
        else
            try_access(fnames[mode], i, FAIL);
    mode++;

    for (i = 0; i < 8; i++)
        if (i < 2)
            try_access(fnames[mode], i, OK);
        else
            try_access(fnames[mode], i, FAIL);
    mode++;

    for (i = 0; i < 8; i++)
        if (i == 0 || i == 2)
            try_access(fnames[mode], i, OK);
        else
            try_access(fnames[mode], i, FAIL);
    mode++;

    for (i = 0; i < 8; i++)
        if (i < 4)
            try_access(fnames[mode], i, OK);

```

```

        else
            try_access(fnames[mode], i, FAIL);
mode++;

for (i = 0; i < 8; i++)
    if (i == 0 || i == 4)
        try_access(fnames[mode], i, OK);
    else
        try_access(fnames[mode], i, FAIL);
mode++;

for (i = 0; i < 8; i++)
    if (i == 0 || i == 1 || i == 4 || i == 5)
        try_access(fnames[mode], i, OK);
    else
        try_access(fnames[mode], i, FAIL);
mode++;

for (i = 0; i < 8; i++)
    if (i % 2 == 0)
        try_access(fnames[mode], i, OK);
    else
        try_access(fnames[mode], i, FAIL);
mode++;

for (i = 0; i < 8; i++) try_access(fnames[mode], i, OK);
}
/* access_standards */

void try_access(fname, mode, test)
int mode, test;
char *fname;
{
    if (access(fname, mode) != test)
        err(100, ACCESS, "incorrect access on a file (try_access)");
}
/* try_access */

/* "support.c", created by Rene Montsma and Menno Wilcke */

/* Err, make_and_fill_dirs, init_array, clear_array, comp_array,
   try_close, try_unlink, Remove, get_mode, check, open_alot,
   close_alot, clean_up_the_mess.
*/

/*****
 *
 * EXTENDED FIONS
 *
 *****/
/* First extended functions (i.e. not oldfashioned monixcalls.
   e(), nlcr(), octal.*/

void e(string)
char *string;
{
    printf("Test program error: %s\n", string);
    errct++;
}

void nlcr()
{
    printf("\n");
}

void str(s)
char *s;
{
    printf(s);
}

/*****
 *
 * ERR(or) messages
 *
 *****/
void err(number, scall, name)
/* Give nice error messages */

```

```
char *scall, *name;
int number;

{
    errct++;
    if (errct > MAXERR) {
        printf( "Too many errors; test aborted\n" );
        chdir( ".." );
        system( "rm -rf DIR*" );
        quit();
    }
    e( "" );
    str( "\t" );
    switch (number) {
        case 0:
            str( scall );
            str( ": illegal initial value." );
            break;
        case 1:
            str( scall );
            str( ": " );
            str( name );
            str( " value returned." );
            break;
        case 2:
            str( scall );
            str( ": accepting illegal " );
            str( name );
            str( "." );
            break;
        case 3:
            str( scall );
            str( ": accepting non-existing file." );
            break;
        case 4:
            str( scall );
            str( ": could search non-searchable dir ( " );
            str( name );
            str( ")." );
            break;
        case 5:
            str( scall );
            str( ": cannot " );
            str( scall );
            str( " " );
            str( name );
            str( "." );
            break;
        case 7:
            str( scall );
            str( ": incorrect " );
            str( name );
            str( "." );
            break;
        case 8:
            str( scall );
            str( ": wrong values." );
            break;
        case 9:
            str( scall );
            str( ": accepting too many " );
            str( name );
            str( " files." );
            break;
        case 10:
            str( scall );
            str( ": even a superuser can't do anything!" );
            break;
        case 11:
            str( scall );
            str( ": could " );
            str( name );
            str( "." );
    }
}
```

```

        break;
    case 12:
        str(scall);
        str(": could write in non-writable dir (");
        str(name);
        str(").");
        break;
    case 13:
        str(scall);
        str(": wrong filedes returned (");
        str(name);
        str(").");
        break;
    case 100:
        str(scall);           /* very common */
        str(": ");
        str(name);
        str(".");
        break;
    default:
        str("error number does not exist!\n");
}
nlcr();
}                               /* err */

/*****
*
*                               MAKE_AND_FILL_DIRS
*
*****/

void make_and_fill_dirs()
/* Create 8 dir.'s: "d---", "d--x", "d-w-", "d-wx", "dr--", "dr-x",
 * "drw-", "drwx".                               * Then create 8 files
 * in "drwx", and some needed files in other dirs. */
{
    int mode, i;

    for (i = 0; i < 8; i++) {
        mkdir(dir[i], 0700);
        chown(dir[i], USER_ID, GROUP_ID);
    }
    setuid(USER_ID);
    setgid(GROUP_ID);

    for (mode = 0; mode < 8; mode++) put_file_in_dir("drwx", mode);

    put_file_in_dir("d-wx", RWX);
    put_file_in_dir("dr-x", RWX);
    put_file_in_dir("drw-", RWX);

    chmod_8_dirs(8);           /* 8 means; 8 different modes */
}                               /* make_and_fill_dirs */

void put_file_in_dir(dirname, mode)
char *dirname;
int mode;
/* Fill directory 'dirname' with file with mode 'mode'. */
{
    int nr;

    if (chdir(dirname) != OK)
        err(5, CHDIR, "to dirname (put_f_in_dir)");
    else {
        /* Create the file */
        if ((nr = creat(fnames[mode], mode * 0100)) < 0)
            err(13, CREAT, fnames[mode]);
        else
            try_close(nr, fnames[mode]);

        if (chdir("..") != OK)
            err(5, CHDIR, "to previous dir (put_f_in_dir)");
    }
}                               /* put_file_in_dir */

```

```
/******  
*  
*                               MISCELLANEOUS  
*  
*(all about arrays, 'try_close', 'try_unlink', 'Remove', 'get_mode')  
*  
*****/  
  
void init_array(a)  
char *a;  
{  
    int i;  
  
    i = 0;  
    while (i++ < ARSIZE) *a++ = 'a' + (i % 26);  
}  
/* init_array */  
  
void clear_array(b)  
char *b;  
{  
    int i;  
  
    i = 0;  
    while (i++ < ARSIZE) *b++ = '0';  
}  
/* clear_array */  
  
int comp_array(a, b, range)  
char *a, *b;  
int range;  
{  
    if ((range < 0) || (range > ARSIZE)) {  
        err(100, "comp_array", "illegal range");  
        return(FAIL);  
    } else {  
        while (range-- && (*a++ == *b++));  
        if (*--a == *--b)  
            return(OK);  
        else  
            return(FAIL);  
    }  
}  
/* comp_array */  
  
void try_close(filedes, name)  
int filedes;  
char *name;  
{  
    if (close(filedes) != OK) err(5, CLOSE, name);  
}  
/* try_close */  
  
void try_unlink(fname)  
char *fname;  
{  
    if (unlink(fname) != 0) err(5, UNLINK, fname);  
}  
/* try_unlink */  
  
void Remove(fdes, fname)  
int fdes;  
char *fname;  
{  
    try_close(fdes, fname);  
    try_unlink(fname);  
}  
/* Remove */  
  
int get_mode(name)  
char *name;  
{  
    struct stat stbfl;  
  
    if (stat(name, &stbfl) != OK) {  
        err(5, STAT, name);  
        return(stbfl.st_mode); /* return a mode which will cause *  
                                * error in the calling function *  
    }  
}
```

```

        * (file/dir bit)                                */
    } else
        return(stbfl.st_mode & 07777); /* take last 4 bits */
    }
    /* get_mode */

/*****
*
*                                CHECK
*
*****/

void check(scall, number)
int number;
char *scall;
{
    if (errno != number) {
        e(NIL);
        str("\t");
        str(scall);
        str(": bad errno-value: ");
        put(errno);
        str(" should have been: ");
        put(number);
        nlcr();
    }
}

/* check */

void put(nr)
int nr;
{
    switch (nr) {
        case 0:  str("unused");          break;
        case 1:  str("EPERM");            break;
        case 2:  str("ENOENT");           break;
        case 3:  str("ESRCH");            break;
        case 4:  str("EINTR");            break;
        case 5:  str("EIO");              break;
        case 6:  str("ENXIO");            break;
        case 7:  str("E2BIG");            break;
        case 8:  str("ENOEXEC");          break;
        case 9:  str("EBADF");            break;
        case 10: str("ECHILD");           break;
        case 11: str("EAGAIN");           break;
        case 12: str("ENOMEM");           break;
        case 13: str("EACCES");           break;
        case 14: str("EFAULT");           break;
        case 15: str("ENOTBLK");          break;
        case 16: str("EBUSY");            break;
        case 17: str("EEXIST");           break;
        case 18: str("EXDEV");            break;
        case 19: str("ENODEV");           break;
        case 20: str("ENOTDIR");          break;
        case 21: str("EISDIR");           break;
        case 22: str("EINVAL");           break;
        case 23: str("ENFILE");           break;
        case 24: str("EMFILE");           break;
        case 25: str("ENOTTY");           break;
        case 26: str("ETXTBSY");          break;
        case 27: str("EFBIG");            break;
        case 28: str("ENOSPC");           break;
        case 29: str("ESPIPE");           break;
        case 30: str("EROFS");            break;
        case 31: str("EMLINK");           break;
        case 32: str("EPIPE");            break;
        case 33: str("EDOM");             break;
        case 34: str("ERANGE");           break;
    }
}

/*****
*
*                                ALLOT-functions
*
*****/

```

```

int open_alot()
{
    int i;

    for (i = 0; i < MAXOPEN; i++)
        if (open(file[i], R) == FAIL) break;
    if (i == 0) err(5, "open_alot", "at all");
    return(i);
}
/* open_alot */

int close_alot(number)
int number;
{
    int i, count = 0;

    if (number > MAXOPEN)
        err(5, "close_alot", "accept this argument");
    else
        for (i = FF; i < number + FF; i++)
            if (close(i) != OK) count++;

    return(number - count);
}
/* return number of closed files */
/* close_alot */

/*****
*
*                               *
*                               *
*                               *
*                               *
*****/
void clean_up_the_mess()
{
    int i;
    char dirname[6];

    /* First remove 'alot' files */
    for (i = 0; i < MAXOPEN; i++) try_unlink(file[i]);

    /* Unlink the files in dir 'drwx' */
    if (chdir("drwx") != OK)
        err(5, CHDIR, "to 'drwx'");
    else {
        for (i = 0; i < 8; i++) try_unlink(fnames[i]);
        if (chdir("..") != OK) err(5, CHDIR, "to '..'");
    }

    /* Before unlinking files in some dirs, make them writable */
    chmod_8_dirs(RWX);

    /* Unlink files in other dirs */
    try_unlink("d-wx/rwx");
    try_unlink("dr-x/rwx");
    try_unlink("drw-rwx");

    /* Unlink dirs */
    for (i = 0; i < 8; i++) {
        strcpy(dirname, "d");
        strcat(dirname, fnames[i]);
        /* 'dirname' contains the directoryname */
        rmdir(dirname);
    }

    /* FINISH */
}
/* clean_up_the_mess */

void chmod_8_dirs(sw)
int sw;
/* if switch == 8, give all different
 * mode, else the same mode */
{
    int mode;
    int i;

    if (sw == 8)

```



```
        mode = 0;
    else
        mode = sw;

    for (i = 0; i < 8; i++) {
        chmod(dir[i], 040000 + mode * 0100);
        if (sw == 8) mode++;
    }
}

void quit()
{
    chdir("..");
    system("rm -rf DIR*");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
        printf("%d errors\n", errct);
        exit(1);
    }
}
```

```
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

#define MAX_ERROR 4
#define NB 30L
#define NBOUNDS 6

int errct, subtest, passes, pipesigs;
long tl;

char aa[100];
char b[4] = {0, 1, 2, 3}, c[4] = {10, 20, 30, 40}, d[4] = {6, 7, 8, 9};
long bounds[NBOUNDS] = {7, 9, 50, 519, 520, 40000L};
char buff[30000];

_PROTOTYPE(int main, (int argc, char *argv[]));
_PROTOTYPE(void test19a, (void));
_PROTOTYPE(void test19b, (void));
_PROTOTYPE(void test19c, (void));
_PROTOTYPE(void test19d, (void));
_PROTOTYPE(void test19e, (void));
_PROTOTYPE(void test19f, (void));
_PROTOTYPE(void test19g, (void));
_PROTOTYPE(void clraa, (void));
_PROTOTYPE(void pipecatcher, (int s));
_PROTOTYPE(void e, (int n));
_PROTOTYPE(void quit, (void));

int main(argc, argv)
int argc;
char *argv[];
{
    int i, m;

    m = (argc == 2 ? atoi(argv[1]) : 0xFFFF);

    if (geteuid() == 0 || getuid() == 0) {
        printf("Test 19 cannot run as root; test aborted\n");
        exit(1);
    }

    system("rm -rf DIR_19; mkdir DIR_19");
    chdir("DIR_19");

    printf("Test 19 ");
    fflush(stdout);
    for (i = 0; i < 4; i++) {
        if (m & 0001) test19a();
        if (m & 0002) test19b();
        if (m & 0004) test19c();
        if (m & 0010) test19d();
        if (m & 0020) test19e();
        if (m & 0040) test19f();
        if (m & 0100) test19g();
        passes++;
    }
    quit();
    return(-1); /* impossible */
}

void test19a()
{
    /* Test open with O_CREAT and O_EXCL. */

    int fd;

    subtest = 1;
```

```
if ( (fd = creat("T19.a1", 0777)) != 3) e(1); /* create test file */
if (close(fd) != 0) e(2);
if ( (fd = open("T19.a1", O_RDONLY)) != 3) e(3);
if (close(fd) != 0) e(4);
if ( (fd = open("T19.a1", O_WRONLY)) != 3) e(5);
if (close(fd) != 0) e(6);
if ( (fd = open("T19.a1", O_RDWR)) != 3) e(7);
if (close(fd) != 0) e(8);

/* See if O_CREAT actually creates a file. */
if ( (fd = open("T19.a2", O_RDONLY)) != -1) e(9); /* must fail */
if ( (fd = open("T19.a2", O_RDONLY | O_CREAT, 0444)) != 3) e(10);
if (close(fd) != 0) e(11);
if ( (fd = open("T19.a2", O_RDONLY)) != 3) e(12);
if (close(fd) != 0) e(13);
if ( (fd = open("T19.a2", O_WRONLY)) != -1) e(14);
if ( (fd = open("T19.a2", O_RDWR)) != -1) e(15);

/* See what O_CREAT does on an existing file. */
if ( (fd = open("T19.a2", O_RDONLY | O_CREAT, 0777)) != 3) e(16);
if (close(fd) != 0) e(17);
if ( (fd = open("T19.a2", O_RDONLY)) != 3) e(18);
if (close(fd) != 0) e(19);
if ( (fd = open("T19.a2", O_WRONLY)) != -1) e(20);
if ( (fd = open("T19.a2", O_RDWR)) != -1) e(21);

/* See if O_EXCL works. */
if ( (fd = open("T19.a2", O_RDONLY | O_EXCL)) != 3) e(22);
if (close(fd) != 0) e(23);
if ( (fd = open("T19.a2", O_WRONLY | O_EXCL)) != -1) e(24);
if ( (fd = open("T19.a3", O_RDONLY | O_EXCL)) != -1) e(25);
if ( (fd = open("T19.a3", O_RDONLY | O_CREAT | O_EXCL, 0444)) != 3) e(26);
if (close(fd) != 0) e(27);
errno = 0;
if ( (fd = open("T19.a3", O_RDONLY | O_CREAT | O_EXCL, 0444)) != -1) e(28);
if (errno != EEXIST) e(29);

if (unlink("T19.a1") != 0) e(30);
if (unlink("T19.a2") != 0) e(31);
if (unlink("T19.a3") != 0) e(32);
}

void test19b()
{
/* Test open with O_APPEND and O_TRUNC. */

int fd;

subtest = 2;

if ( (fd = creat("T19.b1", 0777)) != 3) e(1); /* create test file */
if (write(fd, b, 4) != 4) e(2);
if (close(fd) != 0) e(3);
clr_aa();
if ( (fd = open("T19.b1", O_RDWR | O_APPEND)) != 3) e(4);
if (read(fd, aa, 100) != 4) e(5);
if (aa[0] != 0 || aa[1] != 1 || aa[2] != 2 || aa[3] != 3) e(6);
if (close(fd) != 0) e(7);
if ( (fd = open("T19.b1", O_RDWR | O_APPEND)) != 3) e(8);
if (write(fd, b, 4) != 4) e(9);
if (lseek(fd, 0L, SEEK_SET) != 0L) e(10);
clr_aa();
if (read(fd, aa, 100) != 8) e(11);
if (aa[4] != 0 || aa[5] != 1 || aa[6] != 2 || aa[7] != 3) e(12);
if (close(fd) != 0) e(13);

if ( (fd = open("T19.b1", O_RDWR | O_TRUNC)) != 3) e(14);
if (read(fd, aa, 100) != 0) e(15);
if (close(fd) != 0) e(16);

unlink("T19.b1");
}

void test19c()
```

```
{
/* Test program for open(), close(), creat(), read(), write(), lseek(). */

int i, n, n1, n2;

subtest = 3;
if ((n = creat("foop", 0777)) != 3) e(1);
if ((n1 = creat("foop", 0777)) != 4) e(2);
if ((n2 = creat("/", 0777)) != -1) e(3);
if (close(n) != 0) e(4);
if ((n = open("foop", O_RDONLY)) != 3) e(5);
if ((n2 = open("nofile", O_RDONLY)) != -1) e(6);
if (close(n1) != 0) e(7);

/* N is the only one open now. */
for (i = 0; i < 2; i++) {
    n1 = creat("File2", 0777);
    if (n1 != 4) {
        printf("creat yielded fd=%d, expected 4\n", n1);
        e(8);
    }
    if ((n2 = open("File2", O_RDONLY)) != 5) e(9);
    if (close(n1) != 0) e(10);
    if (close(n2) != 0) e(11);
}
unlink("File2");
if (close(n) != 0) e(12);

/* All files closed now. */
for (i = 0; i < 2; i++) {
    if ((n = creat("foop", 0777)) != 3) e(13);
    if (close(n) != 0) e(14);
    if ((n = open("foop", O_RDWR)) != 3) e(15);

    /* Read/write tests */
    if (write(n, b, 4) != 4) e(16);
    if (read(n, aa, 4) != 0) e(17);
    if (lseek(n, 0L, SEEK_SET) != 0L) e(18);
    if (read(n, aa, 4) != 4) e(19);
    if (aa[0] != 0 || aa[1] != 1 || aa[2] != 2 || aa[3] != 3) e(20);
    if (lseek(n, 0L, SEEK_SET) != 0L) e(21);
    if (lseek(n, 2L, SEEK_CUR) != 2L) e(22);
    if (read(n, aa, 4) != 2) e(23);
    if (aa[0] != 2 || aa[1] != 3 || aa[2] != 2 || aa[3] != 3) e(24);
    if (lseek(n, 2L, SEEK_SET) != 2L) e(25);
    clraa();
    if (write(n, c, 4) != 4) e(26);
    if (lseek(n, 0L, SEEK_SET) != 0L) e(27);
    if (read(n, aa, 10) != 6) e(28);
    if (aa[0] != 0 || aa[1] != 1 || aa[2] != 10 || aa[3] != 20) e(29);
    if (lseek(n, 16L, SEEK_SET) != 16L) e(30);
    if (lseek(n, 2040L, SEEK_END) != 2046L) e(31);
    if (read(n, aa, 10) != 0) e(32);
    if (lseek(n, 0L, SEEK_CUR) != 2046L) e(33);
    clraa();
    if (write(n, c, 4) != 4) e(34);
    if (lseek(n, 0L, SEEK_CUR) != 2050L) e(35);
    if (lseek(n, 2040L, SEEK_SET) != 2040L) e(36);
    clraa();
    if (read(n, aa, 20) != 10) e(37);
    if (aa[0] != 0 || aa[5] != 0 || aa[6] != 10 || aa[9] != 40) e(38);
    if (lseek(n, 10239L, SEEK_SET) != 10239L) e(39);
    if (write(n, d, 2) != 2) e(40);
    if (lseek(n, -2L, SEEK_END) != 10239L) e(41);
    if (read(n, aa, 2) != 2) e(42);
    if (aa[0] != 6 || aa[1] != 7) e(43);
    if (lseek(n, NB * 1024L - 2L, SEEK_SET) != NB * 1024L - 2L) e(44);
    if (write(n, b, 4) != 4) e(45);
    if (lseek(n, 0L, SEEK_SET) != 0L) e(46);
    if (lseek(n, -6L, SEEK_END) != 1024L * NB - 4) e(47);
    clraa();
    if (read(n, aa, 100) != 6) e(48);
    if (aa[0] != 0 || aa[1] != 0 || aa[3] != 1 || aa[4] != 2 || aa[5] != 3)
        e(49);
}
```

```
    if (lseek(n, 20000L, SEEK_SET) != 20000L) e(50);
    if (write(n, c, 4) != 4) e(51);
    if (lseek(n, -4L, SEEK_CUR) != 20000L) e(52);
    if (read(n, aa, 4) != 4) e(53);
    if (aa[0] != 10 || aa[1] != 20 || aa[2] != 30 || aa[3] != 40) e(54);
    if (close(n) != 0) e(55);
    if ((nl = creat("foop", 0777)) != 3) e(56);
    if (close(nl) != 0) e(57);
    unlink("foop");
}
}

void test19d()
{
    /* Test read. */

    int i, fd, pd[2];
    char bb[100];

    subtest = 4;

    for (i = 0; i < 100; i++) bb[i] = i;
    if ( (fd = creat("T19.d1", 0777)) != 3) e(1);  /* create test file */
    if (write(fd, bb, 100) != 100) e(2);
    if (close(fd) != 0) e(3);
    clraa();
    if ( (fd = open("T19.d1", O_RDONLY)) != 3) e(4);
    errno = 1000;
    if (read(fd, aa, 0) != 0) e(5);
    if (errno != 1000) e(6);
    if (read(fd, aa, 100) != 100) e(7);
    if (lseek(fd, 37L, SEEK_SET) != 37L) e(8);
    if (read(fd, aa, 10) != 10) e(9);
    if (lseek(fd, 0L, SEEK_CUR) != 47L) e(10);
    if (read(fd, aa, 100) != 53) e(11);
    if (aa[0] != 47) e(12);
    if (read(fd, aa, 1) != 0) e(13);
    if (close(fd) != 0) e(14);

    /* Read from pipe with no writer open. */
    if (pipe(pd) != 0) e(15);
    if (close(pd[1]) != 0) e(16);
    errno = 2000;
    if (read(pd[0], aa, 1) != 0) e(17);  /* must return EOF */
    if (errno != 2000) e(18);

    /* Read from a pipe with O_NONBLOCK set. */
    if (fcntl(pd[0], F_SETFL, O_NONBLOCK) != 0) e(19);  /* set O_NONBLOCK */
    /*
    if (read(pd[0], aa, 1) != -1) e(20);
    if (errno != EAGAIN) e(21);
    */
    if (close(pd[0]) != 0) e(19);
    if (unlink("T19.d1") != 0) e(20);
}

void test19e()
{
    /* Test link, unlink, stat, fstat, dup, umask. */

    int i, j, n, nl, flag;
    char a[255], b[255];
    struct stat s, sl;

    subtest = 5;
    for (i = 0; i < 2; i++) {
        umask(0);

        if ((n = creat("T3", 0702)) < 0) e(1);
        if (link("T3", "newT3") < 0) e(2);
        if ((nl = open("newT3", O_RDWR)) < 0) e(3);
        for (j = 0; j < 255; j++) a[j] = j;
        if (write(n, a, 255) != 255) e(4);
    }
}
```

```

if (read(n1, b, 255) != 255) e(5);
flag = 0;
for (j = 0; j < 255; j++)
    if (a[j] != b[j]) flag++;
if (flag) e(6);
if (unlink("T3") < 0) e(7);
if (close(n) < 0) e(8);
if (close(n1) < 0) e(9);
if ((n1 = open("newT3", O_RDONLY)) < 0) e(10);
if (read(n1, b, 255) != 255) e(11);
flag = 0;
for (j = 0; j < 255; j++)
    if (a[j] != b[j]) flag++;
if (flag) e(12);

/* Now check out stat, fstat. */
if (stat("newT3", &s) < 0) e(13);
if (s.st_mode != (mode_t) 0100702) e(14);
/* The cast was because regular modes are
 * negative :-(. Anyway, the magic number
 * should be (S_IFREG | S_IRWXU | S_IWOTH)
 * for POSIX.
 */
if (s.st_nlink != 1) e(15);
if (s.st_size != 255L) e(16);
if (fstat(n1, &s1) < 0) e(17);
if (s.st_dev != s1.st_dev) e(18);
if (s.st_ino != s1.st_ino) e(19);
if (s.st_mode != s1.st_mode) e(20);
if (s.st_nlink != s1.st_nlink) e(21);
if (s.st_uid != s1.st_uid) e(22);
if (s.st_gid != s1.st_gid) e(23);
if (s.st_rdev != s1.st_rdev) e(24);
if (s.st_size != s1.st_size) e(25);
if (s.st_atime != s1.st_atime) e(26);
if (close(n1) < 0) e(27);
if (unlink("newT3") < 0) e(28);

umask(040);
if ((n = creat("T3a", 0777)) < 0) e(29);
if (stat("T3a", &s) < 0) e(30);
if (s.st_mode != (mode_t) 0100737) e(31); /* negative :-( */
if (unlink("T3a") < 0) e(32);
if (close(n1) < 0) e(33);

/* Dup */
if ((n = creat("T3b", 0777)) < 0) e(34);
if (close(n) < 0) e(35);
if ((n = open("T3b", O_RDWR)) < 0) e(36);
if ((n1 = dup(n)) != n + 1) e(37);
if (write(n, a, 255) != 255) e(38);
read(n1, b, 20);
if (lseek(n, 0L, SEEK_SET) != 0L) e(39);
if ((j = read(n1, b, 512)) != 255) e(40);
if (unlink("T3b") < 0) e(41);
if (close(n) < 0) e(42);
if (close(n1) < 0) e(43);
}
}

```

```

void test19f()
{
/* Test large files to see if indirect block stuff works. */

```

```

    int fd, i;
    long pos;

```

```

    subtest = 6;

```

```

    if (passes > 0) return; /* takes too long to repeat this test */
    for (i = 0; i < NBOUNDS; i++) {
        pos = 1024L * bounds[i];
        fd = creat("T19f", 0777);

```

```
    if (fd < 0) e(10*i+1);
    if (lseek(fd, pos, 0) < 0) e(10*i+2);
    if (write(fd, buff, 30720) != 30720) e(10*i+3);
    if (close(fd) < 0) e(10*i+3);
    if (unlink("T19f") < 0) e(10*i+4);
}
}

void test19g()
{
/* Test POSIX calls for pipe, read, write, lseek and close. */

    int pipefd[2], n, i, fd;
    char buf[512], buf2[512];

    subtest = 7;

    for (i = 0; i < 512; i++) buf[i] = i % 128;

    if (pipe(pipefd) < 0) e(1);
    if (write(pipefd[1], buf, 512) != 512) e(2);
    if (read(pipefd[0], buf2, 512) != 512) e(3);
    if (close(pipefd[1]) != 0) e(4);
    if (close(pipefd[1]) >= 0) e(5);
    if (read(pipefd[0], buf2, 1) != 0) e(6);
    if (close(pipefd[0]) != 0) e(7);

    /* Test O_NONBLOCK on pipes. */
    if (pipe(pipefd) < 0) e(8);
    if (fcntl(pipefd[0], F_SETFL, O_NONBLOCK) != 0) e(9);
    if (read(pipefd[0], buf2, 1) != -1) e(10);
    if (errno != EAGAIN) e(11);
    if (close(pipefd[0]) != 0) e(12);
    if (close(pipefd[1]) != 0) e(13);

    /* Test read and lseek. */
    if ( (fd = creat("T19.gl", 0777)) != 3) e(14);          /* create test file */
    if (write(fd, buf, 512) != 512) e(15);
    errno = 3000;
    if (read(fd, buf2, 512) != -1) e(16);
    if (errno != EBADF) e(17);
    if (close(fd) != 0) e(18);
    if ( (fd = open("T19.gl", O_RDWR)) != 3) e(19);
    if (read(fd, buf2, 512) != 512) e(20);
    if (read(fd, buf2, 512) != 0) e(21);
    if (lseek(fd, 100L, SEEK_SET) != 100L) e(22);
    if (read(fd, buf2, 512) != 412) e(23);
    if (lseek(fd, 1000L, SEEK_SET) != 1000L) e(24);

    /* Test write. */
    if (lseek(fd, -1000L, SEEK_CUR) != 0) e(25);
    if (write(fd, buf, 512) != 512) e(26);
    if (lseek(fd, 2L, SEEK_SET) != 2) e(27);
    if (write(fd, buf, 3) != 3) e(28);
    if (lseek(fd, -2L, SEEK_CUR) != 3) e(29);
    if (write(fd, &buf[30], 1) != 1) e(30);
    if (lseek(fd, 2L, SEEK_CUR) != 6) e(31);
    if (write(fd, &buf[60], 1) != 1) e(32);
    if (lseek(fd, -512L, SEEK_END) != 0) e(33);
    if (read(fd, buf2, 8) != 8) e(34);
    errno = 4000;
    if (buf2[0] != 0 || buf2[1] != 1 || buf2[2] != 0 || buf2[3] != 30) e(35);
    if (buf2[4] != 2 || buf2[5] != 5 || buf2[6] != 60 || buf2[7] != 7) e(36);

    /* Turn the O_APPEND flag on. */
    if (fcntl(fd, F_SETFL, O_APPEND) != 0) e(37);
    if (lseek(fd, 0L, SEEK_SET) != 0) e(38);
    if (write(fd, &buf[100], 1) != 1) e(39);
    if (lseek(fd, 0L, SEEK_SET) != 0) e(40);
    if (read(fd, buf2, 10) != 10) e(41);
    if (buf2[0] != 0) e(42);
    if (lseek(fd, -1L, SEEK_END) != 512) e(43);
    if (read(fd, buf2, 10) != 1) e(44);
    if (buf2[0] != 100) e(45);
}
```

```

if (close(fd) != 0) e(46);

/* Now try write with O_NONBLOCK. */
if (pipe(pipefd) != 0) e(47);
if (fcntl(pipefd[1], F_SETFL, O_NONBLOCK) != 0) e(48);
if (write(pipefd[1], buf, 512) != 512) e(49);
if (write(pipefd[1], buf, 512) != 512) e(50);
errno = 0;
for (i = 1; i < 20; i++) {
    n = write(pipefd[1], buf, 512);
    if (n == 512) continue;
    if (n != -1 || errno != EAGAIN) {e(51); break;}
}
if (read(pipefd[0], buf, 512) != 512) e(52);
if (close(pipefd[0]) != 0) e(53);

/* Write to a pipe with no reader. This should generate a signal. */
signal(SIGPIPE, pipecatcher);
errno = 0;
if (write(pipefd[1], buf, 1) != -1) e(54);
if (errno != EPIPE) e(55);
if (pipesigs != passes + 1) e(56); /* we should have had the sig now */
if (close(pipefd[1]) != 0) e(57);
errno = 0;
if (write(100, buf, 512) != -1) e(58);
if (errno != EBADF) e(59);
if (unlink("T19.gl") != 0) e(60);
}

void clraa()
{
    int i;
    for (i = 0; i < 100; i++) aa[i] = 0;
}

void pipecatcher(s)
int s; /* it is supposed to have an arg */
{
    pipesigs++;
}

void e(n)
int n;
{
    int err_num = errno; /* save errno in case printf clobbers it */

    printf("Subtest %d, error %d errno=%d ", subtest, n, errno);
    fflush(stdout); /* aargh! Most results go to stdout and are
                    * messed up by perror going to stderr.
                    * Should replace perror by printf and strerror
                    * in all the tests.
                    */
    errno = err_num; /* restore errno, just in case */
    perror("");
    if (errct++ > MAX_ERROR) {
        printf("Too many errors; test aborted\n");
        chdir("..");
        system("rm -rf DIR*");
        exit(1);
    }
}

void quit()
{
    chdir("..");
    system("rm -rf DIR*");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
        printf("%d errors\n", errct);
        exit(1);
    }
}

```



```
}  
}
```

```
/* test 2 */

#include <sys/types.h>
#include <sys/times.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <stdio.h>

#define ITERATIONS 5
#define MAX_ERROR 4

int is, array[4], parsigs, parcum, sigct, cumsig, errct, subtest;
int iteration, kk = 0, errct = 0;
char buf[2048];

_PROTOTYPE(int main, (int argc, char *argv []));
_PROTOTYPE(void test2a, (void));
_PROTOTYPE(void test2b, (void));
_PROTOTYPE(void test2c, (void));
_PROTOTYPE(void test2d, (void));
_PROTOTYPE(void test2e, (void));
_PROTOTYPE(void test2f, (void));
_PROTOTYPE(void test2g, (void));
_PROTOTYPE(void test2h, (void));
_PROTOTYPE(void sigpip, (int s));
_PROTOTYPE(void quit, (void));
_PROTOTYPE(void e, (int n));

int main(argc, argv)
int argc;
char *argv[];
{
    int i, m = 0xFFFF;

    sync();

    if (argc == 2) m = atoi(argv[1]);

    printf("Test 2 ");
    fflush(stdout);          /* have to flush for child's benefit */

    system("rm -rf DIR_02; mkdir DIR_02");
    chdir("DIR_02");

    for (i = 0; i < ITERATIONS; i++) {
        iteration = i;
        if (m & 0001) test2a();
        if (m & 0002) test2b();
        if (m & 0004) test2c();
        if (m & 0010) test2d();
        if (m & 0020) test2e();
        if (m & 0040) test2f();
        if (m & 0100) test2g();
        if (m & 0200) test2h();
    }
    subtest = 100;
    if (cumsig != ITERATIONS) e(101);
    quit();
    return(-1);              /* impossible */
}

void test2a()
{
    /* Test pipes */

    int fd[2];
    int n, i, j, q = 0;

    subtest = 1;
    if (pipe(fd) < 0) {
```

```

        printf("pipe error. errno= %d\n", errno);
        errct++;
        quit();
    }
    i = fork();
    if (i < 0) {
        printf("fork failed\n");
        errct++;
        quit();
    }
    if (i != 0) {
        /* Parent code */
        close(fd[0]);
        for (i = 0; i < 2048; i++) buf[i] = i & 0377;
        for (q = 0; q < 8; q++) {
            if (write(fd[1], buf, 2048) < 0) {
                printf("write pipe err. errno=%d\n", errno);
                errct++;
                quit();
            }
        }
        close(fd[1]);
        wait(&q);
        if (q != 256 * 58) {
            printf("wrong exit code %d\n", q);
            errct++;
            quit();
        }
    } else {
        /* Child code */
        close(fd[1]);
        for (q = 0; q < 32; q++) {
            n = read(fd[0], buf, 512);
            if (n != 512) {
                printf("read yielded %d bytes, not 512\n", n);
                errct++;
                quit();
            }
            for (j = 0; j < n; j++)
                if ((buf[j] & 0377) != (kk & 0377)) {
                    printf("wrong data: %d %d %d\n",
                        j, buf[j] & 0377, kk & 0377);
                } else {
                    kk++;
                }
        }
        exit(58);
    }
}

void test2b()
{
    int fd[2], n;
    char buf[4];

    subtest = 2;
    sigct = 0;
    signal(SIGPIPE, sigpip);
    pipe(fd);
    if (fork()) {
        /* Parent */
        close(fd[0]);
        while (sigct == 0) {
            write(fd[1], buf, 1);
        }
        wait(&n);
    } else {
        /* Child */
        close(fd[0]);
        close(fd[1]);
        exit(0);
    }
}

```

```
void test2c()
{
    int n;

    subtest = 3;
    signal(SIGINT, SIG_DFL);
    is = 0;
    if ((array[is++] = fork()) > 0) {
        if ((array[is++] = fork()) > 0) {
            if ((array[is++] = fork()) > 0) {
                if ((array[is++] = fork()) > 0) {
                    signal(SIGINT, SIG_IGN);
                    kill(array[0], SIGINT);
                    kill(array[1], SIGINT);
                    kill(array[2], SIGINT);
                    kill(array[3], SIGINT);
                    wait(&n);
                    wait(&n);
                    wait(&n);
                    wait(&n);
                } else {
                    pause();
                }
            } else {
                pause();
            }
        } else {
            pause();
        }
    } else {
        pause();
    }
}

void test2d()
{
    int pid, stat_loc, s;

    /* Test waitpid. */
    subtest = 4;

    /* Test waitpid(pid, arg2, 0) */
    pid = fork();
    if (pid < 0) e(1);
    if (pid > 0) {
        /* Parent. */
        s = waitpid(pid, &stat_loc, 0);
        if (s != pid) e(2);
        if (WIFEXITED(stat_loc) == 0) e(3);
        if (WIFSIGNALED(stat_loc) != 0) e(4);
        if (WEXITSTATUS(stat_loc) != 22) e(5);
    } else {
        /* Child */
        exit(22);
    }

    /* Test waitpid(-1, arg2, 0) */
    pid = fork();
    if (pid < 0) e(6);
    if (pid > 0) {
        /* Parent. */
        s = waitpid(-1, &stat_loc, 0);
        if (s != pid) e(7);
        if (WIFEXITED(stat_loc) == 0) e(8);
        if (WIFSIGNALED(stat_loc) != 0) e(9);
        if (WEXITSTATUS(stat_loc) != 33) e(10);
    } else {
        /* Child */
        exit(33);
    }

    /* Test waitpid(0, arg2, 0) */
    pid = fork();
```

```
if (pid < 0) e(11);
if (pid > 0) {
    /* Parent. */
    s = waitpid(0, &stat_loc, 0);
    if (s != pid) e(12);
    if (WIFEXITED(stat_loc) == 0) e(13);
    if (WIFSIGNALED(stat_loc) != 0) e(14);
    if (WEXITSTATUS(stat_loc) != 44) e(15);
} else {
    /* Child */
    exit(44);
}

/* Test waitpid(0, arg2, WNOHANG) */
signal(SIGTERM, SIG_DFL);
pid = fork();
if (pid < 0) e(16);
if (pid > 0) {
    /* Parent. */
    s = waitpid(0, &stat_loc, WNOHANG);
    if (s != 0) e(17);
    if (kill(pid, SIGTERM) != 0) e(18);
    if (waitpid(pid, &stat_loc, 0) != pid) e(19);
    if (WIFEXITED(stat_loc) != 0) e(20);
    if (WIFSIGNALED(stat_loc) == 0) e(21);
    if (WTERMSIG(stat_loc) != SIGTERM) e(22);
} else {
    /* Child */
    pause();
}

/* Test some error conditions. */
errno = 9999;
if (waitpid(0, &stat_loc, 0) != -1) e(23);
if (errno != ECHILD) e(24);
errno = 9999;
if (waitpid(0, &stat_loc, WNOHANG) != -1) e(25);
if (errno != ECHILD) e(26);
}

void test2e()
{
    int pid1, pid2, stat_loc, s;

    /* Test waitpid with two children. */
    subtest = 5;
    if (iteration > 1) return;          /* slow test, don't do it too much */
    if ( (pid1 = fork()) ) {
        /* Parent. */
        if ( (pid2 = fork()) ) {
            /* Parent. Collect second child first. */
            s = waitpid(pid2, &stat_loc, 0);
            if (s != pid2) e(1);
            if (WIFEXITED(stat_loc) == 0) e(2);
            if (WIFSIGNALED(stat_loc) != 0) e(3);
            if (WEXITSTATUS(stat_loc) != 222) e(4);

            /* Now collect first child. */
            s = waitpid(pid1, &stat_loc, 0);
            if (s != pid1) e(5);
            if (WIFEXITED(stat_loc) == 0) e(6);
            if (WIFSIGNALED(stat_loc) != 0) e(7);
            if (WEXITSTATUS(stat_loc) != 111) e(8);
        } else {
            /* Child 2. */
            sleep(2);          /* child 2 delays before exiting. */
            exit(222);
        }
    } else {
        /* Child 1. */
        exit(111);          /* child 1 exits immediately */
    }
}
```

```
}

void test2f()
{
    /* test getpid, getppid, getuid, etc. */

    pid_t pid, pid1, ppid, cpid, stat_loc, err;

    subtest = 6;
    errno = -2000;
    err = 0;
    pid = getpid();
    if ( (pid1 = fork()) ) {
        /* Parent. Do nothing. */
        if (wait(&stat_loc) != pid1) e(1);
        if (WEXITSTATUS(stat_loc) != (pid1 & 0377)) e(2);
    } else {
        /* Child. Get ppid. */
        cpid = getpid();
        ppid = getppid();
        if (ppid != pid) err = 3;
        if (cpid == ppid) err = 4;
        exit(cpid & 0377);
    }
    if (err != 0) e(err);
}

void test2g()
{
    /* test time(), times() */

    time_t t1, t2;
    clock_t t3, t4;
    struct tms tmsbuf;

    subtest = 7;
    errno = -7000;

    /* First time(). */
    t1 = -1;
    t2 = -2;
    t1 = time(&t2);
    if (t1 < 6500000000L) e(1);    /* 6500000000 is Sept. 1990 */
    if (t1 != t2) e(2);
    t1 = -1;
    t1 = time( (time_t *) NULL);
    if (t1 < 6500000000L) e(3);
    t3 = times(&tmsbuf);
    sleep(1);
    t2 = time( (time_t *) NULL);
    if (t2 < 0L) e(4);
    if (t2 - t1 < 1) e(5);

    /* Now times(). */
    t4 = times(&tmsbuf);
    if ( t4 == (clock_t) -1) e(6);
    if (t4 - t3 < CLOCKS_PER_SEC) e(7);
    if (tmsbuf.tms_utime < 0) e(8);
    if (tmsbuf.tms_stime < 0) e(9);
    if (tmsbuf.tms_cutime < 0) e(10);
    if (tmsbuf.tms_cstime < 0) e(11);
}

void test2h()
{
    /* Test getgroups(). */

    gid_t g[10];

    subtest = 8;
    errno = -8000;
    if (getgroups(10, g) != 0) e(1);
    if (getgroups(1, g) != 0) e(2);
    if (getgroups(0, g) != 0) e(3);
}
```

```
}

void sigpip(s)
int s;                /* for ANSI */
{
    sigct++;
    cumsig++;
}

void quit()
{
    chdir("..");
    system("rm -rf DIR*");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
        printf("%d errors\n", errct);
        exit(4);
    }
}

void e(n)
int n;
{
    int err_num = errno;        /* save errno in case printf clobbers it */

    printf("Subtest %d, error %d errno=%d ", subtest, n, errno);
    errno = err_num;           /* restore errno, just in case */
    perror("");
    if (errct++ > MAX_ERROR) {
        printf("Too many errors; test aborted\n");
        chdir("..");
        system("rm -rf DIR*");
        exit(1);
    }
}
```

```

/* test20: fcntl()                               Author: Jan-Mark Wams (jms@cs.vu.nl) */

/* Some things have to be checked for ``exec()'' call's. Therefor
** there is a check routine called ``do_check()'' that will be
** called if the first argument (``argv[0]'' equals ``DO CHECK.''
** Note that there is no way the shell (``/bin/sh'') will set
** ``argv[0]'' to this funny value. (Unless we rename ``test20''
** to ``DO CHECK'' ;-)
*/

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <errno.h>
#include <time.h>
#include <stdio.h>

#define MAX_ERROR          4
#define ITERATIONS         10

#define System(cmd)        if (system(cmd) != 0) printf("``%s'' failed\n", cmd)
#define Chdir(dir)         if (chdir(dir) != 0) printf("Can't goto %s\n", dir)
#define Stat(a,b)          if (stat(a,b) != 0) printf("Can't stat %s\n", a)

int errct = 0;
int subtest = 1;
int superuser;
char MaxName[NAME_MAX + 1]; /* Name of maximum length */
char MaxPath[PATH_MAX];     /* Same for path */
char ToLongName[NAME_MAX + 2]; /* Name of maximum +1 length */
char ToLongPath[PATH_MAX + 1]; /* Same for path, both too long */

_PROTOTYPE(void main, (int argc, char *argv[]));
_PROTOTYPE(void test20a, (void));
_PROTOTYPE(void test20b, (void));
_PROTOTYPE(void test20c, (void));
_PROTOTYPE(void test20d, (void));
_PROTOTYPE(int do_check, (void));
_PROTOTYPE(void makelongnames, (void));
_PROTOTYPE(void e, (int number));
_PROTOTYPE(void quit, (void));

char executable[1024];

void main(argc, argv)
int argc;
char *argv[];
{
    int i, m = 0xFFFF;

    sync();
    if (argc == 2) m = atoi(argv[1]);

    /* If we have to check things, call do_check(). */
    if (strcmp(argv[0], "DO CHECK") == 0) exit(do_check());

    /* Get the path of the executable. */
    strcpy(executable, "./");
    strcat(executable, argv[0]);

    printf("Test 20 ");
    fflush(stdout);
    System("rm -rf DIR_20; mkdir DIR_20");
    Chdir("DIR_20");
    makelongnames();
    superuser = (geteuid() == 0);

    for (i = 0; i < ITERATIONS; i++) {
        test20a();
    }
}

```



```
        test20b();
        test20c();
        test20d();
    }
    quit();
}

void test20a()
{
    /* Test normal operation. */
    subtest = 1;
    System("rm -rf ../DIR_20/*");
}

void test20b()
{
    subtest = 2;
    System("rm -rf ../DIR_20/*");
}

void test20c()
{
    subtest = 3;
    System("rm -rf ../DIR_20/*");
}

/* Open fds 3, 4, 5 and 6. Set FD_CLOEXEC on 5 and 6. Exclusively lock the
** first 10 bytes of fd no. 3. Shared lock fd no. 7. Lock fd no. 8 after
** the fork. Do a 'exec()' call with a funny argv[0] and check the return
** value.
*/
void test20d()
{
    /* Test locks with 'fork()' and 'exec()' */
    int fd3, fd4, fd5, fd6, fd7, fd8;
    int stat_loc;
    int do_check_retval;
    char *argv[2];
    struct flock fl;

    subtest = 4;

    argv[0] = "DO CHECK";
    argv[1] = (char *) NULL;

    fl.l_whence = SEEK_SET;
    fl.l_start = 0;
    fl.l_len = 10;

    /* Make a dummy files and open them. */
    System("echo 'Great Balls Of Fire!' > file3");
    System("echo 'Great Balls Of Fire!' > file4");
    System("echo 'Great Balls Of Fire!' > file7");
    System("echo 'Great Balls Of Fire!' > file8");
    System("echo 'Great Balls Of Fire!' > file");
    if ((fd3 = open("file3", O_RDWR)) != 3) e(1);
    if ((fd4 = open("file4", O_RDWR)) != 4) e(2);
    if ((fd5 = open("file", O_RDWR)) != 5) e(3);
    if ((fd6 = open("file", O_RDWR)) != 6) e(4);
    if ((fd7 = open("file7", O_RDWR)) != 7) e(5);
    if ((fd8 = open("file8", O_RDWR)) != 8) e(6);

    /* Set FD_CLOEXEC flags on fd5 and fd6. */
    if (fcntl(fd5, F_SETFD, FD_CLOEXEC) == -1) e(7);
    if (fcntl(fd6, F_SETFD, FD_CLOEXEC) == -1) e(8);

    /* Lock the first ten bytes from fd3 (for writing). */
    fl.l_type = F_WRLCK;
    if (fcntl(fd3, F_SETLK, &fl) == -1) e(9);

    /* Lock (for reading) fd7. */
    fl.l_type = F_RDLCK;
    if (fcntl(fd7, F_SETLK, &fl) == -1) e(10);

    switch (fork()) {
        case -1: printf("Can't fork\n"); break;
    }
}
```

```

case 0:
    alarm(20);

    /* Lock fd8. */
    fl.l_type = F_WRLCK;
    if (fcntl(fd8, F_SETLK, &fl) == -1) e(11);

    /* Check the lock on fd3 and fd7. */
    fl.l_type = F_WRLCK;
    if (fcntl(fd3, F_GETLK, &fl) == -1) e(12);
    if (fl.l_type != F_WRLCK) e(13);
    if (fl.l_pid != getpid()) e(14);
    fl.l_type = F_WRLCK;
    if (fcntl(fd7, F_GETLK, &fl) == -1) e(15);
    if (fl.l_type != F_RDLCK) e(16);
    if (fl.l_pid != getpid()) e(17);

    /* Check FD_CLOEXEC flags. */
    if ((fcntl(fd3, F_GETFD) & FD_CLOEXEC) != 0) e(18);
    if ((fcntl(fd4, F_GETFD) & FD_CLOEXEC) != 0) e(19);
    if ((fcntl(fd5, F_GETFD) & FD_CLOEXEC) != FD_CLOEXEC) e(20);
    if ((fcntl(fd6, F_GETFD) & FD_CLOEXEC) != FD_CLOEXEC) e(21);
    if ((fcntl(fd7, F_GETFD) & FD_CLOEXEC) != 0) e(22);
    if ((fcntl(fd8, F_GETFD) & FD_CLOEXEC) != 0) e(23);

    execlp(executable + 3, "DO CHECK", (char *) NULL);
    execlp(executable, "DO CHECK", (char *) NULL);
    printf("Can't exec %s or %s\n", executable + 3, executable);
    exit(0);

default:
    wait(&stat_loc);
    if (WIFSIGNALED(stat_loc)) e(24);          /* Alarm? */
    if (WIFEXITED(stat_loc) == 0) {
        errct=10000;
        quit();
    }
}

/* Check the return value of do_check(). */
do_check_retval = WEXITSTATUS(stat_loc);
if ((do_check_retval & 0x11) == 0x11) e(25);
if ((do_check_retval & 0x12) == 0x12) e(26);
if ((do_check_retval & 0x14) == 0x14) e(27);
if ((do_check_retval & 0x18) == 0x18) e(28);
if ((do_check_retval & 0x21) == 0x21) e(29);
if ((do_check_retval & 0x22) == 0x22) e(30);
if ((do_check_retval & 0x24) == 0x24) e(31);
if ((do_check_retval & 0x28) == 0x28) e(32);
if ((do_check_retval & 0x41) == 0x41) e(33);
if ((do_check_retval & 0x42) == 0x42) e(34);
if ((do_check_retval & 0x44) == 0x44) e(35);
if ((do_check_retval & 0x48) == 0x48) e(36);
if ((do_check_retval & 0x81) == 0x81) e(37);
if ((do_check_retval & 0x82) == 0x82) e(38);
if ((do_check_retval & 0x84) == 0x84) e(39);
if ((do_check_retval & 0x88) == 0x88) e(40);

switch (fork()) {
    case -1: printf("Can't fork\n");      break;
    case 0:
        alarm(20);

        /* Lock fd8. */
        fl.l_type = F_WRLCK;
        if (fcntl(fd8, F_SETLK, &fl) == -1) e(41);

        execvp(executable + 3, argv);
        execvp(executable, argv);
        printf("Can't exec %s or %s\n", executable + 3, executable);
        exit(0);

    default:
        wait(&stat_loc);

```

```

    if (WIFSIGNALED(stat_loc)) e(48);      /* Alarm? */
}

/* Check the return value of do_check(). */
do_check_retval = WEXITSTATUS(stat_loc);
if ((do_check_retval & 0x11) == 0x11) e(49);
if ((do_check_retval & 0x12) == 0x12) e(50);
if ((do_check_retval & 0x14) == 0x14) e(51);
if ((do_check_retval & 0x18) == 0x18) e(52);
if ((do_check_retval & 0x21) == 0x21) e(53);
if ((do_check_retval & 0x22) == 0x22) e(54);
if ((do_check_retval & 0x24) == 0x24) e(55);
if ((do_check_retval & 0x28) == 0x28) e(56);
if ((do_check_retval & 0x41) == 0x41) e(57);
if ((do_check_retval & 0x42) == 0x42) e(58);
if ((do_check_retval & 0x44) == 0x44) e(59);
if ((do_check_retval & 0x48) == 0x48) e(60);
if ((do_check_retval & 0x81) == 0x81) e(61);
if ((do_check_retval & 0x82) == 0x82) e(62);
if ((do_check_retval & 0x84) == 0x84) e(63);
if ((do_check_retval & 0x88) == 0x88) e(64);

fl.l_type = F_UNLCK;
if (fcntl(fd3, F_SETLK, &fl) == -1) e(65);
if (fcntl(fd7, F_SETLK, &fl) == -1) e(66);

if (close(fd3) != 0) e(67);
if (close(fd4) != 0) e(68);
if (close(fd5) != 0) e(69);
if (close(fd6) != 0) e(70);
if (close(fd7) != 0) e(71);
if (close(fd8) != 0) e(72);

System("rm -f ./DIR_20/*\n");
}

/* This routine checks that fds 0 through 4, 7 and 8 are open and the rest
** is closed. It also checks if we can lock the first 10 bytes on fd no. 3
** and 4. It should not be possible to lock fd no. 3, but it should be
** possible to lock fd no. 4. See ``test20d()`` for usage of this routine.
*/
int do_check()
{
    int i;
    int retval = 0;
    struct flock fl;

    fl.l_whence = SEEK_SET;
    fl.l_start = 0;
    fl.l_len = 10;

    /* All std.. are open. */
    if (fcntl(0, F_GETFD) == -1) retval |= 0x11;
    if (fcntl(1, F_GETFD) == -1) retval |= 0x11;
    if (fcntl(2, F_GETFD) == -1) retval |= 0x11;

    /* Fd no. 3, 4, 7 and 8 are open. */
    if (fcntl(3, F_GETFD) == -1) retval |= 0x12;
    if (fcntl(4, F_GETFD) == -1) retval |= 0x12;
    if (fcntl(7, F_GETFD) == -1) retval |= 0x12;

    /* Fd no. 5, 6 and 9 trough OPEN_MAX are closed. */
    if (fcntl(5, F_GETFD) != -1) retval |= 0x14;
    if (fcntl(6, F_GETFD) != -1) retval |= 0x14;
    for (i = 9; i < OPEN_MAX; i++)
        if (fcntl(i, F_GETFD) != -1) retval |= 0x18;

#ifdef 0
    /* Fd no. 3 is WRLCKed. */
    fl.l_type = F_WRLCK;
    if (fcntl(3, F_SETLK, &fl) != -1) retval |= 0x21;
    if (errno != EACCES && errno != EAGAIN) retval |= 0x22;
    fl.l_type = F_RDLCK;
    if (fcntl(3, F_SETLK, &fl) != -1) retval |= 0x24;
#endif
}

```

```

if (errno != EACCES && errno != EAGAIN) retval |= 0x22;
fl.l_type = F_RDLCK;
if (fcntl(3, F_GETLK, &fl) == -1) retval |= 0x28;
if (fl.l_type != F_WRLCK) retval |= 0x28;
if (fl.l_pid != getpid()) retval |= 0x28;
fl.l_type = F_WRLCK;
if (fcntl(3, F_GETLK, &fl) == -1) retval |= 0x28;
if (fl.l_type != F_WRLCK) retval |= 0x28;
if (fl.l_pid != getpid()) retval |= 0x28;
#endif

/* Fd no. 4 is not locked. */
fl.l_type = F_WRLCK;
if (fcntl(4, F_SETLK, &fl) == -1) retval |= 0x41;
if (fcntl(4, F_GETLK, &fl) == -1) retval |= 0x42;
#if 0 /* XXX - see test7.c */
if (fl.l_type != F_WRLCK) retval |= 0x42;
if (fl.l_pid != getpid()) retval |= 0x42;
#endif /* 0 */

/* Fd no. 8 is locked after the fork, it is ours. */
fl.l_type = F_WRLCK;
if (fcntl(8, F_SETLK, &fl) == -1) retval |= 0x44;
if (fcntl(8, F_GETLK, &fl) == -1) retval |= 0x48;
#if 0 /* XXX - see test7.c */
if (fl.l_type != F_WRLCK) retval |= 0x48;
if (fl.l_pid != getpid()) retval |= 0x48;
#endif /* 0 */

#if 0
/* Fd no. 7 is RDLCKed. */
fl.l_type = F_WRLCK;
if (fcntl(7, F_SETLK, &fl) != -1) retval |= 0x81;
if (errno != EACCES && errno != EAGAIN) retval |= 0x82;
fl.l_type = F_RDLCK;
if (fcntl(7, F_SETLK, &fl) == -1) retval |= 0x84;
fl.l_type = F_RDLCK;
if (fcntl(7, F_GETLK, &fl) == -1) retval |= 0x88;
if (fl.l_type != F_UNLCK) retval |= 0x88;
fl.l_type = F_WRLCK;
if (fcntl(7, F_GETLK, &fl) == -1) retval |= 0x88;
if (fl.l_type != F_RDLCK) retval |= 0x88;
if (fl.l_pid != getppid()) retval |= 0x88;
#endif

return retval;
}

void makelongnames()
{
    register int i;

    memset(MaxName, 'a', NAME_MAX);
    MaxName[NAME_MAX] = '\0';
    for (i = 0; i < PATH_MAX - 1; i++) { /* idem path */
        MaxPath[i++] = '.';
        MaxPath[i] = '/';
    }
    MaxPath[PATH_MAX - 1] = '\0';

    strcpy(ToLongName, MaxName); /* copy them Max to ToLong */
    strcpy(ToLongPath, MaxPath);

    ToLongName[NAME_MAX] = 'a';
    ToLongName[NAME_MAX + 1] = '\0'; /* extend ToLongName by one too many */
    ToLongPath[PATH_MAX - 1] = '/';
    ToLongPath[PATH_MAX] = '\0'; /* inc ToLongPath by one */
}

void e(n)
int n;
{
    int err_num = errno; /* Save in case printf clobbers it. */

```

```
printf("Subtest %d, error %d errno=%d: ", subtest, n, errno);
errno = err_num;
perror("");
if (errct++ > MAX_ERROR) {
    printf("Too many errors; test aborted\n");
    chdir("..");
    system("rm -rf DIR*");
    exit(1);
}
errno = 0;
}

void quit()
{
    Chdir("..");
    System("rm -rf DIR_20");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else if (errct < 10000) {
        printf("%d errors\n", errct);
        exit(1);
    } else {
        printf("errors\n");
        exit(2);
    }
}
```

```
/* POSIX test program (21).                               Author: Andy Tanenbaum */

/* The following POSIX calls are tested:
 *
 *      rename(), mkdir(), rmdir()
 */

#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
#include <limits.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <stdio.h>

#define ITERATIONS      1
#define MAX_ERROR 4

int subtest, errct;

_PROTOTYPE(int main, (int argc, char *argv []));
_PROTOTYPE(void test21a, (void));
_PROTOTYPE(void test21b, (void));
_PROTOTYPE(void test21c, (void));
_PROTOTYPE(void test21d, (void));
_PROTOTYPE(void test21e, (void));
_PROTOTYPE(void test21f, (void));
_PROTOTYPE(void test21g, (void));
_PROTOTYPE(void test21h, (void));
_PROTOTYPE(void test21i, (void));
_PROTOTYPE(void test21k, (void));
_PROTOTYPE(void test21l, (void));
_PROTOTYPE(void test21m, (void));
_PROTOTYPE(void test21n, (void));
_PROTOTYPE(void test21o, (void));
_PROTOTYPE(int get_link, (char *name));
_PROTOTYPE(void e, (int n));
_PROTOTYPE(void quit, (void));

int main(argc, argv)
int argc;
char *argv[];
{
    int i, m = 0xFFFF;

    sync();
    if (geteuid() == 0 || getuid() == 0) {
        printf("Test 21 cannot run as root; test aborted\n");
        exit(1);
    }

    if (argc == 2) m = atoi(argv[1]);
    printf("Test 21 ");
    fflush(stdout);

    system("rm -rf DIR_21; mkdir DIR_21");
    chdir("DIR_21");

    for (i = 0; i < ITERATIONS; i++) {
        if (m & 00001) test21a();
        if (m & 00002) test21b();
        if (m & 00004) test21c();
        if (m & 00010) test21d();
        if (m & 00020) test21e();
        if (m & 00040) test21f();
        if (m & 01000) test21g();
        if (m & 00200) test21h();
        if (m & 00400) test21i();
        if (m & 01000) test21k();
        if (m & 02000) test21l();
        if (m & 04000) test21m();
    }
}
```

```
    if (m & 010000) test21n();
    if (m & 020000) test21o();
}
quit();
return(-1);          /* impossible */
}

void test21a()
{
    /* Test rename(). */

    int fd, fd2;
    char buf[PATH_MAX+1], buf1[PATH_MAX+1], buf2[PATH_MAX+1];
    struct stat stat1, stat2;

    subtest = 1;

    unlink("A1");          /* get rid of it if it exists */
    unlink("A2");          /* get rid of it if it exists */
    unlink("A3");          /* get rid of it if it exists */
    unlink("A4");          /* get rid of it if it exists */
    unlink("A5");          /* get rid of it if it exists */
    unlink("A6");          /* get rid of it if it exists */
    unlink("A7");          /* get rid of it if it exists */

    /* Basic test. Rename A1 to A2 and then A2 to A3. */
    if ( (fd=creat("A1", 0666)) < 0) e(1);
    if (write(fd, buf, 20) != 20) e(2);
    if (close(fd) < 0) e(3);
    if (rename("A1", "A2") < 0) e(4);
    if ( (fd=open("A2", O_RDONLY)) < 0) e(5);
    if (rename("A2", "A3") < 0) e(6);
    if ( (fd2=open("A3", O_RDONLY)) < 0) e(7);
    if (close(fd) != 0) e(8);
    if (close(fd2) != 0) e(9);
    if (unlink("A3") != 0) e(10);

    /* Now get the absolute path name of the current directory using getcwd()
     * and use it to test RENAME using different combinations of relative and
     * absolute path names.
     */
    if (getcwd(buf, PATH_MAX) == (char *) NULL) e(11);
    if ( (fd = creat("A4", 0666)) < 0) e(12);
    if (write(fd, buf, 30) != 30) e(13);
    if (close(fd) != 0) e(14);
    strcpy(buf1, buf);
    strcat(buf1, "/A4");
    if (rename(buf1, "A5") != 0) e(15);    /* rename(absolute, relative) */
    if (access("A5", 6) != 0) e(16);      /* use access to see if file exists */
    strcpy(buf2, buf);
    strcat(buf2, "/A6");
    if (rename("A5", buf2) != 0) e(17);    /* rename(relative, absolute) */
    if (access("A6", 6) != 0) e(18);
    if (access(buf2, 6) != 0) e(19);
    strcpy(buf1, buf);
    strcat(buf1, "/A6");
    strcpy(buf2, buf);
    strcat(buf2, "/A7");
    if (rename(buf1, buf2) != 0) e(20);    /* rename(absolute, absolute) */
    if (access("A7", 6) != 0) e(21);
    if (access(buf2, 6) != 0) e(22);

    /* Try renaming using names like "./A8" */
    if (rename("A7", "./A8") != 0) e(23);
    if (access("A8", 6) != 0) e(24);
    if (rename("./A8", "A9") != 0) e(25);
    if (access("A9", 6) != 0) e(26);
    if (rename("./A9", "./A10") != 0) e(27);
    if (access("A10", 6) != 0) e(28);
    if (access("./A10", 6) != 0) e(29);
    if (unlink("A10") != 0) e(30);

    /* Now see if directories can be renamed. */
    if (system("rm -rf ?uzzy scsi") != 0) e(31);
}
```

```
if (system("mkdir fuzzy") != 0) e(32);
if (rename("fuzzy", "wuzzy") != 0) e(33);
if ( (fd=creat("wuzzy/was_a_bear", 0666)) < 0) e(34);
if (access("wuzzy/was_a_bear", 6) != 0) e(35);
if (unlink("wuzzy/was_a_bear") != 0) e(36);
if (close(fd) != 0) e(37);
if (rename("wuzzy", "buzzy") != 0) e(38);
if (system("rmdir buzzy") != 0) e(39);

/* Now start testing the case that 'new' exists. */
if ( (fd = creat("horse", 0666)) < 0) e(40);
if ( (fd2 = creat("sheep", 0666)) < 0) e(41);
if (write(fd, buf, PATH_MAX) != PATH_MAX) e(42);
if (write(fd2, buf, 23) != 23) e(43);
if (stat("horse", &stat1) != 0) e(44);
if (rename("horse", "sheep") != 0) e(45);
if (stat("sheep", &stat2) != 0) e(46);
if (stat1.st_dev != stat2.st_dev) e(47);
if (stat1.st_ino != stat2.st_ino) e(48);
if (stat2.st_size != PATH_MAX) e(49);
if (access("horse", 6) == 0) e(50);
if (close(fd) != 0) e(51);
if (close(fd2) != 0) e(52);
if (rename("sheep", "sheep") != 0) e(53);
if (unlink("sheep") != 0) e(54);

/* Now try renaming something to a directory that already exists. */
if (system("mkdir fuzzy wuzzy") != 0) e(55);
if ( (fd = creat("fuzzy/was_a_bear", 0666)) < 0) e(56);
if (close(fd) != 0) e(57);
if (rename("fuzzy", "wuzzy") != 0) e(58); /* 'new' is empty dir */
if (system("mkdir scsi") != 0) e(59);
if (rename("scsi", "wuzzy") == 0) e(60); /* 'new' is full dir */
if (errno != EEXIST && errno != ENOTEMPTY) e(61);

/* Test 14 character names--always tricky. */
if (rename("wuzzy/was_a_bear", "wuzzy/was_not_a_bear") != 0) e(62);
if (access("wuzzy/was_not_a_bear", 6) != 0) e(63);
if (rename("wuzzy/was_not_a_bear", "wuzzy/was_not_a_duck") != 0) e(64);
if (access("wuzzy/was_not_a_duck", 6) != 0) e(65);
if (rename("wuzzy/was_not_a_duck", "wuzzy/was_a_bird") != 0) e(65);
if (access("wuzzy/was_a_bird", 6) != 0) e(66);

/* Test moves between directories. */
if (rename("wuzzy/was_a_bird", "beast") != 0) e(67);
if (access("beast", 6) != 0) e(68);
if (rename("beast", "wuzzy/was_a_cat") != 0) e(69);
if (access("wuzzy/was_a_cat", 6) != 0) e(70);

/* Test error conditions. 'scsi' and 'wuzzy/was_a_cat' exist now. */
if (rename("wuzzy/was_a_cat", "wuzzy/was_a_dog") != 0) e(71);
if (access("wuzzy/was_a_dog", 6) != 0) e(72);
if (chmod("wuzzy", 0) != 0) e(73);

errno = 0;
if (rename("wuzzy/was_a_dog", "wuzzy/was_a_pig") != -1) e(74);
if (errno != EACCES) e(75);

errno = 0;
if (rename("wuzzy/was_a_dog", "doggie") != -1) e(76);
if (errno != EACCES) e(77);

errno = 0;
if ( (fd = creat("beast", 0666)) < 0) e(78);
if (close(fd) != 0) e(79);
if (rename("beast", "wuzzy/was_a_twit") != -1) e(80);
if (errno != EACCES) e(81);

errno = 0;
if (rename("beast", "wuzzy") != -1) e(82);
if (errno != EISDIR) e(83);

errno = 0;
if (rename("beest", "baste") != -1) e(84);
```



```
if (errno != ENOENT) e(85);

errno = 0;
if (rename("wuzzy", "beast") != -1) e(86);
if (errno != ENOTDIR) e(87);

/* Test prefix rule. */
errno = 0;
if (chmod("wuzzy", 0777) != 0) e(88);
if (unlink("wuzzy/was_a_dog") != 0) e(89);
strcpy(buf1, buf);
strcat(buf1, "/wuzzy");
if (rename(buf, buf1) != -1) e(90);
if (errno != EINVAL) e(91);

if (system("rm -rf wuzzy beast scsi") != 0) e(92);
}

void test21b()
{
/* Test mkdir() and rmdir(). */

int i;
char name[3];
struct stat statbuf;

subtest = 2;

/* Simple stuff. */
if (mkdir("D1", 0700) != 0) e(1);
if (stat("D1", &statbuf) != 0) e(2);
if (!S_ISDIR(statbuf.st_mode)) e(3);
if ((statbuf.st_mode & 0777) != 0700) e(4);
if (rmdir("D1") != 0) e(5);

/* Make and remove 40 directories. By doing so, the directory has to
 * grow to 2 blocks. That presents plenty of opportunity for bugs.
 */
name[0] = 'D';
name[2] = '\0';
for (i = 0; i < 40; i++) {
    name[1] = 'A' + i;
    if (mkdir(name, 0700 + i%7) != 0) e(10+i); /* for simplicity */
}
for (i = 0; i < 40; i++) {
    name[1] = 'A' + i;
    if (rmdir(name) != 0) e(50+i);
}
}

void test21c()
{
/* Test mkdir() and rmdir(). */

subtest = 3;

if (mkdir("D1", 0777) != 0) e(1);
if (mkdir("D1/D2", 0777) != 0) e(2);
if (mkdir("D1/D2/D3", 0777) != 0) e(3);
if (mkdir("D1/D2/D3/D4", 0777) != 0) e(4);
if (mkdir("D1/D2/D3/D4/D5", 0777) != 0) e(5);
if (mkdir("D1/D2/D3/D4/D5/D6", 0777) != 0) e(6);
if (mkdir("D1/D2/D3/D4/D5/D6/D7", 0777) != 0) e(7);
if (mkdir("D1/D2/D3/D4/D5/D6/D7/D8", 0777) != 0) e(8);
if (mkdir("D1/D2/D3/D4/D5/D6/D7/D8/D9", 0777) != 0) e(9);
if (access("D1/D2/D3/D4/D5/D6/D7/D8/D9", 7) != 0) e(10);
if (rmdir("D1/D2/D3/D4/D5/D6/D7/D8/D9") != 0) e(11);
if (rmdir("D1/D2/D3/D4/D5/D6/D7/D8") != 0) e(12);
if (rmdir("D1/D2/D3/D4/D5/D6/D7") != 0) e(13);
if (rmdir("D1/D2/D3/D4/D5/D6") != 0) e(11);
if (rmdir("D1/D2/D3/D4/D5") != 0) e(13);
if (rmdir("D1/D2/D3/D4") != 0) e(14);
```

```
if (rmdir("D1/D2/D3") != 0) e(15);
if (rmdir("D1/D2") != 0) e(16);
if (rmdir("D1") != 0) e(17);
}

void test21d()
{
/* Test making directories with files and directories in them. */

    int fd1, fd2, fd3, fd4, fd5, fd6, fd7, fd8, fd9;

    subtest = 4;

    if (mkdir("D1", 0777) != 0) e(1);
    if (mkdir("D1/D2", 0777) != 0) e(2);
    if (mkdir("./D1/D3", 0777) != 0) e(3);
    if (mkdir("../D1/D4", 0777) != 0) e(4);
    if ( (fd1 = creat("D1/D2/x", 0700)) < 0) e(5);
    if ( (fd2 = creat("D1/D2/y", 0700)) < 0) e(6);
    if ( (fd3 = creat("D1/D2/z", 0700)) < 0) e(7);
    if ( (fd4 = creat("D1/D3/x", 0700)) < 0) e(8);
    if ( (fd5 = creat("D1/D3/y", 0700)) < 0) e(9);
    if ( (fd6 = creat("D1/D3/z", 0700)) < 0) e(10);
    if ( (fd7 = creat("D1/D4/x", 0700)) < 0) e(11);
    if ( (fd8 = creat("D1/D4/y", 0700)) < 0) e(12);
    if ( (fd9 = creat("D1/D4/z", 0700)) < 0) e(13);
    if (unlink("D1/D2/z") != 0) e(14);
    if (unlink("D1/D2/y") != 0) e(15);
    if (unlink("D1/D2/x") != 0) e(16);
    if (unlink("D1/D3/x") != 0) e(17);
    if (unlink("D1/D3/z") != 0) e(18);
    if (unlink("D1/D3/y") != 0) e(19);
    if (unlink("D1/D4/y") != 0) e(20);
    if (unlink("D1/D4/z") != 0) e(21);
    if (unlink("D1/D4/x") != 0) e(22);
    if (rmdir("D1/D2") != 0) e(23);
    if (rmdir("D1/D3") != 0) e(24);
    if (rmdir("D1/D4") != 0) e(25);
    if (rmdir("D1") != 0) e(26);
    if (close(fd1) != 0) e(27);
    if (close(fd2) != 0) e(28);
    if (close(fd3) != 0) e(29);
    if (close(fd4) != 0) e(30);
    if (close(fd5) != 0) e(31);
    if (close(fd6) != 0) e(32);
    if (close(fd7) != 0) e(33);
    if (close(fd8) != 0) e(34);
    if (close(fd9) != 0) e(35);
}

void test21e()
{
/* Test error conditions. */

    subtest = 5;

    if (mkdir("D1", 0677) != 0) e(1);
    errno = 0;
    if (mkdir("D1/D2", 0777) != -1) e(2);
    if (errno != EACCES) e(3);
    if (chmod("D1", 0577) != 0) e(4);
    errno = 0;
    if (mkdir("D1/D2", 0777) != -1) e(5);
    if (errno != EACCES) e(6);
    if (chmod("D1", 0777) != 0) e(7);
    errno = 0;
    if (mkdir("D1", 0777) != -1) e(8);
    if (errno != EEXIST) e(9);
    #if NAME_MAX == 14
    if (mkdir("D1/ABCDEFGHIJKLMNOPQRSTUVWXYZ", 0777) != 0) e(10);
    if (access("D1/ABCDEFGHIJKLMN", 7) != 0) e(11);
    if (rmdir("D1/ABCDEFGHIJKLMNOPQ") != 0) e(12);
    if (access("D1/ABCDEFGHIJKLMN", 7) != -1) e(13);
    #endif
}
```

```
#endif
errno = 0;
if (mkdir("D1/D2/x", 0777) != -1) e(14);
if (errno != ENOENT) e(15);

/* A particularly nasty test is when the parent has mode 0.  Although
 * this is unlikely to work, it had better not muck up the file system
 */
if (mkdir("D1/D2", 0777) != 0) e(16);
if (chmod("D1", 0) != 0) e(17);
errno = 0;
if (rmdir("D1/D2") != -1) e(18);
if (errno != EACCES) e(19);
if (chmod("D1", 0777) != 0) e(20);
if (rmdir("D1/D2") != 0) e(21);
if (rmdir("D1") != 0) e(22);
}

void test21f()
{
/* The rename() function affects the link count of all the files and
 * directories it goes near.  Test to make sure it gets everything ok.
 * There are four cases:
 *
 * 1. rename("d1/file1", "d1/file2"); - rename file without moving it
 * 2. rename("d1/file1", "d2/file2"); - move a file to another dir
 * 3. rename("d1/dir1", "d2/dir2"); - rename a dir without moving it
 * 4. rename("d1/dir1", "d2/dir2"); - move a dir to another dir
 *
 * Furthermore, a distinction has to be made when the target file exists
 * and when it does not exist, giving 8 cases in all.
 */

int fd, D1_before, D1_after, x_link, y_link;

/* Test case 1: renaming a file within the same directory. */
subtest = 6;
if (mkdir("D1", 0777) != 0) e(1);
if ( (fd = creat("D1/x", 0777)) < 0) e(2);
if (close(fd) != 0) e(3);
D1_before = get_link("D1");
x_link = get_link("D1/x");
if (rename("D1/x", "D1/y") != 0) e(4);
y_link = get_link("D1/y");
D1_after = get_link("D1");
if (D1_before != 2) e(5);
if (D1_after != 2) e(6);
if (x_link != 1) e(7);
if (y_link != 1) e(8);
if (access("D1/y", 7) != 0) e(9);
if (unlink("D1/y") != 0) e(10);
if (rmdir("D1") != 0) e(11);
}

void test21g()
{
int fd, D1_before, D1_after, D2_before, D2_after, x_link, y_link;

/* Test case 2: move a file to a new directory. */
subtest = 7;
if (mkdir("D1", 0777) != 0) e(1);
if (mkdir("D2", 0777) != 0) e(2);
if ( (fd = creat("D1/x", 0777)) < 0) e(3);
if (close(fd) != 0) e(4);
D1_before = get_link("D1");
D2_before = get_link("D2");
x_link = get_link("D1/x");
if (rename("D1/x", "D2/y") != 0) e(5);
y_link = get_link("D2/y");
D1_after = get_link("D1");
D2_after = get_link("D2");
if (D1_before != 2) e(6);
if (D2_before != 2) e(7);
if (D1_after != 2) e(8);
```

```
    if (D2_after != 2) e(9);
    if (x_link != 1) e(10);
    if (y_link != 1) e(11);
    if (access("D2/y", 7) != 0) e(12);
    if (unlink("D2/y") != 0) e(13);
    if (rmdir("D1") != 0) e(14);
    if (rmdir("D2") != 0) e(15);
}

void test21h()
{
    int D1_before, D1_after, x_link, y_link;

    /* Test case 3: renaming a directory within the same directory. */
    subtest = 8;
    if (mkdir("D1", 0777) != 0) e(1);
    if (mkdir("D1/X", 0777) != 0) e(2);
    D1_before = get_link("D1");
    x_link = get_link("D1/X");
    if (rename("D1/X", "D1/Y") != 0) e(3);
    y_link = get_link("D1/Y");
    D1_after = get_link("D1");
    if (D1_before != 3) e(4);
    if (D1_after != 3) e(5);
    if (x_link != 2) e(6);
    if (y_link != 2) e(7);
    if (access("D1/Y", 7) != 0) e(8);
    if (rmdir("D1/Y") != 0) e(9);
    if (get_link("D1") != 2) e(10);
    if (rmdir("D1") != 0) e(11);
}

void test21i()
{
    int D1_before, D1_after, D2_before, D2_after, x_link, y_link;

    /* Test case 4: move a directory to a new directory. */
    subtest = 9;
    if (mkdir("D1", 0777) != 0) e(1);
    if (mkdir("D2", 0777) != 0) e(2);
    if (mkdir("D1/X", 0777) != 0) e(3);
    D1_before = get_link("D1");
    D2_before = get_link("D2");
    x_link = get_link("D1/X");
    if (rename("D1/X", "D2/Y") != 0) e(4);
    y_link = get_link("D2/Y");
    D1_after = get_link("D1");
    D2_after = get_link("D2");
    if (D1_before != 3) e(5);
    if (D2_before != 2) e(6);
    if (D1_after != 2) e(7);
    if (D2_after != 3) e(8);
    if (x_link != 2) e(9);
    if (y_link != 2) e(10);
    if (access("D2/Y", 7) != 0) e(11);
    if (rename("D2/Y", "D1/Z") != 0) e(12);
    if (get_link("D1") != 3) e(13);
    if (get_link("D2") != 2) e(14);
    if (rmdir("D1/Z") != 0) e(15);
    if (get_link("D1") != 2) e(16);
    if (rmdir("D1") != 0) e(17);
    if (rmdir("D2") != 0) e(18);
}

void test21k()
{
    /* Now test the same 4 cases, except when the target exists. */

    int fd, D1_before, D1_after, x_link, y_link;

    /* Test case 5: renaming a file within the same directory. */
    subtest = 10;
    if (mkdir("D1", 0777) != 0) e(1);
    if ( (fd = creat("D1/x", 0777)) < 0) e(2);
```

```
    if (close(fd) != 0) e(3);
    if ( (fd = creat("D1/y", 0777)) < 0) e(3);
    if (close(fd) != 0) e(4);
    D1_before = get_link("D1");
    x_link = get_link("D1/x");
    if (rename("D1/x", "D1/y") != 0) e(5);
    y_link = get_link("D1/y");
    D1_after = get_link("D1");
    if (D1_before != 2) e(6);
    if (D1_after != 2) e(7);
    if (x_link != 1) e(8);
    if (y_link != 1) e(9);
    if (access("D1/y", 7) != 0) e(10);
    if (unlink("D1/y") != 0) e(11);
    if (rmdir("D1") != 0) e(12);
}

void test21l()
{
    int fd, D1_before, D1_after, D2_before, D2_after, x_link, y_link;

    /* Test case 6: move a file to a new directory. */
    subtest = 11;
    if (mkdir("D1", 0777) != 0) e(1);
    if (mkdir("D2", 0777) != 0) e(2);
    if ( (fd = creat("D1/x", 0777)) < 0) e(3);
    if (close(fd) != 0) e(4);
    if ( (fd = creat("D2/y", 0777)) < 0) e(5);
    if (close(fd) != 0) e(6);
    D1_before = get_link("D1");
    D2_before = get_link("D2");
    x_link = get_link("D1/x");
    if (rename("D1/x", "D2/y") != 0) e(7);
    y_link = get_link("D2/y");
    D1_after = get_link("D1");
    D2_after = get_link("D2");
    if (D1_before != 2) e(8);
    if (D2_before != 2) e(9);
    if (D1_after != 2) e(10);
    if (D2_after != 2) e(11);
    if (x_link != 1) e(12);
    if (y_link != 1) e(13);
    if (access("D2/y", 7) != 0) e(14);
    if (unlink("D2/y") != 0) e(15);
    if (rmdir("D1") != 0) e(16);
    if (rmdir("D2") != 0) e(17);
}

void test21m()
{
    int D1_before, D1_after, x_link, y_link;

    /* Test case 7: renaming a directory within the same directory. */
    subtest = 12;
    if (mkdir("D1", 0777) != 0) e(1);
    if (mkdir("D1/X", 0777) != 0) e(2);
    if (mkdir("D1/Y", 0777) != 0) e(3);
    D1_before = get_link("D1");
    x_link = get_link("D1/X");
    if (rename("D1/X", "D1/Y") != 0) e(4);
    y_link = get_link("D1/Y");
    D1_after = get_link("D1");
    if (D1_before != 4) e(5);
    if (D1_after != 3) e(6);
    if (x_link != 2) e(7);
    if (y_link != 2) e(8);
    if (access("D1/Y", 7) != 0) e(9);
    if (rmdir("D1/Y") != 0) e(10);
    if (get_link("D1") != 2) e(11);
    if (rmdir("D1") != 0) e(12);
}

void test21n()
{

```

```

int D1_before, D1_after, D2_before, D2_after, x_link, y_link;

/* Test case 8: move a directory to a new directory. */
subtest = 13;
if (mkdir("D1", 0777) != 0) e(1);
if (mkdir("D2", 0777) != 0) e(2);
if (mkdir("D1/X", 0777) != 0) e(3);
if (mkdir("D2/Y", 0777) != 0) e(4);
D1_before = get_link("D1");
D2_before = get_link("D2");
x_link = get_link("D1/X");
if (rename("D1/X", "D2/Y") != 0) e(5);
y_link = get_link("D2/Y");
D1_after = get_link("D1");
D2_after = get_link("D2");
if (D1_before != 3) e(6);
if (D2_before != 3) e(7);
if (D1_after != 2) e(8);
if (D2_after != 3) e(9);
if (x_link != 2) e(10);
if (y_link != 2) e(11);
if (access("D2/Y", 7) != 0) e(12);
if (rename("D2/Y", "D1/Z") != 0) e(13);
if (get_link("D1") != 3) e(14);
if (get_link("D2") != 2) e(15);
if (rmdir("D1/Z") != 0) e(16);
if (get_link("D1") != 2) e(17);
if (rmdir("D1") != 0) e(18);
if (rmdir("D2") != 0) e(19);
}

void test21o()
{
    /* Test trying to remove . and .. */
    subtest = 14;
    if (mkdir("D1", 0777) != 0) e(1);
    if (chdir("D1") != 0) e(2);
    if (rmdir(".") == 0) e(3);
    if (rmdir("..") == 0) e(4);
    if (mkdir("D2", 0777) != 0) e(5);
    if (mkdir("D3", 0777) != 0) e(6);
    if (mkdir("D4", 0777) != 0) e(7);
    if (rmdir("D2../D3../D4") != 0) e(8);           /* legal way to remove D4 */
    if (rmdir("D2../D3../D2../") == 0) e(9);       /* removing self is illegal */
    if (rmdir("D2../D3../D2../..") == 0) e(10);    /* removing parent is illegal */
    if (rmdir("../D1../D1/D3") != 0) e(11);       /* legal way to remove D3 */
    if (rmdir("../D2../D2") != 0) e(12);          /* legal way to remove D2 */
    if (chdir("..") != 0) e(13);
    if (rmdir("D1") != 0) e(14);
}

int get_link(name)
char *name;
{
    struct stat statbuf;

    if (stat(name, &statbuf) != 0) {
        printf("Unable to stat %s\n", name);
        errct++;
        return(-1);
    }
    return(statbuf.st_nlink);
}

void e(n)
int n;
{
    int err_num = errno;           /* save errno in case printf clobbers it */

    printf("Subtest %d, error %d errno=%d ", subtest, n, errno);
    errno = err_num;              /* restore errno, just in case */
    perror("");
    if (errct++ > MAX_ERROR) {
        printf("Too many errors; test aborted\n");
    }
}

```

```
        chdir("../");
        system("rm -rf DIR*");
        exit(1);
    }
}

void quit()
{
    chdir("../");
    system("rm -rf DIR*");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
        printf("%d errors\n", errct);
        exit(1);
    }
}
```

```

/* test22: umask()                                (p) Jan-Mark Wams. email: jms@cs.vu.nl */

#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <stdio.h>

#define MAX_ERROR 4                                /* Stop after ``MAX_ERROR`` errors. */
#define ITERATIONS 2

#define System(cmd)    if (system(cmd) != 0) printf("``%s`` failed\n", cmd)
#define Chdir(dir)     if (chdir(dir) != 0) printf("Can't goto %s\n", dir)
#define Stat(a,b)      if (stat(a,b) != 0) printf("Can't stat %s\n", a)

int errct = 0;                                     /* Total error counter. */
int subtest = 1;

_PROTOTYPE(void main, (int argc, char *argv[]));
_PROTOTYPE(void test22a, (void));
_PROTOTYPE(int mode, (char *filename));
_PROTOTYPE(int umode, (char *filename));
_PROTOTYPE(void e, (int number));
_PROTOTYPE(void quit, (void));

void main(argc, argv)
int argc;
char *argv[];
{
    int i, m = 0xFFFF;

    sync();
    if (argc == 2) m = atoi(argv[1]);
    printf("Test 22 ");
    fflush(stdout);
    system("chmod 777 DIR_22/* DIR_22/*/* > /dev/null 2>&1");
    System("rm -rf DIR_22; mkdir DIR_22");
    Chdir("DIR_22");

    for (i = 0; i < ITERATIONS; i++) {
        if (m & 0001) test22a();
    }

    quit();
}

void test22a()
{
    int fd1, fd2;
    int i, oldmask;
    int stat_loc;                                /* For the wait sys call. */

    subtest = 1;

    system("chmod 777 ../DIR_22/* ../DIR_22/*/* > /dev/null 2>&1");
    System("rm -rf ../DIR_22/*");

    oldmask = 0123;                                /* Set oldmask and umask. */
    umask(oldmask);                                /* Set oldmask and umask. */

    /* Check all the possible values of umask. */
    for (i = 0000; i <= 0777; i++) {
        if (oldmask != umask(i)) e(1); /* set umask() */
        fd1 = open("open", O_CREAT, 0777);
        if (fd1 != 3) e(2); /* test open(), */
        fd2 = creat("creat", 0777);
        if (fd2 != 4) e(3); /* creat(), */
        if (mkdir("dir", 0777) != 0) e(4); /* mkdir(), */
        if (mkfifo("fifo", 0777) != 0) e(5); /* and mkfifo(). */

        if (umode("open") != i) e(6); /* see if they have */
    }
}

```



```
    if (umode("creat") != i) e(7);    /* the proper mode */
    if (umode("dir") != i) e(8);
    if (umode("fifo") != i) e(9);

    /* Clean up */
    if (close(fd1) != 0) e(10);
    if (close(fd2) != 0) e(11);    /* close fd's and */
    unlink("open");    /* clean the dir */
    unlink("creat");
    rmdir("dir");
    unlink("fifo");
    oldmask = i;    /* save current mask */
}

/* Check-reset mask */
if (umask(0124) != 0777) e(12);

/* Check if a umask of 0000 leaves the modes alone. */
if (umask(0000) != 0124) e(13);
for (i = 0000; i <= 0777; i++) {
    fd1 = open("open", O_CREAT, i);
    if (fd1 != 3) e(14);    /* test open(), */
    fd2 = creat("creat", i);
    if (fd2 != 4) e(15);    /* creat(), */
    if (mkdir("dir", i) != 0) e(16);    /* mkdir(), */
    if (mkfifo("fifo", i) != 0) e(17);    /* and mkfifo(). */

    if (mode("open") != i) e(18);    /* see if they have */
    if (mode("creat") != i) e(19);    /* the proper mode */
    if (mode("dir") != i) e(20);
    if (mode("fifo") != i) e(21);

    /* Clean up */
    if (close(fd1) != 0) e(22);
    if (close(fd2) != 0) e(23);
    if (unlink("open") != 0) e(24);
    unlink("creat");
    rmdir("dir");
    unlink("fifo");
}

/* Check if umask survives a fork() */
if (umask(0124) != 0000) e(25);
switch (fork()) {
    case -1: fprintf(stderr, "Can't fork\n");    break;
    case 0:
        mkdir("bar", 0777);    /* child makes a dir */
        exit(0);
    default:
        if (wait(&stat_loc) == -1) e(26);
}
if (umode("bar") != 0124) e(27);
rmdir("bar");

/* Check if umask in child changes umask in parent. */
switch (fork()) {
    case -1: fprintf(stderr, "Can't fork\n");    break;
    case 0:
        switch (fork()) {
            case -1:
                fprintf(stderr, "Can't fork\n");
                break;
            case 0:
                if (umask(0432) != 0124) e(28);
                exit(0);
            default:
                if (wait(&stat_loc) == -1) e(29);
        }
        if (umask(0423) != 0124) e(30);
        exit(0);
    default:
        if (wait(&stat_loc) == -1) e(31);
}
if (umask(0342) != 0124) e(32);
```

```
/* See if extra bits are ignored */
if (umask(0xFFFF) != 0342) e(33);
if (umask(0xFE00) != 0777) e(34);
if (umask(01777) != 0000) e(35);
if (umask(0022) != 0777) e(36);
}

int mode(arg)
char *arg;
{
    /* return the file mode. */
    struct stat st;
    Stat(arg, &st);
    return st.st_mode & 0777;
}

int umode(arg)
char *arg;
{
    /* return the umask used for this file */
    return 0777 ^ mode(arg);
}

void e(n)
int n;
{
    int err_num = errno;    /* Save in case printf clobbers it. */

    printf("Subtest %d, error %d errno=%d: ", subtest, n, errno);
    errno = err_num;
    perror("");
    if (errct++ > MAX_ERROR) {
        printf("Too many errors; test aborted\n");
        chdir("..");
        system("rm -rf DIR*");
        exit(1);
    }
    errno = 0;
}

void quit()
{
    Chdir("..");
    system("chmod 777 ../DIR_22/* ../DIR_22/*/* > /dev/null 2>&1");
    System("rm -rf DIR_22");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
        printf("%d errors\n", errct);
        exit(1);
    }
}
```

```

/* test23: chdir(), getcwd()      Author: Jan-Mark Wams (jms@cs.vu.nl) */

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/dir.h>
#include <limits.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <stdio.h>

#define MAX_ERROR 4
#define ITERATIONS 3

#define System(cmd)      if (system(cmd) != 0) printf("'%s' failed\n", cmd)
#define Chdir(dir)      if (chdir(dir) != 0) printf("Can't goto %s\n", dir)

int errct;
int subtest;
int superuser;                /* True if we are root. */

char cwd[PATH_MAX];           /* Space for path names. */
char cwd2[PATH_MAX];
char buf[PATH_MAX];
char MaxName[NAME_MAX + 1];   /* Name of maximum length */
char MaxPath[PATH_MAX];       /* Same for path */
char ToLongName[NAME_MAX + 2]; /* Name of maximum +1 length */
char ToLongPath[PATH_MAX + 1]; /* Same for path, both too long */

_PROTOTYPE(void main, (int argc, char *argv[]));
_PROTOTYPE(void test23a, (void));
_PROTOTYPE(void test23b, (void));
_PROTOTYPE(void test23c, (void));
_PROTOTYPE(void makelongnames, (void)); /* Fill MaxName etc. */
_PROTOTYPE(char *last_index, (char *string, int ch));
_PROTOTYPE(char *my_getcwd, (char *buf, int size));
_PROTOTYPE(void e, (int number));
_PROTOTYPE(void quit, (void));

void main(argc, argv)
int argc;
char *argv[];
{
    int i, m = 0xFFFF;

    sync();
    if (argc == 2) m = atoi(argv[1]);
    printf("Test 23 ");
    fflush(stdout);
    System("rm -rf DIR_23; mkdir DIR_23");
    Chdir("DIR_23");
    makelongnames();
    superuser = (geteuid() == 0);

    for (i = 0; i < ITERATIONS; i++) {
        if (m & 0001) test23a();           /* Test normal operation */
        if (m & 0002) test23b();           /* Test critical operation */
        if (m & 0004) test23c();           /* Test error operation */
    }

    quit();
}

void test23a()
{
    register int i;                /* Test normal operation. */

    subtest = 1;

    System("rm -rf ../DIR_23/*");

    /* Let's do some fiddeling with path names. */

```

```

if (getcwd(cwd, PATH_MAX) != cwd) e(1);
if (chdir(cwd) != 0) e(2);
if (getcwd(buf, PATH_MAX) != buf) e(3);
if (strcmp(buf, cwd) != 0) e(4);
if (chdir(".") != 0) e(5);
if (getcwd(buf, PATH_MAX) != buf) e(6);
if (strcmp(buf, cwd) != 0) e(7);
if (chdir("../.") != 0) e(8);
if (getcwd(buf, PATH_MAX) != buf) e(9);
if (strcmp(buf, cwd) != 0) e(10);

/* Creat a working dir named "foo", remove any previous residues. */
System("rm -rf foo");
if (mkdir("foo", 0777) != 0) e(11);

/* Do some more fiddeling with path names. */
if (chdir("foo../foo..") != 0) e(12);          /* change to cwd */
if (getcwd(buf, PATH_MAX) != buf) e(13);
if (strcmp(buf, cwd) != 0) e(13);
if (chdir("foo") != 0) e(14); /* change to foo */
if (chdir("..") != 0) e(15); /* and back again */
if (getcwd(buf, PATH_MAX) != buf) e(16);
if (strcmp(buf, cwd) != 0) e(17);

/* Make 30 sub dirs, eg. ./bar/bar/bar/bar/bar..... */
System("rm -rf bar");          /* get ridd of bar */
for (i = 0; i < 30; i++) {
    if (mkdir("bar", 0777) != 0) e(18);
    if (chdir("bar") != 0) e(19); /* change to bar */
}
for (i = 0; i < 30; i++) {
    if (chdir("..") != 0) e(20); /* and back again */
    if (rmdir("bar") != 0) e(21);
}

/* Make sure we are back where we started. */
if (getcwd(buf, PATH_MAX) != buf) e(22);
if (strcmp(buf, cwd) != 0) e(23);
System("rm -rf bar");          /* just incase */

/* Do some normal checks on 'Chdir()' and 'getcwd()' */
if (chdir("/") != 0) e(24);
if (getcwd(buf, PATH_MAX) != buf) e(25);
if (strcmp(buf, "/") != 0) e(26);
if (chdir("..") != 0) e(27); /* go to parent of / */
if (getcwd(buf, PATH_MAX) != buf) e(28);
if (strcmp(buf, "/") != 0) e(29);
if (chdir(cwd) != 0) e(30);
if (getcwd(buf, PATH_MAX) != buf) e(31);
if (strcmp(buf, cwd) != 0) e(32);
if (chdir("/etc") != 0) e(33); /* /etc might be on RAM */
if (getcwd(buf, PATH_MAX) != buf) e(34); /* might make a difference */
if (strcmp(buf, "/etc") != 0) e(35);
if (chdir(cwd) != 0) e(36);
if (getcwd(buf, PATH_MAX) != buf) e(37);
if (strcmp(buf, cwd) != 0) e(38);
if (chdir("../..") != 0) e(39); /* ../.. == current dir */
if (getcwd(buf, PATH_MAX) != buf) e(40);
if (strcmp(buf, cwd) != 0) e(41); /* we might be at '/' */
#ifdef _MINIX
/* XXX - my_getcwd() is old rubbish. It reads the directory directly instead
 * of through the directory library. It uses a fixed size buffer instead of
 * a size related to PATH_MAX, NAME_MAX or the size required.
 */
if (my_getcwd(buf, PATH_MAX) != buf) e(42); /* get cwd my way */
if (strcmp(cwd, buf) != 0) e(43);
#endif
System("rm -rf foo");
}

void test23b()
{
    subtest = 2;          /* Test critical values. */
}

```

```
System("rm -rf ../DIR_23/*");

/* Fiddle with the size (2nt) parameter of 'getcwd()'. */
if (getcwd(cwd, PATH_MAX) != cwd) e(1);          /* get cwd */
if (getcwd(buf, strlen(cwd)) != (char *) 0) e(2); /* size 1 to small */
if (errno != ERANGE) e(3);
if (getcwd(buf, PATH_MAX) != buf) e(4);
if (strcmp(buf, cwd) != 0) e(5);
Chdir(cwd);                                       /* getcwd might cd / */
if (getcwd(buf, strlen(cwd) + 1) != buf) e(6);   /* size just ok */
if (getcwd(buf, PATH_MAX) != buf) e(7);
if (strcmp(buf, cwd) != 0) e(8);

/* Let's see how "MaxName" and "ToLongName" are handled. */
if (mkdir(MaxName, 0777) != 0) e(9);
if (chdir(MaxName) != 0) e(10);
if (chdir("..") != 0) e(11);
if (rmdir(MaxName) != 0) e(12);
if (getcwd(buf, PATH_MAX) != buf) e(13);
if (strcmp(buf, cwd) != 0) e(14);
if (chdir(MaxPath) != 0) e(15);
if (getcwd(buf, PATH_MAX) != buf) e(16);
if (strcmp(buf, cwd) != 0) e(17);

if (chdir(ToLongName) != -1) e(18);
#ifdef _POSIX_NO_TRUNC
# if _POSIX_NO_TRUNC - 0 != -1
#  if (errno != ENAMETOOLONG) e(20);
# else
#  if (errno != ENOENT) e(20);
# endif
#else
# include "error, this case requires dynamic checks and is not handled"
#endif

if (getcwd(buf, PATH_MAX) != buf) e(21);
if (strcmp(buf, cwd) != 0) e(22);
if (chdir(ToLongPath) != -1) e(23);
if (errno != ENAMETOOLONG) e(24);
if (getcwd(buf, PATH_MAX) != buf) e(25);
if (strcmp(buf, cwd) != 0) e(26);
}

void test23c()
{
    /* Check reaction to errors */
    subtest = 3;

    System("rm -rf ../DIR_23/*");

    if (getcwd(cwd, PATH_MAX) != cwd) e(1);          /* get cwd */

    /* Creat a working dir named "foo", remove any previous residues. */
    System("rm -rf foo; mkdir foo");

    /* Check some obviouse errors. */
    if (chdir("") != -1) e(2);
    if (errno != ENOENT) e(3);
    if (getcwd(buf, PATH_MAX) != buf) e(4);
    if (strcmp(buf, cwd) != 0) e(5);
    if (getcwd(buf, 0) != (char *) 0) e(6);
    if (errno != EINVAL) e(7);
    if (getcwd(buf, PATH_MAX) != buf) e(8);
    if (strcmp(buf, cwd) != 0) e(9);
    if (getcwd(buf, 0) != (char *) 0) e(10);
    if (errno != EINVAL) e(11);
    if (getcwd(buf, PATH_MAX) != buf) e(12);
    if (strcmp(buf, cwd) != 0) e(13);
    if (chdir(cwd) != 0) e(14); /* getcwd might be buggy. */

    /* Change the mode of foo, and check the effect. */
    if (chdir("foo") != 0) e(15); /* change to foo */
    if (mkdir("bar", 0777) != 0) e(16); /* make a new dir bar */
    if (getcwd(cwd2, PATH_MAX) != cwd2) e(17); /* get the new cwd */
    if (getcwd(buf, 3) != (char *) 0) e(18); /* size is too small */
}
```

```

if (errno != ERANGE) e(19);
if (getcwd(buf, PATH_MAX) != buf) e(20);
if (strcmp(buf, cwd2) != 0) e(21);
Chdir(cwd2); /* getcwd() might cd */
System("chmod 377."); /* make foo unreadable */
if (getcwd(buf, PATH_MAX) != buf) e(22); /* dir not readable */
if (getcwd(buf, PATH_MAX) != buf) e(23);
if (strcmp(buf, cwd2) != 0) e(24);
if (chdir("bar") != 0) e(25); /* at ../foo/bar */
if (!superuser) {
    if (getcwd(buf, PATH_MAX) != (char *) 0) e(26);
    if (errno != EACCES) e(27);
}
if (superuser) {
    if (getcwd(buf, PATH_MAX) != buf) e(28);
}
if (chdir(cwd2) != 0) e(29);
if (getcwd(buf, PATH_MAX) != buf) e(30);
if (strcmp(buf, cwd2) != 0) e(31);
System("chmod 677."); /* make foo inaccessible */
if (!superuser) {
    if (getcwd(buf, PATH_MAX) != (char *) 0) e(32); /* try to get cwd */
    if (errno != EACCES) e(33); /* but no access */
    if (chdir("..") != -1) e(34); /* try to get back */
    if (errno != EACCES) e(35); /* again no access */
    if (chdir(cwd) != 0) e(36); /* get back to cwd */
    /* 'Chdir()' might do path optimizing, it shouldn't. */
    if (chdir("foo/..") != -1) e(37); /* no op */
    if (chdir("foo") != -1) e(38); /* try to cd to foo */
    if (errno != EACCES) e(39); /* no have access */
    if (getcwd(buf, PATH_MAX) != buf) e(40);
    if (strcmp(buf, cwd) != 0) e(41);
}
if (superuser) {
    if (getcwd(buf, PATH_MAX) != buf) e(42);
    if (strcmp(buf, cwd2) != 0) e(43);
    if (chdir("..") != 0) e(44); /* get back to cwd */
    if (chdir("foo") != 0) e(45); /* get back to foo */
    if (chdir(cwd) != 0) e(46); /* get back to cwd */
}
if (getcwd(buf, PATH_MAX) != buf) e(47); /* check we are */
if (strcmp(buf, cwd) != 0) e(48); /* back at cwd. */
Chdir(cwd); /* just in case... */

if (chdir("/etc/passwd") != -1) e(49); /* try to change to a file */
if (errno != ENOTDIR) e(50);
if (getcwd(buf, PATH_MAX) != buf) e(51);
if (strcmp(buf, cwd) != 0) e(52);
if (chdir("/notexist") != -1) e(53);
if (errno != ENOENT) e(54);
if (getcwd(buf, PATH_MAX) != buf) e(55);
if (strcmp(buf, cwd) != 0) e(56);
System("chmod 777 foo");
if (chdir("foo") != 0) e(57);

/* XXX - this comment was botched by 'pretty'. */
/* * Since 'foo' is the cwd, it should not be removeable but * if it
 * were, this code would be found here; *
 *
 * System("cd .. ; rm -rf foo"); remove foo * if (chdir("."))
 * != -1) e(); try go to. * if (errno != ENOENT) e();
 * hould not be an entry * if (chdir("..") != -1) e(); try
 * to get back * if (errno != ENOENT) e(); should not be
 * an entry * if (getcwd(buf, PATH_MAX) != (char *)0) e(); don't
 * know where we are *
 *
 * What should errno be now ? The cwd might be gone if te superuser *
 * removed the cwd. (Might even have linked it first.) But this *
 * testing should be done by the test program for 'rmdir()'. */
if (chdir(cwd) != 0) e(58);
}

void makelongnames()
{

```

```
register int i;

memset(MaxName, 'a', NAME_MAX);
MaxName[NAME_MAX] = '\0';
for (i = 0; i < PATH_MAX - 1; i++) { /* idem path */
    MaxPath[i++] = '.';
    MaxPath[i] = '/';
}
MaxPath[PATH_MAX - 1] = '\0';

strcpy(ToLongName, MaxName); /* copy them Max to ToLong */
strcpy(ToLongPath, MaxPath);

ToLongName[NAME_MAX] = 'a';
ToLongName[NAME_MAX + 1] = '\0'; /* extend ToLongName by one too many */
ToLongPath[PATH_MAX - 1] = '/';
ToLongPath[PATH_MAX] = '\0'; /* inc ToLongPath by one */
}

/* The following code, is take from pwd written by Adri Koppes */

/* My_getcwd() helper. */
char *last_index(string, ch)
char *string;
char ch;
{
    register char *retval = '\0';

    while (*string != '\0') {
        if (*string == ch) retval = string;
        string++;
    }
    return(retval);
}

char *my_getcwd(buf, size)
char *buf;
int size;
{
    /* should be like getcwd() */
    int sd;
    register int fd;
    register char *n;
    char name[128];
    struct stat s, st, sk;
    struct direct d;

    if (size <= 0) return(char *) 0;

    *buf = '\0';
    *name = '\0';
    stat(".", &s);
    do {
        if ((fd = open(".", O_RDONLY)) < 0) return(char *) 0;
        st.st_dev = s.st_dev;
        st.st_ino = s.st_ino;
        stat(".", &s);
        Chdir(".");
        sd = sizeof(struct direct);
        if (s.st_dev == st.st_dev) {
            do {
                if (read(fd, (char *) &d, sd) < sd) return(char *) 0;
            } while (d.d_ino != st.st_ino);
        } else {
            do {
                if (read(fd, (char *) &d, sd) < sd) return(char *) 0;
                stat(d.d_name, &sk);
            } while ((sk.st_dev != st.st_dev) ||
                (sk.st_ino != st.st_ino));
        }
        close(fd);
        if (strcmp(".", d.d_name) != 0) {
            strcat(name, "/");
            strcat(name, d.d_name);
        }
    }
}
```

```
} while ((s.st_ino != st.st_ino) || (s.st_dev != st.st_dev));
if (*name == '\0')
    strncat(buf, "/", size);
else
    while ((n = last_index(name, '/')) != NULL) {
        n[NAME_MAX] = '\0';
        strncat(buf, n, size - strlen(buf));
        *n = '\0';
    }
strncat(buf, name, size - strlen(buf));
buf[size - 1] = '\0';
Chdir(buf);                /* get back there */
return buf;
}

void e(n)
int n;
{
    int err_num = errno;    /* Save in case printf clobbers it. */

    printf("Subtest %d, error %d errno=%d: ", subtest, n, errno);
    errno = err_num;
    perror("");
    if (errct++ > MAX_ERROR) {
        printf("Too many errors; test aborted\n");
        chdir("..");
        system("rm -rf DIR*");
        exit(1);
    }
    errno = 0;
}

void quit()
{
    Chdir("..");
    System("rm -rf DIR_23");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
        printf("%d errors\n", errct);
        exit(1);
    }
}
```



```

/* Test24: opendir, readdir, rewinddir, closedir                                Author: Jan-Mark Wams */

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <limits.h>
#include <fcntl.h>
#include <dirent.h>
#include <errno.h>
#include <time.h>
#include <stdio.h>

_PROTOTYPE(void main, (int argc, char *argv[]));
_PROTOTYPE(void chk_dir, (DIR * dirpnr));
_PROTOTYPE(void test24a, (void));
_PROTOTYPE(void test24b, (void));
_PROTOTYPE(void test24c, (void));
_PROTOTYPE(void makelongnames, (void));
_PROTOTYPE(void e, (int number));
_PROTOTYPE(void quit, (void));

#define OVERFLOW_DIR_NR (OPEN_MAX + 1)
#define MAX_ERROR      4
#define ITERATIONS 5

#define DIRENT0 ((struct dirent *) NULL)
#define System(cmd)      if (system(cmd) != 0) printf("“%s” failed\n", cmd)
#define Chdir(dir)       if (chdir(dir) != 0) printf("Can't goto %s\n", dir)

int errct = 0;
int subtest = 1;
int superuser;

char MaxName[NAME_MAX + 1]; /* Name of maximum length */
char MaxPath[PATH_MAX];    /* Same for path */
char ToLongName[NAME_MAX + 2]; /* Name of maximum +1 length */
char ToLongPath[PATH_MAX + 1]; /* Same for path, both too long */

void main(argc, argv)
int argc;
char *argv[];
{
    int i, m = 0xFFFF;

    sync();
    if (argc == 2) m = atoi(argv[1]);
    printf("Test 24 ");
    fflush(stdout);
    System("rm -rf DIR_24; mkdir DIR_24");
    Chdir("DIR_24");
    makelongnames();
    superuser = (geteuid() == 0);

    for (i = 0; i < ITERATIONS; i++) {
        if (m & 0001) test24a();
        if (m & 0002) test24b();
        if (m & 0004) test24c();
    }
    quit();
}

void test24a()
{
    /* Test normal operations. */
    int fd3, fd4, fd5;
    DIR *dirp;
    int j, ret, fd, flags;
    struct stat st1, st2;
    int stat_loc;
    time_t timel;

    subtest = 1;

```

```

System("rm -rf ../DIR_24/*");

if ((fd = dup(0)) != 3) e(1); /* dup stdin */
close(fd);                  /* free the fd again */
dirp = opendir("/");        /* open "/" */
if (dirp == ((DIR *) NULL)) e(2); /* has to succseed */
if ((fd = dup(0)) <= 2) e(3); /* dup stdin */
if (fd > 3) {                /* if opendir() uses fd 3 */
    flags = fcntl(3, F_GETFD); /* get fd fags of 3 */
    if (!(flags & FD_CLOEXEC)) e(4); /* it should be closed on */
}
close(fd);                  /* exec..() calls */
ret = closedir(dirp);       /* close, we don't need it */
if (ret == -1) e(5);        /* closedir() unsucces full */
if (ret != 0) e(6);         /* should be 0 or -1 */
if ((fd = dup(0)) != 3) e(7); /* see if next fd is same */
close(fd);                  /* free the fd again */

System("rm -rf foo; mkdir foo");
Chdir("foo");
System("touch f1 f2 f3 f4 f5"); /* make f1 .. f5 */
System("rm f[24]");           /* creat 'holes' in entrys */
Chdir("..");

if ((dirp = opendir("foo")) == ((DIR *) NULL)) e(8); /* open foo */
chk_dir(dirp); /* test if foo's ok */
for (j = 0; j < 10; j++) {
    errno = j * 47 % 7; /* there should */
    if (readdir(dirp) != DIRENT0) e(9); /* be nomore dir */
    if (errno != j * 47 % 7) e(10); /* entrys */
}
rewinddir(dirp); /* rewind foo */
chk_dir(dirp); /* test foosok */
for (j = 0; j < 10; j++) {
    errno = j * 23 % 7; /* there should */
    if (readdir(dirp) != DIRENT0) e(11); /* be nomore dir */
    if (errno != j * 23 % 7) e(12); /* entrys */
}
if ((fd4 = creat("foo/f4", 0666)) <= 2) e(13); /* Open a file. */
System("rm foo/f4"); /* Kill entry. */
rewinddir(dirp); /* Rewind foo. */
if ((fd3 = open("foo/f3", O_WRONLY)) <= 2) e(14); /* Open more files. */
if ((fd5 = open("foo/f5", O_WRONLY)) <= 2) e(15);
if (write(fd3, "Hello", 6) != 6) e(16);
if (write(fd4, "Hello", 6) != 6) e(17); /* write some data */
if (close(fd5) != 0) e(18);
chk_dir(dirp);
for (j = 0; j < 10; j++) {
    errno = j * 101 % 7; /* there should */
    if (readdir(dirp) != DIRENT0) e(19); /* be nomore dir */
    if (errno != j * 101 % 7) e(20); /* entrys */
}
if (close(fd4) != 0) e(21); /* shouldn't matter */
if (close(fd3) != 0) e(22); /* when we do this */
if (closedir(dirp) != 0) e(23); /* close foo again */

Chdir("foo");
if ((dirp = opendir("./")) == ((DIR *) NULL)) e(24); /* open foo again */
Chdir("..");
chk_dir(dirp); /* foosok? */
for (j = 0; j < 10; j++) {
    errno = (j * 101) % 7; /* there should */
    if (readdir(dirp) != DIRENT0) e(25); /* be nomore dir */
    if (errno != (j * 101) % 7) e(26); /* entrys */
}

if (closedir(dirp) != 0) e(27); /* It should be closable */

stat("foo", &st1); /* get stat */
time(&timel);
while (timel >= time((time_t *)0))
    ;
if ((dirp = opendir("foo")) == ((DIR *) NULL)) e(28); /* open, */

```

```

if (readdir(dirp) == DIRENT0) e(29); /* read and */
stat("foo", &st2); /* get new stat */
if (st1.st_atime > st2.st_atime) e(30); /* st_atime check */

switch (fork()) {
    case -1: printf("Can't fork\n"); break;
    case 0:
        rewinddir(dirp); /* rewind childs dirp */
        if (readdir(dirp) == DIRENT0) e(31); /* read should be ok */
        if (closedir(dirp) != 0) e(32); /* close child'd foo */
        exit(0); /* 0 stops here */
    default:
        if (wait(&stat_loc) == -1) e(33); /* PARENT wait()'s */
        break;
}
if (closedir(dirp) != 0) e(34); /* close parent's foo */
}

void test24b()
{
    /* See what happens with too many dir's open. Check if file size seems ok,
    * and independency.
    */

    int i, j; /* i = highest open dir count */
    DIR *dirp[OVERFLOW_DIR_NR], *dp;
    struct dirent *dep, *dep1, *dep2;
    char name[NAME_MAX + 2]; /* buffer for file name, and count */
    int dot = 0, dotdot = 0;

    subtest = 2;

    System("rm -rf ../DIR_24/*");

    for (i = 0; i < OVERFLOW_DIR_NR; i++) {
        dirp[i] = opendir("/");
        if (dirp[i] == ((DIR *) NULL)) {
            if (errno != EMFILE) e(1);
            break;
        }
    }
    if (i <= 4) e(2); /* sounds resanable */
    if (i >= OVERFLOW_DIR_NR) e(3); /* might be to small */
    for (j = 0; j < i; j++) {
        if (closedir(dirp[(j + 5) % i]) != 0) e(4); /* neat! */
    }

    /* Now check if number of bytes in d_name can go up till NAME_MAX */
    System("rm -rf foo; mkdir foo");
    Chdir("foo");
    name[0] = 0;
    for (i = 0; i <= NAME_MAX; i++) {
        if (strcat(name, "X") != name) e(5);
        close(creat(name, 0666)); /* fails once on */
    } /* XX..XX, 1 too long */
    Chdir("..");
    /* Now change i-th X to Y in name buffer record file of length i. */
    if ((dp = opendir("foo")) == ((DIR *) NULL)) e(6);
    while ((dep = readdir(dp)) != DIRENT0) {
        if (strcmp("..", dep->d_name) == 0)
            dotdot++;
        else if (strcmp(".", dep->d_name) == 0)
            dot++;
        else
            name[strlen(dep->d_name)] += 1; /* 'X' + 1 == 'Y' */
    }
    if (closedir(dp) != 0) e(7);
    for (i = 1; i <= NAME_MAX; i++) { /* Check if every length */
        if (name[i] != 'Y') e(8); /* has been seen once. */
    }

    /* Check upper and lower bound. */
    if (name[0] != 'X') e(9);
    if (name[NAME_MAX + 1] != '\0') e(10);
}

```

```

/* Now check if two simultaneous open dirs do the same */
if ((dirp[1] = opendir("foo")) == ((DIR *) NULL)) e(11);
if ((dirp[2] = opendir("foo")) == ((DIR *) NULL)) e(12);
if ((dep1 = readdir(dirp[1])) == DIRENT0) e(13);
if ((dep2 = readdir(dirp[2])) == DIRENT0) e(14);
if (dep1->d_name == dep2->d_name) e(15); /* 1 & 2 Should be */
strcpy(name, dep2->d_name); /* differand buffers */
if (strcmp(dep1->d_name, name) != 0) e(16); /* But hold the same */
if ((dep1 = readdir(dirp[1])) == DIRENT0) e(17);
if ((dep1 = readdir(dirp[1])) == DIRENT0) e(18); /* lose some entries */
if ((dep1 = readdir(dirp[1])) == DIRENT0) e(19); /* Using dirp 1 has */
if (dep1->d_name == dep2->d_name) e(20); /* no effect on 2 */
if (strcmp(dep2->d_name, name) != 0) e(21);
rewinddir(dirp[1]); /* Rewinding dirp 1 */
if ((dep2 = readdir(dirp[2])) == DIRENT0) e(22); /* can't effect 2 */
if (strcmp(dep2->d_name, name) == 0) e(23); /* Must be next */
if (closedir(dirp[1]) != 0) e(24); /* Closing dirp 1 */
if ((dep2 = readdir(dirp[2])) == DIRENT0) e(25); /* can't effect 2 */
if (strcmp(dep2->d_name, name) == 0) e(26); /* Must be next */
if (closedir(dirp[2]) != 0) e(27);
}

void test24c()
{
/* Test whether wrong things go wrong right. */

DIR *dirp;

subtest = 3;

System("rm -rf ../DIR_24/*");

if (opendir("foo/bar/nono") != ((DIR *) NULL)) e(1); /* nonexistent */
if (errno != ENOENT) e(2);
System("mkdir foo; chmod 677 foo"); /* foo inaccesable */
if (opendir("foo/bar/nono") != ((DIR *) NULL)) e(3);
if (superuser) {
    if (errno != ENOENT) e(4); /* su has access */
    System("chmod 377 foo");
    if ((dirp = opendir("foo")) == ((DIR *) NULL)) e(5);
    if (closedir(dirp) != 0) e(6);
}
if (!superuser) {
    if (errno != EACCES) e(7); /* we don't ;- ) */
    System("chmod 377 foo");
    if (opendir("foo") != ((DIR *) NULL)) e(8);
}
System("chmod 777 foo");

if (mkdir(MaxName, 0777) != 0) e(9); /* make longdir */
if ((dirp = opendir(MaxName)) == ((DIR *) NULL)) e(10); /* open it */
if (closedir(dirp) != 0) e(11); /* close it */
if (rmdir(MaxName) != 0) e(12); /* then remove it */
if ((dirp = opendir(MaxPath)) == ((DIR *) NULL)) e(13); /* open '.' */
if (closedir(dirp) != 0) e(14); /* close it */
#ifdef XXX /* XXX - anything could happen with the bad pointer */
if (closedir(dirp) != -1) e(15); /* close it again */
if (closedir(dirp) != -1) e(16); /* and again */
#endif /* 0 */
if (opendir(ToLongName) != ((DIR *) NULL)) e(17); /* is too long */
#ifdef _POSIX_NO_TRUNC
# if _POSIX_NO_TRUNC - 0 != -1
if (errno != ENAMETOOLONG) e(18);
# else
if (errno != ENOENT) e(19);
# endif
#else
# include "error, this case requires dynamic checks and is not handled"
#endif
if (opendir(ToLongPath) != ((DIR *) NULL)) e(20); /* path is too long */
if (errno != ENAMETOOLONG) e(21);
System("touch foo/abc"); /* make a file */
if (opendir("foo/abc") != ((DIR *) NULL)) e(22); /* not a dir */

```

```

    if (errno != ENOTDIR) e(23);
}

void chk_dir(dirp)                /* dir should contain */
DIR *dirp;                       /* ('f1', 'f3', 'f5', '.', '..') */
{                                /* no more, no less */
    int f1 = 0, f2 = 0, f3 = 0, f4 = 0, f5 = 0, /* counters for all */
        other = 0, dot = 0, dotdot = 0; /* possible entrys */
    int i;
    struct dirent *dep;
    char *fname;
    int oldsubtest = subtest;

    subtest = 4;

    for (i = 0; i < 5; i++) {     /* 3 files and '.' and '..' == 5 entrys */
        dep = readdir(dirp);
        if (dep == DIRENT0) {     /* not einough */
            if (dep == DIRENT0) e(1);
            break;
        }
        fname = dep->d_name;
        if (strcmp(fname, ".") == 0)
            dot++;
        else if (strcmp(fname, "..") == 0)
            dotdot++;
        else if (strcmp(fname, "f1") == 0)
            f1++;
        else if (strcmp(fname, "f2") == 0)
            f2++;
        else if (strcmp(fname, "f3") == 0)
            f3++;
        else if (strcmp(fname, "f4") == 0)
            f4++;
        else if (strcmp(fname, "f5") == 0)
            f5++;
        else
            other++;
    }                               /* do next dir entry */

    if (dot != 1) e(2);           /* Check the entrys */
    if (dotdot != 1) e(3);
    if (f1 != 1) e(4);
    if (f3 != 1) e(5);
    if (f5 != 1) e(6);
    if (f2 != 0) e(7);
    if (f4 != 0) e(8);
    if (other != 0) e(9);

    subtest = oldsubtest;
}

void makelongnames()
{
    register int i;

    memset(MaxName, 'a', NAME_MAX);
    MaxName[NAME_MAX] = '\0';
    for (i = 0; i < PATH_MAX - 1; i++) { /* idem path */
        MaxPath[i++] = '.';
        MaxPath[i] = '/';
    }
    MaxPath[PATH_MAX - 1] = '\0';

    strcpy(ToLongName, MaxName); /* copy them Max to ToLong */
    strcpy(ToLongPath, MaxPath);

    ToLongName[NAME_MAX] = 'a';
    ToLongName[NAME_MAX + 1] = '\0'; /* extend ToLongName by one too many */
    ToLongPath[PATH_MAX - 1] = '/';
    ToLongPath[PATH_MAX] = '\0'; /* inc ToLongPath by one */
}

void e(n)

```

```
int n;
{
    int err_num = errno;          /* Save in case printf clobbers it. */

    printf("Subtest %d, error %d errno=%d: ", subtest, n, errno);
    errno = err_num;
    perror("");
    if (errct++ > MAX_ERROR) {
        printf("Too many errors; test aborted\n");
        chdir("..");
        system("rm -rf DIR*");
        exit(1);
    }
    errno = 0;
}

void quit()
{
    Chdir("..");
    System("rm -rf DIR_24");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
        printf("%d errors\n", errct);
        exit(1);
    }
}
```

```

/* test25: open (), close ()      (p) Jan-Mark Wams. email: jms@cs.vu.nl */

/* Not tested: O_NONBLOCK on special files, supporting it.
** On a read-only file system, some error reports are to be expected.
*/

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <errno.h>
#include <time.h>
#include <stdio.h>

#define MAX_ERROR      4
#define ITERATIONS     2

#define System(cmd)     if (system(cmd) != 0) printf("“%s” failed\n", cmd)
#define Chdir(dir)     if (chdir(dir) != 0) printf("Can't goto %s\n", dir)
#define Stat(a,b)      if (stat(a,b) != 0) printf("Can't stat %s\n", a)
#define Creat(f)       if (close(creat(f,0777))!=0) printf("Can't creat %s\n", f)
#define Report(s,n)    printf("Subtest %d" s,subtest,(n))

int errct = 0;
int subtest = 1;
int superuser;
char MaxName[NAME_MAX + 1]; /* Name of maximum length */
char MaxPath[PATH_MAX];    /* Same for path */
char ToLongName[NAME_MAX + 2]; /* Name of maximum +1 length */
char ToLongPath[PATH_MAX + 1]; /* Same for path, both too long */

_PROTOTYPE(void main, (int argc, char *argv[]));
_PROTOTYPE(void test25a, (void));
_PROTOTYPE(void test25b, (void));
_PROTOTYPE(void test25c, (void));
_PROTOTYPE(void test25d, (void));
_PROTOTYPE(void test25e, (void));
_PROTOTYPE(void makelongnames, (void));
_PROTOTYPE(void e, (int number));
_PROTOTYPE(void quit, (void));

void main(argc, argv)
int argc;
char *argv[];
{
    int i, m = 0xFFFF;

    sync();
    if (geteuid() == 0 || getuid() == 0) {
        printf("Test 25 cannot run as root; test aborted\n");
        exit(1);
    }

    if (argc == 2) m = atoi(argv[1]);
    printf("Test 25 ");
    fflush(stdout);
    System("rm -rf DIR_25; mkdir DIR_25");
    Chdir("DIR_25");
    makelongnames();
    superuser = (geteuid() == 0);

    /* Close all files, the parent might have opened. */
    for (i = 3; i < 100; i++) close(i);

    for (i = 0; i < ITERATIONS; i++) {
        if (m & 001) test25a();
        if (m & 002) test25b();
        if (m & 004) test25c();
        if (m & 010) test25d();
        if (m & 020) test25e();
    }
}

```

```

    }
    quit();
}

void test25a()
{
    /* Test fcntl flags. */
    subtest = 1;

#define EXCLUDE(a,b)      (((a)^(b)) == ((a)|(b)))
#define ADDIT             (O_APPEND | O_CREAT | O_EXCL | O_NONBLOCK | O_TRUNC)

    /* If this compiles all flags are defined but they have to be or-able. */
    if (!(EXCLUDE(O_NONBLOCK, O_TRUNC))) e(1);
    if (!(EXCLUDE(O_EXCL, O_NONBLOCK | O_TRUNC))) e(2);
    if (!(EXCLUDE(O_CREAT, O_EXCL | O_NONBLOCK | O_TRUNC))) e(3);
    if (!(EXCLUDE(O_APPEND, O_CREAT | O_EXCL | O_NONBLOCK | O_TRUNC))) e(4);
    if (!(EXCLUDE(O_RDONLY, ADDIT))) e(5);
    if (!(EXCLUDE(O_WRONLY, ADDIT))) e(6);
    if (!(EXCLUDE(O_RDWR, ADDIT))) e(7);
}

void test25b()
{
    /* Test normal operation. */

#define BUF_SIZE 1024

    int fd1, fd2, fd3, fd4, fd5;
    char buf[BUF_SIZE];
    struct stat st1, st2, st3;
    time_t time1, time2;
    int stat_loc;

    subtest = 2;

    System("rm -rf ../DIR_25/*");

    System("echo Hello>he");          /* make test files */
    System("echo Hello>ha");          /* size 6 bytes */
    System("echo Hello>hi");
    System("echo Hello>ho");

    /* Check path resolution. Check if lowest fds are returned */
    if ((fd1 = open("he", O_RDONLY)) != 3) e(1);
    if (read(fd1, buf, BUF_SIZE) != 6) e(2);
    if ((fd2 = open("./ha", O_RDONLY)) != 4) e(3);
    if ((fd3 = open("../DIR_25/he", O_RDWR)) != 5) e(4);
    if ((fd4 = open("ho", O_WRONLY)) != 6) e(5);
    if (close(fd4) != 0) e(6);
    if (close(fd1) != 0) e(7);
    if ((fd1 = open("../ho", O_RDWR)) != 3) e(8);
    if ((fd4 = open("../DIR_25/he", O_RDONLY)) != 6) e(9);
    if (close(fd2) != 0) e(10);
    if (close(fd3) != 0) e(11);
    if ((fd2 = open("ha", O_RDONLY)) != 4) e(12);
    if ((fd3 = open("/etc/passwd", O_RDONLY)) != 5) e(13);
    if (close(fd4) != 0) e(14); /* close all */
    if (close(fd1) != 0) e(15);
    if (close(fd3) != 0) e(16);

    /* Check if processes share fd2, and if they have independent new fds */
    System("rm -rf /tmp/sema.25");
    switch (fork()) {
        case -1: printf("Can't fork\n"); break;

        case 0:
            if ((fd1 = open("he", O_WRONLY)) != 3) e(17);
            if ((fd3 = open("../DIR_25/ha", O_WRONLY)) != 5) e(18);
            if ((fd4 = open("../DIR_25/hi", O_WRONLY)) != 6) e(19);
            if ((fd5 = open("ho", O_WRONLY)) != 7) e(20);
            system("while test !-f /tmp/sema.25; do sleep 1; done"); /* parent */
            if (read(fd2, buf, BUF_SIZE) != 3) e(21); /* gets Hel */
            if (strncmp(buf, "lo\n", 3) != 0) e(22); /* we get lo */
            if (close(fd1) != 0) e(23);
            if (close(fd2) != 0) e(24);
    }
}

```



```

    if (close(fd3) != 0) e(25);
    if (close(fd4) != 0) e(26);
    if (close(fd5) != 0) e(27);
    exit(0);

default:
    if ((fd1 = open("ha", O_RDONLY)) != 3) e(28);
    if ((fd3 = open("./he", O_RDONLY)) != 5) e(29);
    if ((fd4 = open("../DIR_25/hi", O_RDWR)) != 6) e(30);
    if ((fd5 = open("ho", O_WRONLY)) != 7) e(31);
    if (close(fd1) != 0) e(32);
    if (read(fd2, buf, 3) != 3) e(33);      /* get Hel */
    Creat("/tmp/sema.25");
    if (strncmp(buf, "Hel", 3) != 0) e(34);
    if (close(fd2) != 0) e(35);
    if (close(fd3) != 0) e(36);
    if (close(fd4) != 0) e(37);
    if (close(fd5) != 0) e(38);
    if (wait(&stat_loc) == -1) e(39);
    if (stat_loc != 0) e(40);
}
System("rm -f/tmp/sema.25");

/* Check if the file status information is updated correctly */
Stat("hi", &st1);      /* get info */
Stat("ha", &st2);      /* of files */
time(&timel);
while (timel >= time((time_t *)0))
    ;                  /* wait a sec */
if ((fd1 = open("hi", O_RDONLY)) != 3) e(41); /* open files */
if ((fd2 = open("ha", O_WRONLY)) != 4) e(42);
if (read(fd1, buf, 1) != 1) e(43);      /* read one */
if (close(fd1) != 0) e(44);      /* close one */
Stat("hi", &st3);      /* get info */
if (st1.st_uid != st3.st_uid) e(45);
if (st1.st_gid != st3.st_gid) e(46); /* should be same */
if (st1.st_mode != st3.st_mode) e(47);
if (st1.st_size != st3.st_size) e(48);
if (st1.st_nlink != st3.st_nlink) e(49);
if (st1.st_mtime != st3.st_mtime) e(50);
if (st1.st_ctime != st3.st_ctime) e(51);
#ifdef V1_FILESYSTEM
    if (st1.st_atime >= st3.st_atime) e(52);      /* except for atime. */
#endif
if (write(fd2, "Howdy\n", 6) != 6) e(53);      /* Update c & mtime. */
if ((fd1 = open("ha", O_RDWR)) != 3) e(54);
if (read(fd1, buf, 6) != 6) e(55);      /* Update atime. */
if (strncmp(buf, "Howdy\n", 6) != 0) e(56);
if (close(fd1) != 0) e(57);
Stat("ha", &st3);
if (st2.st_uid != st3.st_uid) e(58);
if (st2.st_gid != st3.st_gid) e(59); /* should be same */
if (st2.st_mode != st3.st_mode) e(60);
if (st2.st_nlink != st3.st_nlink) e(61);
if (st2.st_ctime >= st3.st_ctime) e(62);
#ifdef V1_FILESYSTEM
    if (st2.st_atime >= st3.st_atime) e(63);
#endif
#ifdef V1_FILESYSTEM
    if (st2.st_mtime >= st3.st_mtime) e(64);
    if (st2.st_size != st3.st_size) e(65);
    if (close(fd2) != 0) e(66);

    /* Let's see if RDONLY files are read only. */
    if ((fd1 = open("hi", O_RDONLY)) != 3) e(67);
    if (write(fd1, " again", 7) != -1) e(68);      /* we can't write */
    if (errno != EBADF) e(69);      /* a read only fd */
    if (read(fd1, buf, 7) != 6) e(70);      /* but we can read */
    if (close(fd1) != 0) e(71);

    /* Let's see if WRONLY files are write only. */
    if ((fd1 = open("hi", O_WRONLY)) != 3) e(72);
    if (read(fd1, buf, 7) != -1) e(73);      /* we can't read */
    if (errno != EBADF) e(74);      /* a write only fd */
    if (write(fd1, "HELLO", 6) != 6) e(75);      /* but we can write */

```

```
if (close(fdl) != 0) e(76);

/* Let's see if files are closable only once. */
if (close(fdl) != -1) e(77);
if (errno != EBADF) e(78);

/* Let's see how calling close() with bad fds is handled. */
if (close(10) != -1) e(79);
if (errno != EBADF) e(80);
if (close(111) != -1) e(81);
if (errno != EBADF) e(82);
if (close(-432) != -1) e(83);
if (errno != EBADF) e(84);

/* Let's see if RDWR files are read & write able. */
if ((fdl = open("hi", O_RDWR)) != 3) e(85);
if (read(fdl, buf, 6) != 6) e(86); /* we can read */
if (strncmp(buf, "HELLO", 6) != 0) e(87); /* and we can write */
if (write(fdl, "Hello", 6) != 6) e(88); /* a read write fd */
if (close(fdl) != 0) e(89);

/* Check if APPENDED files are really appended */
if ((fdl = open("hi", O_RDWR | O_APPEND)) != 3) e(90); /* open hi */

/* An open should set the file offset to 0. */
if (lseek(fdl, (off_t) 0, SEEK_CUR) != 0) e(91);

/* Writing 0 bytes should not have an effect. */
if (write(fdl, "", 0) != 0) e(92);
if (lseek(fdl, (off_t) 0, SEEK_CUR) != 0) e(93); /* the end? */

/* A seek before a write should not matter with O_APPEND. */
Stat("hi", &st1);
if (lseek(fdl, (off_t) - 3, SEEK_END) != st1.st_size - 3) e(94);

/* By writing 1 byte, we force the offset to the end of the file */
if (write(fdl, "1", 1) != 1) e(95);
Stat("hi", &st1);
if (lseek(fdl, (off_t) 0, SEEK_CUR) != st1.st_size) e(96);
if (write(fdl, "2", 1) != 1) e(97);
Stat("hi", &st1);
if (lseek(fdl, (off_t) 0, SEEK_CUR) != st1.st_size) e(98);
if (write(fdl, "3", 1) != 1) e(99);
Stat("hi", &st1);
if (lseek(fdl, (off_t) 0, SEEK_CUR) != st1.st_size) e(100);
if (lseek(fdl, (off_t) - 2, SEEK_CUR) <= 0) e(101);
if (write(fdl, "4", 1) != 1) e(102);

/* Since the mode was O_APPEND, the offset should be reset to EOF */
Stat("hi", &st1);
if (lseek(fdl, (off_t) 0, SEEK_CUR) != st1.st_size) e(103);
if (lseek(fdl, (off_t) - 4, SEEK_CUR) != st1.st_size - 4) e(104);
if (read(fdl, buf, BUF_SIZE) != 4) e(105);
if (strncmp(buf, "1234", 4) != 0) e(106);
if (close(fdl) != 0) e(107);

/* Check the effect of O_CREAT */
Stat("ho", &st1);
fdl = open("ho", O_RDWR | O_CREAT, 0000);
if (fdl != 3) e(108);
Stat("ho", &st2);
if (memcmp(&st1, &st2, sizeof(struct stat)) != 0) e(109);
if (read(fdl, buf, 6) != 6) e(110);
if (strncmp(buf, "Hello\n", 6) != 0) e(111);
if (write(fdl, "@", 1) != 1) e(112);
if (close(fdl) != 0) e(113);
(void) umask(0000);
fdl = open("ho", O_RDWR | O_CREAT | O_EXCL, 0777);
if (fdl != -1) e(114); /* ho exists */
System("rm -rf new");
time(&timel);
while (timel >= time((time_t *)0))
;
fdl = open("new", O_RDWR | O_CREAT, 0716);
```

```

if (fd1 != 3) e(115);          /* new file */
Stat("new", &st1);
time(&time2);
while (time2 >= time((time_t *)0))
    ;
time(&time2);
if (st1.st_uid != geteuid()) e(116); /* try this as superuser. */
if (st1.st_gid != getegid()) e(117);
if ((st1.st_mode & 0777) != 0716) e(118);
if (st1.st_nlink != 1) e(119);
if (st1.st_mtime <= time1) e(120);
if (st1.st_mtime >= time2) e(121);
#ifdef V1_FILESYSTEM
if (st1.st_atime != st1.st_mtime) e(122);
#endif
if (st1.st_ctime != st1.st_mtime) e(123);
if (st1.st_size != 0) e(124);
if (write(fd1, "I'm new in town", 16) != 16) e(125);
if (lseek(fd1, (off_t) - 5, SEEK_CUR) != 11) e(126);
if (read(fd1, buf, 5) != 5) e(127);
if (strncmp(buf, "town", 5) != 0) e(128);
if (close(fd1) != 0) e(129);

/* Let's test the O_TRUNC flag on this new file. */
time(&time1);
while (time1 >= time((time_t *)0));
if ((fd1 = open("new", O_RDWR | O_TRUNC)) != 3) e(130);
Stat("new", &st1);
time(&time2);
while (time2 >= time((time_t *)0));
time(&time2);
if ((st1.st_mode & 0777) != 0716) e(131);
if (st1.st_size != (size_t) 0) e(132);          /* TRUNCed ? */
if (st1.st_mtime <= time1) e(133);
if (st1.st_mtime >= time2) e(134);
if (st1.st_ctime != st1.st_mtime) e(135);
if (close(fd1) != 0) e(136);

/* Test if file permission bits and the file ownership are unchanged. */
/* So we will see if 'O_CREAT' has no effect if the file exists. */
if (superuser) {
    System("echo > bar; chmod 077 bar");          /* Make bar 077 */
    System("chown daemon bar");
    System("chgrp daemon bar");          /* Daemon's bar */
    fd1 = open("bar", O_RDWR | O_CREAT | O_TRUNC, 0777);          /* knock knock */
    if (fd1 == -1) e(137);
    if (write(fd1, "foo", 3) != 3) e(138);          /* rewrite bar */
    if (close(fd1) != 0) e(139);
    Stat("bar", &st1);
    if (st1.st_uid != 1) e(140);          /* bar is still */
    if (st1.st_gid != 1) e(141);          /* owned by daemon */
    if ((st1.st_mode & 0777) != 077) e(142);          /* mode still is 077 */
    if (st1.st_size != (size_t) 3) e(143);          /* 3 bytes long */

    /* We do the whole thing again, but with O_WRONLY */
    fd1 = open("bar", O_WRONLY | O_CREAT | O_TRUNC, 0777);
    if (fd1 == -1) e(144);
    if (write(fd1, "foobar", 6) != 6) e(145);          /* rewrite bar */
    if (close(fd1) != 0) e(146);
    Stat("bar", &st1);
    if (st1.st_uid != 1) e(147);          /* bar is still */
    if (st1.st_gid != 1) e(148);          /* owned by daemon */
    if ((st1.st_mode & 0777) != 077) e(149);          /* mode still is 077 */
    if (st1.st_size != (size_t) 6) e(150);          /* 6 bytes long */
}
}

void test25c()
{
    /* Test normal operation Part two. */
    int fd1, fd2;
    char buf[BUF_SIZE];
    struct stat st;
    int stat_loc;
    static int iteration=0;

```

```
subtest = 3;
iteration++;

System("rm -rf ../DIR_25/*");

/* Fifo file test here. */
if (mkfifo("fifo", 0777) != 0) e(1);
switch (fork()) {
    case -1: printf("Can't fork\n"); break;
    case 0:
        alarm(20); /* Give child 20 seconds to live. */
        if ((fd1 = open("fifo", O_RDONLY)) != 3) e(2);
        if (read(fd1, buf, BUF_SIZE) != 23) e(3);
        if (strncmp(buf, "1 2 3 testing testing\n", 23) != 0) e(4);
        if (close(fd1) != 0) e(5);
        exit(0);
    default:
        if ((fd1 = open("fifo", O_WRONLY)) != 3) e(6);
        if (write(fd1, "1 2 3 testing testing\n", 23) != 23) e(7);
        if (close(fd1) != 0) e(8);
        if (wait(&stat_loc) == -1) e(9);
        if (stat_loc != 0) e(10); /* The alarm went off? */
}

/* Try opening for writing with O_NONBLOCK. */
fd1 = open("fifo", O_WRONLY | O_NONBLOCK);
if (fd1 != -1) e(11);
if (errno != ENXIO) e(12);
close(fd1);

/* Try opening for writing with O_NONBLOCK and O_CREAT. */
fd1 = open("fifo", O_WRONLY | O_CREAT | O_NONBLOCK, 0777);
if (fd1 != -1) e(13);
if (errno != ENXIO) e(14);
close(fd1);

/* Both the NONBLOCK and the EXCLUSIVE give raise to error. */
fd1 = open("fifo", O_WRONLY | O_CREAT | O_EXCL | O_NONBLOCK, 0777);
if (fd1 != -1) e(15);
if (errno != EEXIST && errno != ENXIO) e(16);
close(fd1); /* Just in case. */

/* Try opening for reading with O_NONBLOCK. */
fd1 = open("fifo", O_RDONLY | O_NONBLOCK);
if (fd1 != 3) e(17);
if (close(fd1) != 0) e(18);

/* Nopt runs out of memory. ;-< We just cut out some valid code */
/* FIFO's should always append. (They have no file position.) */
switch (fork()) {
    case -1: printf("Can't fork\n"); break;
    case 0:
        alarm(20); /* Give child 20 seconds to live. */
        if ((fd1 = open("fifo", O_WRONLY)) != 3) e(19);
        if ((fd2 = open("fifo", O_WRONLY)) != 4) e(20);
        if (write(fd1, "I did see Elvis.\n", 18) != 18) e(21);
        if (write(fd2, "I DID.\n", 8) != 8) e(22);
        if (close(fd2) != 0) e(23);
        if (close(fd1) != 0) e(24);
        exit(0);
    default:
        if ((fd1 = open("fifo", O_RDONLY)) != 3) e(25);
        if (read(fd1, buf, 18) != 18) e(26);
        if (strncmp(buf, "I did see Elvis.\n", 18) != 0) e(27);
        if (read(fd1, buf, BUF_SIZE) != 8) e(28);
        if (strncmp(buf, "I DID.\n", 8) != 0) e(29);
        if (close(fd1) != 0) e(30);
        if (wait(&stat_loc) == -1) e(31);
        if (stat_loc != 0) e(32); /* The alarm went off? */
}

/* O_TRUNC should have no effect on FIFO files. */
switch (fork()) {
```

```

case -1:  printf("Can't fork\n");      break;
case 0:
    alarm(20);                          /* Give child 20 seconds to live. */
    if ((fd1 = open("fifo", O_WRONLY)) != 3) e(33);
    if (write(fd1, "I did see Elvis.\n", 18) != 18) e(34);
    if ((fd2 = open("fifo", O_WRONLY | O_TRUNC)) != 4) e(35);
    if (write(fd2, "IDID.\n", 8) != 8) e(36);
    if (close(fd2) != 0) e(37);
    if (close(fd1) != 0) e(38);
    exit(0);
default:
    if ((fd1 = open("fifo", O_RDONLY)) != 3) e(39);
    if (read(fd1, buf, 18) != 18) e(40);
    if (strncmp(buf, "I did see Elvis.\n", 18) != 0) e(41);
    if (read(fd1, buf, BUF_SIZE) != 8) e(42);
    if (strncmp(buf, "IDID.\n", 8) != 0) e(43);
    if (close(fd1) != 0) e(44);
    if (wait(&stat_loc) == -1) e(45);
    if (stat_loc != 0) e(46);          /* The alarm went off? */
}

/* Closing the last fd should flush all data to the bitbucket. */
System("rm -rf /tmp/sema.25");
switch (fork()) {
    case -1:  printf("Can't fork\n");      break;

    case 0:
        alarm(20);                      /* Give child 20 seconds to live. */
        if ((fd1 = open("fifo", O_WRONLY)) != 3) e(47);
        if (write(fd1, "I did see Elvis.\n", 18) != 18) e(48);
        Creat("/tmp/sema.25");
        sleep(2);                       /* give parent a chance to open */
        /* this was sleep(1), but that's too short: child also sleeps(1) */
        if (close(fd1) != 0) e(49);
        exit(0);

        default:
            if ((fd1 = open("fifo", O_RDONLY)) != 3) e(50);
            /* Make 'sure' write has closed. */
            while (stat("/tmp/sema.25", &st) != 0) sleep(1);
            if (close(fd1) != 0) e(51);
            if ((fd1 = open("fifo", O_RDONLY | O_NONBLOCK)) != 3) e(52);
            if (read(fd1, buf, BUF_SIZE) != 18) e(53);
            if (close(fd1) != 0) e(54);
            if (wait(&stat_loc) == -1) e(55);
            if (stat_loc != 0) e(56);          /* The alarm went off? */
}

/* Let's try one too many. */
System("rm -rf /tmp/sema.25");
switch (fork()) {
    case -1:  printf("Can't fork\n");      break;
    case 0:
        alarm(20);                      /* Give child 20 seconds to live. */
        if ((fd1 = open("fifo", O_WRONLY)) != 3) e(57);
        if (write(fd1, "I did see Elvis.\n", 18) != 18) e(58);

        /* Keep open till second reader is opened. */
        while (stat("/tmp/sema.25", &st) != 0) sleep(1);
        if (close(fd1) != 0) e(59);
        exit(0);

        default:
            if ((fd1 = open("fifo", O_RDONLY)) != 3) e(60);
            if (read(fd1, buf, 2) != 2) e(61);
            if (strncmp(buf, "I ", 2) != 0) e(62);
            if (close(fd1) != 0) e(63);
            if ((fd1 = open("fifo", O_RDONLY)) != 3) e(64);

            /* Signal second reader is open. */
            Creat("/tmp/sema.25");
            if (read(fd1, buf, 4) != 4) e(65);
            if (strncmp(buf, "did ", 4) != 0) e(66);
            if ((fd2 = open("fifo", O_RDONLY)) != 4) e(67);
            if (read(fd2, buf, BUF_SIZE) != 12) e(68);

```

```
    if (strcmp(buf, "see Elvis.\n", 12) != 0) e(69);
    if (close(fd2) != 0) e(70);
    if (close(fd1) != 0) e(71);
    if (wait(&stat_loc) == -1) e(72);
    if (stat_loc != 0) e(73);          /* The alarm went off? */
}
System("rm -rf fifo /tmp/sema.25");

/* O_TRUNC should have no effect on directroys. */
System("mkdir dir; touch dir/f1 dir/f2 dir/f3");
if ((fd1 = open("dir", O_WRONLY | O_TRUNC)) != -1) e(74);
if (errno != EISDIR) e(75);
close(fd1);

/* Opening a directory for reading should be possible. */
if ((fd1 = open("dir", O_RDONLY)) != 3) e(76);
if (close(fd1) != 0) e(77);
if (unlink("dir/f1") != 0) e(78);      /* Should still be there. */
if (unlink("dir/f2") != 0) e(79);
if (unlink("dir/f3") != 0) e(80);
if (rmdir("dir") != 0) e(81);

if (!superuser) {
    /* Test if O_CREAT is not usable to open files with the wrong mode */
    (void) umask(0200);      /* nono has no */
    System("touch nono");    /* write bit */
    (void) umask(0000);
    fd1 = open("nono", O_RDWR | O_CREAT, 0777);    /* try to open */
    if (fd1 != -1) e(82);
    if (errno != EACCES) e(83);    /* but no access */
}
}

void test25d()
{
    int fd;

    subtest = 4;

    System("rm -rf ../DIR_25/*");

    /* Test maximal file name length. */
    if ((fd = open(MaxName, O_RDWR | O_CREAT, 0777)) != 3) e(1);
    if (close(fd) != 0) e(2);
    MaxPath[strlen(MaxPath) - 2] = '/';
    MaxPath[strlen(MaxPath) - 1] = 'a';    /* make ../.../a */
    if ((fd = open(MaxPath, O_RDWR | O_CREAT, 0777)) != 3) e(3);
    if (close(fd) != 0) e(4);
    MaxPath[strlen(MaxPath) - 1] = '/';    /* make ../.../a */
}

void test25e()
{
    int fd;
    char *noread = "noread";    /* Name for unreadable file. */
    char *nowrite = "nowrite";    /* Same for unwritable. */
    int stat_loc;

    subtest = 5;

    System("rm -rf ../DIR_25/*");

    mkdir("bar", 0777);    /* make bar */

    /* Check if no access on part of path generates the correct error. */
    System("chmod 677 bar");    /* rw-rwxrwx */
    if (open("bar/nono", O_RDWR | O_CREAT, 0666) != -1) e(1);
    if (errno != EACCES) e(2);

    /* Ditto for no write permission. */
    System("chmod 577 bar");    /* r-xrwxrwx */
    if (open("bar/nono", O_RDWR | O_CREAT, 0666) != -1) e(3);
    if (errno != EACCES) e(4);
}
```

```
/* Clean up bar. */
System("rm -rf bar");

/* Improper flags set on existing file. */
System("touch noread; chmod 377 noread"); /* noread */
if (open(noread, O_RDONLY) != -1) e(5);
if (open(noread, O_RDWR) != -1) e(6);
if (open(noread, O_RDWR | O_CREAT, 0777) != -1) e(7);
if (open(noread, O_RDWR | O_CREAT | O_TRUNC, 0777) != -1) e(8);
if ((fd = open(noread, O_WRONLY)) != 3) e(9);
if (close(fd) != 0) e(10);
System("touch nowrite; chmod 577 nowrite"); /* nowrite */
if (open(nowrite, O_WRONLY) != -1) e(11);
if (open(nowrite, O_RDWR) != -1) e(12);
if (open(nowrite, O_RDWR | O_CREAT, 0777) != -1) e(13);
if (open(nowrite, O_RDWR | O_CREAT | O_TRUNC, 0777) != -1) e(14);
if ((fd = open(nowrite, O_RDONLY)) != 3) e(15);
if (close(fd) != 0) e(16);
if (superuser) {
    /* If we can make a file ownd by some one else, test access again. */
    System("chmod 733 noread");
    System("chown bin noread");
    System("chgrp system noread");
    System("chmod 755 nowrite");
    System("chown bin nowrite");
    System("chgrp system nowrite");
    switch (fork()) {
        case -1: printf("Can't fork\n"); break;
        case 0:
            setuid(1);
            setgid(1); /* become daemon */
            if (open(noread, O_RDONLY) != -1) e(17);
            if (open(noread, O_RDWR) != -1) e(18);
            if (open(noread, O_RDWR | O_CREAT, 0777) != -1) e(19);
            fd = open(noread, O_RDWR | O_CREAT | O_TRUNC, 0777);
            if (fd != -1) e(20);
            if ((fd = open(noread, O_WRONLY)) != 3) e(21);
            if (close(fd) != 0) e(22);
            if (open(nowrite, O_WRONLY) != -1) e(23);
            if (open(nowrite, O_RDWR) != -1) e(24);
            if (open(nowrite, O_RDWR | O_CREAT, 0777) != -1) e(25);
            fd = open(nowrite, O_RDWR | O_CREAT | O_TRUNC, 0777);
            if (fd != -1) e(26);
            if ((fd = open(nowrite, O_RDONLY)) != 3) e(27);
            if (close(fd) != 0) e(28);
            exit(0);
        default:
            if (wait(&stat_loc) == -1) e(29);
    }
}

/* Clean up the noread and nowrite files. */
System("rm -rf noread nowrite");

/* Test the O_EXCL flag. */
System("echo > exists");
if (open("exists", O_RDWR | O_CREAT | O_EXCL, 0777) != -1) e(30);
if (errno != EEXIST) e(31);
if (open("exists", O_RDONLY | O_CREAT | O_EXCL, 0777) != -1) e(32);
if (errno != EEXIST) e(33);
if (open("exists", O_WRONLY | O_CREAT | O_EXCL, 0777) != -1) e(34);
if (errno != EEXIST) e(35);
fd = open("exists", O_RDWR | O_CREAT | O_EXCL | O_TRUNC, 0777);
if (fd != -1) e(36);
if (errno != EEXIST) e(37);
fd = open("exists", O_RDONLY | O_CREAT | O_EXCL | O_TRUNC, 0777);
if (fd != -1) e(38);
if (errno != EEXIST) e(39);
fd = open("exists", O_WRONLY | O_CREAT | O_EXCL | O_TRUNC, 0777);
if (fd != -1) e(40);
if (errno != EEXIST) e(41);

/* Test ToLongName and ToLongPath */
if ((fd = open(ToLongName, O_RDWR | O_CREAT, 0777)) != 3) e(45);
```

```
    if (close(fd) != 0) e(46);
    ToLongPath[PATH_MAX - 2] = '/';
    ToLongPath[PATH_MAX - 1] = 'a';
    if ((fd = open(ToLongPath, O_RDWR | O_CREAT, 0777)) != -1) e(47);
    if (errno != ENAMETOOLONG) e(48);
    if (close(fd) != -1) e(49);
    ToLongPath[PATH_MAX - 1] = '/';
}

void makelongnames()
{
    register int i;

    memset(MaxName, 'a', NAME_MAX);
    MaxName[NAME_MAX] = '\0';
    for (i = 0; i < PATH_MAX - 1; i++) { /* idem path */
        MaxPath[i++] = '.';
        MaxPath[i] = '/';
    }
    MaxPath[PATH_MAX - 1] = '\0';

    strcpy(ToLongName, MaxName); /* copy them Max to ToLong */
    strcpy(ToLongPath, MaxPath);

    ToLongName[NAME_MAX] = 'a';
    ToLongName[NAME_MAX + 1] = '\0'; /* extend ToLongName by one too many */
    ToLongPath[PATH_MAX - 1] = '/';
    ToLongPath[PATH_MAX] = '\0'; /* inc ToLongPath by one */
}

void e(n)
int n;
{
    int err_num = errno; /* Save in case printf clobbers it. */

    printf("Subtest %d, error %d errno=%d: ", subtest, n, errno);
    errno = err_num;
    perror("");
    if (errct++ > MAX_ERROR) {
        printf("Too many errors; test aborted\n");
        chdir("..");
        system("rm -rf DIR*");
        exit(1);
    }
    errno = 0;
}

void quit()
{
    Chdir("..");
    System("rm -rf DIR_25");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
        printf("%d errors\n", errct);
        exit(1);
    }
}
```



```

/* test26: lseek()                                Author: Jan-Mark Wams (jms@cs.vu.nl) */

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <errno.h>
#include <time.h>
#include <stdio.h>

#define MAX_ERROR      4
#define ITERATIONS     10

#define System(cmd)    if (system(cmd) != 0) printf("“%s” failed\n", cmd)
#define Chdir(dir)     if (chdir(dir) != 0) printf("Can't goto %s\n", dir)
#define Stat(a,b)      if (stat(a,b) != 0) printf("Can't stat %s\n", a)
#define Mkfifo(f)      if (mkfifo(f,0777)!=0) printf("Can't make fifo %s\n", f)

int errct = 0;
int subtest = 1;
char MaxName[NAME_MAX + 1]; /* Name of maximum length */
char MaxPath[PATH_MAX];    /* Same for path */
char ToLongName[NAME_MAX + 2]; /* Name of maximum +1 length */
char ToLongPath[PATH_MAX + 1]; /* Same for path, both too long */

_PROTOTYPE(void main, (int argc, char *argv[]));
_PROTOTYPE(void test26a, (void));
_PROTOTYPE(void test26b, (void));
_PROTOTYPE(void test26c, (void));
_PROTOTYPE(void e, (int number));
_PROTOTYPE(void quit, (void));

void main(argc, argv)
int argc;
char *argv[];
{
    int i, m = 0xFFFF;

    sync();
    if (argc == 2) m = atoi(argv[1]);
    printf("Test 26 ");
    fflush(stdout);
    System("rm -rf DIR_26; mkdir DIR_26");
    Chdir("DIR_26");

    for (i = 0; i < 10; i++) {
        if (m & 0001) test26a();
        if (m & 0002) test26b();
        if (m & 0004) test26c();
    }
    quit();
}

void test26a()
{
    /* Test normal operation. */
    int fd;
    char buf[20];
    int i, j;
    struct stat st;

    subtest = 1;
    System("rm -rf ../DIR_26/*");

    System("echo -n hihaho > hihaho");
    if ((fd = open("hihaho", O_RDONLY)) != 3) e(1);
    if (lseek(fd, (off_t) 3, SEEK_SET) != (off_t) 3) e(2);
    if (read(fd, buf, 1) != 1) e(3);
    if (buf[0] != 'a') e(4);
    if (lseek(fd, (off_t) - 1, SEEK_END) != 5) e(5);
    if (read(fd, buf, 1) != 1) e(6);
}

```

```
if (buf[0] != 'o') e(7);

/* Seek past end of file. */
if (lseek(fd, (off_t) 1000, SEEK_END) != 1006) e(8);
if (read(fd, buf, 1) != 0) e(9);

/* Lseek() should not extend the file. */
if (fstat(fd, &st) != 0) e(10);
if (st.st_size != (off_t) 6) e(11);
if (close(fd) != 0) e(12);

/* Probeer lseek met write. */
if ((fd = open("hihaho", O_WRONLY)) != 3) e(13);
if (lseek(fd, (off_t) 3, SEEK_SET) != (off_t) 3) e(14);
if (write(fd, "e", 1) != 1) e(15);
if (lseek(fd, (off_t) 1000, SEEK_END) != 1006) e(16);

/* Lseek() should not extend the file. */
if (fstat(fd, &st) != 0) e(17);
if (st.st_size != (off_t) 6) e(18);
if (write(fd, "e", 1) != 1) e(19);

/* Lseek() and a subsequent write should! */
if (fstat(fd, &st) != 0) e(20);
if (st.st_size != (off_t) 1007) e(21);

if (close(fd) != 0) e(22);

/* Check the file, it should start with hiheho. */
if ((fd = open("hihaho", O_RDONLY)) != 3) e(23);
if (read(fd, buf, 6) != 6) e(24);
if (strncmp(buf, "hiheho", 6) != 0) e(25);

/* The should be zero bytes and a trailing '\0'. */
if (sizeof(buf) < 10) e(26);
for (i = 1; i <= 20; i++) {
    if (read(fd, buf, 10) != 10) e(27);
    for (j = 0; j < 10; j++)
        if (buf[j] != '\0') break;
    if (j != 10) e(28);
    if (lseek(fd, (off_t) 15, SEEK_CUR) != (off_t) i * 25 + 6) e(29);
}

if (lseek(fd, (off_t) 1006, SEEK_SET) != (off_t) 1006) e(30);
if (read(fd, buf, sizeof(buf)) != 1) e(31);
if (buf[0] != 'e') e(32);

if (lseek(fd, (off_t) - 1, SEEK_END) != (off_t) 1006) e(33);
if (read(fd, buf, sizeof(buf)) != 1) e(34);
if (buf[0] != 'e') e(35);

/* Closing time. */
if (close(fd) != 0) e(36);
}

void test26b()
{
    int fd1, fd2, fd3;
    int stat_loc;

    subtest = 2;
    System("rm -rf ../DIR_26/*");

    /* See if childs lseek() is effecting the parent. * See also if
     * lseeking() on same file messes things up. */

    /* Creat a file of 11 bytes. */
    if ((fd1 = open("santa", O_WRONLY | O_CREAT, 0777)) != 3) e(1);
    if (write(fd1, "ho ho ho ho", 11) != 11) e(2);
    if (close(fd1) != 0) e(3);

    /* Open it multiple times. */
    if ((fd1 = open("santa", O_RDONLY)) != 3) e(4);
    if ((fd2 = open("santa", O_WRONLY)) != 4) e(5);
```

```
if ((fd3 = open("santa", O_RDWR)) != 5) e(6);

/* Set all offsets different. */
if (lseek(fd1, (off_t) 2, SEEK_SET) != 2) e(7);
if (lseek(fd2, (off_t) 4, SEEK_SET) != 4) e(8);
if (lseek(fd3, (off_t) 7, SEEK_SET) != 7) e(9);

/* Have a child process do additional offset changes. */
switch (fork()) {
    case -1: printf("Can't fork\n"); break;
    case 0:
        alarm(20);
        if (lseek(fd1, (off_t) 1, SEEK_CUR) != 3) e(10);
        if (lseek(fd2, (off_t) 5, SEEK_SET) != 5) e(11);
        if (lseek(fd3, (off_t) - 4, SEEK_END) != 7) e(12);
        exit(0);
    default:
        wait(&stat_loc);
        if (stat_loc != 0) e(13); /* Alarm? */
}

/* Check if the new offsets are correct. */
if (lseek(fd1, (off_t) 0, SEEK_CUR) != 3) e(14);
if (lseek(fd2, (off_t) 0, SEEK_CUR) != 5) e(15);
if (lseek(fd3, (off_t) 0, SEEK_CUR) != 7) e(16);

/* Close the file. */
if (close(fd1) != 0) e(17);
if (close(fd2) != 0) e(18);
if (close(fd3) != 0) e(19);
}

void test26c()
{
    /* Test error returns. */
    int fd;
    int tube[2];
    int i, stat_loc;

    subtest = 3;
    System("rm -rf ../DIR_26/*");

    /* Fifo's can't be lseeked(). */
    Mkfifo("fifo");
    switch (fork()) {
        case -1: printf("Can't fork\n"); break;
        case 0:
            alarm(3); /* Try for max 3 secs. */
            if ((fd = open("fifo", O_RDONLY)) != 3) e(1);
            if (lseek(fd, (off_t) 0, SEEK_SET) != (off_t) - 1) e(2);
            if (errno != ESPIPE) e(3);
            if (close(fd) != 0) e(4);
            exit(0);
        default:
            if ((fd = open("fifo", O_WRONLY)) != 3) e(5);
            wait(&stat_loc);
            if (stat_loc != 0) e(6); /* Alarm? */
            if (close(fd) != 0) e(7);
    }

    /* Pipes can't be lseeked() either. */
    if (pipe(tube) != 0) e(8);
    switch (fork()) {
        case -1: printf("Can't fork\n"); break;
        case 0:
            alarm(3); /* Max 3 sconds wait. */
            if (lseek(tube[0], (off_t) 0, SEEK_SET) != (off_t) - 1) e(9);
            if (errno != ESPIPE) e(10);
            if (lseek(tube[1], (off_t) 0, SEEK_SET) != (off_t) - 1) e(11);
            if (errno != ESPIPE) e(12);
            exit(0);
        default:
            wait(&stat_loc);
            if (stat_loc != 0) e(14); /* Alarm? */
    }
}
```

```
/* Close the pipe. */
if (close(tube[0]) != 0) e(15);
if (close(tube[1]) != 0) e(16);

/* Whence argument invalid. */
System("echo -n contact > file");
if ((fd = open("file", O_RDWR)) != 3) e(17);
for (i = -1000; i < 1000; i++) {
    if (i == SEEK_SET || i == SEEK_END || i == SEEK_CUR) continue;
    if (lseek(fd, (off_t) 0, i) != (off_t) -1) e(18);
    if (errno != EINVAL) e(19);
}
if (close(fd) != 0) e(20);

/* EBADF for bad fides. */
for (i = -1000; i < 1000; i++) {
    if (i >= 0 && i < OPEN_MAX) continue;
    if (lseek(i, (off_t) 0, SEEK_SET) != (off_t) -1) e(21);
    if (lseek(i, (off_t) 0, SEEK_END) != (off_t) -1) e(22);
    if (lseek(i, (off_t) 0, SEEK_CUR) != (off_t) -1) e(23);
}
}

void e(n)
int n;
{
    int err_num = errno;          /* Save in case printf clobbers it. */

    printf("Subtest %d, error %d errno=%d: ", subtest, n, errno);
    errno = err_num;
    perror("");
    if (errct++ > MAX_ERROR) {
        printf("Too many errors; test aborted\n");
        chdir("..");
        system("rm -rf DIR*");
        exit(1);
    }
    errno = 0;
}

void quit()
{
    Chdir("..");
    System("rm -rf DIR_26");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
        printf("%d errors\n", errct);
        exit(1);
    }
}
```

```

/* test27: stat() fstat()                      Author: Jan-Mark Wams (jms@cs.vu.nl) */

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <limits.h>
#include <string.h>
#include <limits.h>
#include <time.h>
#include <stdio.h>

#define MODE_MASK      (S_IRWXU | S_IRWXG | S_IRWXO | S_ISUID | S_ISGID)
#define MAX_ERROR      4
#define ITERATIONS     2

#define System(cmd)     if (system(cmd) != 0) printf("“%s” failed\n", cmd)
#define Chdir(dir)      if (chdir(dir) != 0) printf("Can't goto %s\n", dir)

int errct = 0;
int subtest = 1;
int superuser;
char MaxName[NAME_MAX + 1]; /* Name of maximum length */
char MaxPath[PATH_MAX];
char ToLongName[NAME_MAX + 2]; /* Name of maximum +1 length */
char ToLongPath[PATH_MAX + 1];

_PROTOTYPE(void main, (int argc, char *argv[]));
_PROTOTYPE(void test27a, (void));
_PROTOTYPE(void test27b, (void));
_PROTOTYPE(void test27c, (void));
_PROTOTYPE(void makelongnames, (void));
_PROTOTYPE(void e, (int __n));
_PROTOTYPE(void quit, (void));

void main(argc, argv)
int argc;
char *argv[];
{
    int i, m = 0xFFFF;

    sync();
    if (argc == 2) m = atoi(argv[1]);
    printf("Test 27 ");
    fflush(stdout);
    System("rm -rf DIR_27; mkdir DIR_27");
    Chdir("DIR_27");
    superuser = (getuid() == 0);
    makelongnames();

    for (i = 0; i < ITERATIONS; i++) {
        if (m & 0001) test27a();
        if (m & 0002) test27b();
        if (m & 0004) test27c();
    }
    quit();
}

void test27a()
{
    /* Test Normal operation. */
    struct stat st1, st2;
    time_t time1, time2;
    int fd, pfd[2];

    subtest = 1;

    time(&time1); /* get time before */
    while (time1 >= time((time_t *)0))
        ; /* Wait for time to change. */
    System("echo 7bytes > foo; chmod 4750 foo");
    if (stat("foo", &st1) != 0) e(1); /* get foo's info */
    time(&time2);

```

```

while (time2 >= time((time_t *)0))
    ; /* Wait for next second. */
time(&time2); /* get time after */
if ((st1.st_mode & MODE_MASK) != 04750) e(2);
if (st1.st_nlink != 1) e(3); /* check stat */
if (st1.st_uid != geteuid()) e(4);
#if defined(NGROUPS_MAX) && NGROUPS_MAX == 0
if (st1.st_gid != getegid()) e(5);
#endif /* defined(NGROUPS_MAX) && NGROUPS_MAX == 0 */
if (st1.st_size != (size_t) 7) e(6);
if (st1.st_atime <= time1) e(7);
if (st1.st_atime >= time2) e(8);
if (st1.st_ctime <= time1) e(9);
if (st1.st_ctime >= time2) e(10);
if (st1.st_mtime <= time1) e(11);
if (st1.st_mtime >= time2) e(12);

/* Compar stat and fstat. */
System("echo 7bytes>bar");
fd = open("bar", O_RDWR | O_APPEND); /* the bar is open! */
if (fd != 3) e(13); /* should be stderr + 1 */
if (stat("bar", &st1) != 0) e(14); /* get bar's info */
if (fstat(fd, &st2) != 0) e(15); /* get bar's info */

/* St1 en st2 should be the same. */
if (st1.st_dev != st2.st_dev) e(16);
if (st1.st_ino != st2.st_ino) e(17);
if (st1.st_mode != st2.st_mode) e(18);
if (st1.st_nlink != st2.st_nlink) e(19);
if (st1.st_uid != st2.st_uid) e(20);
if (st1.st_gid != st2.st_gid) e(21);
if (st1.st_size != st2.st_size) e(22);
if (st1.st_atime != st2.st_atime) e(23);
if (st1.st_ctime != st2.st_ctime) e(24);
if (st1.st_mtime != st2.st_mtime) e(25);
time(&time1); /* wait a sec. */
while (time1 >= time((time_t *)0))
    ;
System("chmod 755 bar"); /* change mode */
System("rm -f foobar; ln bar foobar"); /* change # links */
if (write(fd, "foo", 4) != 4) e(26); /* write a bit (or two) */
if (stat("bar", &st2) != 0) e(27); /* get new info */
if (st2.st_dev != st1.st_dev) e(28);
if (st2.st_ino != st1.st_ino) e(29); /* compare the fields */
if ((st2.st_mode & MODE_MASK) != 0755) e(30);
if (!S_ISREG(st2.st_mode)) e(31);
if (st2.st_nlink != st1.st_nlink + 1) e(32);
if (st2.st_uid != st1.st_uid) e(33);
if (st2.st_gid != st1.st_gid) e(34);
if (st2.st_size != (size_t) 11) e(35);
if (st2.st_atime != st1.st_atime) e(36);
if (st2.st_ctime <= st1.st_ctime) e(37);
if (st2.st_mtime <= st1.st_mtime) e(38);
if (close(fd) != 0) e(39); /* sorry the bar is closed */

/* Check special file. */
if (stat("/dev/tty", &st1) != 0) e(40);
if (!S_ISCHR(st1.st_mode)) e(41);
#ifdef _MINIX
if (stat("/dev/ram", &st1) != 0) e(42);
if (!S_ISBLK(st1.st_mode)) e(43);
#endif

/* Check fifos. */
time(&time1);
while (time1 >= time((time_t *)0))
    ;
if (mkfifo("fifo", 0640) != 0) e(44);
if (stat("fifo", &st1) != 0) e(45); /* get fifo's info */
time(&time2);
while (time2 >= time((time_t *)0))
    ;
time(&time2);
if (!S_ISFIFO(st1.st_mode)) e(46);

```

```
if (st1.st_nlink != 1) e(47); /* check the stat info */
if (st1.st_uid != geteuid()) e(48);
#if defined(NGROUPS_MAX) && NGROUPS_MAX == 0
if (st1.st_gid != getegid()) e(49);
#endif /* defined(NGROUPS_MAX) && NGROUPS_MAX == 0 */
if (st1.st_size != (size_t) 0) e(50);
if (st1.st_atime <= time1) e(51);
if (st1.st_atime >= time2) e(52);
if (st1.st_ctime <= time1) e(53);
if (st1.st_ctime >= time2) e(54);
if (st1.st_mtime <= time1) e(55);
if (st1.st_mtime >= time2) e(56);

/* Note: the st_mode of a fstat on a pipe should contain a isfifo bit. */
/* Check pipes. */
time(&time1);
while (time1 >= time((time_t *)0))
;
if (pipe(pfd) != 0) e(57);
if (fstat(pfd[0], &st1) != 0) e(58); /* get pipe input info */
time(&time2);
while (time2 >= time((time_t *)0))
;
time(&time2);
if (!(S_ISFIFO(st1.st_mode))) e(59); /* check stat struct */
if (st1.st_uid != geteuid()) e(60);
if (st1.st_gid != getegid()) e(61);
if (st1.st_size != (size_t) 0) e(62);
if (st1.st_atime <= time1) e(63);
if (st1.st_atime >= time2) e(64);
if (st1.st_ctime <= time1) e(65);
if (st1.st_ctime >= time2) e(66);
if (st1.st_mtime <= time1) e(67);
if (st1.st_mtime >= time2) e(68);
if (fstat(pfd[1], &st1) != 0) e(69); /* get pipe output info */
if (!(S_ISFIFO(st1.st_mode))) e(70);
if (st1.st_uid != geteuid()) e(71);
if (st1.st_gid != getegid()) e(72);
if (st1.st_size != (size_t) 0) e(73);
if (st1.st_atime < time1) e(74);
if (st1.st_atime > time2) e(75);
if (st1.st_ctime < time1) e(76);
if (st1.st_ctime > time2) e(77);
if (st1.st_mtime < time1) e(78);
if (st1.st_mtime > time2) e(79);
if (close(pfd[0]) != 0) e(80);
if (close(pfd[1]) != 0) e(81); /* close pipe */

/* Check dirs. */
time(&time1);
while (time1 >= time((time_t *)0))
;
System("mkdir dir");
if (stat("dir", &st1) != 0) e(82); /* get dir info */
time(&time2);
while (time2 >= time((time_t *)0))
;
time(&time2);
if (!(S_ISDIR(st1.st_mode))) e(83); /* check stat struct */
if (st1.st_uid != geteuid()) e(84);
#if defined(NGROUPS_MAX) && NGROUPS_MAX == 0
if (st1.st_gid != getegid()) e(85);
#endif /* defined(NGROUPS_MAX) && NGROUPS_MAX == 0 */
if (st1.st_atime < time1) e(86);
if (st1.st_atime > time2) e(87);
if (st1.st_ctime < time1) e(88);
if (st1.st_ctime > time2) e(89);
if (st1.st_mtime < time1) e(90);
if (st1.st_mtime > time2) e(91);
System("rm -rf ../DIR_27/*");
}

void test27b()
{
    /* Test maxima. */
}
```

```

struct stat st;
int fd;

subtest = 2;

/* Check stats on maximum length files names. */
if (mkdir(MaxName, 0777) != 0) e(1);
if (stat(MaxName, &st) != 0) e(2);
if ((fd = open(MaxName, O_RDONLY)) != 3) e(3);
if (fstat(fd, &st) != 0) e(4);
if (close(fd) != 0) e(5);
if (rmdir(MaxName) != 0) e(6);
if (stat(MaxPath, &st) != 0) e(7);
if ((fd = open(MaxPath, O_RDONLY)) != 3) e(8);
if (fstat(fd, &st) != 0) e(9);
if (close(fd) != 0) e(10);
System("rm -rf ../DIR_27/*");
}

void test27c()
{
    /* Test error response. */
    struct stat st;
    int fd, i;

    subtest = 3;

    System("echo Hi>foo"); /* Make a file called foo. */
    /* Check if a un searchable dir is handled ok. */
    Chdir(".."); /* cd .. */
    System("chmod 677 DIR_27"); /* no search permission */
    if (stat("DIR_27/nono", &st) != -1) e(1);
    if (superuser) {
        if (errno != ENOENT) e(2); /* su has access */
    }
    if (!superuser) {
        if (errno != EACCES) e(3); /* we don't ;-) */
    }
    System("chmod 777 DIR_27");
    Chdir("DIR_27"); /* back to test dir */

    /* Check on ToLongName etc. */
#ifdef _POSIX_NO_TRUNC
# if _POSIX_NO_TRUNC - 0 != -1
    if (stat(ToLongName, &st) != -1) e(4); /* name is too long */
    if (errno != ENAMETOOLONG) e(5);
# endif
#else
# include "error, this case requires dynamic checks and is not handled"
#endif
    if (stat(ToLongPath, &st) != -1) e(6); /* path is too long */
    if (errno != ENAMETOOLONG) e(7);

    /* Test some common errors. */
    if (stat("nono", &st) != -1) e(8); /* nono nonexistent */
    if (errno != ENOENT) e(9);
    if (stat("", &st) != -1) e(10); /* try empty */
    if (errno != ENOENT) e(11);
    if (stat("foo/bar", &st) != -1) e(12); /* foo is a file */
    if (errno != ENOTDIR) e(13);

    /* Test fstat on file descriptors that are not open. */
    for (i = 3; i < 6; i++) {
        if (fstat(i, &st) != -1) e(14);
        if (errno != EBADF) e(15);
    }

    /* Test if a just closed file is 'fstat()'-able. */
    if ((fd = open("foo", O_RDONLY)) != 3) e(16); /* open foo */
    if (fstat(fd, &st) != 0) e(17); /* get stat */
    if (close(fd) != 0) e(18); /* close it */
    if (fstat(fd, &st) != -1) e(19); /* get stat */
    if (errno != EBADF) e(20);
    System("rm -rf ../DIR_27/*");
}

```



```
void makelongnames()
{
    register int i;

    memset(MaxName, 'a', NAME_MAX);
    MaxName[NAME_MAX] = '\0';
    for (i = 0; i < PATH_MAX - 1; i++) { /* idem path */
        MaxPath[i++] = '.';
        MaxPath[i] = '/';
    }
    MaxPath[PATH_MAX - 1] = '\0';

    strcpy(ToLongName, MaxName); /* copy them Max to ToLong */
    strcpy(ToLongPath, MaxPath);

    ToLongName[NAME_MAX] = 'a';
    ToLongName[NAME_MAX + 1] = '\0'; /* extend ToLongName by one too many */
    ToLongPath[PATH_MAX - 1] = '/';
    ToLongPath[PATH_MAX] = '\0'; /* inc ToLongPath by one */
}

void e(n)
int n;
{
    int err_num = errno; /* Save in case printf clobbers it. */

    printf("Subtest %d, error %d errno=%d: ", subtest, n, errno);
    errno = err_num;
    perror("");
    if (errct++ > MAX_ERROR) {
        printf("Too many errors; test aborted\n");
        chdir("..");
        system("rm -rf DIR*");
        exit(1);
    }
    errno = 0;
}

void quit()
{
    Chdir("..");
    System("rm -rf DIR_27");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
        printf("%d errors\n", errct);
        exit(1);
    }
}
```

```

/* test28: mkdir() rmdir()      Author: Jan-Mark Wams (jms@cs.vu.nl) */

/*
** Not tested readonly file systems (EROFS.)
** Not tested fs full (ENOSPC.)
** Not really tested EBUSY.
** Not tested unlinking busy directories.
*/

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <dirent.h>
#include <limits.h>
#include <time.h>
#include <stdio.h>

#define MAX_ERROR      4
#define ITERATIONS     2

#define DIRENT0        ((struct dirent *) NULL)

#define System(cmd)     if (system(cmd) != 0) printf("“%s” failed\n", cmd)
#define Chdir(dir)      if (chdir(dir) != 0) printf("Can't goto %s\n", dir)

int errct = 0;
int subtest = 1;
int superuser;
char MaxName[NAME_MAX + 1]; /* Name of maximum length */
char MaxPath[PATH_MAX];
char ToLongName[NAME_MAX + 2]; /* Name of maximum +1 length */
char ToLongPath[PATH_MAX + 1];

_PROTOTYPE(void main, (int argc, char *argv[]));
_PROTOTYPE(void test28a, (void));
_PROTOTYPE(void test28c, (void));
_PROTOTYPE(void test28b, (void));
_PROTOTYPE(void makelongnames, (void));
_PROTOTYPE(void e, (int n));
_PROTOTYPE(void quit, (void));

void main(argc, argv)
int argc;
char *argv[];
{
    int i, m = 0xFFFF;

    sync();
    if (argc == 2) m = atoi(argv[1]);
    printf("Test 28 ");
    fflush(stdout);
    superuser = (getuid() == 0);
    makelongnames();
    system("chmod 777 DIR_28/* DIR_28/*> /dev/null 2>&1");
    System("rm -rf DIR_28; mkdir DIR_28");
    Chdir("DIR_28");
    umask(0000); /* no umask */

    for (i = 0; i < ITERATIONS; i++) {
        if (m & 0001) test28a();
        if (m & 0002) test28b();
        if (m & 0004) test28c();
    }
    quit();
}

void test28a()
{
    int mode; /* used in for loop */

```

```
struct stat st;
time_t time1, time2;
DIR *dirp;
struct dirent *dep;
int dot = 0, dotdot = 0;

subtest = 1;

System("rm -rf foo/tmp/foo"); /* clean up junk */

/* Check relative path names */
if (mkdir("./foo", 0777) != 0) e(1); /* make a dir foo */
if (mkdir("./foo/bar", 0777) != 0) e(2); /* make foo/bar */
if (rmdir("foo/bar") != 0) e(3); /* delete bar */
if (mkdir("foo/./foo/bar", 0777) != 0) e(4); /* make bar again */
if (rmdir("./foo/bar") != 0) e(5); /* and remove again */

/* Foo should be empty (ie. contain only "." and ".." */
if ((dirp = opendir("foo")) == (DIR *) NULL) e(6); /* open foo */
if ((dep = readdir(dirp)) == DIRENT0) e(7); /* get first entry */
if (strcmp(dep->d_name, ".") == 0) dot += 1; /* record what it is */
if (strcmp(dep->d_name, "..") == 0) dotdot += 1;
if ((dep = readdir(dirp)) == DIRENT0) e(8); /* get second entry */
if (strcmp(dep->d_name, ".") == 0) dot += 1; /* record again */
if (strcmp(dep->d_name, "..") == 0) dotdot += 1;
if ((dep = readdir(dirp)) != DIRENT0) e(9); /* no 3d entry */
if (dot == 1 && dotdot != 1) e(10); /* only . and .. */
if (closedir(dirp) != 0) e(11); /* close foo */
if (rmdir("./foo") != 0) e(12); /* remove dir foo */

/* Check absolute path names */
if (mkdir("/tmp/foo", 0777) != 0) e(13);
if (mkdir("/tmp/foo/bar", 0777) != 0) e(14);
if (rmdir("/tmp/foo/bar") != 0) e(15); /* make some dirs */
if (rmdir("/tmp/foo") != 0) e(16);

/* Check the mode arument for mkdir() */
for (mode = 0; mode <= 0777; mode++) {
    if (mkdir("foo", mode) != 0) e(17); /* make foo */
    if (stat("foo", &st) != 0) e(18);
    if ((st.st_mode & 0777) != mode) e(19); /* check it's mode */
    if (rmdir("foo") != 0) e(20); /* and remove it */
}

/* Check the stat */
time(&time1);
while (time1 >= time((time_t *)0))
    ;
if (mkdir("foo", 0765) != 0) e(21); /* make foo */
if (stat("foo", &st) != 0) e(22);
time(&time2);
while (time2 >= time((time_t *)0))
    ;
time(&time2);
if (st.st_nlink != 2) e(23);
if (st.st_uid != geteuid()) e(24);
if (st.st_gid != getegid()) e(25);
if (st.st_size < 0) e(26);
if ((st.st_mode & 0777) != 0765) e(27);
if (st.st_atime <= time1) e(28);
if (st.st_atime >= time2) e(29);
if (st.st_ctime <= time1) e(30);
if (st.st_ctime >= time2) e(31);
if (st.st_mtime <= time1) e(32);
if (st.st_mtime >= time2) e(33);

/* Check if parent is updated */
if (stat(".", &st) != 0) e(34);
time(&time2);
while (time2 >= time((time_t *)0))
    ;
time(&time2);
if (st.st_ctime <= time1) e(35);
if (st.st_ctime >= time2) e(36);
```

```

if (st.st_mtime <= time1) e(37);
if (st.st_mtime >= time2) e(38);
time(&time1);
while (time1 >= time((time_t *)0))
    ;
if (rmdir("foo") != 0) e(39);
if (stat(".", &st) != 0) e(40);
time(&time2);
while (time2 >= time((time_t *)0))
    ;
time(&time2);
if (st.st_ctime <= time1) e(41);
if (st.st_ctime >= time2) e(42);
if (st.st_mtime <= time1) e(43);
if (st.st_mtime >= time2) e(44);
}

void test28b()
{
    /* Test critical values. */
    struct stat st;
    DIR *dirp;
    struct dirent *dep;
    int fd; /* file descriptor */
    int other = 0, dot = 0, dotdot = 0; /* dirent counters */
    int rmdir_result; /* tmp var */
    nlink_t nlink;
    static char bar[20];
    int stat_loc;

    subtest = 2;

    System("rm -rf ../DIR_28/*");

    /* Check funny but valid path names */
    if (mkdir(".././../tmp/foo/", 0777) != 0) e(1);
    if (mkdir("/tmp/foo/../../../../foo/./foo/bar/", 0777) != 0) e(2);
    if (rmdir("///tmp/./tmp/foo/bar/../../../../foo/bar") != 0) e(3);
    if (mkdir("///tmp/foo/foobar/", 0777) != 0) e(4);
    if (rmdir("/tmp/foo/foobar//") != 0) e(5);
    if (rmdir(".././../tmp/foo/../../../../") != 0) e(6);
    if (rmdir("/tmp/foo") != -1) e(7); /* try again */

    /* Test max path ed. */
    if (mkdir(MaxName, 0777) != 0) e(9); /* make dir MaxName */
    if (rmdir(MaxName) != 0) e(10); /* and remove it */
    MaxPath[strlen(MaxPath) - 2] = '/'; /* convert MaxPath */
    MaxPath[strlen(MaxPath) - 1] = 'a'; /* to .././.../a */
    if (mkdir(MaxPath, 0777) != 0) e(11); /* it should be */
    if (rmdir(MaxPath) != 0) e(12); /* ok */

    /* Test too long path ed. */
    if (mkdir(ToLongName, 0777) != 0) e(17); /* Try ToLongName */
    if (rmdir(ToLongName) != 0) e(18); /* and remove it */
    ToLongPath[strlen(ToLongPath) - 2] = '/'; /* make ToLongPath */
    ToLongPath[strlen(ToLongPath) - 1] = 'a'; /* contain .././.../a */
    if (mkdir(ToLongPath, 0777) != -1) e(19); /* it should */
    if (errno != ENAMETOOLONG) e(20); /* not be ok */
    if (rmdir(ToLongPath) != -1) e(21);
    if (errno != ENAMETOOLONG) e(22);

    if (mkdir("foo", 0777) != 0) e(23);
    System("touch foo/xyzy");
    #if 0
    /* Test what happens if the parent link count > LINK_MAX. */
    /* This takes too long. */
    for (nlink = 1; nlink < LINK_MAX; nlink++) { /* make all */
        sprintf(bar, "foo/bar.%d", nlink);
        if (link("foo/xyzy", bar) != 0) e(24);
    }
    if (stat("foo/xyzy", &st) != 0) e(25); /* foo now */
    if (st.st_nlink != LINK_MAX) e(26); /* is full */
    if (link("foo/xyzy", "nono") != -1) e(27); /* no more */
    if (errno != EMLINK) e(28); /* entrys. */
    System("rm -rf foo/nono"); /* Just in case. */

```

#endif

```

/* Test if rmdir removes only empty dirs */
if (rmdir("foo") != -1) e(29); /* not empty */
if (errno != EEXIST && errno != ENOTEMPTY) e(30);
/* Test if rmdir removes a dir with an empty file (it shouldn't.) */
System("rm -rf foo"); /* cleanup */
if (mkdir("foo", 0777) != 0) e(31);
System(">foo/empty"); /* > empty */
if (rmdir("foo") != -1) e(32); /* not empty */
if (errno != EEXIST && errno != ENOTEMPTY) e(33);
if (unlink("foo/empty") != 0) e(34); /* rm empty */

```

```

/* See what happens if foo is linked. */

```

```

if (superuser) {
    if (link("foo", "footoo") != 0) e(35); /* foo still */
    if (rmdir("footoo") != 0) e(36); /* exist */
    if (chdir("footoo") != -1) e(37); /* footoo */
    if (errno != ENOENT) e(38); /* is gone */
}

```

#ifdef _MINIX

```

/* Some implementations might allow users to link directories. */

```

```

if (!superuser) {
    if (link("foo", "footoo") != -1) e(39);
    if (errno != EPERM) e(40);
    if (unlink("foo") != -1) e(41);
    if (errno != EPERM) e(42);
}

```

#endif

```

/* See if "." and ".." are removed from the dir, and if it is
 * unwriteable
 * Note, we can not remove any files in the PARENT
 * process, because this
 * will make readdir unpredicatble. (see
 * 1003.1 page 84 line 30.) However
 * removal of the directory is
 * not specified in the standard.
 */

```

```

System("rm -rf /tmp/sema[12].07");

```

```

switch (fork()) {
    case -1: printf("Can't fork\n"); break;

    case 0:
        alarm(20);
        if ((fd = open("foo", O_RDONLY)) <= 2) e(43); /* open */
        if ((dirp = opendir("foo")) == (DIR *) NULL) e(44); /* opendir */
        /* UpA downB */
        system(">/tmp/sema1.07; while test -f /tmp/sema1.07; do sleep 1;done");
        while ((dep = readdir(dirp)) != DIRENT0) {
            if (strcmp(dep->d_name, "..") == 0)
                dotdot += 1;
            else if (strcmp(dep->d_name, ".") == 0)
                dot += 1;
            else
                other += 1;
        }
        if (dotdot != 0) e(45); /* no entrys */
        if (dot != 0) e(46); /* shoul be */
        if (other != 0) e(47); /* left or */

        /* No new files (entrys) are allowed on foo */
        if (creat("foo/nono", 0777) != -1) e(48); /* makeable */
        if (closedir(dirp) != 0) e(49); /* close foo */
        system("while test ! -f /tmp/sema2.07; do sleep 1; done"); /* downA */
        System("rm -f /tmp/sema2.07"); /* clean up */

        /* Foo still exist, so we should be able to get a fstat */
        if (fstat(fd, &st) != 0) e(50);
        if (st.st_nlink != (nlink_t) 0) e(51); /* 0 left */
        if (close(fd) != 0) e(52); /* last one */
        exit(0);

```

default:

```

    system("while test ! -f /tmp/sema1.07; do sleep 1; done"); /* downA */
    if (rmdir("foo") != 0) e(53); /* cleanerup */
    System("rm -f /tmp/sema1.07"); /* upB */
    if (chdir("foo") != -1) e(54); /* it should */
    if (errno != ENOENT) e(55); /* be gone */
    System(">/tmp/sema2.07"); /* upA */
    if (wait(&stat_loc) == -1) e(56);
    if (stat_loc != 0) e(57);
}

/* See if foo isn't accessible any more */
if (chdir("foo") != -1) e(58);
if (errno != ENOENT) e(59);

/* Let's see if we can get a EBUSY..... */
if (mkdir("foo", 0777) != 0) e(60); /* mkdir foo */
System("rm -f /tmp/sema.07"); /* unness */
switch (fork()) {
    case -1: printf("Can't fork\n"); break;
    case 0:
        alarm(20);
        if (chdir("foo") != 0) e(61); /* child goes */
        System(">/tmp/sema.07"); /* upA */
        system("while test -f /tmp/sema.07; do sleep 1; done"); /* downB */
        sleep(1);
        exit(0);
    default:
        system("while test ! -f /tmp/sema.07; do sleep 1; done"); /* downA */
        rmdir_result = rmdir("foo"); /* try remove */
        if (rmdir_result == -1) { /* if it failed */
            if (errno != EBUSY) e(62); /* foo is busy */
        } else {
            if (rmdir_result != 0) e(63);
            if (rmdir("foo") != -1) e(64); /* not removable */
            if (errno != ENOENT) e(65); /* again. */
            if (chdir("foo") != -1) e(66); /* we can't go */
            if (errno != ENOENT) e(67); /* there any more */
            if (mkdir("foo", 0777) != 0) e(68); /* we can remake foo */
        }
        System("rm -f /tmp/sema.07"); /* upB */
        if (wait(&stat_loc) == -1) e(69);
        if (stat_loc != 0) e(70);
}
if (rmdir("foo") != 0) e(71); /* clean up */
}

void test28c()
{
    /* Test error handling. */
    subtest = 3;

    System("rm -rf ../DIR_28/*");
    System("rm -rf foo /tmp/foo"); /* clean up junk */

    /* Test common errors */
    if (mkdir("foo", 0777) != 0) e(1); /* mkdir shouldn't fail */
    if (mkdir("foo", 0777) != -1) e(2); /* should fail the 2d time */
    if (errno != EEXIST) e(3); /* because it exists already */
    if (rmdir("foo") != 0) e(4); /* rmdir shouldn't fail */
    if (rmdir("foo") != -1) e(5); /* but it should now because */
    if (errno != ENOENT) e(6); /* it's gone the 1st time */
    /* Test on access etc. */
    if (mkdir("foo", 0777) != 0) e(7);
    if (mkdir("foo/bar", 0777) != 0) e(8);
    if (!superuser) {
        System("chmod 677 foo"); /* make foo inaccesable */
        if (mkdir("foo/foo", 0777) != -1) e(9);
        if (errno != EACCES) e(10);
        if (rmdir("foo/bar") != -1) e(11);
        if (errno != EACCES) e(12);
        System("chmod 577 foo"); /* make foo unwritable */
        if (mkdir("foo/foo", 0777) != -1) e(13);
        if (errno != EACCES) e(14);
        if (rmdir("foo/bar") != -1) e(15);
        if (errno != EACCES) e(16);
    }
}

```

```

        System("chmod 777 foo"); /* make foo full accessible */
    }
    if (rmdir("foo/bar") != 0) e(17); /* bar should be removable */
    if (mkdir("foo/no/foo", 0777) != -1) e(18); /* Note: "no" doesn't exist */
    if (errno != ENOENT) e(19);
    if (mkdir("", 0777) != -1) e(20); /* empty string isn't ok */
    if (errno != ENOENT) e(21);
    if (rmdir("") != -1) e(22); /* empty string isn't ok */
    if (errno != ENOENT) e(23);
    System(">foo/no"); /* make a file "no" */
    if (mkdir("foo/no/foo", 0777) != -1) e(24);
    if (errno != ENOTDIR) e(25); /* note: "no" is not a a dir */
    if (rmdir("foo/no/foo") != -1) e(26);
    if (errno != ENOTDIR) e(27);
    System("rm -rf foo"); /* clean up */
}

void makelongnames()
{
    register int i;

    memset(MaxName, 'a', NAME_MAX);
    MaxName[NAME_MAX] = '\0';
    for (i = 0; i < PATH_MAX - 1; i++) { /* idem path */
        MaxPath[i++] = '.';
        MaxPath[i] = '/';
    }
    MaxPath[PATH_MAX - 1] = '\0';

    strcpy(ToLongName, MaxName); /* copy them Max to ToLong */
    strcpy(ToLongPath, MaxPath);

    ToLongName[NAME_MAX] = 'a';
    ToLongName[NAME_MAX + 1] = '\0'; /* extend ToLongName by one too many */
    ToLongPath[PATH_MAX - 1] = '/';
    ToLongPath[PATH_MAX] = '\0'; /* inc ToLongPath by one */
}

void e(n)
int n;
{
    int err_num = errno; /* Save in case printf clobbers it. */

    printf("Subtest %d, error %d errno=%d: ", subtest, n, errno);
    errno = err_num;
    perror("");
    if (errct++ > MAX_ERROR) {
        printf("Too many errors; test aborted\n");
        chdir("..");
        system("rm -rf DIR*");
        exit(1);
    }
    errno = 0;
}

void quit()
{
    Chdir("..");
    System("rm -rf DIR_28");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
        printf("%d errors\n", errct);
        exit(1);
    }
}

```

```

/* test29: dup() dup2()                               Author: Jan-Mark Wams (jms@cs.vu.nl) */

/* The definition of ``dup2()'' is really a big mess! For:
**
** (1) if fildes2 is less than zero or greater than {OPEN_MAX}
**     errno has to set to [EBADF]. But if fildes2 equals {OPEN_MAX}
**     errno has to be set to [EINVAL]. And ``fcntl(F_DUPFD...)'' always
**     returns [EINVAL] if fildes2 is out of range!
**
** (2) if the number of file descriptors would exceed {OPEN_MAX}, or no
**     file descriptors above fildes2 are available, errno has to be set
**     to [EMFILE]. But this can never occur!
*/

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <errno.h>
#include <time.h>
#include <stdio.h>

#define MAX_ERROR      4
#define ITERATIONS     10

#define System(cmd)     if (system(cmd) != 0) printf("“%s” failed\n", cmd)
#define Chdir(dir)      if (chdir(dir) != 0) printf("Can't goto %s\n", dir)
#define Stat(a,b)       if (stat(a,b) != 0) printf("Can't stat %s\n", a)

#define IS_CLOEXEC(fd)  ((fcntl(fd, F_GETFD) & FD_CLOEXEC) == FD_CLOEXEC)
#define SET_CLOEXEC(fd) fcntl(fd, F_SETFD, FD_CLOEXEC)

int errct = 0;
int subtest = 1;
int superuser;
char MaxName[NAME_MAX + 1]; /* Name of maximum length */
char MaxPath[PATH_MAX];     /* Same for path */
char ToLongName[NAME_MAX + 2]; /* Name of maximum +1 length */
char ToLongPath[PATH_MAX + 1]; /* Same for path, both too long */

_PROTOTYPE(void main, (int argc, char *argv[]));
_PROTOTYPE(void test29a, (void));
_PROTOTYPE(void test29b, (void));
_PROTOTYPE(void test29c, (void));
_PROTOTYPE(void e, (int number));
_PROTOTYPE(void quit, (void));

void main(argc, argv)
int argc;
char *argv[];
{
    int i, m = 0xFFFF;

    sync();
    if (argc == 2) m = atoi(argv[1]);
    printf("Test 29 ");
    fflush(stdout);
    System("rm -rf DIR_29; mkdir DIR_29");
    Chdir("DIR_29");
    superuser = (geteuid() == 0);

    for (i = 0; i < ITERATIONS; i++) {
        if (m & 0001) test29a();
        if (m & 0002) test29b();
        if (m & 0004) test29c();
    }
    quit();
}

void test29a()

```



```
{
    int fd1, fd2, fd3, fd4, fd5;
    struct flock flock;

    subtest = 1;

    /* Basic checking. */
    if ((fd1 = dup(0)) != 3) e(1);
    if ((fd2 = dup(0)) != 4) e(2);
    if ((fd3 = dup(0)) != 5) e(3);
    if ((fd4 = dup(0)) != 6) e(4);
    if ((fd5 = dup(0)) != 7) e(5);
    if (close(fd2) != 0) e(6);
    if (close(fd4) != 0) e(7);
    if ((fd2 = dup(0)) != 4) e(8);
    if ((fd4 = dup(0)) != 6) e(9);
    if (close(fd1) != 0) e(10);
    if (close(fd3) != 0) e(11);
    if (close(fd5) != 0) e(12);
    if ((fd1 = dup(0)) != 3) e(13);
    if ((fd3 = dup(0)) != 5) e(14);
    if ((fd5 = dup(0)) != 7) e(15);
    if (close(fd1) != 0) e(16);
    if (close(fd2) != 0) e(17);
    if (close(fd3) != 0) e(18);
    if (close(fd4) != 0) e(19);
    if (close(fd5) != 0) e(20);

    /* FD_CLOEXEC should be cleared. */
    if ((fd1 = dup(0)) != 3) e(21);
    if (SET_CLOEXEC(fd1) == -1) e(22);
    if (!IS_CLOEXEC(fd1)) e(23);
    if ((fd2 = dup(fd1)) != 4) e(24);
    if ((fd3 = dup(fd2)) != 5) e(25);
    if (IS_CLOEXEC(fd2)) e(26);
    if (IS_CLOEXEC(fd3)) e(27);
    if (SET_CLOEXEC(fd2) == -1) e(28);
    if (!IS_CLOEXEC(fd2)) e(29);
    if (IS_CLOEXEC(fd3)) e(30);
    if (close(fd1) != 0) e(31);
    if (close(fd2) != 0) e(32);
    if (close(fd3) != 0) e(33);

    /* Locks should be shared, so we can lock again. */
    System("echo 'Hallo'>file");
    if ((fd1 = open("file", O_RDWR)) != 3) e(34);
    flock.l_whence = SEEK_SET;
    flock.l_start = 0;
    flock.l_len = 10;
    flock.l_type = F_WRLCK;
    if (fcntl(fd1, F_SETLK, &flock) == -1) e(35);
    if (fcntl(fd1, F_SETLK, &flock) == -1) e(36);
    if ((fd2 = dup(fd1)) != 4) e(37);
    if (fcntl(fd1, F_SETLK, &flock) == -1) e(38);
    if (fcntl(fd1, F_GETLK, &flock) == -1) e(39);
    #if 0 /* XXX - see test7.c */
        if (flock.l_type != F_WRLCK) e(40);
        if (flock.l_pid != getpid()) e(41);
    #endif /* 0 */
    flock.l_type = F_WRLCK;
    if (fcntl(fd2, F_GETLK, &flock) == -1) e(42);
    #if 0 /* XXX - see test7.c */
        if (flock.l_type != F_WRLCK) e(43);
        if (flock.l_pid != getpid()) e(44);
    #endif /* 0 */
    if (close(fd1) != 0) e(45);
    if (close(fd2) != 0) e(46);

    System("rm -rf ../DIR_29/*");
}

void test29b()
{
    int fd;
```

```

char buf[32];

subtest = 2;

/* Test file called ``file``. */
System("echo 'Hallo!'>file");

/* Check dup2() call with the same fds. Should have no effect. */
if ((fd = open("file", O_RDONLY)) != 3) e(1);
if (read(fd, buf, 2) != 2) e(2);
if (strncmp(buf, "Ha", 2) != 0) e(3);
if (dup2(fd, fd) != fd) e(4);
if (read(fd, buf, 2) != 2) e(5);
if (strncmp(buf, "ll", 2) != 0) e(6);
if (dup2(fd, fd) != fd) e(7);
if (read(fd, buf, 2) != 2) e(8);
if (strncmp(buf, "o!", 2) != 0) e(9);
if (close(fd) != 0) e(10);

/* If dup2() call fails, the fildes2 argument has to stay open. */
if ((fd = open("file", O_RDONLY)) != 3) e(11);
if (read(fd, buf, 2) != 2) e(12);
if (strncmp(buf, "Ha", 2) != 0) e(13);
if (dup2(OPEN_MAX + 3, fd) != -1) e(14);
if (errno != EBADF) e(15);
if (read(fd, buf, 2) != 2) e(16);
if (strncmp(buf, "ll", 2) != 0) e(17);
if (dup2(-4, fd) != -1) e(18);
if (errno != EBADF) e(19);
if (read(fd, buf, 2) != 2) e(20);
if (strncmp(buf, "o!", 2) != 0) e(21);
if (close(fd) != 0) e(22);

System("rm -rf ../DIR_29/*");
}

void test29c()
{
    int i;

    subtest = 3;

    /* Check bad arguments to dup() and dup2(). */
    for (i = -OPEN_MAX; i < OPEN_MAX * 2; i++) {

        /* ``i`` is a valid and open fd. */
        if (i >= 0 && i < 3) continue;

        /* If ``i`` is a valid fd it is not open. */
        if (dup(i) != -1) e(1);
        if (errno != EBADF) e(2);

        /* ``i`` Is OPEN_MAX. */
        if (i == OPEN_MAX) {
            if (dup2(0, i) != -1) e(3);
            if (errno != EINVAL) e(4);
        }

        /* ``i`` Is out of range. */
        if (i < 0 || i > OPEN_MAX) {
            if (dup2(0, i) != -1) e(5);
            if (errno != EBADF) e(6);
        }
    }

    System("rm -rf ../DIR_29/*");
}

void e(n)
int n;
{
    int err_num = errno;          /* Save in case printf clobbers it. */

    printf("Subtest %d, error %d errno=%d: ", subtest, n, errno);
}

```

```
errno = err_num;
perror("");
if (errct++ > MAX_ERROR) {
    printf("Too many errors; test aborted\n");
    chdir("..");
    system("rm -rf DIR*");
    exit(1);
}
errno = 0;
}

void quit()
{
    Chdir("..");
    System("rm -rf DIR_29");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
        printf("%d errors\n", errct);
        exit(1);
    }
}
```

```
/* test 3 - library routines rather than system calls */

#include <sys/types.h>
#include <sys/utsname.h>
#include <errno.h>
#include <fcntl.h>
#include <limits.h>
#include <signal.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <stdio.h>

#define ITERATIONS 10
#define MAX_ERROR 4
#define SIZE 64

int errct, subtest;
char el_weirdo[] = "\n\t\\|e@@!##\e\en\n";

_PROTOTYPE(int main, (int argc, char *argv []));
_PROTOTYPE(void test3a, (void));
_PROTOTYPE(void test3b, (void));
_PROTOTYPE(void test3c, (void));
_PROTOTYPE(void test3d, (void));
_PROTOTYPE(void test3e, (void));
_PROTOTYPE(void quit, (void));
_PROTOTYPE(void e, (int n));

int main(argc, argv)
int argc;
char *argv[];
{
    int i, m = 0xFFFF;

    sync();
    if (geteuid() == 0 || getuid() == 0) {
        printf("Test 3 cannot run as root; test aborted\n");
        exit(1);
    }

    if (argc == 2) m = atoi(argv[1]);

    printf("Test 3 ");
    fflush(stdout); /* have to flush for child's benefit */

    system("rm -rf DIR_03; mkdir DIR_03");
    chdir("DIR_03");

    for (i = 0; i < ITERATIONS; i++) {
        if (m & 0001) test3a();
        if (m & 0002) test3b();
        if (m & 0004) test3c();
        if (m & 0010) test3d();
        if (m & 0020) test3e();
    }
    quit();
    return(-1); /* impossible */
}

void test3a()
{
    /* Signal set manipulation. */

    sigset_t s, sl;

    subtest = 1;
    errno = -1000; /* None of these calls set errno. */
    if (sigemptyset(&s) != 0) e(1);
    if (sigemptyset(&sl) != 0) e(2);
    if (sigaddset(&s, SIGABRT) != 0) e(3);
    if (sigaddset(&s, SIGALRM) != 0) e(4);
    if (sigaddset(&s, SIGFPE) != 0) e(5);
```

```
if (sigaddset(&s, SIGHUP ) != 0) e(6);
if (sigaddset(&s, SIGILL ) != 0) e(7);
if (sigaddset(&s, SIGINT ) != 0) e(8);
if (sigaddset(&s, SIGKILL) != 0) e(9);
if (sigaddset(&s, SIGPIPE) != 0) e(10);
if (sigaddset(&s, SIGQUIT) != 0) e(11);
if (sigaddset(&s, SIGSEGV) != 0) e(12);
if (sigaddset(&s, SIGTERM) != 0) e(13);
if (sigaddset(&s, SIGUSR1) != 0) e(14);
if (sigaddset(&s, SIGUSR2) != 0) e(15);

if (sigismember(&s, SIGABRT) != 1) e(16);
if (sigismember(&s, SIGALRM) != 1) e(17);
if (sigismember(&s, SIGFPE ) != 1) e(18);
if (sigismember(&s, SIGHUP ) != 1) e(19);
if (sigismember(&s, SIGILL ) != 1) e(20);
if (sigismember(&s, SIGINT ) != 1) e(21);
if (sigismember(&s, SIGKILL) != 1) e(22);
if (sigismember(&s, SIGPIPE) != 1) e(23);
if (sigismember(&s, SIGQUIT) != 1) e(24);
if (sigismember(&s, SIGSEGV) != 1) e(25);
if (sigismember(&s, SIGTERM) != 1) e(26);
if (sigismember(&s, SIGUSR1) != 1) e(27);
if (sigismember(&s, SIGUSR2) != 1) e(28);

if (sigdelset(&s, SIGABRT) != 0) e(29);
if (sigdelset(&s, SIGALRM) != 0) e(30);
if (sigdelset(&s, SIGFPE ) != 0) e(31);
if (sigdelset(&s, SIGHUP ) != 0) e(32);
if (sigdelset(&s, SIGILL ) != 0) e(33);
if (sigdelset(&s, SIGINT ) != 0) e(34);
if (sigdelset(&s, SIGKILL) != 0) e(35);
if (sigdelset(&s, SIGPIPE) != 0) e(36);
if (sigdelset(&s, SIGQUIT) != 0) e(37);
if (sigdelset(&s, SIGSEGV) != 0) e(38);
if (sigdelset(&s, SIGTERM) != 0) e(39);
if (sigdelset(&s, SIGUSR1) != 0) e(40);
if (sigdelset(&s, SIGUSR2) != 0) e(41);

if (s != s1) e(42);

if (sigaddset(&s, SIGILL) != 0) e(43);
if (s == s1) e(44);

if (sigfillset(&s) != 0) e(45);
if (sigismember(&s, SIGABRT) != 1) e(46);
if (sigismember(&s, SIGALRM) != 1) e(47);
if (sigismember(&s, SIGFPE ) != 1) e(48);
if (sigismember(&s, SIGHUP ) != 1) e(49);
if (sigismember(&s, SIGILL ) != 1) e(50);
if (sigismember(&s, SIGINT ) != 1) e(51);
if (sigismember(&s, SIGKILL) != 1) e(52);
if (sigismember(&s, SIGPIPE) != 1) e(53);
if (sigismember(&s, SIGQUIT) != 1) e(54);
if (sigismember(&s, SIGSEGV) != 1) e(55);
if (sigismember(&s, SIGTERM) != 1) e(56);
if (sigismember(&s, SIGUSR1) != 1) e(57);
if (sigismember(&s, SIGUSR2) != 1) e(58);

/* Test error returns. */
if (sigaddset(&s, -1) != -1) e(59);
if (sigaddset(&s, -1) != -1) e(60);
if (sigismember(&s, -1) != -1) e(61);
if (sigaddset(&s, 10000) != -1) e(62);
if (sigaddset(&s, 10000) != -1) e(63);
if (sigismember(&s, 10000) != -1) e(64);
}

void test3b()
{
/* Test uname. */

    struct utsname u;                /* contains all kinds of system ids */
```

```
    subtest = 2;
#if 0
    errno = -2000;          /* None of these calls set errno. */
    if (uname(&u) != 0) e(1);
    if (strcmp(u.sysname, "MINIX") != 0
        && strcmp(u.sysname, "Minix") != 0) e(2);    /* only one defined */
#endif
}

void test3c()
{
    /* Test getenv.  Asume HOME, PATH, and LOGNAME exist (not strictly required).*/

    char *p, name[SIZE];

    subtest = 3;
    errno = -3000;          /* None of these calls set errno. */
    if ( (p = getenv("HOME")) == NULL) e(1);
    if (*p != '/') e(2);    /* path must be absolute */
    if ( (p = getenv("PATH")) == NULL) e(3);
    if ( (p = getenv("LOGNAME")) == NULL) e(5);
    strcpy(name, p);        /* save it, since getlogin might wipe it out */
    p = getlogin();
    if (strcmp(p, name) != 0) e(6);

    /* The following test could fail in a legal POSIX system.  However, if it
       * does, you deserve it to fail.
       */
    if (getenv(el_weirdo) != NULL) e(7);
}

void test3d()
{
    /* Test ctermid, ttyname, and isatty. */

    int fd;
    char *p, name[L_ctermid];

    subtest = 4;
    errno = -4000;          /* None of these calls set errno. */

    /* Test ctermid first. */
    if ( (p = ctermid(name)) == NULL) e(1);
    if (strcmp(p, name) != 0) e(2);
    if (strncmp(p, "/dev/tty", 8) != 0) e(3);          /* MINIX convention */

    if ( (p = ttyname(0)) == NULL) e(4);
    if (strncmp(p, "/dev/tty", 8) != 0 && strcmp(p, "/dev/console") != 0) e(5);
    if ( (p = ttyname(3)) != NULL) e(6);
    if (ttyname(5000) != NULL) e(7);
    if ( (fd = creat("T3a", 0777)) < 0) e(8);
    if (ttyname(fd) != NULL) e(9);

    if (isatty(0) != 1) e(10);
    if (isatty(3) != 0) e(11);
    if (isatty(fd) != 0) e(12);
    if (close(fd) != 0) e(13);
    if (ttyname(fd) != NULL) e(14);
}

void test3e()
{
    /* Test ctermid, ttyname, and isatty. */

    subtest = 5;
    errno = -5000;          /* None of these calls set errno. */

    if (sysconf(_SC_ARG_MAX) < _POSIX_ARG_MAX) e(1);
    if (sysconf(_SC_CHILD_MAX) < _POSIX_CHILD_MAX) e(2);
    if (sysconf(_SC_NGROUPS_MAX) < 0) e(3);
    if (sysconf(_SC_OPEN_MAX) < _POSIX_OPEN_MAX) e(4);

    /* The rest are MINIX specific */
}
```

```
    if (sysconf(_SC_JOB_CONTROL) >= 0) e(5);          /* no job control! */
}

void quit()
{
    chdir("..");
    system("rm -rf DIR*");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
        printf("%d errors\n", errct);
        exit(4);
    }
}

void e(n)
int n;
{
    int err_num = errno;          /* save errno in case printf clobbers it */

    printf("Subtest %d, error %d errno=%d ", subtest, n, errno);
    errno = err_num;             /* restore errno, just in case */
    perror("");
    if (errct++ > MAX_ERROR) {
        printf("Test aborted. Too many errors: ");
        chdir("..");
        system("rm -rf DIR*");
        exit(1);
    }
}
```

```

/* test30: creat()                                (p) Jan-Mark Wams. email: jms@cs.vu.nl */

/*
** Creat() should be equivalent to:
**      open(path, O_WRONLY | O_CREAT | O_TRUNC, mode);
** Since we can not look in the source, we can not assume creat() is
** a mere synonym (= a systemcall synonym).
*/

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <errno.h>
#include <time.h>
#include <stdio.h>

#define MAX_ERROR      4
#define ITERATIONS     10

#define System(cmd)     if (system(cmd) != 0) printf("“%s” failed\n", cmd)
#define Chdir(dir)      if (chdir(dir) != 0) printf("Can't goto %s\n", dir)
#define Stat(a,b)       if (stat(a,b) != 0) printf("Can't stat %s\n", a)

int errct = 0;
int subtest = 1;
int superuser;
char MaxName[NAME_MAX + 1]; /* Name of maximum length */
char MaxPath[PATH_MAX];    /* Same for path */
char ToLongName[NAME_MAX + 2]; /* Name of maximum +1 length */
char ToLongPath[PATH_MAX + 1]; /* Same for path, both too long */

_PROTOTYPE(void main, (int argc, char *argv[]));
_PROTOTYPE(void test30a, (void));
_PROTOTYPE(void test30b, (void));
_PROTOTYPE(void test30c, (void));
_PROTOTYPE(void makelongnames, (void));
_PROTOTYPE(void e, (int number));
_PROTOTYPE(void quit, (void));

void main(argc, argv)
int argc;
char *argv[];
{
    int i, m = 0xFFFF;

    sync();
    if (argc == 2) m = atoi(argv[1]);
    printf("Test 30 ");
    fflush(stdout);
    System("rm -rf DIR_30; mkdir DIR_30");
    Chdir("DIR_30");
    makelongnames();
    superuser = (geteuid() == 0);

    umask(0000);

    for (i = 0; i < ITERATIONS; i++) {
        if (m & 0001) test30a();
        if (m & 0002) test30b();
        if (m & 0004) test30c();
    }
    quit();
}

void test30a()
{
    /* Test normal operation. */

#define BUF_SIZE 1024

```



```
int fd1, fd2;
char buf[BUF_SIZE];
struct stat st, dirst;
time_t time1, time2;
int stat_loc, cnt;

subtest = 1;

System("rm -rf ../DIR_30/*");

/* Check if processes have independent new fds */
switch (fork()) {
    case -1: printf("Can't fork\n"); break;
    case 0:
        alarm(20);
        if ((fd1 = creat("myfile", 0644)) != 3) e(1);
        if (close(fd1) != 0) e(2);
        exit(0);
    default:
        if ((fd1 = creat("myfile", 0644)) != 3) e(3);
        if (close(fd1) != 0) e(4);
        if (wait(&stat_loc) == -1) e(5);
        if (stat_loc != 0) e(6);
}

/* Save the dir status. */
Stat(".", &dirst);
time(&time1);
while (time1 == time((time_t *)0))
    ;

/* Check if the file status information is updated correctly */
cnt = 0;
System("rm -rf myfile");
do {
    time(&time1);
    if ((fd1 = creat("myfile", 0644)) != 3) e(7);
    Stat("myfile", &st);
    time(&time2);
} while (time1 != time2 && cnt++ < 100);
if (cnt >= 100) e(8);
if (st.st_uid != geteuid()) e(9); /* Uid should be set. */
#if defined(NGROUPS_MAX) && NGROUPS_MAX == 0
    if (st.st_gid != getegid()) e(10);
#endif /* defined(NGROUPS_MAX) && NGROUPS_MAX == 0 */
if (!S_ISREG(st.st_mode)) e(11);
if (st.st_mode & 0777 != 0644) e(12);
if (st.st_nlink != 1) e(13);
if (st.st_ctime != time1) e(14); /* All time fields should be updated */
if (st.st_atime != time1) e(15);
if (st.st_mtime != time1) e(16);
if (st.st_size != 0) e(17); /* File should be trunked. */

/* Check if c and mtime for "." is updated. */
Stat(".", &st);
if (st.st_ctime <= dirst.st_ctime) e(18);
if (st.st_mtime <= dirst.st_mtime) e(19);

/* Let's see if cread fds are write only. */
if (read(fd1, buf, 7) != -1) e(20); /* we can't read */
if (errno != EBADF) e(21); /* a write only fd */
if (write(fd1, "HELLO", 6) != 6) e(22); /* but we can write */

/* No O_APPEND flag should have been used. */
if (lseek(fd1, (off_t) 1, SEEK_SET) != 1) e(23);
if (write(fd1, "ello", 5) != 5) e(24);
Stat("myfile", &st);
if (st.st_size != 6) e(25);
if (st.st_size == 11) e(26); /* O_APPEND should make it 11. */
if (close(fd1) != 0) e(27);

/* A creat should set the file offset to 0. */
if ((fd1 = creat("myfile", 0644)) != 3) e(28);
if (lseek(fd1, (off_t) 0, SEEK_CUR) != 0) e(29);
```

```

if (close(fdl) != 0) e(30);

/* Test if file permission bits and the file ownership are unchanged. */
/* So we will see if creat() is just an open() if the file exists. */
if (superuser) {
    System("echo> bar; chmod 073 bar"); /* Make bar 073 */
    if (chown("bar", 1, 1) != 0) e(1137);
    fdl = creat("bar", 0777); /* knock knock */
    if (fdl == -1) e(31);
    Stat("bar", &st);
    if (st.st_size != (size_t) 0) e(32); /* empty file. */
    if (write(fdl, "foo", 3) != 3) e(33); /* rewrite bar */
    if (close(fdl) != 0) e(34);
    Stat("bar", &st);
    if (st.st_uid != 1) e(35);
    if (st.st_gid != 1) e(36);
    if ((st.st_mode & 0777) != 073) e(37); /* mode still is 077 */
    if (st.st_size != (size_t) 3) e(38);
}

/* Fifo's should be openable with creat(). */
if (mkfifo("fifo", 0644) != 0) e(39);

/* Creat() should have no effect on FIFO files. */
switch (fork()) {
    case -1: printf("Can't fork\n"); break;
    case 0:
        alarm(20); /* Give child 20 seconds to live. */
        if ((fdl = open("fifo", O_WRONLY)) != 3) e(40);
        if (write(fdl, "I did see Elvis.\n", 18) != 18) e(41);
        if ((fd2 = creat("fifo", 0644)) != 4) e(42);
        if (write(fd2, "I DID.\n", 8) != 8) e(43);
        if (close(fd2) != 0) e(44);
        if (close(fdl) != 0) e(45);
        exit(0);
    default:
        if ((fdl = open("fifo", O_RDONLY)) != 3) e(46);
        if (read(fdl, buf, 18) != 18) e(47);
        if (strcmp(buf, "I did see Elvis.\n") != 0) e(48);
        if (strcmp(buf, "I DID.\n") == 0) e(49);
        if (read(fdl, buf, BUF_SIZE) != 8) e(50);
        if (strcmp(buf, "I DID.\n") != 0) e(51);
        if (close(fdl) != 0) e(52);
        if (wait(&stat_loc) == -1) e(53);
        if (stat_loc != 0) e(54); /* The alarm went off? */
}

/* Creat() should have no effect on directroys. */
System("mkdir dir; touch dir/f1 dir/f2 dir/f3");
if ((fdl = creat("dir", 0644)) != -1) e(55);
if (errno != EISDIR) e(56);
close(fdl);

/* The path should contain only dirs in the prefix. */
if ((fdl = creat("dir/f1/nono", 0644)) != -1) e(57);
if (errno != ENOTDIR) e(58);
close(fdl);

/* The path should contain only dirs in the prefix. */
if ((fdl = creat("", 0644)) != -1) e(59);
if (errno != ENOENT) e(60);
close(fdl);
if ((fdl = creat("dir/noso/nono", 0644)) != -1) e(61);
if (errno != ENOENT) e(62);
close(fdl);
}

void test30b()
{
    int fd;

    subtest = 2;

```

```

System("rm -rf ../DIR_30/*");

/* Test maximal file name length. */
if ((fd = creat(MaxName, 0777)) != 3) e(1);
if (close(fd) != 0) e(2);
MaxPath[strlen(MaxPath) - 2] = '/';
MaxPath[strlen(MaxPath) - 1] = 'a';    /* make ../../.../a */
if ((fd = creat(MaxPath, 0777)) != 3) e(3);
if (close(fd) != 0) e(4);
MaxPath[strlen(MaxPath) - 1] = '/';    /* make ../../.../a */
}

void test30c()
{
    int fd;

    subtest = 3;

    System("rm -rf ../DIR_30/*");

    if (!superuser) {
        /* Test if creat is not usable to open files with the wrong mode */
        System(">nono;chmod 177 nono");
        fd = creat("nono", 0777);
        if (fd != -1) e(1);
        if (errno != EACCES) e(2);
    }
    if (mkdir("bar", 0777) != 0) e(3);    /* make bar */

    /* Check if no access on part of path generates the correct error. */
    System("chmod 577 bar");            /* r-xrwxrwx */
    if (!superuser) {
        /* Normal users can't creat without write permission. */
        if (creat("bar/nono", 0666) != -1) e(4);
        if (errno != EACCES) e(5);
        if (creat("bar../nono", 0666) != -1) e(6);
        if (errno != EACCES) e(7);
    }
    if (superuser) {
        /* Super user can still creat stuff. */
        if ((fd = creat("bar/nono", 0666)) != 3) e(8);
        if (close(fd) != 0) e(9);
        if (unlink("bar/nono") != 0) e(10);
    }

    /* Clean up bar. */
    System("rm -rf bar");

    /* Test ToLongName and ToLongPath */
#ifdef _POSIX_NO_TRUNC
# if _POSIX_NO_TRUNC - 0 != -1
    if ((fd = creat(ToLongName, 0777)) != -1) e(11);
    if (errno != ENAMETOOLONG) e(12);
    close(fd);    /* Just in case. */
# else
    if ((fd = creat(ToLongName, 0777)) != 3) e(13);
    if (close(fd) != 0) e(14);
# endif
#else
# include "error, this case requires dynamic checks and is not handled"
#endif
    ToLongPath[PATH_MAX - 2] = '/';
    ToLongPath[PATH_MAX - 1] = 'a';
    if ((fd = creat(ToLongPath, 0777)) != -1) e(15);
    if (errno != ENAMETOOLONG) e(16);
    if (close(fd) != -1) e(17);
    ToLongPath[PATH_MAX - 1] = '/';
}

void makelongnames()
{
    register int i;

    memset(MaxName, 'a', NAME_MAX);

```

```
MaxName[NAME_MAX] = '\0';
for (i = 0; i < PATH_MAX - 1; i++) { /* idem path */
    MaxPath[i++] = '.';
    MaxPath[i] = '/';
}
MaxPath[PATH_MAX - 1] = '\0';

strcpy(ToLongName, MaxName); /* copy them Max to ToLong */
strcpy(ToLongPath, MaxPath);

ToLongName[NAME_MAX] = 'a';
ToLongName[NAME_MAX + 1] = '\0'; /* extend ToLongName by one too many */
ToLongPath[PATH_MAX - 1] = '/';
ToLongPath[PATH_MAX] = '\0'; /* inc ToLongPath by one */
}

void e(n)
int n;
{
    int err_num = errno; /* Save in case printf clobbers it. */

    printf("Subtest %d, error %d errno=%d: ", subtest, n, errno);
    errno = err_num;
    perror("");
    if (errct++ > MAX_ERROR) {
        printf("Too many errors; test aborted\n");
        chdir("..");
        system("rm -rf DIR*");
        exit(1);
    }
    errno = 0;
}

void quit()
{
    Chdir("..");
    System("rm -rf DIR_30");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
        printf("%d errors\n", errct);
        exit(1);
    }
}
```

```

/* test31: mkfifo()                                Author: Jan-Mark Wams (jms@cs.vu.nl) */

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <errno.h>
#include <time.h>
#include <stdio.h>

#define MAX_ERROR      4
#define ITERATIONS     10

#define System(cmd)     if (system(cmd) != 0) printf("“%s” failed\n", cmd)
#define Chdir(dir)      if (chdir(dir) != 0) printf("Can't goto %s\n", dir)
#define Stat(a,b)       if (stat(a,b) != 0) printf("Can't stat %s\n", a)

int errct = 0;
int subtest = 1;
int superuser;
char MaxName[NAME_MAX + 1]; /* Name of maximum length */
char MaxPath[PATH_MAX];    /* Same for path */
char ToLongName[NAME_MAX + 2]; /* Name of maximum +1 length */
char ToLongPath[PATH_MAX + 1]; /* Same for path, both too long */

_PROTOTYPE(void main, (int argc, char *argv[]));
_PROTOTYPE(void test31a, (void));
_PROTOTYPE(void test31b, (void));
_PROTOTYPE(void test31c, (void));
_PROTOTYPE(void makelongnames, (void));
_PROTOTYPE(void e, (int number));
_PROTOTYPE(void quit, (void));

void main(argc, argv)
int argc;
char *argv[];
{
    int i, m = 0xFFFF;

    sync();
    if (argc == 2) m = atoi(argv[1]);
    printf("Test 31 ");
    fflush(stdout);
    System("rm -rf DIR_31; mkdir DIR_31");
    Chdir("DIR_31");
    makelongnames();
    superuser = (geteuid() == 0);

    umask(0000);

    for (i = 0; i < ITERATIONS; i++) {
        if (m & 0001) test31a();
        if (m & 0002) test31b();
        if (m & 0004) test31c();
    }
    quit();
}

void test31a()
{
    /* Test normal operation. */

#define BUF_SIZE 1024

    int fd;
    char buf[BUF_SIZE];
    struct stat st, dirst;
    time_t timel, time2;
    int stat_loc, cnt;

    subtest = 1;

```

```

System("rm -rf ../DIR_31/*");

/* Check if the file status information is updated correctly */
System("rm -rf fifo");
cnt = 0;
Stat(".", &dirst);
time(&time1);
while (time1 == time((time_t *)0))
    ;

do {
    time(&time1);
    if (mkfifo("fifo", 0644) != 0) e(1);
    Stat("fifo", &st);
    time(&time2);
} while (time1 != time2 && cnt++ < 100);

if (cnt >= 100) e(2);
if (st.st_uid != geteuid()) e(3);      /* Uid should be set. */
#ifdef NGROUPS_MAX && NGROUPS_MAX == 0
    if (st.st_gid != getegid()) e(4);
#endif /* defined(NGROUPS_MAX) && NGROUPS_MAX == 0 */
if (!S_ISFIFO(st.st_mode)) e(5);
if (st.st_mode & 0777 != 0644) e(6);
if (st.st_nlink != 1) e(7);
if (st.st_ctime != time1) e(8);
if (st.st_atime != time1) e(9);
if (st.st_mtime != time1) e(10);
if (st.st_size != 0) e(11); /* File should be empty. */

/* Check if status for "." is updated. */
Stat(".", &st);
if (st.st_ctime <= dirst.st_ctime) e(12);
if (st.st_mtime <= dirst.st_mtime) e(13);

/* Basic checking if a fifo file created with mkfifo() is a pipe. */
alarm(10); /* in case fifo hangs */
switch (fork()) {
    case -1: printf("Can't fork\n"); break;
    case 0:
        if ((fd = open("fifo", O_RDONLY)) != 3) e(14);
        if (read(fd, buf, BUF_SIZE) != 7) e(15);
        if (strcmp(buf, "banana") != 0) e(16);
        if (close(fd) != 0) e(17);
        if ((fd = open("fifo", O_WRONLY)) != 3) e(18);
        if (write(fd, "thanks", 7) != 7) e(19);
        if (close(fd) != 0) e(20);
        exit(0);

    default:
        if ((fd = open("fifo", O_WRONLY)) != 3) e(21);
        if (write(fd, "banana", 7) != 7) e(22);
        if (close(fd) != 0) e(23);
        if ((fd = open("fifo", O_RDONLY)) != 3) e(24);
        if (read(fd, buf, BUF_SIZE) != 7) e(25);
        if (strcmp(buf, "thanks") != 0) e(26);
        if (close(fd) != 0) e(27);
        wait(&stat_loc);
        if (stat_loc != 0) e(28); /* Alarm? */
}
alarm(0);
}

void test31b()
{
    subtest = 2;

    System("rm -rf ../DIR_31/*");

    /* Test maximal file name length. */
    if (mkfifo(MaxName, 0777) != 0) e(1);
    if (unlink(MaxName) != 0) e(2);
    MaxPath[strlen(MaxPath) - 2] = '/';

```

```

MaxPath[strlen(MaxPath) - 1] = 'a';    /* make ../../.../a */
if (mkfifo(MaxPath, 0777) != 0) e(3);
if (unlink(MaxPath) != 0) e(4);
MaxPath[strlen(MaxPath) - 1] = '/';    /* make ../../.../a */
}

void test31c()
{
    subtest = 3;

    System("rm -rf ../DIR_31/*");

    /* Check if mkfifo() removes, files, fifos, dirs. */
    if (mkfifo("fifo", 0777) != 0) e(1);
    System("mkdir dir; >file");
    if (mkfifo("fifo", 0777) != -1) e(2);
    if (errno != EEXIST) e(3);
    if (mkfifo("dir", 0777) != -1) e(4);
    if (errno != EEXIST) e(5);
    if (mkfifo("file", 0777) != -1) e(6);
    if (errno != EEXIST) e(7);

    /* Test empty path. */
    if (mkfifo("", 0777) != -1) e(8);
    if (errno != ENOENT) e(9);
    if (mkfifo("/tmp/noway/out", 0777) != -1) e(10);
    if (errno != ENOENT) e(11);

    /* Test if path prefix is a directory. */
    if (mkfifo("/etc/passwd/nono", 0777) != -1) e(12);
    if (errno != ENOTDIR) e(13);

    mkdir("bar", 0777);                /* make bar */

    /* Check if no access on part of path generates the correct error. */
    System("chmod 577 bar");           /* r-xrwxrwx */
    if (!superuser) {
        if (mkfifo("bar/nono", 0666) != -1) e(14);
        if (errno != EACCES) e(15);
    }
    if (superuser) {
        if (mkfifo("bar/nono", 0666) != 0) e(14);
        if (unlink("bar/nono") != 0) e(666);
    }
    System("chmod 677 bar");           /* rw-rwxrwx */
    if (!superuser) {
        if (mkfifo("bar/../nono", 0666) != -1) e(16);
        if (errno != EACCES) e(17);
    }
    if (unlink("nono") != -1) e(18);

    /* Clean up bar. */
    System("rm -rf bar");

    /* Test ToLongName and ToLongPath */
#ifdef _POSIX_NO_TRUNC
    #if _POSIX_NO_TRUNC - 0 != -1
    if (mkfifo(ToLongName, 0777) != -1) e(19);
    if (errno != ENAMETOOLONG) e(20);
    #else
    if (mkfifo(ToLongName, 0777) != 0) e(21);
    #endif
#else
    #include "error, this case requires dynamic checks and is not handled"
#endif
    ToLongPath[PATH_MAX - 2] = '/';
    ToLongPath[PATH_MAX - 1] = 'a';
    if (mkfifo(ToLongPath, 0777) != -1) e(22);
    if (errno != ENAMETOOLONG) e(23);
    ToLongPath[PATH_MAX - 1] = '/';
}

void makelongnames()
{

```

```
register int i;

memset(MaxName, 'a', NAME_MAX);
MaxName[NAME_MAX] = '\0';
for (i = 0; i < PATH_MAX - 1; i++) { /* idem path */
    MaxPath[i++] = '.';
    MaxPath[i] = '/';
}
MaxPath[PATH_MAX - 1] = '\0';

strcpy(ToLongName, MaxName); /* copy them Max to ToLong */
strcpy(ToLongPath, MaxPath);

ToLongName[NAME_MAX] = 'a';
ToLongName[NAME_MAX + 1] = '\0'; /* extend ToLongName by one too many */
ToLongPath[PATH_MAX - 1] = '/';
ToLongPath[PATH_MAX] = '\0'; /* inc ToLongPath by one */
}

void e(n)
int n;
{
    int err_num = errno; /* Save in case printf clobbers it. */

    printf("Subtest %d, error %d errno=%d: ", subtest, n, errno);
    errno = err_num;
    perror("");
    if (errct++ > MAX_ERROR) {
        printf("Too many errors; test aborted\n");
        chdir("..");
        system("rm -rf DIR*");
        exit(1);
    }
    errno = 0;
}

void quit()
{
    Chdir("..");
    System("rm -rf DIR_31");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
        printf("%d errors\n", errct);
        exit(1);
    }
}
```



```

/* test32: rename( )                                Author: Jan-Mark Wams (jms@cs.vu.nl) */

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <errno.h>
#include <time.h>
#include <stdio.h>

#define MAX_ERROR      4
#define ITERATIONS     2

#define System(cmd)     if (system(cmd) != 0) printf("“%s” failed\n", cmd)
#define Chdir(dir)      if (chdir(dir) != 0) printf("Can't goto %s\n", dir)
#define Stat(a,b)       if (stat(a,b) != 0) printf("Can't stat %s\n", a)
#define Creat(f)        if (close(creat(f,0777))!=0) printf("Can't creat %s\n", f)

int errct = 0;
int subtest = 1;
int superuser;
char MaxName[NAME_MAX + 1]; /* Name of maximum length */
char MaxPath[PATH_MAX];    /* Same for path */
char ToLongName[NAME_MAX + 2]; /* Name of maximum +1 length */
char ToLongPath[PATH_MAX + 1]; /* Same for path, both too long */

_PROTOTYPE(void main, (int argc, char *argv[]));
_PROTOTYPE(void test32a, (void));
_PROTOTYPE(void test32b, (void));
_PROTOTYPE(void test32c, (void));
_PROTOTYPE(void makelongnames, (void));
_PROTOTYPE(void e, (int number));
_PROTOTYPE(void quit, (void));

void main(argc, argv)
int argc;
char *argv[];
{
    int i, m = 0xFFFF;

    sync();
    if (argc == 2) m = atoi(argv[1]);
    printf("Test 32 ");
    fflush(stdout);
    System("rm -rf DIR_32; mkdir DIR_32");
    Chdir("DIR_32");
    makelongnames();
    superuser = (geteuid() == 0);

    for (i = 0; i < ITERATIONS; i++) {
        if (m & 0001) test32a();
        if (m & 0002) test32b();
        if (m & 0004) test32c();
    }
    quit();
}

#define BUF_SIZE 1024

void test32a()
{
    /* Test normal operation. */
    struct stat st1, st2;
    int fd1, fd2;
    time_t time1, time2, time3;
    char buf[BUF_SIZE];

    subtest = 1;
    System("rm -rf ../DIR_32/*");

    /* Test normal file renamal. */

```

```
System("echo haha > old");
Stat("old", &st1);
if (rename("old", "new") != 0) e(1);
Stat("new", &st2);

/* The status of new should be the same as old. */
if (st1.st_dev != st2.st_dev) e(2);
if (st1.st_ino != st2.st_ino) e(3);
if (st1.st_mode != st2.st_mode) e(4);
if (st1.st_nlink != st2.st_nlink) e(5);
if (st1.st_uid != st2.st_uid) e(6);
if (st1.st_gid != st2.st_gid) e(7);
if (st1.st_rdev != st2.st_rdev) e(8);
if (st1.st_size != st2.st_size) e(9);
if (st1.st_atime != st2.st_atime) e(10);
if (st1.st_mtime != st2.st_mtime) e(11);
if (st1.st_ctime != st2.st_ctime) e(12);

/* If new exists, it should be removed. */
System("ln new new2");
System("echo foobar > old");
Stat("old", &st1);
if (rename("old", "new") != 0) e(13);
Stat("new", &st2);

/* The status of new should be the same as old. */
if (st1.st_dev != st2.st_dev) e(14);
if (st1.st_ino != st2.st_ino) e(15);
if (st1.st_mode != st2.st_mode) e(16);
if (st1.st_nlink != st2.st_nlink) e(17);
if (st1.st_uid != st2.st_uid) e(18);
if (st1.st_gid != st2.st_gid) e(19);
if (st1.st_rdev != st2.st_rdev) e(20);
if (st1.st_size != st2.st_size) e(21);
if (st1.st_atime != st2.st_atime) e(22);
if (st1.st_mtime != st2.st_mtime) e(23);
if (st1.st_ctime != st2.st_ctime) e(24);

/* The link count on new2 should be one since the old new is removed. */
Stat("new2", &st1);
if (st1.st_nlink != 1) e(25);

/* Check if status for "." is updated. */
System(">OLD");
Stat(".", &st1);
time(&time1);
while (time1 == time((time_t *)0))
;
time(&time2);
rename("OLD", "NEW");
Stat(".", &st2);
time(&time3);
while (time3 == time((time_t *)0))
;
time(&time3);
if (st1.st_ctime >= st2.st_ctime) e(26);
if (st1.st_mtime >= st2.st_mtime) e(27);
if (st1.st_ctime > time1) e(28);
if (st1.st_mtime > time1) e(29);
if (st1.st_ctime >= time2) e(30);
if (st1.st_mtime >= time2) e(31);
if (st2.st_ctime < time2) e(32);
if (st2.st_mtime < time2) e(33);
if (st2.st_ctime >= time3) e(34);
if (st2.st_mtime >= time3) e(35);

/* If the new file is removed while it's open it should still be
 * readable. */
System("rm -rf new NEW old OLD");
if ((fd1 = creat("new", 0644)) != 3) e(36);
if (write(fd1, "Hi there! I am Sammy the string", 33) != 33) e(37);
if (close(fd1) != 0) e(38);
if ((fd1 = creat("old", 0644)) != 3) e(39);
if (write(fd1, "I need a new name", 18) != 18) e(40);
```

```
if (close(fd1) != 0) e(41);
if ((fd1 = open("new", O_RDONLY)) != 3) e(42);
if ((fd2 = open("new", O_RDONLY)) != 4) e(43);
if (rename("old", "new") != 0) e(44);
if (stat("old", &st1) == 0) e(45);
if (close(fd1) != 0) e(46);
if ((fd1 = open("new", O_RDONLY)) != 3) e(47);
if (read(fd2, buf, BUF_SIZE) != 33) e(48);
if (strcmp(buf, "Hi there! I am Sammy the string") != 0) e(49);
if (read(fd1, buf, BUF_SIZE) != 18) e(50);
if (strcmp(buf, "I need a new name") != 0) e(51);
if (close(fd1) != 0) e(52);
if (close(fd2) != 0) e(53);
}

void test32b()
{
    char MaxPath2[PATH_MAX];      /* Same for path */
    char MaxName2[NAME_MAX + 1]; /* Name of maximum length */
    int fd, i;
    int stat_loc;
    struct stat st;

    subtest = 2;
    System("rm -rf ../DIR_32/*");

    /* Test maximal file name length. */
    if ((fd = creat(MaxName, 0777)) != 3) e(1);
    if (close(fd) != 0) e(2);
    strcpy(MaxName2, MaxName);
    MaxName2[strlen(MaxName2) - 1] ^= 1;
    if (rename(MaxName, MaxName2) != 0) e(3);
    if (rename(MaxName2, MaxName) != 0) e(4);
    MaxName2[strlen(MaxName2) - 1] ^= 2;
    if (rename(MaxName, MaxName2) != 0) e(5);
    MaxPath[strlen(MaxPath) - 2] = '/';
    MaxPath[strlen(MaxPath) - 1] = 'a'; /* make ../.../a */
    if ((fd = creat(MaxPath, 0777)) != 3) e(6);
    if (close(fd) != 0) e(7);
    strcpy(MaxPath2, MaxPath);
    MaxPath2[strlen(MaxPath2) - 1] ^= 1;
    if (rename(MaxPath, MaxPath2) != 0) e(8);
    if (rename(MaxPath2, MaxPath) != 0) e(9);
    MaxPath2[strlen(MaxPath2) - 1] ^= 2;
    if (rename(MaxPath, MaxPath2) != 0) e(10);
    MaxPath[strlen(MaxPath) - 1] = '/'; /* make ../.../a */

    /* Test if linked files are renamable. */
    System(">foo;ln foo bar");
    if (rename("foo", "bar") != 0) e(11);
    if (rename("bar", "foo") != 0) e(12);
    System("ln foo foobar");
    if (rename("foo", "foobar") != 0) e(13);
    if (rename("bar", "foobar") != 0) e(14);

    /* Since the same files have the same links.... */
    if (rename("bar", "bar") != 0) e(15);
    if (rename("foo", "foo") != 0) e(16);
    if (rename("foobar", "foobar") != 0) e(17);

    /* In ``rename(old, new)'' with new existing, there is always an new
    * entry. */
    for (i = 0; i < 5; i++) {
        System("echo old > old");
        System("echo news > new");
        switch (fork()) {
            case -1: printf("Can't fork\n"); break;
            case 0:
                alarm(20);
                sleep(1);
                rename("old", "new");
                exit(0);
            default:
                while (stat("old", &st) != 0)
```

```

        if (stat("new", &st) != 0) e(18);
        wait(&stat_loc);
        if (stat_loc != 0) e(19);          /* Alarm? */
    }
}

void test32c()
{
    /* Test behavior under error contitions. */
    struct stat st1;
    int stat_loc;

    subtest = 3;
    System("rm -rf ../DIR_32/*");

    /* Test if we have access. */
    system("chmod 777 noacc nowrite > /dev/null 2> /dev/null");
    system("rm -rf noacc nowrite");

    System("mkdir noacc nowrite");
    System("> noacc/file");
    System("> nowrite/file");
    System("> file");
    System("chmod 677 noacc");
    System("chmod 577 nowrite");
    if (!superuser) {
        if (rename("noacc/file", "nono") != -1) e(1);
        if (errno != EACCES) e(2);
        if (rename("nowrite/file", "nono") != -1) e(3);
        if (errno != EACCES) e(4);
        if (rename("file", "noacc/file") != -1) e(5);
        if (errno != EACCES) e(6);
        if (rename("file", "nowrite/file") != -1) e(7);
        if (errno != EACCES) e(8);
    }
    if (superuser) {
        /* Super user heeft access. */
        if (rename("noacc/file", "noacc/yes") != 0) e(9);
        if (rename("nowrite/file", "nowrite/yes") != 0) e(10);
        if (rename("file", "yes") != 0) e(11);
        if (rename("noacc/yes", "noacc/file") != 0) e(12);
        if (rename("nowrite/yes", "nowrite/file") != 0) e(13);
        if (rename("yes", "file") != 0) e(14);
    }
    System("chmod 777 noacc nowrite");

    /* If rmdir() doesn't remove a directory, rename() shouldn't either. */
    System("mkdir newdir olddir");
    System("rm -rf /tmp/sema.11[ab]");
    switch (fork()) {
        case -1: printf("Can't fork\n"); break;
        case 0:
            alarm(20);
            switch (fork()) {
                case -1: printf("Can't fork\n"); break;
                case 0:
                    /* Child A. */
                    alarm(20);
                    if (chdir("newdir") != 0) e(15);
                    Creat("/tmp/sema.11a");
                    while (stat("/tmp/sema.11a", &st1) == -1) sleep(1);
                    exit(0);
                default:
                    wait(&stat_loc);
                    if (stat_loc != 0) e(16);          /* Alarm? */
            }

            /* Child B. */
            if (chdir("olddir") != 0) e(17);
            Creat("/tmp/sema.11b");
            while (stat("/tmp/sema.11b", &st1) == -1) sleep(1);
            exit(0);
        default:

```

```

    /* Wait for child A. It will keep ``newdir`` busy. */
    while (stat("/tmp/sema.11a", &st1) == -1) sleep(1);
    if (rmdir("newdir") == -1) {
        if (rename("olddir", "newdir") != -1) e(18);
        if (errno != EBUSY) e(19);
    }
    (void) unlink("/tmp/sema.11a");

    /* Wait for child B. It will keep ``olddir`` busy. */
    while (stat("/tmp/sema.11b", &st1) == -1) sleep(1);
    if (rmdir("olddir") == -1) {
        if (rename("olddir", "newdir") != -1) e(20);
        if (errno != EBUSY) e(21);
    }
    (void) unlink("/tmp/sema.11b");
    wait(&stat_loc);
    if (stat_loc != 0) e(22);          /* Alarm? */
}
}

void makelongnames()
{
    register int i;

    memset(MaxName, 'a', NAME_MAX);
    MaxName[NAME_MAX] = '\0';
    for (i = 0; i < PATH_MAX - 1; i++) { /* idem path */
        MaxPath[i++] = '.';
        MaxPath[i] = '/';
    }
    MaxPath[PATH_MAX - 1] = '\0';

    strcpy(ToLongName, MaxName); /* copy them Max to ToLong */
    strcpy(ToLongPath, MaxPath);

    ToLongName[NAME_MAX] = 'a';
    ToLongName[NAME_MAX + 1] = '\0'; /* extend ToLongName by one too many */
    ToLongPath[PATH_MAX - 1] = '/';
    ToLongPath[PATH_MAX] = '\0'; /* inc ToLongPath by one */
}

void e(n)
int n;
{
    int err_num = errno;          /* Save in case printf clobbers it. */

    printf("Subtest %d, error %d errno=%d: ", subtest, n, errno);
    errno = err_num;
    perror("");
    if (errct++ > MAX_ERROR) {
        printf("Too many errors; test aborted\n");
        chdir("..");
        system("rm -rf DIR*");
        exit(1);
    }
    errno = 0;
}

void quit()
{
    Chdir("..");
    System("rm -rf DIR_32");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
        printf("%d errors\n", errct);
        exit(1);
    }
}
}

```

```

/* test33: access()                                Author: Jan-Mark Wams (jms@cs.vu.nl) */

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <errno.h>
#include <time.h>
#include <stdio.h>

#define MAX_ERROR      1
#define ITERATIONS     2

#define System(cmd)     if (system(cmd) != 0) printf("'%s' failed\n", cmd)
#define Chdir(dir)      if (chdir(dir) != 0) printf("Can't goto %s\n", dir)
#define Stat(a,b)       if (stat(a,b) != 0) printf("Can't stat %s\n", a)
#define Chmod(a,b)      if (chmod(a,b) != 0) printf("Can't chmod %s\n", a)
#define Mkfifo(f)       if (mkfifo(f,0777)!=0) printf("Can't make fifo %s\n", f)

int errct = 0;
int subtest = 1;
int superuser; /* nonzero if uid == euid (euid == 0 always) */
char MaxName[NAME_MAX + 1]; /* Name of maximum length */
char MaxPath[PATH_MAX]; /* Same for path */
char ToLongName[NAME_MAX + 2]; /* Name of maximum +1 length */
char ToLongPath[PATH_MAX + 1]; /* Same for path, both too long */

_PROTOTYPE(void main, (int argc, char *argv[]));
_PROTOTYPE(void test33a, (void));
_PROTOTYPE(void test33b, (void));
_PROTOTYPE(void test33c, (void));
_PROTOTYPE(void test33d, (void));
_PROTOTYPE(void test_access, (void));
_PROTOTYPE(void makelongnames, (void));
_PROTOTYPE(void e, (int number));
_PROTOTYPE(void quit, (void));

void main(argc, argv)
int argc;
char *argv[];
{
    int i, m = 0xFFFF;

    sync();
    if (argc == 2) m = atoi(argv[1]);
    printf("Test 33 ");
    fflush(stdout);

    if (geteuid() != 0) {
        printf("must be setuid root; test aborted\n");
        exit(1);
    }
    if (getuid() == 0) {
        printf("must be setuid root logged in as someone else; test aborted\n");
        exit(1);
    }

    umask(0000);
    System("rm -rf DIR_33; mkdir DIR_33");
    Chdir("DIR_33");
    makelongnames();
    superuser = (getuid() == 0);

    for (i = 0; i < ITERATIONS; i++) {
        if (m & 0001) test33a();
        if (m & 0002) test33b();
        if (m & 0004) test33c();
        if (m & 0010) test33d();
    }
    quit();
}

```

```

}

void test33a()
{
    /* Test normal operation. */
    int stat_loc;          /* For the wait(&stat_loc) call. */

    subtest = 1;
    System("rm -rf ../DIR_33/*");

    /* To test normal access first make some files for real uid. */
    switch (fork()) {
        case -1: printf("Can't fork\n");      break;
        case 0:
            alarm(20);
            setuid(getuid());          /* (Re)set the effective ids to the
                                         * real ids. */

            setgid(getgid());
            System("> rwx; chmod 700 rwx");
            System("> rw_; chmod 600 rw_");
            System("> r_x; chmod 500 r_x");
            System("> r__; chmod 400 r__");
            System("> _wx; chmod 300 _wx");
            System("> _w_; chmod 200 _w_");
            System("> __x; chmod 100 __x");
            System("> __; chmod 000 __");
            exit(0);

        default:
            wait(&stat_loc);
            if (stat_loc != 0) e(1); /* Alarm? */
    }
    test_access();

    /* Let's test access() on directorys. */
    switch (fork()) {
        case -1: printf("Can't fork\n");      break;
        case 0:
            alarm(20);
            setuid(getuid());          /* (Re)set the effective ids to the
                                         * real ids. */

            setgid(getgid());
            System("rm -rf [_r][_w][_x]");
            System("mkdir rwx; chmod 700 rwx");
            System("mkdir rw_; chmod 600 rw_");
            System("mkdir r_x; chmod 500 r_x");
            System("mkdir r__; chmod 400 r__");
            System("mkdir _wx; chmod 300 _wx");
            System("mkdir _w_; chmod 200 _w_");
            System("mkdir __x; chmod 100 __x");
            System("mkdir __; chmod 000 __");
            exit(0);

        default:
            wait(&stat_loc);
            if (stat_loc != 0) e(2); /* Alarm? */
    }
    test_access();

    switch (fork()) {
        case -1: printf("Can't fork\n");      break;
        case 0:
            alarm(20);
            setuid(getuid());          /* (Re)set the effective ids to the
                                         * real ids. */

            setgid(getgid());
            System("rmdir [_r][_w][_x]");
            Mkfifo("rwx");
            System("chmod 700 rwx");
            Mkfifo("rw_");
            System("chmod 600 rw_");
            Mkfifo("r_x");
            System("chmod 500 r_x");
            Mkfifo("r__");
            System("chmod 400 r__");

```

```

    Mkfifo("_wx");
    System("chmod 300 _wx");
    Mkfifo("_w_");
    System("chmod 200 _w_");
    Mkfifo("__x");
    System("chmod 100 __x");
    Mkfifo("___");
    System("chmod 000 ___");
    exit(0);

    default:
        wait(&stat_loc);
        if (stat_loc != 0) e(3);/* Alarm? */
}
test_access();

/* Remove all the fifos. */
switch (fork()) {
    case -1: printf("Can't fork\n");      break;
    case 0:
        alarm(20);
        setuid(getuid());
        setgid(getgid());
        System("rm -rf [_r][_w][_x]");
        exit(0);

    default:
        wait(&stat_loc);
        if (stat_loc != 0) e(4);/* Alarm? */
}
}

void test33b()
{
    int stat_loc;                /* For the wait(&stat_loc) call. */

    subtest = 2;
    System("rm -rf ../DIR_33/*");

    switch (fork()) {
        case -1: printf("Can't fork\n");      break;
        case 0:
            alarm(20);

            /* (Re)set the effective ids to the real ids. */
            setuid(getuid());
            setgid(getgid());
            System("> _____rwx; chmod 007 _____rwx");
            System("> _____x; chmod 001 _____x");
            System("> _____; chmod 000 _____");
            exit(0);

            default:
                wait(&stat_loc);
                if (stat_loc != 0) e(1);/* Alarm? */
    }

    /* If we are superuser, we have access to all. */
    /* Well, almost, execution access might need at least one X bit. */
    if (superuser) {
        if (access("_____", R_OK) != 0) e(2);
        if (access("_____", W_OK) != 0) e(3);
        if (access("_____x", R_OK) != 0) e(4);
        if (access("_____x", W_OK) != 0) e(5);
        if (access("_____x", X_OK) != 0) e(6);
        if (access("_____rwx", R_OK) != 0) e(7);
        if (access("_____rwx", W_OK) != 0) e(8);
        if (access("_____rwx", X_OK) != 0) e(9);
    }
    if (!superuser) {
        if (access("_____", R_OK) != -1) e(10);
        if (errno != EACCES) e(11);
        if (access("_____", W_OK) != -1) e(12);
        if (errno != EACCES) e(13);
    }
}

```



```
    if (access("_____", X_OK) != -1) e(14);
    if (errno != EACCES) e(15);
    if (access("_____x", R_OK) != -1) e(16);
    if (errno != EACCES) e(17);
    if (access("_____x", W_OK) != -1) e(18);
    if (errno != EACCES) e(19);
    if (access("_____x", X_OK) != -1) e(20);
    if (errno != EACCES) e(21);
    if (access("_____rwx", R_OK) != -1) e(22);
    if (errno != EACCES) e(23);
    if (access("_____rwx", W_OK) != -1) e(24);
    if (errno != EACCES) e(25);
    if (access("_____rwx", X_OK) != -1) e(26);
    if (errno != EACCES) e(27);
}

/* If the real uid != effective uid. */
if (!superuser) {
    System("rm -rf [_r][_w][_x]");
    System(">rwx");
    Chmod("rwx", 0700);
    System(">rw_");
    Chmod("rw_", 0600);
    System(">r_x");
    Chmod("r_x", 0500);
    System(">r__");
    Chmod("r__", 0400);
    System(">_wx");
    Chmod("_wx", 0300);
    System(">_w_");
    Chmod("_w_", 0200);
    System(">__x");
    Chmod("__x", 0100);
    System(">___");
    Chmod("___", 0000);

    if (access("rwx", F_OK) != 0) e(28);
    if (access("rwx", R_OK) != -1) e(29);
    if (errno != EACCES) e(30);
    if (access("rwx", W_OK) != -1) e(31);
    if (errno != EACCES) e(32);
    if (access("rwx", X_OK) != -1) e(33);
    if (errno != EACCES) e(34);

    if (access("rw_", F_OK) != 0) e(35);
    if (access("rw_", R_OK) != -1) e(36);
    if (errno != EACCES) e(37);
    if (access("rw_", W_OK) != -1) e(38);
    if (errno != EACCES) e(39);
    if (access("rw_", X_OK) != -1) e(40);
    if (errno != EACCES) e(41);

    if (access("r_x", F_OK) != 0) e(42);
    if (access("r_x", R_OK) != -1) e(43);
    if (errno != EACCES) e(44);
    if (access("r_x", W_OK) != -1) e(45);
    if (errno != EACCES) e(46);
    if (access("r_x", X_OK) != -1) e(47);
    if (errno != EACCES) e(48);

    if (access("r__", F_OK) != 0) e(49);
    if (access("r__", R_OK) != -1) e(50);
    if (errno != EACCES) e(51);
    if (access("r__", W_OK) != -1) e(52);
    if (errno != EACCES) e(53);
    if (access("r__", X_OK) != -1) e(54);
    if (errno != EACCES) e(55);

    if (access("_wx", F_OK) != 0) e(56);
    if (access("_wx", R_OK) != -1) e(57);
    if (errno != EACCES) e(58);
    if (access("_wx", W_OK) != -1) e(59);
    if (errno != EACCES) e(60);
    if (access("_wx", X_OK) != -1) e(61);
```

```
    if (errno != EACCES) e(62);

    if (access("_w_", F_OK) != 0) e(63);
    if (access("_w_", R_OK) != -1) e(64);
    if (errno != EACCES) e(65);
    if (access("_w_", W_OK) != -1) e(66);
    if (errno != EACCES) e(67);
    if (access("_w_", X_OK) != -1) e(68);
    if (errno != EACCES) e(69);

    if (access("__x", F_OK) != 0) e(70);
    if (access("__x", R_OK) != -1) e(71);
    if (errno != EACCES) e(72);
    if (access("__x", W_OK) != -1) e(73);
    if (errno != EACCES) e(74);
    if (access("__x", X_OK) != -1) e(75);
    if (errno != EACCES) e(76);

    if (access("___", F_OK) != 0) e(77);
    if (access("___", R_OK) != -1) e(78);
    if (errno != EACCES) e(79);
    if (access("___", W_OK) != -1) e(80);
    if (errno != EACCES) e(81);
    if (access("___", X_OK) != -1) e(82);
    if (errno != EACCES) e(83);

    System("rm -rf [_r][_w][_x]");
}
}

void test33c()
{
    /* Test errors returned. */
    int i;

    subtest = 3;
    System("rm -rf ../DIR_33/*");

    /* Test what access() does with non existing files. */
    System("rm -rf nonexistent");
    if (access("noexist", F_OK) != -1) e(1);
    if (errno != ENOENT) e(2);
    if (access("noexist", R_OK) != -1) e(3);
    if (errno != ENOENT) e(4);
    if (access("noexist", W_OK) != -1) e(5);
    if (errno != ENOENT) e(6);
    if (access("noexist", X_OK) != -1) e(7);
    if (errno != ENOENT) e(8);
    if (access("noexist", R_OK | W_OK) != -1) e(9);
    if (errno != ENOENT) e(10);
    if (access("noexist", R_OK | X_OK) != -1) e(11);
    if (errno != ENOENT) e(12);
    if (access("noexist", W_OK | X_OK) != -1) e(13);
    if (errno != ENOENT) e(14);
    if (access("noexist", R_OK | W_OK | X_OK) != -1) e(15);
    if (errno != ENOENT) e(16);

    /* Test access on a nonsearchable path. */
    if (mkdir("nosearch", 0777) != 0) e(1000);
    if ( (i = creat("nosearch/file", 0666)) < 0) e(1001);
    if (close(i) < 0) e(1002);
    if ( (i = creat("file", 0666)) < 0) e(1003);
    if (close(i) < 0) e(1004);
    if (chmod("nosearch/file", 05777) < 0) e(1005);
    if (chmod("file", 05777) < 0) e(1006);
    if (chmod("nosearch", 0677) != 0) e(1007);
    if (access("nosearch/file", F_OK) != 0) e(17);

    /* Test ToLongName and ToLongPath */
#ifdef _POSIX_NO_TRUNC
    # if _POSIX_NO_TRUNC - 0 != -1
        if (access(ToLongName, F_OK) != -1) e(23);
        if (errno != ENAMETOOLONG) e(24);
    # else
        if (close(creat(ToLongName, 0777)) != 0) e(25);
    #endif
}
```

```

    if (access(ToLongName, F_OK) != 0) e(26);
# endif
# else
# include "error, this case requires dynamic checks and is not handled"
# endif
ToLongPath[PATH_MAX - 2] = '/';
ToLongPath[PATH_MAX - 1] = 'a';
if (access(ToLongPath, F_OK) != -1) e(27);
if (errno != ENAMETOOLONG) e(28);
ToLongPath[PATH_MAX - 1] = '/';

/* Test empty strings. */
if (access("", F_OK) != -1) e(29);
if (errno != ENOENT) e(30);
System("rm -rf idonotexist");
if (access("idonotexist", F_OK) != -1) e(31);
if (errno != ENOENT) e(32);

/* Test non directories in prefix of path. */
if (access("/etc/passwd/dir/foo", F_OK) != -1) e(33);
if (errno != ENOTDIR) e(34);
System("rm -rf nodir; > nodir");
if (access("nodir/foo", F_OK) != -1) e(35);
if (errno != ENOTDIR) e(36);

/* Test if invalid amode arguments are signaled. */
System(">allmod");
Chmod("allmod", 05777);
for (i = -1025; i < 1025; i++) {
    if ((mode_t) i != F_OK && ((mode_t) i & ~(R_OK | W_OK | X_OK)) != 0) {
        if (access("allmod", (mode_t) i) != -1) e(37);
        if (errno != EINVAL) e(38);
    } else
        if (access("allmod", (mode_t) i) != 0) e(39);
}
}

void test33d()
{
    /* Test access() flags. */
# define EXCLUDE(a,b) (((a)^(b)) == ((a)|(b)))
    subtest = 4;
    System("rm -rf ../DIR_33/*");

    /* The test are rather strong, stronger that POSIX specifies. */
    /* The should be OR able, this test tests if all the 1 bits */
    /* Are in diferent places. This should be what one wants. */
    if (!EXCLUDE(R_OK, W_OK | X_OK)) e(1);
    if (!EXCLUDE(W_OK, R_OK | X_OK)) e(2);
    if (!EXCLUDE(X_OK, R_OK | W_OK)) e(3);
    if (F_OK == R_OK) e(4);
    if (F_OK == W_OK) e(5);
    if (F_OK == X_OK) e(6);
    if (F_OK == (R_OK | W_OK)) e(7);
    if (F_OK == (W_OK | X_OK)) e(8);
    if (F_OK == (R_OK | X_OK)) e(9);
    if (F_OK == (R_OK | W_OK | X_OK)) e(10);
}

void test_access()
{
    /* Test all [_r][_w][_x] files. */
    if (!superuser) {
        /* Test normal access. */
        if (access("rwx", F_OK) != 0) e(11);
        if (access("rwx", R_OK) != 0) e(12);
        if (access("rwx", W_OK) != 0) e(13);
        if (access("rwx", X_OK) != 0) e(14);
        if (access("rwx", R_OK | W_OK) != 0) e(15);
        if (access("rwx", R_OK | X_OK) != 0) e(16);
        if (access("rwx", W_OK | X_OK) != 0) e(17);
        if (access("rwx", R_OK | W_OK | X_OK) != 0) e(18);

        if (access("rw_", F_OK) != 0) e(19);
        if (access("rw_", R_OK) != 0) e(20);
        if (access("rw_", W_OK) != 0) e(21);
    }
}

```

```
if (access("rw_", X_OK) != -1) e(22);
if (errno != EACCES) e(23);
if (access("rw_", R_OK | W_OK) != 0) e(24);
if (access("rw_", R_OK | X_OK) != -1) e(25);
if (errno != EACCES) e(26);
if (access("rw_", W_OK | X_OK) != -1) e(27);
if (errno != EACCES) e(28);
if (access("rw_", R_OK | W_OK | X_OK) != -1) e(29);
if (errno != EACCES) e(30);

if (access("r_x", F_OK) != 0) e(31);
if (access("r_x", R_OK) != 0) e(32);
if (access("r_x", W_OK) != -1) e(33);
if (errno != EACCES) e(34);
if (access("r_x", X_OK) != 0) e(35);
if (access("r_x", R_OK | W_OK) != -1) e(36);
if (errno != EACCES) e(37);
if (access("r_x", R_OK | X_OK) != 0) e(38);
if (access("r_x", W_OK | X_OK) != -1) e(39);
if (errno != EACCES) e(40);
if (access("r_x", R_OK | W_OK | X_OK) != -1) e(41);
if (errno != EACCES) e(42);

if (access("r__", F_OK) != 0) e(43);
if (access("r__", R_OK) != 0) e(44);
if (access("r__", W_OK) != -1) e(45);
if (errno != EACCES) e(46);
if (access("r__", X_OK) != -1) e(47);
if (errno != EACCES) e(48);
if (access("r__", R_OK | W_OK) != -1) e(49);
if (errno != EACCES) e(50);
if (access("r__", R_OK | X_OK) != -1) e(51);
if (errno != EACCES) e(52);
if (access("r__", W_OK | X_OK) != -1) e(53);
if (errno != EACCES) e(54);
if (access("r__", R_OK | W_OK | X_OK) != -1) e(55);
if (errno != EACCES) e(56);

if (access("_wx", F_OK) != 0) e(57);
if (access("_wx", R_OK) != -1) e(58);
if (errno != EACCES) e(59);
if (access("_wx", W_OK) != 0) e(60);
if (access("_wx", X_OK) != 0) e(61);
if (access("_wx", R_OK | W_OK) != -1) e(62);
if (errno != EACCES) e(63);
if (access("_wx", R_OK | X_OK) != -1) e(64);
if (errno != EACCES) e(65);
if (access("_wx", W_OK | X_OK) != 0) e(66);
if (access("_wx", R_OK | W_OK | X_OK) != -1) e(67);
if (errno != EACCES) e(68);

if (access("_w_", F_OK) != 0) e(69);
if (access("_w_", R_OK) != -1) e(70);
if (errno != EACCES) e(71);
if (access("_w_", W_OK) != 0) e(72);
if (access("_w_", X_OK) != -1) e(73);
if (errno != EACCES) e(74);
if (access("_w_", R_OK | W_OK) != -1) e(75);
if (errno != EACCES) e(76);
if (access("_w_", R_OK | X_OK) != -1) e(77);
if (errno != EACCES) e(78);
if (access("_w_", W_OK | X_OK) != -1) e(79);
if (errno != EACCES) e(80);
if (access("_w_", R_OK | W_OK | X_OK) != -1) e(81);
if (errno != EACCES) e(82);

if (access("__x", F_OK) != 0) e(83);
if (access("__x", R_OK) != -1) e(84);
if (errno != EACCES) e(85);
if (access("__x", W_OK) != -1) e(86);
if (errno != EACCES) e(87);
if (access("__x", X_OK) != 0) e(88);
if (access("__x", R_OK | W_OK) != -1) e(89);
if (errno != EACCES) e(90);
```

```
if (access("__x", R_OK | X_OK) != -1) e(91);
if (errno != EACCES) e(92);
if (access("__x", W_OK | X_OK) != -1) e(93);
if (errno != EACCES) e(94);
if (access("__x", R_OK | W_OK | X_OK) != -1) e(95);
if (errno != EACCES) e(96);

if (access("__", F_OK) != 0) e(97);
if (access("__", R_OK) != -1) e(98);
if (errno != EACCES) e(99);
if (access("__", W_OK) != -1) e(100);
if (errno != EACCES) e(101);
if (access("__", X_OK) != -1) e(102);
if (errno != EACCES) e(103);
if (access("__", R_OK | W_OK) != -1) e(104);
if (errno != EACCES) e(105);
if (access("__", R_OK | X_OK) != -1) e(106);
if (errno != EACCES) e(107);
if (access("__", W_OK | X_OK) != -1) e(108);
if (errno != EACCES) e(109);
if (access("__", R_OK | W_OK | X_OK) != -1) e(110);
if (errno != EACCES) e(111);
}
if (superuser) {
    /* Test root access don't test X_OK on [_r][_w]_ files. */
    if (access("rwx", F_OK) != 0) e(112);
    if (access("rwx", R_OK) != 0) e(113);
    if (access("rwx", W_OK) != 0) e(114);
    if (access("rwx", X_OK) != 0) e(115);
    if (access("rwx", R_OK | W_OK) != 0) e(116);
    if (access("rwx", R_OK | X_OK) != 0) e(117);
    if (access("rwx", W_OK | X_OK) != 0) e(118);
    if (access("rwx", R_OK | W_OK | X_OK) != 0) e(119);

    if (access("rw_", F_OK) != 0) e(120);
    if (access("rw_", R_OK) != 0) e(121);
    if (access("rw_", W_OK) != 0) e(122);
    if (access("rw_", R_OK | W_OK) != 0) e(123);

    if (access("r_x", F_OK) != 0) e(124);
    if (access("r_x", R_OK) != 0) e(125);
    if (access("r_x", W_OK) != 0) e(126);
    if (access("r_x", X_OK) != 0) e(127);
    if (access("r_x", R_OK | W_OK) != 0) e(128);
    if (access("r_x", R_OK | X_OK) != 0) e(129);
    if (access("r_x", W_OK | X_OK) != 0) e(130);
    if (access("r_x", R_OK | W_OK | X_OK) != 0) e(131);

    if (access("r__", F_OK) != 0) e(132);
    if (access("r__", R_OK) != 0) e(133);
    if (access("r__", W_OK) != 0) e(134);
    if (access("r__", R_OK | W_OK) != 0) e(135);

    if (access("_wx", F_OK) != 0) e(136);
    if (access("_wx", R_OK) != 0) e(137);
    if (access("_wx", W_OK) != 0) e(138);
    if (access("_wx", X_OK) != 0) e(139);
    if (access("_wx", R_OK | W_OK) != 0) e(140);
    if (access("_wx", R_OK | X_OK) != 0) e(141);
    if (access("_wx", W_OK | X_OK) != 0) e(142);
    if (access("_wx", R_OK | W_OK | X_OK) != 0) e(143);

    if (access("_w_", F_OK) != 0) e(144);
    if (access("_w_", R_OK) != 0) e(145);
    if (access("_w_", W_OK) != 0) e(146);
    if (access("_w_", R_OK | W_OK) != 0) e(147);

    if (access("__x", F_OK) != 0) e(148);
    if (access("__x", R_OK) != 0) e(149);
    if (access("__x", W_OK) != 0) e(150);
    if (access("__x", X_OK) != 0) e(151);
    if (access("__x", R_OK | W_OK) != 0) e(152);
    if (access("__x", R_OK | X_OK) != 0) e(153);
    if (access("__x", W_OK | X_OK) != 0) e(154);
}
```

```
    if (access("__x", R_OK | W_OK | X_OK) != 0) e(155);

    if (access("___", F_OK) != 0) e(156);
    if (access("___", R_OK) != 0) e(157);
    if (access("___", W_OK) != 0) e(158);
    if (access("___", R_OK | W_OK) != 0) e(159);
}
}

void makelongnames()
{
    register int i;

    memset(MaxName, 'a', NAME_MAX);
    MaxName[NAME_MAX] = '\0';
    for (i = 0; i < PATH_MAX - 1; i++) { /* idem path */
        MaxPath[i++] = '.';
        MaxPath[i] = '/';
    }
    MaxPath[PATH_MAX - 1] = '\0';

    strcpy(ToLongName, MaxName); /* copy them Max to ToLong */
    strcpy(ToLongPath, MaxPath);

    ToLongName[NAME_MAX] = 'a';
    ToLongName[NAME_MAX + 1] = '\0'; /* extend ToLongName by one too many */
    ToLongPath[PATH_MAX - 1] = '/';
    ToLongPath[PATH_MAX] = '\0'; /* inc ToLongPath by one */
}

void e(n)
int n;
{
    int err_num = errno; /* Save in case printf clobbers it. */

    printf("Subtest %d, error %d errno=%d: ", subtest, n, errno);
    errno = err_num;
    perror("");
    if (errct++ > MAX_ERROR) {
        printf("Too many errors; test aborted\n");
        chdir("..");
        system("rm -rf DIR*");
        exit(1);
    }
    errno = 0;
}

void quit()
{
    Chdir("..");
    System("rm -rf DIR_33");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
        printf("%d errors\n", errct);
        exit(1);
    }
}
```

```

/* test34: chmod() chown()          Author: Jan-Mark Wams (jms@cs.vu.nl) */

/* There is a problem getting valid uids and gids, so we use the passwd
** file (ie. /etc/passwd). I don't like this, but I see no other way.
** The read-only-device-error (EROFS) is not checked!
** Supplementary group IDs are ignored.
*/

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <errno.h>
#include <ctype.h>
#include <time.h>
#include <stdio.h>

#define MAX_ERROR          4
#define ITERATIONS         4
#define N 100

#define ALL_RWXB           (S_IRWXU | S_IRWXG | S_IRWXO)
#define ALL_SETB           (S_ISUID | S_ISGID)
#define ALL_BITS           (ALL_RWXB | ALL_SETB)

#define System(cmd)        if (system(cmd) != 0) printf("“%s” failed\n", cmd)
#define Chdir(dir)         if (chdir(dir) != 0) printf("Can't goto %s\n", dir)
#define Stat(a,b)          if (stat(a,b) != 0) printf("Can't stat %s\n", a)
#define Mkfifo(f)          if (mkfifo(f,0777)!=0) printf("Can't make fifo %s\n", f)
#define Mkdir(f)           if (mkdir(f,0777)!=0) printf("Can't make dir %s\n", f)
#define Creat(f)           if (close(creat(f,0777))!=0) printf("Can't creat %s\n",f)

/* This program uses /etc/passwd and assumes things about it's contents. */
#define PASSWD_FILE        "/etc/passwd"

int errct = 0;
int subtest = 1;
int superuser;
int I_can_chown;
char MaxName[NAME_MAX + 1]; /* Name of maximum length */
char MaxPath[PATH_MAX];     /* Same for path */
char NameTooLong[NAME_MAX + 2]; /* Name of maximum +1 length */
char PathTooLong[PATH_MAX + 1]; /* Same for path, both too long */

_PROTOTYPE(void main, (int argc, char *argv[]));
_PROTOTYPE(void test34a, (void));
_PROTOTYPE(void test34b, (void));
_PROTOTYPE(void test34c, (void));
_PROTOTYPE(mode_t mode, (char *file_name));
_PROTOTYPE(void makelongnames, (void));
_PROTOTYPE(void e, (int number));
_PROTOTYPE(void quit, (void));
_PROTOTYPE(void getids, (uid_t * uid, gid_t * gid));

void main(argc, argv)
int argc;
char *argv[];
{
    int i, m = 0xFFFF;

    sync();
    if (argc == 2) m = atoi(argv[1]);
    printf("Test 34 ");
    fflush(stdout);
    (void) system("chmod 777 DIR_34/* > /dev/null 2> /dev/null");
    System("rm -rf DIR_34; mkdir DIR_34");
    if (chdir("DIR_34") != 0) {
        fprintf(stderr, "Can't go to DIR_34\n");
        system("rm -rf DIR_34");
        exit(1);
    }

```

```

}
makelongnames();
superuser = (geteuid() == (uid_t) 0);

#ifdef _POSIX_CHOWN_RESTRICTED
    I_can_chown = superuser;
#else
    I_can_chown = 1;
#endif

    umask(0000);

    for (i = 1; i < ITERATIONS; i++) {
        if (m & 0001) test34a();
        if (m & 0002) test34b();
        if (m & 0004) test34c();
    }
    quit();
}

void test34a()
{
    /* Test normal operation. */
    time_t time1, time2;
    mode_t mod;
    struct stat st1, st2;
    int cnt;
    uid_t uid, uid2;
    gid_t gid, gid2;
    int stat_loc;

    subtest = 1;

    /* Make scratch file. */
    Creat("foo");

    for (mod = 0; mod <= ALL_BITS; mod++) {
        if ((mod & ALL_BITS) != mod) /* If not a valid mod next. */
            continue;
        Stat("foo", &st1);
        if (time(&time1) == (time_t) - 1) e(1);
        if (chmod("foo", mod) != 0) e(2);
        Stat("foo", &st2);
        if (time(&time2) == (time_t) - 1) e(3);
        if (superuser)
            if ((st2.st_mode & ALL_BITS) != mod) e(4);
        if (!superuser)
            if ((st2.st_mode & ALL_RWXB) != (mod & ALL_RWXB)) e(5);

        /* Test the C time feald. */
        if (st1.st_ctime > st2.st_ctime) e(6);
        if (st1.st_ctime > time1) e(7);
        if (st1.st_ctime > time2) e(8);
#ifdef V1_FILESYSTEM
        if (st2.st_ctime < time1) e(9);
#endif
        if (st2.st_ctime > time2) e(10);
        if (st1.st_atime != st2.st_atime) e(11);
        if (st1.st_mtime != st2.st_mtime) e(12);
    }
    /* End for loop. */

    /* Check if chown(file, geteuid(), getegid()) works. */
    for (cnt = 0; cnt < 20; cnt++) {
        /* Set all rights on foo, including the set .id bits. */
        if (chmod("foo", ALL_BITS) != 0) e(13);
        Stat("foo", &st1);
        if (time(&time1) == (time_t) -1) e(14);

        if (chown("foo", geteuid(), getegid()) != 0) e(15);
        Stat("foo", &st2);
        if (time(&time2) == (time_t) -1) e(16);

        /* Check ``chown()`` killed the set .id bits. */
        if (!superuser) {
            if ((st1.st_mode & ALL_RWXB) != ALL_RWXB) e(17);

```



```

        if ((st2.st_mode & ALL_BITS) != ALL_RWXB) e(18);
    }
    if (superuser) {
        if ((st1.st_mode & ALL_BITS) != ALL_BITS) e(19);
        if ((st1.st_mode & ALL_RWXB) != ALL_RWXB) e(20);
    }

    /* Check the timing. */
    if (st1.st_ctime > st2.st_ctime) e(21);
    if (st1.st_ctime > time1) e(22);
    if (st1.st_ctime > time2) e(23);
#ifdef V1_FILESYSTEM
    if (st2.st_ctime < time1) e(24);
#endif
    if (st2.st_ctime > time2) e(25);
    if (st1.st_atime != st2.st_atime) e(26);
    if (st1.st_mtime != st2.st_mtime) e(27);
}
/* End for loop. */

/* Make scratch file. */
if (chmod("foo", ALL_RWXB) != 0) e(28);

if (I_can_chown) {
    /* Do a 20 tests on a gid and uid. */
    for (cnt = 0; cnt < 20; cnt++) {
        /* Get a uid and a gid, test chown. */
        getids(&uid, &gid);
        Stat("foo", &st1);
        if (time(&time1) == (time_t) -1) e(29);
        if (chown("foo", (uid_t) 0, (gid_t) 0) != 0) e(30);
        Stat("foo", &st2);
        if (time(&time2) == (time_t) -1) e(31);

        /* Test the C time field. */
        if (st1.st_ctime > st2.st_ctime) e(32);
        if (st1.st_ctime > time1) e(33);
        if (st1.st_ctime > time2) e(34);
        if (st2.st_ctime < time1) e(35);
        if (st2.st_ctime > time2) e(36);
        if (st1.st_atime != st2.st_atime) e(37);
        if (st1.st_mtime != st2.st_mtime) e(38);

        /* Do additional tests. */
        if (chown("foo", (uid_t) 0, gid) != 0) e(39);
        if (chown("foo", uid, (gid_t) 0) != 0) e(40);
        if (chown("foo", uid, gid) != 0) e(41);
    }
}

if (superuser) {
    /* Check if a non-superuser can change a files gid to gid2 *
     * if gid2 is the current process gid. */
    for (cnt = 0; cnt < 5; cnt++) {
        switch (fork()) {
            case -1:
                printf("Can't fork\n");
                break;
            case 0:
                alarm(20);

                getids(&uid, &gid);
                if (uid == 0) {
                    getids(&uid, &gid);
                    if (uid == 0) e(42);
                }
                getids(&uid2, &gid2);
                if (gid == gid2) e(43);

                /* Creat boo and bar for user uid of group gid. */
                Creat("boo");
                if (chown("boo", uid, gid) != 0) e(44);
                if (chmod("boo", ALL_BITS) != 0) e(45);
                Creat("bar");
                if (chown("bar", uid, gid) != 0) e(46);
                if (chmod("bar", ALL_BITS) != 0) e(47);
            }
        }
    }
}

```

```

        /* We now become user uid of group gid2. */
        setgid(gid2);
        setuid(uid);

        Stat("bar", &st1);
        if (time(&time1) == (time_t) -1) e(48);
        if (chown("bar", uid, gid2) != 0) e(49);
        Stat("bar", &st2);
        if (time(&time2) == (time_t) -1) e(50);

        /* Check if the SET_BITS are cleared. */
        if ((st1.st_mode & ALL_BITS) != ALL_BITS) e(51);
        if ((st2.st_mode & ALL_BITS) != ALL_RWXB) e(52);

        /* Check the st_times. */
        if (st1.st_ctime > st2.st_ctime) e(53);
        if (st1.st_ctime > time1) e(54);
        if (st1.st_ctime > time2) e(55);
        if (st2.st_ctime < time1) e(56);
        if (st2.st_ctime > time2) e(57);
        if (st1.st_atime != st2.st_atime) e(58);
        if (st1.st_mtime != st2.st_mtime) e(59);

        Stat("boo", &st1);
        if (chmod("boo", ALL_BITS) != 0) e(60);
        Stat("boo", &st2);

        /* Check if the set gid bit is cleared. */
        if ((st1.st_mode & ALL_RWXB) != ALL_RWXB) e(61);
        if ((st2.st_mode & S_ISGID) != 0) e(62);

        if (chown("boo", uid, gid2) != 0) e(63);
        Stat("boo", &st1);

        /* Check if the set uid bit is cleared. */
        if ((st1.st_mode & S_ISUID) != 0) e(64);

        exit(0);
    default:
        wait(&stat_loc);
        if (stat_loc != 0) e(65);          /* Alarm? */
    }
}
/* end for loop. */
/* end if (superuser). */
if (chmod("foo", ALL_BITS) != 0) e(66);
Stat("foo", &st1);
if (chown("foo", geteuid(), getegid()) != 0) e(67);
Stat("foo", &st2);
if ((st1.st_mode & ALL_BITS) != ALL_BITS) e(68);          /* See intro! */
if (superuser)
    if ((st2.st_mode & ALL_RWXB) != ALL_RWXB) e(69);
if (!superuser)
    if ((st2.st_mode & ALL_BITS) != ALL_RWXB) e(70);

(void) system("chmod 777 ../DIR_34/*> /dev/null 2> /dev/null");
System("rm -rf ../DIR_34/*");
}

void test34b()
{
    time_t time1, time2;
    mode_t mod;
    struct stat st1, st2;

    subtest = 2;

    /* Test chmod() and chown() on non regular files and on MaxName and
     * MaxPath. * Funny, but dirs should also have S_ISID bits.
     */
    Mkfifo("fifo");
    Mkdir("dir");
    Creat(MaxName);
    MaxPath[strlen(MaxPath) - 2] = '/';

```

```

MaxPath[strlen(MaxPath) - 1] = 'a';    /* make ../../../a */
Creat(MaxPath);

for (mod = 1; mod <= ALL_BITS; mod <= 1) {
    if ((mod & ALL_BITS) != mod) continue; /* bad mod */
    Stat("dir", &st1);
    if (time(&time1) == (time_t) -1) e(1);
    if (chmod("dir", mod) != 0) e(2);
    Stat("dir", &st2);
    if (time(&time2) == (time_t) -1) e(3);
    if (superuser)
        if ((st2.st_mode & ALL_BITS) != mod) e(4);
    if (!superuser)
        if ((st2.st_mode & ALL_RWXB) != (mod & ALL_RWXB)) e(5);

    /* Test the C time field. */
    if (st1.st_ctime > st2.st_ctime) e(6);
    if (st1.st_ctime > time1) e(7);
    if (st1.st_ctime > time2) e(8);
#ifdef V1_FILESYSTEM
    if (st2.st_ctime < time1) e(9);
#endif

    if (st2.st_ctime > time2) e(10);
    if (st1.st_atime != st2.st_atime) e(11);
    if (st1.st_mtime != st2.st_mtime) e(12);

    Stat("fifo", &st1);
    if (time(&time1) == (time_t) -1) e(13);
    if (chmod("fifo", mod) != 0) e(14);
    Stat("fifo", &st2);
    if (time(&time2) == (time_t) -1) e(15);
    if (superuser)
        if ((st2.st_mode & ALL_BITS) != mod) e(16);
    if (!superuser)
        if ((st2.st_mode & ALL_RWXB) != (mod & ALL_RWXB)) e(17);

    /* Test the C time field. */
    if (st1.st_ctime > st2.st_ctime) e(18);
    if (st1.st_ctime > time1) e(19);
    if (st1.st_ctime > time2) e(20);
#ifdef V1_FILESYSTEM
    if (st2.st_ctime < time1) e(21);
#endif

    if (st2.st_ctime > time2) e(22);
    if (st1.st_atime != st2.st_atime) e(23);
    if (st1.st_mtime != st2.st_mtime) e(24);

    Stat(MaxName, &st1);
    if (time(&time1) == (time_t) -1) e(25);
    if (chmod(MaxName, mod) != 0) e(26);
    Stat(MaxName, &st2);
    if (time(&time2) == (time_t) -1) e(27);
    if (superuser)
        if ((st2.st_mode & ALL_BITS) != mod) e(28);
    if (!superuser)
        if ((st2.st_mode & ALL_RWXB) != (mod & ALL_RWXB)) e(29);

    /* Test the C time field. */
    if (st1.st_ctime > st2.st_ctime) e(30);
    if (st1.st_ctime > time1) e(31);
    if (st1.st_ctime > time2) e(32);
#ifdef V1_FILESYSTEM
    if (st2.st_ctime < time1) e(33);
#endif

    if (st2.st_ctime > time2) e(34);
    if (st1.st_atime != st2.st_atime) e(35);
    if (st1.st_mtime != st2.st_mtime) e(36);

    Stat(MaxPath, &st1);
    if (time(&time1) == (time_t) -1) e(37);
    if (chmod(MaxPath, mod) != 0) e(38);
    Stat(MaxPath, &st2);
    if (time(&time2) == (time_t) -1) e(39);
    if (superuser)

```

```

        if ((st2.st_mode & ALL_BITS) != mod) e(40);
    if (!superuser)
        if ((st2.st_mode & ALL_RWXB) != (mod & ALL_RWXB)) e(41);

    /* Test the C time field. */
    if (st1.st_ctime > st2.st_ctime) e(42);
    if (st1.st_ctime > time1) e(43);
    if (st1.st_ctime > time2) e(44);
#ifndef V1_FILESYSTEM
    if (st2.st_ctime < time1) e(45);
#endif
    if (st2.st_ctime > time2) e(46);
    if (st1.st_atime != st2.st_atime) e(47);
    if (st1.st_mtime != st2.st_mtime) e(48);
}

if (chmod("dir", 0777) != 0) e(49);
if (chmod("fifo", 0777) != 0) e(50);
if (chmod(MaxName, 0777) != 0) e(51);
if (chmod(MaxPath, 0777) != 0) e(52);

(void) system("chmod 777 ../DIR_34/* > /dev/null 2> /dev/null");
System("rm -rf ../DIR_34/*");
}

void test34c()
{
    struct stat st;
    uid_t uid, uid2;
    gid_t gid, gid2;
    int stat_loc;

    subtest = 3;

    Mkdir("dir");
    Creat("dir/try_me");

    /* Disalow search permission and see if chmod() and chown() return
     * EACCES.
     */
    if (chmod("dir", ALL_BITS & ~S_IXUSR) != 0) e(1);
    if (!superuser) {
        if (chmod("dir/try_me", 0) != -1) e(2);
        if (errno != EACCES) e(3);
        if (I_can_chown) {
            if (chown("dir/try_me", geteuid(), getegid()) != -1) e(4);
            if (errno != EACCES) e(5);
        }
    }

    /* Check ENOTDIR. */
    Mkfifo("fifo");
    if (chmod("fifo/try_me", 0) != -1) e(6);
    if (errno != ENOTDIR) e(7);
    if (chown("fifo/try_me", geteuid(), getegid()) != -1) e(8);
    if (errno != ENOTDIR) e(9);

    Creat("file");
    if (chmod("file/try_me", 0) != -1) e(10);
    if (errno != ENOTDIR) e(11);
    if (chown("file/try_me", geteuid(), getegid()) != -1) e(12);
    if (errno != ENOTDIR) e(13);

    /* Check empty path. */
    if (chmod("", 0) != -1) e(14);
    if (errno != ENOENT) e(15);
    if (chown("", geteuid(), getegid()) != -1) e(16);
    if (errno != ENOENT) e(17);

    /* Check non existing file name. */
    if (chmod("non_exist", 0) != -1) e(18);
    if (errno != ENOENT) e(19);
    if (chown("non_exist", geteuid(), getegid()) != -1) e(20);
    if (errno != ENOENT) e(21);
}

```

```
/* Check what we get if we do not have permission. */
if (!superuser) {
    Stat("/", &st);
    if (st.st_uid == geteuid()) e(22);

    /* First I had 0, I changed it to st.st_mode 8-). */
    if (chmod("/", st.st_mode) != -1) e(23);
    if (errno != EPERM) e(24);
}
if (!I_can_chown) {
    Stat("/", &st);
    if (st.st_uid == geteuid()) e(25);
    if (chown("/", geteuid(), getegid()) != -1) e(26);
    if (errno != EPERM) e(27);
}

/* If we are superuser, we can test all id combinations. */
if (superuser) {
    switch (fork()) {
        case -1:    printf("Can't fork\n");    break;
        case 0:
            alarm(20);

            getids(&uid, &gid);
            if (uid == 0) {
                getids(&uid, &gid);
                if (uid == 0) e(28);
            }
            getids(&uid2, &gid2);
            if (gid == gid2) e(29);
            if (uid == uid2) e(30);

            /* Creat boo, owned by root. */
            Creat("boo");
            if (chmod("boo", ALL_BITS) != 0) e(31);

            /* Creat boo for user uid2 of group gid2. */
            Creat("bar");
            if (chown("bar", uid2, gid2) != 0) e(32);
            if (chmod("bar", ALL_BITS) != 0) e(33);

            /* Creat my_gid for user uid2 of group gid. */
            Creat("my_gid");
            if (chown("my_gid", uid2, gid) != 0) e(34);
            if (chmod("my_gid", ALL_BITS) != 0) e(35);

            /* Creat my_uid for user uid of uid gid. */
            Creat("my_uid");
            if (chown("my_uid", uid, gid) != 0) e(36);
            if (chmod("my_uid", ALL_BITS) != 0) e(37);

            /* We now become user uid of uid gid. */
            setgid(gid);
            setuid(uid);

            if (chown("boo", uid, gid) != -1) e(38);
            if (errno != EPERM) e(39);
            if (chown("bar", uid, gid) != -1) e(40);
            if (errno != EPERM) e(41);
            if (chown("my_gid", uid, gid) != -1) e(42);
            if (errno != EPERM) e(43);
            if (chown("my_uid", uid, gid2) != -1) e(44);

            /* The EPERM is not strict POSIX. */
            if (errno != EPERM) e(45);

            if (chmod("boo", 0) != -1) e(46);
            if (errno != EPERM) e(47);
            if (chmod("bar", 0) != -1) e(48);
            if (errno != EPERM) e(49);
            if (chmod("my_gid", 0) != -1) e(50);
            if (errno != EPERM) e(51);
    }
}
```

```

        exit(0);
    default:
        wait(&stat_loc);
        if (stat_loc != 0) e(52);          /* Alarm? */
    }
}

/* Check too long path ed. */
Creat(NameTooLong);
if (chmod(NameTooLong, 0777) != 0) e(57);
if (chown(NameTooLong, geteuid(), getegid()) != 0) e(58);

/* Make PathTooLong contain ../.../a */
PathTooLong[strlen(PathTooLong) - 2] = '/';
PathTooLong[strlen(PathTooLong) - 1] = 'a';
Creat("a");
if (chmod(PathTooLong, 0777) != -1) e(59);
if (errno != ENAMETOOLONG) e(60);
if (chown(PathTooLong, geteuid(), getegid()) != -1) e(61);
if (errno != ENAMETOOLONG) e(62);

(void) system("chmod 777 ../DIR_34/* > /dev/null 2> /dev/null");
System("rm -rf ../DIR_34/*");
}

void makelongnames()
{
    register int i;

    memset(MaxName, 'a', NAME_MAX);
    MaxName[NAME_MAX] = '\0';
    for (i = 0; i < PATH_MAX - 1; i++) { /* idem path */
        MaxPath[i++] = '.';
        MaxPath[i] = '/';
    }
    MaxPath[PATH_MAX - 1] = '\0';

    strcpy(NameTooLong, MaxName); /* copy them Max to TooLong */
    strcpy(PathTooLong, MaxPath);

    NameTooLong[NAME_MAX] = 'a';
    NameTooLong[NAME_MAX + 1] = '\0'; /* extend NameTooLong by one too many */
    PathTooLong[PATH_MAX - 1] = '/';
    PathTooLong[PATH_MAX] = '\0'; /* inc PathTooLong by one */
}

void e(n)
int n;
{
    int err_num = errno;          /* Save in case printf clobbers it. */

    printf("Subtest %d, error %d errno=%d: ", subtest, n, errno);
    errno = err_num;
    perror("");
    if (errct++ > MAX_ERROR) {
        printf("Too many errors; test aborted\n");
        chdir("..");
        system("rm -rf DIR*");
        system("rm -rf DIR_34");
        exit(1);
    }
    errno = 0;
}

void quit()
{
    Chdir("..");
    (void) system("chmod 777 DIR_34/* > /dev/null 2> /dev/null");
    System("rm -rf DIR_34");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {

```

```

    printf("%d errors\n", errct);
    exit(1);
}
}

/* Getids returns a valid uid and gid. Is used PASSWD FILE.
 * It assumes the following format for a passwd file line:
 * <user_name>:<passwd>:<uid>:<gid>:<other_stuff>
 * If no uids and gids can be found, it will only return 0 ids.
 */
void getids(r_uid, r_gid)
uid_t * r_uid;
gid_t * r_gid;
{
    char line[N];
    char *p;
    uid_t uid;
    gid_t gid;
    FILE *fp;
    int i;

    static uid_t a_uid[N];      /* Array for uids. */
    static gid_t a_gid[N];      /* Array for gids. */
    static int nuid = 0, ngid = 0; /* The number of user & group ids. */
    static int cuid = 0, cgid = 0; /* The current id index. */

    /* If we don't have any uids go read some from the passwd file. */
    if (nuid == 0) {
        a_uid[nuid++] = 0;      /* Root uid and gid. */
        a_gid[ngid++] = 0;
        if ((fp = fopen(PASSWD_FILE, "r")) == NULL) {
            printf("Can't open ");
            perror(PASSWD_FILE);
        }
        while (fp != NULL && fgets(line, sizeof(line), fp) != NULL) {
            p = strchr(line, ':');
            if (p != NULL) p = strchr(p + 1, ':');
            if (p != NULL) {
                p++;
                uid = 0;
                while (isdigit(*p)) {
                    uid *= 10;
                    uid += (uid_t) (*p - '0');
                    p++;
                }
                if (*p != ':') continue;
                p++;
                gid = 0;
                while (isdigit(*p)) {
                    gid *= 10;
                    gid += (gid_t) (*p - '0');
                    p++;
                }
                if (*p != ':') continue;
                if (nuid < N) {
                    for (i = 0; i < nuid; i++)
                        if (a_uid[i] == uid) break;
                    if (i == nuid) a_uid[nuid++] = uid;
                }
                if (ngid < N) {
                    for (i = 0; i < ngid; i++)
                        if (a_gid[i] == gid) break;
                    if (i == ngid) a_gid[ngid++] = gid;
                }
                if (nuid >= N && ngid >= N) break;
            }
        }
        if (fp != NULL) fclose(fp);
    }

    /* We now have uids and gids in a_uid and a_gid. */
    if (cuid >= nuid) cuid = 0;
    if (cgid >= ngid) cgid = 0;
    *r_uid = a_uid[cuid++];

```

```
*r_gid = a_gid[cgid++];  
}
```



```

/* test35: utime()                                Author: Jan-Mark Wams (jms@cs.vu.nl) */

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <utime.h>
#include <errno.h>
#include <time.h>
#include <ctype.h>
#include <stdio.h>

#define MAX_ERROR      1
#define ITERATIONS     10
#define N 100

#define System(cmd)    if (system(cmd) != 0) printf("“%s” failed\n", cmd)
#define Chdir(dir)    if (chdir(dir) != 0) printf("Can't goto %s\n", dir)
#define Stat(a,b)     if (stat(a,b) != 0) printf("Can't stat %s\n", a)
#define Mkfifo(f)     if (mkfifo(f,0777)!=0) printf("Can't make fifo %s\n", f)
#define Mkdir(f)      if (mkdir(f,0777)!=0) printf("Can't make dir %s\n", f)
#define Creat(f)      if (close(creat(f,0777))!=0) printf("Can't creat %s\n", f)
#define Time(t)       if (time(t) == (time_t)-1) printf("Time error\n")
#define Chown(f,u,g)  if (chown(f,u,g) != 0) printf("Can't chown %s\n", f)
#define Chmod(f,m)    if (chmod(f,m) != 0) printf("Can't chmod %s\n", f)

#define PASSWD_FILE    "/etc/passwd"

int errct = 0;
int subtest = 1;
int I_can_chown;
int superuser;
char MaxName[NAME_MAX + 1]; /* Name of maximum length */
char MaxPath[PATH_MAX]; /* Same for path */
char NameTooLong[NAME_MAX + 2]; /* Name of maximum +1 length */
char PathTooLong[PATH_MAX + 1]; /* Same for path, both too long */

_PROTOTYPE(void main, (int argc, char *argv[]));
_PROTOTYPE(void test35a, (void));
_PROTOTYPE(void test35b, (void));
_PROTOTYPE(void test35c, (void));
_PROTOTYPE(void makelongnames, (void));
_PROTOTYPE(void e, (int number));
_PROTOTYPE(void quit, (void));
_PROTOTYPE(void getids, (uid_t * uid, gid_t * gid));

void main(argc, argv)
int argc;
char *argv[];
{
    int i, m = 0xFFFF;

    sync();
    if (argc == 2) m = atoi(argv[1]);
    printf("Test 35 ");
    fflush(stdout);
    System("rm -rf DIR_35; mkdir DIR_35");
    Chdir("DIR_35");
    makelongnames();
    superuser = (geteuid() == 0);

#ifdef _POSIX_CHOWN_RESTRICTED
# if _POSIX_CHOWN_RESTRICTED - 0 != -1
    I_can_chown = superuser;
# else
    I_can_chown = 1;
# endif
#else
# include "error, this case requires dynamic checks and is not handled"
#endif
}

```

```

for (i = 0; i < ITERATIONS; i++) {
    if (m & 0001) test35a();
    if (m & 0002) test35b();
    if (m & 0004) test35c();
}
quit();
}

void test35a()
{
    struct stat st;
    struct utimbuf ub;
    time_t time1, time2;
    int cnt;

    subtest = 1;

    /* Creat scratch file. */
    Creat("foo");

    /* Set file times back two seconds. */
    Stat("foo", &st);
    ub.actime = st.st_atime - 2;
    ub.modtime = st.st_mtime - 2;
    Time(&time1);
    utime("foo", &ub);
    Time(&time2);
    Stat("foo", &st);
    if (ub.actime != st.st_atime) e(1);
    if (ub.modtime != st.st_mtime) e(2);

    /* The status changed time should be changed. */
#ifdef V1_FILESYSTEM
    if (st.st_ctime < time1) e(3);
#endif
    if (st.st_ctime > time2) e(4);

    /* Add twenty seconds. */
    Stat("foo", &st);
    ub.actime = st.st_atime + 20;
    ub.modtime = st.st_mtime + 20;
    Time(&time1);
    utime("foo", &ub);
    Time(&time2);
    Stat("foo", &st);
    if (ub.actime != st.st_atime) e(5);
    if (ub.modtime != st.st_mtime) e(6);
    if (st.st_ctime < time1) e(7);
#ifdef V1_FILESYSTEM
    if (st.st_ctime > time2) e(8);
#endif

    /* Try 100 times to do utime in less than one second. */
    cnt = 0;
    do {
        Time(&time1);
        utime("foo", (struct utimbuf *) NULL);
        Time(&time2);
    } while (time1 != time2 && cnt++ < 100);
    if (time1 == time2) {
        Stat("foo", &st);
        Time(&time2);
        if (st.st_atime != time1) e(9);
        if (st.st_mtime != time1) e(10);
    } else {
        Stat("foo", &st);
        if (st.st_atime > time2) e(11);
        if (st.st_mtime > time2) e(12);
        Time(&time2);
        if (st.st_atime < time1) e(13);
        if (st.st_mtime < time1) e(14);
    }
    if (st.st_ctime < time1) e(15);
}

```

```
    if (st.st_ctime > time2) e(16);

    System("rm -rf ../DIR_35/*");
}

void test35b()
{
    subtest = 2;

    /* MaxPath and MaxName checkup. */
    Creat(MaxName);
    MaxPath[strlen(MaxPath) - 2] = '/';
    MaxPath[strlen(MaxPath) - 1] = 'a';    /* make ../.../a */
    Creat(MaxPath);
    if (utime(MaxName, NULL) != 0) e(1);
    if (utime(MaxPath, NULL) != 0) e(2);

    /* The owner doesn't need write permission to set times. */
    Creat("foo");
    if (chmod("foo", 0) != 0) e(3);
    if (utime("foo", NULL) != 0) e(4);
    if (chmod("foo", 0777) != 0) e(5);
    if (utime("foo", NULL) != 0) e(6);

    System("rm -rf ../DIR_35/*");
}

void test35c()
{
    gid_t gid, gid2;
    uid_t uid, uid2;
    struct utimbuf ub;
    int stat_loc;

    subtest = 3;

    /* Access problems. */
    Mkdir("bar");
    Creat("bar/tryme");
    if (superuser) {
        Chmod("bar", 0000);    /* No search permission at all. */
        if (utime("bar/tryme", NULL) != 0) e(1);
    }
    if (!superuser) {
        Chmod("bar", 0677);    /* No search permission. */
        if (utime("bar/tryme", NULL) != -1) e(2);
        if (errno != EACCES) e(3);
    }
    Chmod("bar", 0777);

    if (I_can_chown) {
        switch (fork()) {
            case -1:    printf("Can't fork\n");    break;
            case 0:
                alarm(20);

                /* Get two differend non root uids. */
                if (superuser) {
                    getids(&uid, &gid);
                    if (uid == 0) getids(&uid, &gid);
                    if (uid == 0) e(4);
                }
                if (!superuser) {
                    uid = geteuid();
                    gid = getegid();
                }
                getids(&uid2, &gid2);
                if (uid == uid2) getids(&uid2, &gid2);
                if (uid == uid2) e(5);

                /* Creat a number of files for root, user and user2. */
                Creat("rootfile");    /* Owned by root. */
                Chmod("rootfile", 0600);
                Chown("rootfile", 0, 0);
            }
        }
    }
}
```

```

        Creat("user2file");          /* Owned by user 2, writeable. */
        Chmod("user2file", 0020);
        Chown("user2file", uid2, gid);
        Creat("user2private");        /* Owned by user 2, privately. */
        Chmod("user2private", 0600);
        Chown("user2private", uid2, gid);

        if (superuser) {
            setgid(gid);
            setuid(uid);
        }

        /* We now are user ``uid`` from group ``gid``. */
        ub.actime = (time_t) 12345L;
        ub.modtime = (time_t) 12345L;

        if (utime("rootfile", NULL) != -1) e(6);
        if (errno != EACCES) e(7);
        if (utime("rootfile", &ub) != -1) e(8);
        if (errno != EPERM) e(9);

        if (utime("user2file", NULL) != 0) e(10);
        if (utime("user2file", &ub) != -1) e(11);
        if (errno != EPERM) e(12);

        if (utime("user2private", NULL) != -1) e(13);
        if (errno != EACCES) e(14);
        if (utime("user2private", &ub) != -1) e(15);
        if (errno != EPERM) e(16);

        exit(errct ? 1 : 0);
    default:
        wait(&stat_loc);
        if (stat_loc != 0) e(17);          /* Alarm? */
    }
}

/* Test names that are too long. */
#ifdef _POSIX_NO_TRUNC
# if _POSIX_NO_TRUNC - 0 != -1
    /* Not exist might also be a propper response? */
    if (utime(NameTooLong, NULL) != -1) e(18);
    if (errno != ENAMETOOLONG) e(19);
# else
    Creat(NameTooLong);
    if (utime(NameTooLong, NULL) != 0) e(20);
# endif
#else
# include "error, this case requires dynamic checks and is not handled"
#endif

/* Make PathTooLong contain ../../../a */
PathTooLong[strlen(PathTooLong) - 2] = '/';
PathTooLong[strlen(PathTooLong) - 1] = 'a';
Creat("a");
if (utime(PathTooLong, NULL) != -1) e(21);
if (errno != ENAMETOOLONG) e(22);

/* Non existing file name. */
if (utime("nonexist", NULL) != -1) e(23);
if (errno != ENOENT) e(24);

/* Empty file name. */
if (utime("", NULL) != -1) e(25);
if (errno != ENOENT) e(26);

System("rm -rf ../DIR_35/*");
}

void makelongnames()
{
    register int i;

    memset(MaxName, 'a', NAME_MAX);

```

```

MaxName[NAME_MAX] = '\0';
for (i = 0; i < PATH_MAX - 1; i++) { /* idem path */
    MaxPath[i++] = '.';
    MaxPath[i] = '/';
}
MaxPath[PATH_MAX - 1] = '\0';

strcpy(NameTooLong, MaxName); /* copy them Max to TooLong */
strcpy(PathTooLong, MaxPath);

NameTooLong[NAME_MAX] = 'a';
NameTooLong[NAME_MAX + 1] = '\0'; /* extend NameTooLong by one too many*/
PathTooLong[PATH_MAX - 1] = '/';
PathTooLong[PATH_MAX] = '\0'; /* inc PathTooLong by one */
}

void e(n)
int n;
{
    int err_num = errno; /* Save in case printf clobbers it. */

    printf("Subtest %d, error %d errno=%d: ", subtest, n, errno);
    errno = err_num;
    perror("");
    if (errct++ > MAX_ERROR) {
        printf("Too many errors; test aborted\n");
        chdir("..");
        system("rm -rf DIR* > /dev/null 2> /dev/null");
        exit(1);
    }
    errno = 0;
}

void quit()
{
    Chdir("..");
    System("rm -rf DIR_35");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
        printf("%d errors\n", errct);
        exit(1);
    }
}

/* Getids returns a valid uid and gid. Is used PASSWD FILE.
** It assumes the following format for a passwd file line:
** <user_name>:<passwd>:<uid>:<gid>:<other_stuff>
** If no uids and gids can be found, it will only return 0 ids.
*/
void getids(r_uid, r_gid)
uid_t *r_uid;
gid_t *r_gid;
{
    char line[N];
    char *p;
    uid_t uid;
    gid_t gid;
    FILE *fp;
    int i;

    static uid_t a_uid[N]; /* Array for uids. */
    static gid_t a_gid[N]; /* Array for gids. */
    static int nuid = 0, ngid = 0; /* The number of user & group ids. */
    static int cuid = 0, cgid = 0; /* The current id index. */

    /* If we don't have any uids go read some from the passwd file. */
    if (nuid == 0) {
        a_uid[nuid++] = 0; /* Root uid and gid. */
        a_gid[ngid++] = 0;
        if ((fp = fopen(PASSWD_FILE, "r")) == NULL) {
            printf("Can't open ");

```

```
        perror(PASSWD_FILE);
    }
    while (fp != NULL && fgets(line, sizeof(line), fp) != NULL) {
        p = strchr(line, ':');
        if (p != NULL) p = strchr(p + 1, ':');
        if (p != NULL) {
            p++;
            uid = 0;
            while (isdigit(*p)) {
                uid *= 10;
                uid += (uid_t) (*p - '0');
                p++;
            }
            if (*p != ':') continue;
            p++;
            gid = 0;
            while (isdigit(*p)) {
                gid *= 10;
                gid += (gid_t) (*p - '0');
                p++;
            }
            if (*p != ':') continue;
            if (nuid < N) {
                for (i = 0; i < nuid; i++)
                    if (a_uid[i] == uid) break;
                if (i == nuid) a_uid[nuid++] = uid;
            }
            if (ngid < N) {
                for (i = 0; i < ngid; i++)
                    if (a_gid[i] == gid) break;
                if (i == ngid) a_gid[ngid++] = gid;
            }
            if (nuid >= N && ngid >= N) break;
        }
    }
    if (fp != NULL) fclose(fp);
}

/* We now have uids and gids in a_uid and a_gid. */
if (cuid >= nuid) cuid = 0;
if (cgid >= ngid) cgid = 0;
*r_uid = a_uid[cuid++];
*r_gid = a_gid[cgid++];
}
```

```

/* test36: pathconf() fpathconf()      Author: Jan-mark Wams */

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <errno.h>
#include <time.h>
#include <stdio.h>

#define MAX_ERROR      4
#define ITERATIONS     10

#define System(cmd)     if (system(cmd) != 0) printf("“%s” failed\n", cmd)
#define Chdir(dir)     if (chdir(dir) != 0) printf("Can't goto %s\n", dir)
#define Stat(a,b)      if (stat(a,b) != 0) printf("Can't stat %s\n", a)

int errct = 0;
int subtest = 1;
int superuser;
char MaxName[NAME_MAX + 1]; /* Name of maximum length */
char MaxPath[PATH_MAX];    /* Same for path */
char ToLongName[NAME_MAX + 2]; /* Name of maximum +1 length */
char ToLongPath[PATH_MAX + 1]; /* Same for path, both too long */

_PROTOTYPE(void main, (int argc, char *argv[]));
_PROTOTYPE(void test36a, (void));
_PROTOTYPE(void test36b, (void));
_PROTOTYPE(void test36c, (void));
_PROTOTYPE(void test36d, (void));
_PROTOTYPE(void makelongnames, (void));
_PROTOTYPE(void e, (int number));
_PROTOTYPE(void quit, (void));
_PROTOTYPE(int not_provided_option, (int _option));
_PROTOTYPE(int provided_option, (int _option, int _minimum_value));
_PROTOTYPE(int variating_option, (int _option, int _minimum_value));

char *testdirs[] = {
    "/",
    "/etc",
    "/tmp",
    "/usr",
    "/usr/bin",
    ".",
    NULL
};

char *testfiles[] = {
    "/",
    "/etc",
    "/etc/passwd",
    "/tmp",
    "/dev/tty",
    "/usr",
    "/usr/bin",
    ".",
    NULL
};

void main(argc, argv)
int argc;
char *argv[];
{
    int i, m = 0xFFFF;

    sync();
    if (argc == 2) m = atoi(argv[1]);
    printf("Test 36 ");
    fflush(stdout);
    System("rm -rf DIR_36; mkdir DIR_36");

```

```
Chdir("DIR_36");
makelongnames();
superuser = (geteuid() == 0);

for (i = 0; i < ITERATIONS; i++) {
    if (m & 0001) test36a();
    if (m & 0002) test36b();
    if (m & 0004) test36c();
    if (m & 0010) test36d();
}
quit();
}

void test36a()
{
    /* Test normal operation. */
    subtest = 1;
    System("rm -rf ../DIR_36/*");

#ifdef _POSIX_CHOWN_RESTRICTED
# if _POSIX_CHOWN_RESTRICTED - 0 == -1
    if (not_provided_option(_PC_CHOWN_RESTRICTED) != 0) e(1);
# else
    if (provided_option(_PC_CHOWN_RESTRICTED, 0) != 0) e(2);
# endif
#else
    if (varying_option(_PC_CHOWN_RESTRICTED, 0) != 0) e(3);
#endif

#ifdef _POSIX_NO_TRUNC
# if _POSIX_NO_TRUNC - 0 == -1
    if (not_provided_option(_PC_NO_TRUNC) != 0) e(4);
# else
    if (provided_option(_PC_NO_TRUNC, 0) != 0) e(5);
# endif
#else
    if (varying_option(_PC_NO_TRUNC, 0) != 0) e(6);
#endif

#ifdef _POSIX_VDISABLE
# if _POSIX_VDISABLE - 0 == -1
    if (not_provided_option(_PC_VDISABLE) != 0) e(7);
# else
    if (provided_option(_PC_VDISABLE, 0) != 0) e(8);
# endif
#else
    if (varying_option(_PC_VDISABLE, 0) != 0) e(9);
#endif
}

void test36b()
{
    subtest = 2;
    System("rm -rf ../DIR_36/*");
}

void test36c()
{
    subtest = 3;
    System("rm -rf ../DIR_36/*");
}

void test36d()
{
    subtest = 4;
    System("rm -rf ../DIR_36/*");
}

void makelongnames()
{
    register int i;

    memset(MaxName, 'a', NAME_MAX);
    MaxName[NAME_MAX] = '\0';
}
```



```
for (i = 0; i < PATH_MAX - 1; i++) { /* idem path */
    MaxPath[i++] = '.';
    MaxPath[i] = '/';
}
MaxPath[PATH_MAX - 1] = '\0';

strcpy(ToLongName, MaxName); /* copy them Max to ToLong */
strcpy(ToLongPath, MaxPath);

ToLongName[NAME_MAX] = 'a';
ToLongName[NAME_MAX + 1] = '\0'; /* extend ToLongName by one too many */
ToLongPath[PATH_MAX - 1] = '/';
ToLongPath[PATH_MAX] = '\0'; /* inc ToLongPath by one */
}

void e(n)
int n;
{
    int err_num = errno; /* Save in case printf clobbers it. */

    printf("Subtest %d, error %d errno=%d: ", subtest, n, errno);
    errno = err_num;
    perror("");
    if (errct++ > MAX_ERROR) {
        printf("Too many errors; test aborted\n");
        chdir("..");
        system("rm -rf DIR*");
        exit(1);
    }
    errno = 0;
}

void quit()
{
    Chdir("..");
    System("rm -rf DIR_36");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
        printf("%d errors\n", errct);
        exit(1);
    }
}

int not_provided_option(option)
int option;
{
    char **p;

    for (p = testfiles; *p != (char *) NULL; p++) {
        if (pathconf(*p, option) != -1) return printf("*p== %s\n", *p), 1;
    }
    return 0;
}

int provided_option(option, minimum)
int option, minimum;
{
    char **p;

    /* These three options are only defined on directories. */
    if (option == _PC_NO_TRUNC
        || option == _PC_NAME_MAX
        || option == _PC_PATH_MAX)
        p = testdirs;
    else
        p = testfiles;

    for (; *p != NULL; p++) {
        if (pathconf(*p, option) < minimum)
            return printf("*p== %s\n", *p), 1;
    }
}
```

```
    return 0;
}

int variating_option(option, minimum)
int option, minimum;
{
    char **p;

    /* These three options are only defined on directorys. */
    if (option == _PC_NO_TRUNC
        || option == _PC_NAME_MAX
        || option == _PC_PATH_MAX)
        p = testdirs;
    else
        p = testfiles;

    for (; *p != NULL; p++) {
        if (pathconf(*p, option) < minimum)
            return printf("p== %s\n", *p), 1;
    }
    return 0;
}
```

```
/* test 37 - signals */

#include <sys/types.h>
#include <sys/times.h>
#ifdef _MINIX
#include <sys/sigcontext.h>
#endif
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <setjmp.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

#define ITERATIONS 2
#define SIGS 14
#define MAX_ERROR 4

int iteration, cumsig, subtest, errct = 0, sig1, sig2;

int sigarray[SIGS] = {SIGHUP, SIGILL, SIGTRAP, SIGABRT, SIGIOT,
                      SIGFPE, SIGUSR1, SIGSEGV, SIGUSR2, SIGPIPE, SIGALRM,
                      SIGTERM};

/* Prototypes produced automatically by mkptypes. */
_PROTOTYPE(int main, (int argc, char *argv []));
_PROTOTYPE(void test37a, (void));
_PROTOTYPE(void func1, (int sig));
_PROTOTYPE(void func2, (int sig));
_PROTOTYPE(void test37b, (void));
_PROTOTYPE(void catch1, (int signo));
_PROTOTYPE(void catch2, (int signo));
_PROTOTYPE(void test37c, (void));
_PROTOTYPE(void catch3, (int signo));
_PROTOTYPE(void test37d, (void));
_PROTOTYPE(void catch4, (int signo));
_PROTOTYPE(void test37e, (void));
_PROTOTYPE(void catch5, (int signo));
_PROTOTYPE(void test37f, (void));
_PROTOTYPE(void sigint_handler, (int signo));
_PROTOTYPE(void sigpipe_handler, (int signo));
_PROTOTYPE(void test37g, (void));
_PROTOTYPE(void sighup8, (int signo));
_PROTOTYPE(void sigpip8, (int signo));
_PROTOTYPE(void sigter8, (int signo));
_PROTOTYPE(void test37h, (void));
_PROTOTYPE(void sighup9, (int signo));
_PROTOTYPE(void sigter9, (int signo));
_PROTOTYPE(void test37i, (void));
_PROTOTYPE(void sighup10, (int signo));
_PROTOTYPE(void sigalrm_handler10, (int signo));
_PROTOTYPE(void test37j, (void));
_PROTOTYPE(void test37k, (void));
_PROTOTYPE(void test37l, (void));
_PROTOTYPE(void func_m1, (void));
_PROTOTYPE(void func_m2, (void));
_PROTOTYPE(void test37m, (void));
_PROTOTYPE(void longjerr, (void));
_PROTOTYPE(void catch14, (int signo, int code, struct sigcontext * scp));
_PROTOTYPE(void test37n, (void));
_PROTOTYPE(void catch15, (int signo));
_PROTOTYPE(void test37o, (void));
_PROTOTYPE(void clearsigsigstate, (void));
_PROTOTYPE(void quit, (void));
_PROTOTYPE(void wait_for, (int pid));
_PROTOTYPE(void e, (int n));

int main(argc, argv)
int argc;
char *argv[];
{
    int i, m = 0xFFFF;
```

```
sync();

if (argc == 2) m = atoi(argv[1]);

printf("Test 37 ");
fflush(stdout);          /* have to flush for child's benefit */

system("rm -rf DIR_37; mkdir DIR_37");
chdir("DIR_37");

for (i = 0; i < ITERATIONS; i++) {
    iteration = i;
    if (m & 0000001) test37a();
    if (m & 0000002) test37b();
    if (m & 0000004) test37c();
    if (m & 0000010) test37d();
    if (m & 0000020) test37e();
    if (m & 0000040) test37f();
    if (m & 0000100) test37g();
    if (m & 0000200) test37h();
    if (m & 0000400) test37i();
    if (m & 0001000) test37j();
    if (m & 0002000) test37k();
    if (m & 0004000) test37l();
    if (m & 0010000) test37m();
    if (m & 0020000) test37n();
    if (m & 0040000) test37o();
}

quit();
return(-1);              /* impossible */
}

void test37a()
{
    /* Test signal set management. */

    sigset_t s;

    subtest = 1;
    clearsigtstate();

    /* Create an empty set and see if any bits are on. */
    if (sigemptyset(&s) != 0) e(1);
    if (sigismember(&s, SIGHUP) != 0) e(2);
    if (sigismember(&s, SIGINT) != 0) e(3);
    if (sigismember(&s, SIGQUIT) != 0) e(4);
    if (sigismember(&s, SIGILL) != 0) e(5);
    if (sigismember(&s, SIGTRAP) != 0) e(6);
    if (sigismember(&s, SIGABRT) != 0) e(7);
    if (sigismember(&s, SIGIOT) != 0) e(8);
    if (sigismember(&s, SIGFPE) != 0) e(10);
    if (sigismember(&s, SIGKILL) != 0) e(11);
    if (sigismember(&s, SIGUSR1) != 0) e(12);
    if (sigismember(&s, SIGSEGV) != 0) e(13);
    if (sigismember(&s, SIGUSR2) != 0) e(14);
    if (sigismember(&s, SIGPIPE) != 0) e(15);
    if (sigismember(&s, SIGALRM) != 0) e(16);
    if (sigismember(&s, SIGTERM) != 0) e(17);

    /* Create a full set and see if any bits are off. */
    if (sigfillset(&s) != 0) e(19);
    if (sigemptyset(&s) != 0) e(20);
    if (sigfillset(&s) != 0) e(21);
    if (sigismember(&s, SIGHUP) != 1) e(22);
    if (sigismember(&s, SIGINT) != 1) e(23);
    if (sigismember(&s, SIGQUIT) != 1) e(24);
    if (sigismember(&s, SIGILL) != 1) e(25);
    if (sigismember(&s, SIGTRAP) != 1) e(26);
    if (sigismember(&s, SIGABRT) != 1) e(27);
    if (sigismember(&s, SIGIOT) != 1) e(28);
    if (sigismember(&s, SIGFPE) != 1) e(30);
    if (sigismember(&s, SIGKILL) != 1) e(31);
    if (sigismember(&s, SIGUSR1) != 1) e(32);
}
```

```
if (sigismember(&s, SIGSEGV) != 1) e(33);
if (sigismember(&s, SIGUSR2) != 1) e(34);
if (sigismember(&s, SIGPIPE) != 1) e(35);
if (sigismember(&s, SIGALRM) != 1) e(36);
if (sigismember(&s, SIGTERM) != 1) e(37);

/* Create an empty set, then turn on bits individually. */
if (sigemptyset(&s) != 0) e(39);
if (sigaddset(&s, SIGHUP) != 0) e(40);
if (sigaddset(&s, SIGINT) != 0) e(41);
if (sigaddset(&s, SIGQUIT) != 0) e(42);
if (sigaddset(&s, SIGILL) != 0) e(43);
if (sigaddset(&s, SIGTRAP) != 0) e(44);

/* See if the bits just turned on are indeed on. */
if (sigismember(&s, SIGHUP) != 1) e(45);
if (sigismember(&s, SIGINT) != 1) e(46);
if (sigismember(&s, SIGQUIT) != 1) e(47);
if (sigismember(&s, SIGILL) != 1) e(48);
if (sigismember(&s, SIGTRAP) != 1) e(49);

/* The others should be turned off. */
if (sigismember(&s, SIGABRT) != 0) e(50);
if (sigismember(&s, SIGIOT) != 0) e(51);
if (sigismember(&s, SIGFPE) != 0) e(53);
if (sigismember(&s, SIGKILL) != 0) e(54);
if (sigismember(&s, SIGUSR1) != 0) e(55);
if (sigismember(&s, SIGSEGV) != 0) e(56);
if (sigismember(&s, SIGUSR2) != 0) e(57);
if (sigismember(&s, SIGPIPE) != 0) e(58);
if (sigismember(&s, SIGALRM) != 0) e(59);
if (sigismember(&s, SIGTERM) != 0) e(60);

/* Now turn them off and see if all are off. */
if (sigdelset(&s, SIGHUP) != 0) e(62);
if (sigdelset(&s, SIGINT) != 0) e(63);
if (sigdelset(&s, SIGQUIT) != 0) e(64);
if (sigdelset(&s, SIGILL) != 0) e(65);
if (sigdelset(&s, SIGTRAP) != 0) e(66);

if (sigismember(&s, SIGHUP) != 0) e(67);
if (sigismember(&s, SIGINT) != 0) e(68);
if (sigismember(&s, SIGQUIT) != 0) e(69);
if (sigismember(&s, SIGILL) != 0) e(70);
if (sigismember(&s, SIGTRAP) != 0) e(71);
if (sigismember(&s, SIGABRT) != 0) e(72);
if (sigismember(&s, SIGIOT) != 0) e(73);
if (sigismember(&s, SIGFPE) != 0) e(75);
if (sigismember(&s, SIGKILL) != 0) e(76);
if (sigismember(&s, SIGUSR1) != 0) e(77);
if (sigismember(&s, SIGSEGV) != 0) e(78);
if (sigismember(&s, SIGUSR2) != 0) e(79);
if (sigismember(&s, SIGPIPE) != 0) e(80);
if (sigismember(&s, SIGALRM) != 0) e(81);
if (sigismember(&s, SIGTERM) != 0) e(82);
}

void func1(sig)
int sig;
{
    sig1++;
}

void func2(sig)
int sig;
{
    sig2++;
}

void test37b()
{
    /* Test sigprocmask and sigpending. */
    int i;
    pid_t p;
```

```

sigset_t s, s1, s_empty, s_full, s_ill, s_ill_pip, s_nokill, s_nokill_stop;
struct sigaction sa, osa;

subtest = 2;
clearsigstate();

/* Construct s_ill = {SIGILL} and s_ill_pip {SIGILL | SIGPIPE}, etc. */
if (sigemptyset(&s_empty) != 0) e(1);
if (sigemptyset(&s_ill) != 0) e(2);
if (sigemptyset(&s_ill_pip) != 0) e(3);
if (sigaddset(&s_ill, SIGILL) != 0) e(4);
if (sigaddset(&s_ill_pip, SIGILL) != 0) e(5);
if (sigaddset(&s_ill_pip, SIGPIPE) != 0) e(6);
if (sigfillset(&s_full) != 0) e(7);
s_nokill = s_full;
if (sigdelset(&s_nokill, SIGKILL) != 0) e(8);
s_nokill_stop = s_nokill;
if (sigdelset(&s_nokill_stop, SIGSTOP) != 0) e(8);
#ifdef _MINIX /* XXX - should unsupported signals be <= _NSIG? */
if (SIGSTOP > _NSIG) e(666);
if (SIGSTOP <= _NSIG && sigdelset(&s_nokill, SIGSTOP) != 0) e(888);
#endif /* _MINIX */

/* Now get most of the signals into default state. Don't change SIGINT
 * or SIGQUIT, so this program can be killed. SIGKILL is also special.
 */
sa.sa_handler = SIG_DFL;
sa.sa_mask = s_empty;
sa.sa_flags = 0;
for (i = 0; i < SIGS; i++) sigaction(i, &sa, &osa);

/* The second argument may be zero. See if it wipes out the system. */
for (i = 0; i < SIGS; i++) sigaction(i, (struct sigaction *) NULL, &osa);

/* Install a signal handler. */
sa.sa_handler = func1;
sa.sa_mask = s_ill;
sa.sa_flags = SA_NODEFER | SA_NOCLDSTOP;
osa.sa_handler = SIG_IGN;
osa.sa_mask = s_empty;
osa.sa_flags = 0;
if (sigaction(SIGHUP, &sa, &osa) != 0) e(9);
if (osa.sa_handler != SIG_DFL) e(10);
if (osa.sa_mask != 0) e(11);
if (osa.sa_flags != s_empty) e(12);

/* Replace action and see if old value is read back correctly. */
sa.sa_handler = func2;
sa.sa_mask = s_ill_pip;
sa.sa_flags = SA_RESETHAND | SA_NODEFER;
osa.sa_handler = SIG_IGN;
osa.sa_mask = s_empty;
osa.sa_flags = 0;
if (sigaction(SIGHUP, &sa, &osa) != 0) e(13);
if (osa.sa_handler != func1) e(14);
if (osa.sa_mask != s_ill) e(15);
if (osa.sa_flags != SA_NODEFER
    && osa.sa_flags != (SA_NODEFER | SA_NOCLDSTOP)) e(16);

/* Replace action once more and check what is read back. */
sa.sa_handler = SIG_DFL;
sa.sa_mask = s_empty;
osa.sa_handler = SIG_IGN;
osa.sa_mask = s_empty;
osa.sa_flags = 0;
if (sigaction(SIGHUP, &sa, &osa) != 0) e(17);
if (osa.sa_handler != func2) e(18);
if (osa.sa_mask != s_ill_pip) e(19);
if (osa.sa_flags != (SA_RESETHAND | SA_NODEFER)) e(20);

/* Test sigprocmask(SIG_SETMASK, ...). */
if (sigprocmask(SIG_SETMASK, &s_full, &s1) != 0) e(18); /* block all */
if (sigemptyset(&s1) != 0) e(19);
errno = 0;

```

```
if (sigprocmask(SIG_SETMASK, &s_empty, &s1) != 0) e(20); /* block none */
if (s1 != s_nokill_stop) e(21);
if (sigprocmask(SIG_SETMASK, &s_ill, &s1) != 0) e(22); /* block SIGILL */
errno = 0;
if (s1 != s_empty) e(23);
if (sigprocmask(SIG_SETMASK, &s_ill_pip, &s1) != 0) e(24); /* SIGILL+PIP */
if (s1 != s_ill) e(25);
if (sigprocmask(SIG_SETMASK, &s_full, &s1) != 0) e(26); /* block all */
if (s1 != s_ill_pip) e(27);

/* Test sigprocmask(SIG_UNBLOCK, ...) */
if (sigprocmask(SIG_UNBLOCK, &s_ill, &s1) != 0) e(28);
if (s1 != s_nokill_stop) e(29);
if (sigprocmask(SIG_UNBLOCK, &s_ill_pip, &s1) != 0) e(30);
s = s_nokill_stop;
if (sigdelset(&s, SIGILL) != 0) e(31);
if (s != s1) e(32);
if (sigprocmask(SIG_UNBLOCK, &s_empty, &s1) != 0) e(33);
s = s_nokill_stop;
if (sigdelset(&s, SIGILL) != 0) e(34);
if (sigdelset(&s, SIGPIPE) != 0) e(35);
if (s != s1) e(36);
s1 = s_nokill_stop;
if (sigprocmask(SIG_SETMASK, &s_empty, &s1) != 0) e(37);
if (s != s1) e(38);

/* Test sigprocmask(SIG_BLOCK, ...) */
if (sigprocmask(SIG_BLOCK, &s_ill, &s1) != 0) e(39);
if (s1 != s_empty) e(40);
if (sigprocmask(SIG_BLOCK, &s_ill_pip, &s1) != 0) e(41);
if (s1 != s_ill) e(42);
if (sigprocmask(SIG_SETMASK, &s_full, &s1) != 0) e(43);
if (s1 != s_ill_pip) e(44);

/* Check error condition. */
errno = 0;
if (sigprocmask(20000, &s_full, &s1) != -1) e(45);
if (errno != EINVAL) e(46);
if (sigprocmask(SIG_SETMASK, &s_full, &s1) != 0) e(47);
if (s1 != s_nokill_stop) e(48);

/* If second arg is 0, nothing is set. */
if (sigprocmask(SIG_SETMASK, (sigset_t *) NULL, &s1) != 0) e(49);
if (s1 != s_nokill_stop) e(50);
if (sigprocmask(SIG_SETMASK, &s_ill_pip, &s1) != 0) e(51);
if (s1 != s_nokill_stop) e(52);
if (sigprocmask(SIG_SETMASK, (sigset_t *) NULL, &s1) != 0) e(53);
if (s1 != s_ill_pip) e(54);
if (sigprocmask(SIG_BLOCK, (sigset_t *) NULL, &s1) != 0) e(55);
if (s1 != s_ill_pip) e(56);
if (sigprocmask(SIG_UNBLOCK, (sigset_t *) NULL, &s1) != 0) e(57);
if (s1 != s_ill_pip) e(58);

/* Trying to block SIGKILL is not allowed, but is not an error, either. */
s = s_empty;
if (sigaddset(&s, SIGKILL) != 0) e(59);
if (sigprocmask(SIG_BLOCK, &s, &s1) != 0) e(60);
if (s1 != s_ill_pip) e(61);
if (sigprocmask(SIG_SETMASK, &s_full, &s1) != 0) e(62);
if (s1 != s_ill_pip) e(63);

/* Test sigpending. At this moment, all signals are blocked. */
sa.sa_handler = func2;
sa.sa_mask = s_empty;
if (sigaction(SIGHUP, &sa, &osa) != 0) e(64);
p = getpid();
kill(p, SIGHUP); /* send SIGHUP to self */
if (sigpending(&s) != 0) e(65);
if (sigemptyset(&s1) != 0) e(66);
if (sigaddset(&s1, SIGHUP) != 0) e(67);
if (s != s1) e(68);
sa.sa_handler = SIG_IGN;
if (sigaction(SIGHUP, &sa, &osa) != 0) e(69);
if (sigpending(&s) != 0) e(70);
```

```

    if (s != s_empty) e(71);
}

/*-----*/
int x;
sigset_t glo_vol_set;

void catch1(signo)
int signo;
{
    x = 42;
}

void catch2(signo)
int signo;
{
    if (sigprocmask(SIG_BLOCK, (sigset_t *)NULL, (sigset_t *) &glo_vol_set) != 0)
        e(1);
}

/* Verify that signal(2), which is now built on top of sigaction(2), still
 * works.
 */
void test37c()
{
    pid_t pid;
    sigset_t sigset_var;

    subtest = 3;
    clearsigtstate();
    x = 0;

    /* Verify an installed signal handler persists across a fork(2). */
    if (signal(SIGTERM, catch1) == SIG_ERR) e(1);
    switch (pid = fork()) {
        case 0: /* child */
            errct = 0;
            while (x == 0);
            if (x != 42) e(2);
            exit(errct == 0 ? 0 : 1);
        case -1: e(3); break;
        default: /* parent */
            sleep(1);
            if (kill(pid, SIGTERM) != 0) e(4);
            wait_for(pid);
            break;
    }

    /* Verify that the return value is the previous handler. */
    signal(SIGINT, SIG_IGN);
    if (signal(SIGINT, catch2) != SIG_IGN) e(5);
    if (signal(SIGINT, catch1) != catch2) e(6);
    if (signal(SIGINT, SIG_DFL) != catch1) e(7);
    if (signal(SIGINT, catch1) != SIG_DFL) e(8);
    if (signal(SIGINT, SIG_DFL) != catch1) e(9);
    if (signal(SIGINT, SIG_DFL) != SIG_DFL) e(10);
    if (signal(SIGINT, catch1) != SIG_DFL) e(11);

    /* Verify that SIG_ERR is correctly generated. */
    if (signal(_NSIG + 1, catch1) != SIG_ERR) e(12);
    if (signal(0, catch1) != SIG_ERR) e(13);
    if (signal(-1, SIG_DFL) != SIG_ERR) e(14);

    /* Verify that caught signals are automatically reset to the default,
     * and that further instances of the same signal are not blocked here
     * or in the signal handler.
     */
    if (signal(SIGTERM, catch1) == SIG_ERR) e(15);
    switch ((pid = fork())) {
        case 0: /* child */
            errct = 0;
            while (x == 0);
            if (x != 42) e(16);
            if (sigismember((sigset_t *) &glo_vol_set, SIGTERM)) e(17);

```



```

    if (sigprocmask(SIG_BLOCK, (sigset_t *)NULL, &sigset_var) != 0) e(18);
    if (sigismember(&sigset_var, SIGTERM)) e(19);

#if 0
/* Use this if you have compiled signal() to have the broken SYSV behaviour. */
    if (signal(SIGTERM, catch1) != SIG_DFL) e(20);
#else
    if (signal(SIGTERM, catch1) != catch1) e(20);
#endif
    exit(errct == 0 ? 0 : 1);
    default: /* parent */
        sleep(1);
        if (kill(pid, SIGTERM) != 0) e(21);
        wait_for(pid);
        break;
    case -1: e(22); break;
}
}

/*-----*/
/* Test that the signal handler can be invoked recursively with the
 * state being properly saved and restored.
 */

static int y;
static int z;

void catch3(signo)
int signo;
{
    if (z == 1) { /* catching a nested signal */
        y = 2;
        return;
    }
    z = 1;
    if (kill(getpid(), SIGHUP) != 0) e(1);
    while (y != 2);
    y = 1;
}

void test37d()
{
    struct sigaction act;

    subtest = 4;
    clearsigtstate();
    y = 0;
    z = 0;

    act.sa_handler = catch3;
    act.sa_mask = 0;
    act.sa_flags = SA_NODEFER; /* Otherwise, nested occurrence of
                               * SIGINT is blocked. */
    if (sigaction(SIGHUP, &act, (struct sigaction *) NULL) != 0) e(2);
    if (kill(getpid(), SIGHUP) != 0) e(3);
    if (y != 1) e(4);
}

/*-----*/
/* Test that the signal mask in effect for the duration of a signal handler
 * is as specified in POSIX Section 3, lines 718 -724. Test that the
 * previous signal mask is restored when the signal handler returns.
 */

void catch4(signo)
int signo;
{
    sigset_t oset;
    sigset_t set;

    if (sigemptyset(&set) == -1) e(5001);
    if (sigaddset(&set, SIGTERM) == -1) e(5002);
    if (sigaddset(&set, SIGHUP) == -1) e(5003);

```

```

if (sigaddset(&set, SIGINT) == -1) e(5004);
if (sigaddset(&set, SIGPIPE) == -1) e(5005);
if (sigprocmask(SIG_BLOCK, (sigset_t *)NULL, &oset) != 0) e(5006);
if (oset != set) e(5007);
}

void test37e()
{
    struct sigaction act, oact;
    sigset_t set, oset;

    subtest = 5;
    clearsigtstate();

    act.sa_handler = catch4;
    sigemptyset(&act.sa_mask);
    sigaddset(&act.sa_mask, SIGTERM);
    sigaddset(&act.sa_mask, SIGHUP);
    act.sa_flags = 0;
    if (sigaction(SIGINT, &act, &oact) == -1) e(2);

    if (sigemptyset(&set) == -1) e(3);
    if (sigaddset(&set, SIGPIPE) == -1) e(4);
    if (sigprocmask(SIG_SETMASK, &set, &oset) == -1) e(5);
    if (kill(getpid(), SIGINT) == -1) e(6);
    if (sigprocmask(SIG_BLOCK, (sigset_t *)NULL, &oset) == -1) e(7);
    if (sigemptyset(&set) == -1) e(8);
    if (sigaddset(&set, SIGPIPE) == -1) e(9);
    if (set != oset) e(10);
}

/*-----*/

/* Test the basic functionality of sigsuspend(2). */

void catch5(signo)
int signo;
{
    x = 1;
}

void test37f()
{
    sigset_t set;
    int r;
    struct sigaction act;
    pid_t pid;

    subtest = 6;
    clearsigtstate();

    switch (pid = fork()) {
        case 0: /* child */
            errct = 0;
            sleep(1);
            if (kill(getppid(), SIGINT) == -1) e(1);
            exit(errct == 0 ? 0 : 1);
        case -1: e(2); break;
        default: /* parent */
            if (sigemptyset(&act.sa_mask) == -1) e(3);
            act.sa_flags = 0;
            act.sa_handler = catch5;
            if (sigaction(SIGINT, &act, (struct sigaction *) NULL) == -1) e(4);

            if (sigemptyset(&set) == -1) e(5);
            r = sigsuspend(&set);

            if (r != -1 || errno != EINTR || x != 1) e(6);
            wait_for(pid);
            break;
    }
}

/*-----*/

```

```

/* Test that sigsuspend() does block the signals specified in its
 * argument, and after sigsuspend returns, the previous signal
 * mask is restored.
 *
 * The child sends two signals to the parent SIGINT and then SIGPIPE,
 * separated by a long delay. The parent executes sigsuspend() with
 * SIGINT blocked. It is expected that the parent's SIGPIPE handler
 * will be invoked, then sigsuspend will return restoring the
 * original signal mask, and then the SIGPIPE handler will be
 * invoked.
 */

void sigint_handler(signo)
int signo;
{
    x = 1;
    z++;
}

void sigpipe_handler(signo)
int signo;
{
    x = 2;
    z++;
}

void test37g()
{
    sigset_t set;
    int r;
    struct sigaction act;
    pid_t pid;

    subtest = 7;
    clearsigsigstate();
    x = 0;
    z = 0;

    switch (pid = fork()) {
        case 0: /* child */
            errct = 0;
            sleep(1);
            if (kill(getppid(), SIGINT) == -1) e(1);
            sleep(1);
            if (kill(getppid(), SIGPIPE) == -1) e(2);
            exit(errct == 0 ? 0 : 1);
        case -1: e(3); break;
        default: /* parent */
            if (sigemptyset(&act.sa_mask) == -1) e(3);
            act.sa_flags = 0;
            act.sa_handler = sigint_handler;
            if (sigaction(SIGINT, &act, (struct sigaction *) NULL) == -1) e(4);

            act.sa_handler = sigpipe_handler;
            if (sigaction(SIGPIPE, &act, (struct sigaction *) NULL) == -1) e(5);

            if (sigemptyset(&set) == -1) e(6);
            if (sigaddset(&set, SIGINT) == -1) e(7);
            r = sigsuspend(&set);
            if (r != -1) e(8);
            if (errno != EINTR) e(9);
            if (z != 2) e(10);
            if (x != 1) e(11);
            wait_for(pid);
            break;
    }
}

/*-----*/

/* Test that sigsuspend() does block the signals specified in its
 * argument, and after sigsuspend returns, the previous signal
 * mask is restored.

```

```
*
* The child sends three signals to the parent: SIGHUP, then SIGPIPE,
* and then SIGTERM, separated by a long delay. The parent executes
* sigsuspend() with SIGHUP and SIGPIPE blocked. It is expected that
* the parent's SIGTERM handler will be invoked first, then sigsuspend()
* will return restoring the original signal mask, and then the other
* two handlers will be invoked.
*/

void sighup8(signo)
int signo;
{
    x = 1;
    z++;
}

void sigpip8(signo)
int signo;
{
    x = 1;
    z++;
}

void sigter8(signo)
int signo;
{
    x = 2;
    z++;
}

void test37h()
{
    sigset_t set;
    int r;
    struct sigaction act;
    pid_t pid;

    subtest = 8;
    clearsigtstate();
    x = 0;
    z = 0;

    switch (pid = fork()) {
        case 0: /* child */
            errct = 0;
            sleep(1);
            if (kill(getppid(), SIGHUP) == -1) e(1);
            sleep(1);
            if (kill(getppid(), SIGPIPE) == -1) e(2);
            sleep(1);
            if (kill(getppid(), SIGTERM) == -1) e(3);
            exit(errct == 0 ? 0 : 1);
        case -1: e(5); break;
        default: /* parent */
            if (sigemptyset(&act.sa_mask) == -1) e(6);
            act.sa_flags = 0;
            act.sa_handler = sighup8;
            if (sigaction(SIGHUP, &act, (struct sigaction *) NULL) == -1) e(7);

            act.sa_handler = sigpip8;
            if (sigaction(SIGPIPE, &act, (struct sigaction *) NULL) == -1) e(8);

            act.sa_handler = sigter8;
            if (sigaction(SIGTERM, &act, (struct sigaction *) NULL) == -1) e(9);

            if (sigemptyset(&set) == -1) e(10);
            if (sigaddset(&set, SIGHUP) == -1) e(11);
            if (sigaddset(&set, SIGPIPE) == -1) e(12);
            r = sigsuspend(&set);
            if (r != -1) e(13);
            if (errno != EINTR) e(14);
            if (z != 3) e(15);
            if (x != 1) e(16);
            wait_for(pid);
    }
}
```

```
        break;
    }
}

/*-----*/

/* Block SIGHUP and SIGTERM with sigprocmask(), send ourself SIGHUP
 * and SIGTERM, unblock these signals with sigprocmask, and verify
 * that these signals are delivered.
 */

void sighup9(signo)
int signo;
{
    y++;
}

void sigter9(signo)
int signo;
{
    z++;
}

void test37i()
{
    sigset_t set;
    struct sigaction act;

    subtest = 9;
    clearsigsigstate();
    y = 0;
    z = 0;

    if (sigemptyset(&act.sa_mask) == -1) e(1);
    act.sa_flags = 0;

    act.sa_handler = sighup9;
    if (sigaction(SIGHUP, &act, (struct sigaction *) NULL) == -1) e(2);

    act.sa_handler = sigter9;
    if (sigaction(SIGTERM, &act, (struct sigaction *) NULL) == -1) e(3);

    if (sigemptyset(&set) == -1) e(4);
    if (sigaddset(&set, SIGTERM) == -1) e(5);
    if (sigaddset(&set, SIGHUP) == -1) e(6);
    if (sigprocmask(SIG_SETMASK, &set, (sigset_t *)NULL) == -1) e(7);

    if (kill(getpid(), SIGHUP) == -1) e(8);
    if (kill(getpid(), SIGTERM) == -1) e(9);
    if (y != 0) e(10);
    if (z != 0) e(11);

    if (sigemptyset(&set) == -1) e(12);
    if (sigprocmask(SIG_SETMASK, &set, (sigset_t *)NULL) == -1) e(12);
    if (y != 1) e(13);
    if (z != 1) e(14);
}

/*-----*/

/* Block SIGINT and then send this signal to ourself.
 *
 * Install signal handlers for SIGALRM and SIGINT.
 *
 * Set an alarm for 6 seconds, then sleep for 7.
 *
 * The SIGALRM should interrupt the sleep, but the SIGINT
 * should remain pending.
 */

void sighupl0(signo)
int signo;
{
    y++;
}
```

```
}

void sigalrm_handler10(signo)
int signo;
{
    z++;
}

void test37j()
{
    sigset_t set, set2;
    struct sigaction act;

    subtest = 10;
    clearsigtstate();
    y = 0;
    z = 0;

    if (sigemptyset(&act.sa_mask) == -1) e(1);
    act.sa_flags = 0;

    act.sa_handler = sighup10;
    if (sigaction(SIGHUP, &act, (struct sigaction *) NULL) == -1) e(2);

    act.sa_handler = sigalrm_handler10;
    if (sigaction(SIGALRM, &act, (struct sigaction *) NULL) == -1) e(3);

    if (sigemptyset(&set) == -1) e(4);
    if (sigaddset(&set, SIGHUP) == -1) e(5);
    if (sigprocmask(SIG_SETMASK, &set, (sigset_t *)NULL) == -1) e(6);

    if (kill(getpid(), SIGHUP) == -1) e(7);
    if (sigpending(&set) == -1) e(8);
    if (sigemptyset(&set2) == -1) e(9);
    if (sigaddset(&set2, SIGHUP) == -1) e(10);
    if (set2 != set) e(11);
    alarm(6);
    sleep(7);
    if (sigpending(&set) == -1) e(12);
    if (set != set2) e(13);
    if (y != 0) e(14);
    if (z != 1) e(15);
}

/*-----*/

void test37k()
{
    subtest = 11;
}

void test37l()
{
    subtest = 12;
}

/*-----*/

/* Basic test for setjmp/longjmp. This includes testing that the
 * signal mask is properly restored.
 */

void test37m()
{
    jmp_buf jb;
    sigset_t ss;

    subtest = 13;
    clearsigtstate();

    ss = 0x32;
    if (sigprocmask(SIG_SETMASK, &ss, (sigset_t *)NULL) == -1) e(1);
    if (setjmp(jb)) {
        if (sigprocmask(SIG_BLOCK, (sigset_t *)NULL, &ss) == -1) e(2);
        if (ss != 0x32) e(388);
    }
}
```

```
        return;
    }
    ss = 0x3abc;
    if (sigprocmask(SIG_SETMASK, &ss, (sigset_t *)NULL) == -1) e(4);
    longjmp(jb, 1);
}

void longjerr()
{
    e(5);
}

/*-----*/

/* Test for setjmp/longjmp.
 *
 * Catch a signal. While in signal handler do setjmp/longjmp.
 */

void catch14(signo, code, scp)
int signo;
int code;
struct sigcontext *scp;
{
    jmp_buf jb;

    if (setjmp(jb)) {
        x++;
        sigreturn(scp);
        e(1);
    }
    y++;
    longjmp(jb, 1);
    e(2);
}

void test37n()
{
    struct sigaction act;
    typedef _PROTOTYPE( void (*sighandler_t), (int sig) );

    subtest = 14;
    clearsigsigstate();
    x = 0;
    y = 0;

    act.sa_flags = 0;
    act.sa_mask = 0;
    act.sa_handler = (sighandler_t) catch14; /* fudge */
    if (sigaction(SIGSEGV, &act, (struct sigaction *) NULL) == -1) e(3);
    if (kill(getpid(), SIGSEGV) == -1) e(4);

    if (x != 1) e(5);
    if (y != 1) e(6);
}

/*-----*/

/* Test for setjmp/longjmp.
 *
 * Catch a signal. Longjmp out of signal handler.
 */
jmp_buf glo_jb;

void catch15(signo)
int signo;
{
    z++;
    longjmp(glo_jb, 7);
    e(1);
}

void test37o()
```

```
{
    struct sigaction act;
    int k;

    subtest = 15;
    clearsigtstate();
    z = 0;

    act.sa_flags = 0;
    act.sa_mask = 0;
    act.sa_handler = catch15;
    if (sigaction(SIGALRM, &act, (struct sigaction *) NULL) == -1) e(2);

    if ((k = setjmp(glo_jb))) {
        if (z != 1) e(399);
        if (k != 7) e(4);
        return;
    }
    if (kill(getpid(), SIGALRM) == -1) e(5);
}

void clearsigtstate()
{
    int i;
    sigset_t sigset_var;

    /* Clear the signal state. */
    for (i = 1; i <= _NSIG; i++) signal(i, SIG_IGN);
    for (i = 1; i <= _NSIG; i++) signal(i, SIG_DFL);
    sigfillset(&sigset_var);
    sigprocmask(SIG_UNBLOCK, &sigset_var, (sigset_t *)NULL);
}

void quit()
{
    chdir("..");
    system("rm -rf DIR*");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
        printf("%d errors\n", errct);
        exit(4);
    }
}

void wait_for(pid)
pid_t pid;
{
    /* Expect exactly one child, and that it exits with 0. */

    int r;
    int status;

    errno = 0;
    while (1) {
        errno = 0;
        r = wait(&status);
        if (r == pid) {
            errno = 0;
            if (status != 0) e(90);
            return;
        }
        if (r < 0) {
            e(91);
            return;
        }
        e(92);
    }
}

void e(n)
```



```
int n;
{
    char msgbuf[80];

    sprintf(msgbuf, "Subtest %d, error %d errno=%d ", subtest, n, errno);
    perror(msgbuf);
    if (errct++ > MAX_ERROR) {
        fprintf(stderr, "Too many errors; test aborted\n");
        chdir("..");
        system("rm -rf DIR*");
        exit(1);
    }
}
```

```

/* Many of the tests require 1.6.n, n > 16, so we may as well assume that
 * POSIX signals are implemented.
 */
#define SIGACTION

/* test38: read(), write()          Author: Jan-Mark Wams (jms@cs.vu.nl) */

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <errno.h>
#include <time.h>
#include <signal.h>
#include <stdio.h>

#define MAX_ERROR      4
#define ITERATIONS     3
#define BUF_SIZE 1024

#define System(cmd)     if (system(cmd) != 0) printf("'%s' failed\n", cmd)
#define Chdir(dir)      if (chdir(dir) != 0) printf("Can't goto %s\n", dir)
#define Stat(a,b)       if (stat(a,b) != 0) printf("Can't stat %s\n", a)

int errct = 0;
int subtest = 1;
int superuser;
int signumber = 0;

_PROTOTYPE(void main, (int argc, char *argv[]));
_PROTOTYPE(void test38a, (void));
_PROTOTYPE(void test38b, (void));
_PROTOTYPE(void test38c, (void));
_PROTOTYPE(void setsignumber, (int _signumber));
_PROTOTYPE(void e, (int number));
_PROTOTYPE(void quit, (void));

void main(argc, argv)
int argc;
char *argv[];
{
    int i, m = 0xFFFF;

    sync();
    if (argc == 2) m = atoi(argv[1]);
    printf("Test 38 ");
    fflush(stdout);
    System("rm -rf DIR_38; mkdir DIR_38");
    Chdir("DIR_38");
    superuser = (geteuid() == 0);
    umask(0000);

    for (i = 0; i < ITERATIONS; i++) {
        if (m & 0001) test38a();
        if (m & 0002) test38b();
        if (m & 0004) test38c();
    }
    quit();
}

void test38a()
{
    /* Try normal operation. */
    int fd1;
    struct stat st1, st2;
    time_t timel;
    char buf[BUF_SIZE];
    int stat_loc;
    int i, j;
    int tube[2];

```

```
subtest = 1;
System("rm -rf ../DIR_38/*");

/* Let's open bar. */
if ((fd1 = open("bar", O_RDWR | O_CREAT, 0777)) != 3) e(1);
Stat("bar", &st1);

/* Writing nothing should not affect the file at all. */
if (write(fd1, "", 0) != 0) e(2);
Stat("bar", &st2);
if (st1.st_uid != st2.st_uid) e(3);
if (st1.st_gid != st2.st_gid) e(4); /* should be same */
if (st1.st_mode != st2.st_mode) e(5);
if (st1.st_size != st2.st_size) e(6);
if (st1.st_nlink != st2.st_nlink) e(7);
if (st1.st_mtime != st2.st_mtime) e(8);
if (st1.st_ctime != st2.st_ctime) e(9);
if (st1.st_atime != st2.st_atime) e(10);

/* A write should update some status fields. */
time(&timel);
while (timel >= time((time_t *)0))
;
if (write(fd1, "foo", 4) != 4) e(11);
Stat("bar", &st2);
if (st1.st_mode != st2.st_mode) e(12);
if (st1.st_size >= st2.st_size) e(13);
if ((off_t) 4 != st2.st_size) e(14);
if (st1.st_nlink != st2.st_nlink) e(15);
if (st1.st_mtime >= st2.st_mtime) e(16);
if (st1.st_ctime >= st2.st_ctime) e(17);
if (st1.st_atime != st2.st_atime) e(18);

/* Lseek should not change the file status. */
if (lseek(fd1, (off_t) - 2, SEEK_END) != 2) e(19);
Stat("bar", &st1);
if (st1.st_mode != st2.st_mode) e(20);
if (st1.st_size != st2.st_size) e(21);
if (st1.st_nlink != st2.st_nlink) e(22);
if (st1.st_mtime != st2.st_mtime) e(23);
if (st1.st_ctime != st2.st_ctime) e(24);
if (st1.st_atime != st2.st_atime) e(25);

/* Writing should start at the current (2) position. */
if (write(fd1, "foo", 4) != 4) e(26);
Stat("bar", &st2);
if (st1.st_mode != st2.st_mode) e(27);
if (st1.st_size >= st2.st_size) e(28);
if ((off_t) 6 != st2.st_size) e(29);
if (st1.st_nlink != st2.st_nlink) e(30);
if (st1.st_mtime > st2.st_mtime) e(31);
if (st1.st_ctime > st2.st_ctime) e(32);
if (st1.st_atime != st2.st_atime) e(33);

/* A read of zero bytes should not affect anything. */
if (read(fd1, buf, 0) != 0) e(34);
Stat("bar", &st1);
if (st1.st_uid != st2.st_uid) e(35);
if (st1.st_gid != st2.st_gid) e(36); /* should be same */
if (st1.st_mode != st2.st_mode) e(37);
if (st1.st_size != st2.st_size) e(38);
if (st1.st_nlink != st2.st_nlink) e(39);
if (st1.st_mtime != st2.st_mtime) e(40);
if (st1.st_ctime != st2.st_ctime) e(41);
if (st1.st_atime != st2.st_atime) e(42);

/* The file now should contain 'fofoo\0' Let's check that. */
if (lseek(fd1, (off_t) 0, SEEK_SET) != 0) e(43);
if (read(fd1, buf, BUF_SIZE) != 6) e(44);
if (strcmp(buf, "fofoo") != 0) e(45);

/* Only the Access Time should be updated. */
Stat("bar", &st2);
if (st1.st_mtime != st2.st_mtime) e(46);
```

```
if (st1.st_ctime != st2.st_ctime) e(47);
if (st1.st_atime >= st2.st_atime) e(48);

/* A read of zero bytes should do nothing even at the end of the file. */
time(&timel);
while (timel >= time((time_t *)0))
;
if (read(fdl, buf, 0) != 0) e(49);
Stat("bar", &st1);
if (st1.st_size != st2.st_size) e(50);
if (st1.st_mtime != st2.st_mtime) e(51);
if (st1.st_ctime != st2.st_ctime) e(52);
if (st1.st_atime != st2.st_atime) e(53);

/* Reading should be done from the current offset. */
if (read(fdl, buf, BUF_SIZE) != 0) e(54);
if (lseek(fdl, (off_t) 2, SEEK_SET) != 2) e(55);
if (read(fdl, buf, BUF_SIZE) != 4) e(56);
if (strcmp(buf, "foo") != 0) e(57);

/* Reading should effect the current file position. */
if (lseek(fdl, (off_t) 2, SEEK_SET) != 2) e(58);
if (read(fdl, buf, 1) != 1) e(59);
if (*buf != 'f') e(60);
if (lseek(fdl, (off_t) 0, SEEK_CUR) != 3) e(61);
if (read(fdl, buf, 1) != 1) e(62);
if (*buf != 'o') e(63);
if (lseek(fdl, (off_t) 0, SEEK_CUR) != 4) e(64);
if (read(fdl, buf, 1) != 1) e(65);
if (*buf != 'o') e(66);
if (lseek(fdl, (off_t) 0, SEEK_CUR) != 5) e(67);
if (read(fdl, buf, 1) != 1) e(68);
if (*buf != '\0') e(69);
if (lseek(fdl, (off_t) 0, SEEK_CUR) != 6) e(70);

/* Read's at EOF should return 0. */
if (read(fdl, buf, BUF_SIZE) != 0) e(71);
if (lseek(fdl, (off_t) 0, SEEK_CUR) != 6) e(72);
if (read(fdl, buf, BUF_SIZE) != 0) e(73);
if (lseek(fdl, (off_t) 0, SEEK_CUR) != 6) e(74);
if (read(fdl, buf, BUF_SIZE) != 0) e(75);
if (lseek(fdl, (off_t) 0, SEEK_CUR) != 6) e(76);
if (read(fdl, buf, BUF_SIZE) != 0) e(77);
if (lseek(fdl, (off_t) 0, SEEK_CUR) != 6) e(78);
if (read(fdl, buf, BUF_SIZE) != 0) e(79);
if (lseek(fdl, (off_t) 0, SEEK_CUR) != 6) e(80);

/* Writing should not always change the file size. */
if (lseek(fdl, (off_t) 2, SEEK_SET) != 2) e(81);
if (write(fdl, "ba", 2) != 2) e(82);
if (lseek(fdl, (off_t) 0, SEEK_CUR) != 4) e(83);
Stat("bar", &st1);
if (st1.st_size != 6) e(84);

/* Kill the \0 at the end. */
if (lseek(fdl, (off_t) 5, SEEK_SET) != 5) e(85);
if (write(fdl, "x", 1) != 1) e(86);

/* And close the bar. */
if (close(fdl) != 0) e(87);

/* Try some stuff with O_APPEND. Bar contains ``fobaiox'' */
if ((fdl = open("bar", O_RDWR | O_APPEND)) != 3) e(88);

/* No matter what the file position is. Writes should append. */
if (lseek(fdl, (off_t) 2, SEEK_SET) != 2) e(89);
if (write(fdl, "y", 1) != 1) e(90);
Stat("bar", &st1);
if (st1.st_size != (off_t) 7) e(91);
if (lseek(fdl, (off_t) 0, SEEK_CUR) != 7) e(92);
if (lseek(fdl, (off_t) 2, SEEK_SET) != 2) e(93);
if (write(fdl, "z", 2) != 2) e(94);

/* The file should contain ``fobaioxyz\0'' == 9 chars long. */
```

```

Stat("bar", &st1);
if (st1.st_size != (off_t) 9) e(95);
if (lseek(fdl, (off_t) 0, SEEK_CUR) != 9) e(96);

/* Reading on a O_APPEND flag should be from the current offset. */
if (lseek(fdl, (off_t) 0, SEEK_SET) != 0) e(97);
if (read(fdl, buf, BUF_SIZE) != 9) e(98);
if (strcmp(buf, "fobaoxyz") != 0) e(99);
if (lseek(fdl, (off_t) 0, SEEK_CUR) != 9) e(100);

if (close(fdl) != 0) e(101);

/* Let's test fifo writes. First blocking. */
if (mkfifo("fifo", 0777) != 0) e(102);

/* Read from fifo but no writer. */
System("rm -rf /tmp/sema.38a");
switch (fork()) {
    case -1: printf("Can't fork\n"); break;

    case 0:
        alarm(20);
        if ((fdl = open("fifo", O_RDONLY)) != 3) e(103);
        system(">/tmp/sema.38a");
        system("while test -f /tmp/sema.38a; do sleep 1; done");
errno = 0;
        if (read(fdl, buf, BUF_SIZE) != 0) e(104);
        if (read(fdl, buf, BUF_SIZE) != 0) e(105);
        if (read(fdl, buf, BUF_SIZE) != 0) e(106);
        if (close(fdl) != 0) e(107);
        exit(0);

    default:
        if ((fdl = open("fifo", O_WRONLY)) != 3) e(108);
        while (stat("/tmp/sema.38a", &st1) != 0) sleep(1);
        if (close(fdl) != 0) e(109);
        unlink("/tmp/sema.38a");
        if (wait(&stat_loc) == -1) e(110);
        if (stat_loc != 0) e(111); /* Alarm? */
}

/* Read from fifo should wait for writer. */
switch (fork()) {
    case -1: printf("Can't fork\n"); break;

    case 0:
        alarm(20);
        if ((fdl = open("fifo", O_RDONLY)) != 3) e(112);
        if (read(fdl, buf, BUF_SIZE) != 10) e(113);
        if (strcmp(buf, "Hi reader") != 0) e(114);
        if (close(fdl) != 0) e(115);
        exit(0);

    default:
        if ((fdl = open("fifo", O_WRONLY)) != 3) e(116);
        sleep(1);
        if (write(fdl, "Hi reader", 10) != 10) e(117);
        if (close(fdl) != 0) e(118);
        if (wait(&stat_loc) == -1) e(119);
        if (stat_loc != 0) e(120); /* Alarm? */
}

#if DEAD_CODE
/* Does this test test what it is supposed to test??? */

/* Read from fifo should wait for all writers to close. */
switch (fork()) {
    case -1: printf("Can't fork\n"); break;

    case 0:
        alarm(60);
        switch (fork()) {
            case -1: printf("Can't fork\n"); break;
            case 0:

```

```

        alarm(20);
        if ((fd1 = open("fifo", O_WRONLY)) != 3) e(121);
        printf("C2 did open\n");
        if (close(fd1) != 0) e(122);
        printf("C2 did close\n");
        exit(0);
    default:
        printf("C1 scheduled\n");
        if ((fd1 = open("fifo", O_WRONLY)) != 3) e(123);
        printf("C1 did open\n");
        sleep(2);
        if (close(fd1) != 0) e(124);
        printf("C1 did close\n");
        sleep(1);
        if (wait(&stat_loc) == -1) e(125);
        if (stat_loc != 0) e(126);          /* Alarm? */
    }
    exit(stat_loc);

default: {
    int wait_status;
    printf("Parent running\n");
    sleep(1);                               /* open in childs first */
    if ((fd1 = open("fifo", O_RDONLY)) != 3) e(127);
    if (read(fd1, buf, BUF_SIZE) != 0) e(128);
    if (close(fd1) != 0) e(129);
    printf("Parent closed\n");
    if ((wait_status=wait(&stat_loc)) == -1) e(130);

    printf("wait_status %d, stat_loc %d:", wait_status, stat_loc);
    if (WIFSIGNALED(stat_loc)) {
        printf(" killed, signal number %d\n", WTERMSIG(stat_loc));
    }
    else if (WIFEXITED(stat_loc)) {
        printf(" normal exit, status %d\n", WEXITSTATUS(stat_loc));
    }

    if (stat_loc != 0) e(131);          /* Alarm? */
}
#endif

/* PIPE_BUF has to have a nice value. */
if (PIPE_BUF < 5) e(132);
if (BUF_SIZE < 1000) e(133);

/* Writes of blocks smaller than PIPE_BUF should be atomic. */
System("rm -rf /tmp/sema.38b;> /tmp/sema.38b");
switch (fork()) {
    case -1: printf("Can't fork\n");    break;

    case 0:
        alarm(20);
        switch (fork()) {
            case -1: printf("Can't fork\n");    break;

            case 0:
                alarm(20);
                if ((fd1 = open("fifo", O_WRONLY)) != 3) e(134);
                for (i = 0; i < 100; i++) write(fd1, "1234 ", 5);
                system("while test -f /tmp/sema.38b; do sleep 1; done");
                if (close(fd1) != 0) e(135);
                exit(0);

            default:
                if ((fd1 = open("fifo", O_WRONLY)) != 3) e(136);
                for (i = 0; i < 100; i++) write(fd1, "1234 ", 5);
                while (stat("/tmp/sema.38b", &st1) == 0) sleep(1);
                if (close(fd1) != 0) e(137);
                if (wait(&stat_loc) == -1) e(138);
                if (stat_loc != 0) e(139);          /* Alarm? */
            }
        }
    exit(stat_loc);
}

```

```

default:
    if ((fd1 = open("fifo", O_RDONLY)) != 3) e(140);
    i = 0;
    memset(buf, '\0', BUF_SIZE);

    /* Read buffer full or till EOF. */
    do {
        j = read(fd1, buf + i, BUF_SIZE - i);
        if (j > 0) {
            if (j % 5 != 0) e(141);
            i += j;
        }
    } while (j > 0 && i < 1000);

    /* Signal the children to close write ends. This should not be */
    /* Necessary. But due to a bug in 1.16.6 this is necessary. */
    unlink("/tmp/sema.38b");
    if (j < 0) e(142);
    if (i != 1000) e(143);
    if (wait(&stat_loc) == -1) e(144);
    if (stat_loc != 0) e(145); /* Alarm? */

    /* Check 200 times 1234. */
    for (i = 0; i < 200; i++)
        if (strncmp(buf + (i * 5), "1234", 5) != 0) break;
    if (i != 200) e(146);
    if (buf[1000] != '\0') e(147);
    if (buf[1005] != '\0') e(148);
    if (buf[1010] != '\0') e(149);
    if (read(fd1, buf, BUF_SIZE) != 0) e(150);
    if (close(fd1) != 0) e(151);
}

/* Read from pipe should wait for writer. */
if (pipe(tube) != 0) e(152);
switch (fork()) {
    case -1: printf("Can't fork\n"); break;
    case 0:
        alarm(20);
        if (close(tube[1]) != 0) e(153);
        if (read(tube[0], buf, BUF_SIZE) != 10) e(154);
        if (strcmp(buf, "Hi reader") != 0) e(155);
        if (close(tube[0]) != 0) e(156);
        exit(0);
    default:
        if (close(tube[0]) != 0) e(157);
        sleep(1);
        if (write(tube[1], "Hi reader", 10) != 10) e(158);
        if (close(tube[1]) != 0) e(159);
        if (wait(&stat_loc) == -1) e(160);
        if (stat_loc != 0) e(161); /* Alarm? */
}

/* Read from pipe should wait for all writers to close. */
if (pipe(tube) != 0) e(162);
switch (fork()) {
    case -1: printf("Can't fork\n"); break;
    case 0:
        alarm(20);
        if (close(tube[0]) != 0) e(163);
        switch (fork()) {
            case -1: printf("Can't fork\n"); break;
            case 0:
                alarm(20);
                if (close(tube[1]) != 0) e(164);
                exit(0);
            default:
                sleep(1);
                if (close(tube[1]) != 0) e(165);
                if (wait(&stat_loc) == -1) e(166);
                if (stat_loc != 0) e(167); /* Alarm? */
        }
        exit(stat_loc);
    default:

```

```

    if (close(tube[1]) != 0) e(168);
    if (read(tube[0], buf, BUF_SIZE) != 0) e(169);
    if (close(tube[0]) != 0) e(170);
    if (wait(&stat_loc) == -1) e(171);
    if (stat_loc != 0) e(172);      /* Alarm? */
}

/* Writes of blocks smaller than PIPE_BUF should be atomic. */
System("rm -rf /tmp/sema.38c;>/tmp/sema.38c");
if (pipe(tube) != 0) e(173);
switch (fork()) {
    case -1: printf("Can't fork\n");      break;
    case 0:
        alarm(20);
        if (close(tube[0]) != 0) e(174);
        switch (fork()) {
            case -1: printf("Can't fork\n");      break;
            case 0:
                alarm(20);
                for (i = 0; i < 100; i++) write(tube[1], "1234 ", 5);
                system("while test -f /tmp/sema.38c; do sleep 1; done");
                if (close(tube[1]) != 0) e(175);
                exit(0);
            default:
                for (i = 0; i < 100; i++) write(tube[1], "1234 ", 5);
                while (stat("/tmp/sema.38c", &st1) == 0) sleep(1);
                if (close(tube[1]) != 0) e(176);
                if (wait(&stat_loc) == -1) e(177);
                if (stat_loc != 0) e(178);      /* Alarm? */
        }
        exit(stat_loc);
    default:
        i = 0;
        if (close(tube[1]) != 0) e(179);
        memset(buf, '\0', BUF_SIZE);
        do {
            j = read(tube[0], buf + i, BUF_SIZE - i);
            if (j > 0) {
                if (j % 5 != 0) e(180);
                i += j;
            } else
                break; /* EOF seen. */
        } while (i < 1000);
        unlink("/tmp/sema.38c");
        if (j < 0) e(181);
        if (i != 1000) e(182);
        if (close(tube[0]) != 0) e(183);
        if (wait(&stat_loc) == -1) e(184);
        if (stat_loc != 0) e(185);      /* Alarm? */

        /* Check 200 times 1234. */
        for (i = 0; i < 200; i++)
            if (strncmp(buf + (i * 5), "1234 ", 5) != 0) break;
        if (i != 200) e(186);
    }
}

void test38b()
{
    int i, fd, stat_loc;
    char buf[BUF_SIZE];
    char buf2[BUF_SIZE];
    struct stat st;

    subtest = 2;
    System("rm -rf ../DIR_38/*");

    /* Lets try sequential writes. */
    system("rm -rf /tmp/sema.38d");
    System(">testing");
    switch (fork()) {
        case -1: printf("Can't fork\n");      break;
        case 0:
            alarm(20);

```



```

    if ((fd = open("testing", O_WRONLY | O_APPEND)) != 3) e(1);
    if (write(fd, "one", 4) != 4) e(2);
    if (close(fd) != 0) e(3);
    system(">/tmp/sema.38d");
    system("while test -f /tmp/sema.38d; do sleep 1; done");
    if ((fd = open("testing", O_WRONLY | O_APPEND)) != 3) e(4);
    if (write(fd, "three", 6) != 6) e(5);
    if (close(fd) != 0) e(6);
    system(">/tmp/sema.38d");
    exit(0);
default:
    while (stat("/tmp/sema.38d", &st) != 0) sleep(1);
    if ((fd = open("testing", O_WRONLY | O_APPEND)) != 3) e(7);
    if (write(fd, "two", 4) != 4) e(8);
    if (close(fd) != 0) e(9);
    unlink("/tmp/sema.38d");
    while (stat("/tmp/sema.38d", &st) != 0) sleep(1);
    if ((fd = open("testing", O_WRONLY | O_APPEND)) != 3) e(10);
    if (write(fd, "four", 5) != 5) e(11);
    if (close(fd) != 0) e(12);
    if (wait(&stat_loc) == -1) e(13);
    if (stat_loc != 0) e(14);          /* The alarm went off? */
    unlink("/tmp/sema.38d");
}
if ((fd = open("testing", O_RDONLY)) != 3) e(15);
if (read(fd, buf, BUF_SIZE) != 19) e(16);
if (strcmp(buf, "one two three four") != 0) e(17);
if (close(fd) != 0) e(18);

/* Non written bytes in regular files should be zero. */
memset(buf2, '\0', BUF_SIZE);
if ((fd = open("bigfile", O_RDWR | O_CREAT, 0644)) != 3) e(19);
if (lseek(fd, (off_t) 102400, SEEK_SET) != (off_t) 102400L) e(20);
if (read(fd, buf, BUF_SIZE) != 0) e(21);
if (write(fd, ".", 1) != 1) e(22);
Stat("bigfile", &st);
if (st.st_size != (off_t) 102401) e(23);
if (lseek(fd, (off_t) 0, SEEK_SET) != 0) e(24);
for (i = 0; i < 102400 / BUF_SIZE; i++) {
    if (read(fd, buf, BUF_SIZE) != BUF_SIZE) e(25);
    if (memcmp(buf, buf2, BUF_SIZE) != 0) e(26);
}
if (close(fd) != 0) e(27);
}

void test38c()
{
    /* Test correct error behavior. */
    char buf[BUF_SIZE];
    int fd, tube[2], stat_loc;
    struct stat st;
    pid_t pid;
#ifdef SIGACTION
    struct sigaction act, oact;
#else
    if _ANSI
        void (*oldfunc) (int);
    else
        void (*oldfunc) ();
#endif
    subtest = 3;
    System("rm -rf ../DIR_38/*");

    /* To test if writing processes on closed pipes are signumbered. */
#ifdef SIGACTION
    act.sa_handler = setsignumber;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    if (sigaction(SIGPIPE, &act, &oact) != 0) e(1);
#else
    oldfunc = signal(SIGPIPE, setsignumber);
#endif
}

```

```

/* Non valid file descriptors should be an error. */
for (fd = -111; fd < 0; fd++) {
    errno = 0;
    if (read(fd, buf, BUF_SIZE) != -1) e(2);
    if (errno != EBADF) e(3);
}
for (fd = 3; fd < 111; fd++) {
    errno = 0;
    if (read(fd, buf, BUF_SIZE) != -1) e(4);
    if (errno != EBADF) e(5);
}
for (fd = -111; fd < 0; fd++) {
    errno = 0;
    if (write(fd, buf, BUF_SIZE) != -1) e(6);
    if (errno != EBADF) e(7);
}
for (fd = 3; fd < 111; fd++) {
    errno = 0;
    if (write(fd, buf, BUF_SIZE) != -1) e(8);
    if (errno != EBADF) e(9);
}

/* Writing a pipe with no readers should trigger SIGPIPE. */
if (pipe(tube) != 0) e(10);
close(tube[0]);
switch (fork()) {
    case -1: printf("Can't fork\n");      break;
    case 0:
        alarm(20);
        signumber = 0;
        if (write(tube[1], buf, BUF_SIZE) != -1) e(11);
        if (errno != EPIPE) e(12);
        if (signumber != SIGPIPE) e(13);
        if (close(tube[1]) != 0) e(14);
        exit(0);
    default:
        close(tube[1]);
        if (wait(&stat_loc) == -1) e(15);
        if (stat_loc != 0) e(16);      /* Alarm? */
}

/* Writing a fifo with no readers should trigger SIGPIPE. */
System(">/tmp/sema.38e");
if (mkfifo("fifo", 0666) != 0) e(17);
switch (fork()) {
    case -1: printf("Can't fork\n");      break;
    case 0:
        alarm(20);
        if ((fd = open("fifo", O_WRONLY)) != 3) e(18);
        system("while test -f /tmp/sema.38e; do sleep 1; done");
        signumber = 0;
        if (write(fd, buf, BUF_SIZE) != -1) e(19);
        if (errno != EPIPE) e(20);
        if (signumber != SIGPIPE) e(21);
        if (close(fd) != 0) e(22);
        exit(0);
    default:
        if ((fd = open("fifo", O_RDONLY)) != 3) e(23);
        if (close(fd) != 0) e(24);
        unlink("/tmp/sema.38e");
        if (wait(&stat_loc) == -1) e(25);
        if (stat_loc != 0) e(26);      /* Alarm? */
}

#ifdef SIGACTION
/* Restore normal (re)action to SIGPIPE. */
if (sigaction(SIGPIPE, &oact, NULL) != 0) e(27);
#else
signal(SIGPIPE, oldfunc);
#endif

/* Read from fifo should return -1 and set errno to EAGAIN. */
System("rm -rf /tmp/sema.38[fgh]");
switch (fork()) {

```

```

    case -1: printf("Can't fork\n"); break;
    case 0:
        alarm(20);
        system("while test ! -f /tmp/sema.38f; do sleep 1; done");
        System("rm -rf /tmp/sema.38f");
        if ((fd = open("fifo", O_WRONLY | O_NONBLOCK)) != 3) e(28);
        close(creat("/tmp/sema.38g", 0666));
        system("while test ! -f /tmp/sema.38h; do sleep 1; done");
        if (close(fd) != 0) e(38);
        System("rm -rf /tmp/sema.38h");
        exit(0);
    default:
        if ((fd = open("fifo", O_RDONLY | O_NONBLOCK)) != 3) e(30);
        close(creat("/tmp/sema.38f", 0666));
        system("while test ! -f /tmp/sema.38g; do sleep 1; done");
        System("rm -rf /tmp/sema.38g");
        if (read(fd, buf, BUF_SIZE) != -1) e(31);
        if (errno != EAGAIN) e(32);
        if (read(fd, buf, BUF_SIZE) != -1) e(33);
        if (errno != EAGAIN) e(34);
        if (read(fd, buf, BUF_SIZE) != -1) e(35);
        if (errno != EAGAIN) e(36);
        close(creat("/tmp/sema.38h", 0666));
        while (stat("/tmp/sema.38h", &st) == 0) sleep(1);
        if (read(fd, buf, BUF_SIZE) != 0) e(37);
        if (close(fd) != 0) e(38);
        if (wait(&stat_loc) == -1) e(39);
        if (stat_loc != 0) e(40); /* Alarm? */
}
System("rm -rf fifo");

/* If a read is interrupted by a SIGNAL. */
if (pipe(tube) != 0) e(41);
switch (pid = fork()) {
    case -1: printf("Can't fork\n"); break;
    case 0:
        alarm(20);
#ifdef SIGACTION
        act.sa_handler = setsignumber;
        sigemptyset(&act.sa_mask);
        act.sa_flags = 0;
        if (sigaction(SIGUSR1, &act, &oact) != 0) e(42);
#else
        oldfunc = signal(SIGUSR1, setsignumber);
#endif
        if (read(tube[0], buf, BUF_SIZE) != -1) e(43);
        if (errno != EINTR) e(44);
        if (signumber != SIGUSR1) e(45);
#ifdef SIGACTION
        /* Restore normal (re)action to SIGPIPE. */
        if (sigaction(SIGUSR1, &oact, NULL) != 0) e(46);
#else
        signal(SIGUSR1, oldfunc);
#endif
        close(tube[0]);
        close(tube[1]);
        exit(0);
    default:
        /* The sleep 1 should give the child time to start the read. */
        sleep(1);
        close(tube[0]);
        kill(pid, SIGUSR1);
        wait(&stat_loc);
        if (stat_loc != 0) e(47); /* Alarm? */
        close(tube[1]);
}
}

void setsignumber(signum)
int signum;
{
    signumber = signum;
}

```

```
void e(n)
int n;
{
    int err_num = errno;          /* Save in case printf clobbers it. */

    printf("Subtest %d, error %d errno=%d: ", subtest, n, errno);
    errno = err_num;
    perror("");
    if (errct++ > MAX_ERROR) {
        printf("Too many errors; test aborted\n");
        chdir("..");
        system("rm -rf DIR*");
        exit(1);
    }
    errno = 0;
}

void quit()
{
    Chdir("..");
    System("rm -rf DIR_38");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
        printf("%d errors\n", errct);
        exit(1);
    }
}
```

```

/* POSIX test program (39).                                     Author: Andy Tanenbaum */

/* The following POSIX calls are tested:
 *
 *      opendir()
 *      readdir()
 *      rewinddir()
 *      closedir()
 *      chdir()
 *      getcwd()
 */

#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <fcntl.h>
#include <errno.h>
#include <limits.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <utime.h>
#include <stdio.h>

#define DIR_NULL (DIR*) NULL
#define ITERATIONS 3      /* LINK_MAX is high, so time consuming */
#define MAX_FD 100        /* must be large enough to cause error */
#define BUF_SIZE PATH_MAX+20
#define ERR_CODE -1       /* error return */
#define RD_BUF 200
#define MAX_ERROR 4

char str[] = {"The time has come the walrus said to talk of many things.\n"};
char str2[] = {"Of ships and shoes and sealing wax, of cabbages and kings.\n"};
char str3[] = {"Of why the sea is boiling hot and whether pigs have wings\n"};

int subtest, errct;

_PROTOTYPE(int main, (int argc, char *argv []));
_PROTOTYPE(void test39a, (void));
_PROTOTYPE(void checkdir, (DIR *dirp, int t));
_PROTOTYPE(void test39b, (void));
_PROTOTYPE(void test39c, (void));
_PROTOTYPE(void test39d, (void));
_PROTOTYPE(void test39e, (void));
_PROTOTYPE(void test39f, (void));
_PROTOTYPE(void test39g, (void));
_PROTOTYPE(void test39h, (void));
_PROTOTYPE(void test39i, (void));
_PROTOTYPE(void test39j, (void));
_PROTOTYPE(void e, (int n));
_PROTOTYPE(void quit, (void));

int main(argc, argv)
int argc;
char *argv[];
{
    int i, m = 0xFFFF;

    sync();
    if (geteuid() == 0 || getuid() == 0) {
        printf("Test 39 cannot run as root; test aborted\n");
        exit(1);
    }

    if (argc == 2) m = atoi(argv[1]);
    printf("Test 39 ");
    fflush(stdout);

    system("rm -rf DIR_39; mkdir DIR_39");
    chdir("DIR_39");

```

```

for (i = 0; i < ITERATIONS; i++) {
    if (m & 00001) test39a();      /* test for correct operation */
    if (m & 00002) test39b();      /* test general error handling */
    if (m & 00004) test39c();      /* test for EMFILE error */
    if (m & 00010) test39d();      /* test chdir() and getcwd() */
    if (m & 00020) test39e();      /* test open() */
    if (m & 00040) test39f();      /* test umask(), stat(), fstat() */
    if (m & 00100) test39g();      /* test link() and unlink() */
    if (m & 00200) test39h();      /* test access() */
    if (m & 00400) test39i();      /* test chmod() and chown() */
    if (m & 01000) test39j();      /* test utime() */
}
quit();
return(-1);                      /* impossible */
}

void test39a()
{
/* Subtest 1. Correct operation */

    int f1, f2, f3, f4, f5;
    DIR *dirp;

    /* Remove any residue of previous tests. */
    subtest = 1;

    system("rm -rf foo");

    /* Create a directory foo with 5 files in it. */
    mkdir("foo", 0777);
    if ((f1 = creat("foo/f1", 0666)) < 0) e(1);
    if ((f2 = creat("foo/f2", 0666)) < 0) e(2);
    if ((f3 = creat("foo/f3", 0666)) < 0) e(3);
    if ((f4 = creat("foo/f4", 0666)) < 0) e(4);
    if ((f5 = creat("foo/f5", 0666)) < 0) e(5);

    /* Now remove 2 files to create holes in the directory. */
    if (unlink("foo/f2") < 0) e(6);
    if (unlink("foo/f4") < 0) e(7);

    /* Close the files. */
    close(f1);
    close(f2);
    close(f3);
    close(f4);
    close(f5);

    /* Open the directory. */
    dirp = opendir("./foo");
    if (dirp == DIR_NULL) e(6);

    /* Read the 5 files from it. */
    checkdir(dirp, 2);

    /* Rewind dir and test again. */
    rewinddir(dirp);
    checkdir(dirp, 3);

    /* We're done. Close the directory stream. */
    if (closedir(dirp) < 0) e(7);

    /* Remove dir for next time. */
    system("rm -rf foo");
}

void checkdir(DIR *dirp, int t)
/* pointer to directory stream */
/* subtest number to use */
{
    int i, f1, f2, f3, f4, f5, dot, dotdot, subt;
    struct dirent *d;
    char *s;

```

```
/* Save subtest number */
subt = subtest;
subtest = t;

/* Clear the counters. */
f1 = 0;
f2 = 0;
f3 = 0;
f4 = 0;
f5 = 0;
dot = 0;
dotdot = 0;

/* Read the directory. It should contain 5 entries, ".", ".." and 3
 * files. */
for (i = 0; i < 5; i++) {
    d = readdir(dirp);
    if (d == (struct dirent *) NULL) {
        e(1);
        subtest = subt; /* restore subtest number */
        return;
    }
    s = d->d_name;
    if (strcmp(s, ".") == 0) dot++;
    if (strcmp(s, "..") == 0) dotdot++;
    if (strcmp(s, "f1") == 0) f1++;
    if (strcmp(s, "f2") == 0) f2++;
    if (strcmp(s, "f3") == 0) f3++;
    if (strcmp(s, "f4") == 0) f4++;
    if (strcmp(s, "f5") == 0) f5++;
}

/* Check results. */
d = readdir(dirp);
if (d != (struct dirent *) NULL) e(2);
if (f1 != 1 || f3 != 1 || f5 != 1) e(3);
if (f2 != 0 || f4 != 0) e(4);
if (dot != 1 || dotdot != 1) e(5);
subtest = subt;
return;
}

void test39b()
{
    /* Subtest 4. Test error handling. */

    int fd;
    DIR *dirp;

    subtest = 4;

    if (opendir("foo/xyz/---") != DIR_NULL) e(1);
    if (errno != ENOENT) e(2);
    if (mkdir("foo", 0777) < 0) e(3);
    if (chmod("foo", 0) < 0) e(4);
    if (opendir("foo/xyz/--") != DIR_NULL) e(5);
    if (errno != EACCES) e(6);
    if (chmod("foo", 0777) != 0) e(7);
    if (rmdir("foo") != 0) e(8);
    if ((fd = creat("abc", 0666)) < 0) e(9);
    if (close(fd) < 0) e(10);
    if (opendir("abc/xyz") != DIR_NULL) e(11);
    if (errno != ENOTDIR) e(12);
    if ((dirp = opendir(".")) == DIR_NULL) e(13);
    if (closedir(dirp) != 0) e(14);
    if (unlink("abc") != 0) e(15);
}

void test39c()
{
    /* Subtest 5. See what happens if we open too many directory streams. */

    int i, j;
```

```
DIR *dirp[MAX_FD];

subtest = 5;

for (i = 0; i < MAX_FD; i++) {
    dirp[i] = opendir(".");
    if (dirp[i] == (DIR *) NULL) {
        /* We have hit the limit. */
        if (errno != EMFILE && errno != ENOMEM) e(1);
        for (j = 0; j < i; j++) {
            if (closedir(dirp[j]) != 0) e(2);          /* close */
        }
        return;
    }
}

/* Control should never come here. This is an error. */
e(3);
for (i = 0; i < MAX_FD; i++) closedir(dirp[i]);        /* don't check */
}

void test39d()
{
    /* Test chdir and getcwd(). */

    int fd;
    char *s;
    char base[BUF_SIZE], buf2[BUF_SIZE], tmp[BUF_SIZE];

    subtest = 6;

    if (getcwd(base, BUF_SIZE) == (char *) NULL) e(1); /* get test dir's path */
    if (system("rm -rf Dir") != 0) e(2);             /* remove residue of previous test */
    if (mkdir("Dir", 0777) < 0) e(3);                 /* create directory called "Dir" */

    /* Change to Dir and verify that it worked. */
    if (chdir("Dir") < 0) e(4);                       /* go to Dir */
    s = getcwd(buf2, BUF_SIZE);                       /* get full path of Dir */
    if (s == (char *) NULL) e(5);                     /* check for error return */
    if (s != buf2) e(6);                             /* if successful, first arg is returned */
    strcpy(tmp, base);                                /* concatenate base name and "/Dir" */
    strcat(tmp, "/");
    strcat(tmp, "Dir");
    if (strcmp(tmp, s) != 0) e(7);

    /* Change to ".." and verify that it worked. */
    if (chdir("..") < 0) e(8);
    if (getcwd(buf2, BUF_SIZE) != buf2) e(9);
    if (strcmp(buf2, base) != 0) e(10);

    /* Now make calls that do nothing, but do it in a strange way. */
    if (chdir("Dir/..") < 0) e(11);
    if (getcwd(buf2, BUF_SIZE) != buf2) e(12);
    if (strcmp(buf2, base) != 0) e(13);

    if (chdir("Dir/../Dir/..") < 0) e(14);
    if (getcwd(buf2, BUF_SIZE) != buf2) e(15);
    if (strcmp(buf2, base) != 0) e(16);

    if (chdir("Dir/../Dir/../Dir/../Dir/../Dir/../Dir/..") < 0) e(17);
    if (getcwd(buf2, BUF_SIZE) != buf2) e(18);
    if (strcmp(buf2, base) != 0) e(19);

    /* Make Dir unreadable and unsearchable. Check error message. */
    if (chmod("Dir", 0) < 0) e(20);
    if (chdir("Dir") >= 0) e(21);
    if (errno != EACCES) e(22);

    /* Check error message for bad path. */
    if (chmod("Dir", 0777) < 0) e(23);
    if (chdir("Dir/x/y") != ERR_CODE) e(24);
    if (errno != ENOENT) e(25);

    if ((fd=creat("Dir/x", 0777)) < 0) e(26);
}
```



```
if (close(fd) != 0) e(27);
if (chdir("Dir/x/y") != ERR_CODE) e(28);
if (errno != ENOTDIR) e(29);

/* Check empty string. */
if (chdir("") != ERR_CODE) e(30);
if (errno != ENOENT) e(31);

/* Remove the directory. */
if (unlink("Dir/x") != 0) e(32);
if (system("rmdir Dir") != 0) e(33);
}

void test39e()
{
/* Test open. */

int fd, bytes, bytes2;
char buf[RD_BUF];

subtest = 7;

unlink("T39");          /* get rid of it in case it exists */

/* Create a test file. */
bytes = strlen(str);
bytes2 = strlen(str2);
if ((fd = creat("T39", 0777)) < 0) e(1);
if (write(fd, str, bytes) != bytes) e(2);      /* T39 now has 'bytes' bytes */
if (close(fd) != 0) e(3);

/* Test opening a file with O_RDONLY. */
if ((fd = open("T39", O_RDONLY)) < 0) e(4);
buf[0] = '\0';
if (read(fd, buf, RD_BUF) != bytes) e(5);
if (strncmp(buf, str, bytes) != 0) e(6);
if (close(fd) < 0) e(7);

/* Test the same thing, only with O_RDWR now. */
if ((fd = open("T39", O_RDWR)) < 0) e(8);
buf[0] = '\0';
if (read(fd, buf, RD_BUF) != bytes) e(9);
if (strncmp(buf, str, bytes) != 0) e(10);
if (close(fd) < 0) e(11);

/* Try opening and reading with O_WRONLY. It should fail. */
if ((fd = open("T39", O_WRONLY)) < 0) e(12);
buf[0] = '\0';
if (read(fd, buf, RD_BUF) >= 0) e(13);
if (close(fd) != 0) e(14);

/* Test O_APPEND. */
if ((fd = open("T39", O_RDWR | O_APPEND)) < 0) e(15);
if (lseek(fd, 0L, SEEK_SET) < 0) e(16);      /* go to start of file */
if (write(fd, str2, bytes2) != bytes2) e(17); /* write at start of file */
if (lseek(fd, 0L, SEEK_SET) < 0) e(18);      /* go back to start again */
if (read(fd, buf, RD_BUF) != bytes + bytes2) e(19); /* read whole file */
if (strncmp(buf, str, bytes) != 0) e(20);
if (close(fd) != 0) e(21);

/* Get rid of the file. */
if (unlink("T39") < 0) e(22);
}

void test39f()
{
/* Test stat, fstat, umask. */
int i, fd;
mode_t ml;
struct stat stbuf1, stbuf2;
time_t t, t1;

subtest = 8;
```

```
ml = umask(~0777);
if (system("rm -rf foo xxx") != 0) e(1);
if ((fd = creat("foo", 0777)) < 0) e(2);
if (stat("foo", &stbuf1) < 0) e(3);
if (fstat(fd, &stbuf2) < 0) e(4);
if (stbuf1.st_mode != stbuf2.st_mode) e(5);
if (stbuf1.st_ino != stbuf2.st_ino) e(6);
if (stbuf1.st_dev != stbuf2.st_dev) e(7);
if (stbuf1.st_nlink != stbuf2.st_nlink) e(8);
if (stbuf1.st_uid != stbuf2.st_uid) e(9);
if (stbuf1.st_gid != stbuf2.st_gid) e(10);
if (stbuf1.st_size != stbuf2.st_size) e(11);
if (stbuf1.st_atime != stbuf2.st_atime) e(12);
if (stbuf1.st_mtime != stbuf2.st_mtime) e(13);
if (stbuf1.st_ctime != stbuf2.st_ctime) e(14);

if (!S_ISREG(stbuf1.st_mode)) e(15);
if (S_ISDIR(stbuf1.st_mode)) e(16);
if (S_ISCHR(stbuf1.st_mode)) e(17);
if (S_ISBLK(stbuf1.st_mode)) e(18);
if (S_ISFIFO(stbuf1.st_mode)) e(19);

if ((stbuf1.st_mode & (S_IRWXU | S_IRWXG | S_IRWXO)) != 0777) e(20);
if (stbuf1.st_nlink != 1) e(21);
if (stbuf1.st_uid != getuid()) e(22);
if (stbuf1.st_gid != getgid()) e(23);
if (stbuf1.st_size != 0L) e(24);

/* First unlink, then close -- harder test */
if (unlink("foo") < 0) e(25);
if (close(fd) < 0) e(26);

/* Now try umask a bit more. */
fd = 0;
if ((i = umask(~0704)) != 0) e(27);
if ((fd = creat("foo", 0777)) < 0) e(28);
if (stat("foo", &stbuf1) < 0) e(29);
if (fstat(fd, &stbuf2) < 0) e(30);
if (stbuf1.st_mode != stbuf2.st_mode) e(31);
if ((stbuf1.st_mode & (S_IRWXU | S_IRWXG | S_IRWXO)) != 0704) e(32);

/* First unlink, then close -- harder test */
if (unlink("foo") < 0) e(33);
if (close(fd) < 0) e(34);
if (umask(ml) != 073) e(35);

/* Test some errors. */
if (system("mkdir Dir; date >Dir/x; chmod 666 Dir") != 0) e(36);
if (stat("Dir/x", &stbuf1) >= 0) e(37);
if (errno != EACCES) e(38);
if (stat(".....", &stbuf1) >= 0) e(39);
if (errno != ENOENT) e(40);
if (stat("", &stbuf1) >= 0) e(41);
if (errno != ENOENT) e(42);
if (stat("xxx/yy/zzz", &stbuf1) >= 0) e(43);
if (errno != ENOENT) e(44);
if (fstat(10000, &stbuf1) >= 0) e(45);
if (errno != EBADF) e(46);
if (chmod("Dir", 0777) != 0) e(47);
if (system("rm -rf foo Dir") != 0) e(48);

/* See if time looks reasonable. */
errno = 0;
t = time(&t1); /* current time */
if (t < 6500000000L) e(49); /* 6500000000 is Sept. 1990 */
unlink("T39f");
fd = creat("T39f", 0777);
if (fd < 0) e(50);
if (close(fd) < 0) e(51);
if (stat("T39f", &stbuf1) < 0) e(52);
if (stbuf1.st_mtime < t) e(53);
if (unlink("T39f") < 0) e(54);
}
```

```
void test39g()
{
    /* Test link and unlink. */
    int i, fd;
    struct stat stbuf;
    char name[20];

    subtest = 9;

    if (system("rm -rf L? L?? Dir; mkdir Dir") != 0) e(1);
    if ( (fd = creat("L1", 0666)) < 0) e(2);
    if (fstat(fd, &stbuf) != 0) e(3);
    if (stbuf.st_nlink != 1) e(4);
    if (link("L1", "L2") != 0) e(5);
    if (fstat(fd, &stbuf) != 0) e(6);
    if (stbuf.st_nlink != 2) e(7);
    if (unlink("L2") != 0) e(8);
    if (link("L1", "L2") != 0) e(9);
    if (unlink("L1") != 0) e(10);
    if (close(fd) != 0) e(11);

    /* L2 exists at this point. */
    if ( (fd = creat("L1", 0666)) < 0) e(12);
    if (stat("L1", &stbuf) != 0) e(13);
    if (stbuf.st_nlink != 1) e(14);
    if (link("L1", "Dir/L2") != 0) e(15);
    if (stat("L1", &stbuf) != 0) e(16);
    if (stbuf.st_nlink != 2) e(17);
    if (stat("Dir/L2", &stbuf) != 0) e(18);
    if (stbuf.st_nlink != 2) e(19);

    /* L1, L2, and Dir/L2 exist at this point. */
    if (unlink("Dir/L2") != 0) e(20);
    if (link("L1", "Dir/L2") != 0) e(21);
    if (unlink("L1") != 0) e(22);
    if (close(fd) != 0) e(23);
    if (chdir("Dir") != 0) e(24);
    if (unlink("L2") != 0) e(25);
    if (chdir("..") != 0) e(26);

    /* L2 exists at this point. Test linking to unsearchable dir. */
    if (link("L2", "Dir/L2") != 0) e(27);
    if (chmod("Dir", 0666) != 0) e(27);
    if (link("L2", "Dir/L2") != -1) e(28);
    if (errno != EACCES) e(29);
    errno = 0;
    if (link("Dir/L2", "L3") != -1) e(30);
    if (errno != EACCES) e(31);
    if (chmod("Dir", 0777) != 0) e(32);
    if (unlink("Dir/L2") != 0) e(33);
    if (unlink("L3") == 0) e(34);

    /* L2 exists at this point. Test linking to unwriteable dir. */
    if (chmod("Dir", 0555) != 0) e(35);
    if (link("L2", "Dir/L2") != -1) e(36);
    if (errno != EACCES) e(37);
    if (chmod("Dir", 0777) != 0) e(38);

    /* L2 exists at this point. Test linking mode 0 file. */
    if (chmod("L2", 0) != 0) e(39);
    if (link("L2", "L3") != 0) e(40);
    if (stat("L3", &stbuf) != 0) e(41);
    if (stbuf.st_nlink != 2) e(42);
    if (unlink("L2") != 0) e(43);

    /* L3 exists at this point. Test linking to an existing file. */
    if ( (fd = creat("L1", 0666)) < 0) e(44);
    if (link("L1", "L3") != -1) e(45);
    if (errno != EEXIST) e(46);
    errno = 0;
    if (link("L1", "L1") != -1) e(47);
    if (errno != EEXIST) e(48);
    if (unlink("L3") != 0) e(49);
```

```
/* L1 exists at this point. Test creating too many links. */
for (i = 2; i <= LINK_MAX; i++) {
    sprintf(name, "Lx%d", i);
    if (link("L1", name) != 0) e(50);
}
if (stat("L1", &stbuf) != 0) e(51);
if (stbuf.st_nlink != LINK_MAX) e(52);
if (link("L1", "L2") != -1) e(53);
if (errno != EMLINK) e(54);
for (i = 2; i <= LINK_MAX; i++) {
    sprintf(name, "Lx%d", i);
    if (unlink(name) != 0) e(55);
}

if (stat("L1", &stbuf) != 0) e(56);
if (stbuf.st_nlink != 1) e(57);

/* L1 exists. Test ENOENT. */
errno = 0;
if (link("xx/L1", "L2") != -1) e(58);
if (errno != ENOENT) e(59);
errno = 0;
if (link("L1", "xx/L2") != -1) e(60);
if (errno != ENOENT) e(61);
errno = 0;
if (link("L4", "L5") != -1) e(62);
if (errno != ENOENT) e(63);
errno = 0;
if (link("", "L5") != -1) e(64);
if (errno != ENOENT) e(65);
errno = 0;
if (link("L1", "") != -1) e(66);
if (errno != ENOENT) e(67);

/* L1 exists. Test ENOTDIR. */
errno = 0;
if (link("/dev/tty/x", "L2") != -1) e(68);
if (errno != ENOTDIR) e(69);

/* L1 exists. Test EPERM. */
if (link(".", "L2") != -1) e(70);
if (errno != EPERM) e(71);

/* L1 exists. Test unlink. */
if (link("L1", "Dir/L1") != 0) e(72);
if (chmod("Dir", 0666) != 0) e(73);
if (unlink("Dir/L1") != -1) e(74);
if (errno != EACCES) e(75);
errno = 0;
if (chmod("Dir", 0555) != 0) e(76);
if (unlink("Dir/L1") != -1) e(77);
if (errno != EACCES) e(78);

if (unlink("L7") != -1) e(79);
if (errno != ENOENT) e(80);
errno = 0;
if (unlink("") != -1) e(81);
if (errno != ENOENT) e(82);

if (unlink("Dir/L1/L2") != -1) e(83);
if (errno != ENOTDIR) e(84);

if (chmod("Dir", 0777) != 0) e(85);
if (unlink("Dir/L1") != 0) e(86);
if (unlink("Dir") != -1) e(87);
if (errno != EPERM) e(88);
if (unlink("L1") != 0) e(89);
if (system("rm -rf Dir") != 0) e(90);
if (close(fd) != 0) e(91);
}

void test39h()
{
    /* Test access. */
}
```

```

int fd;

subtest = 10;
system("rm -rf A1");
if ( (fd = creat("A1", 0777)) < 0) e(1);
if (close(fd) != 0) e(2);
if (access("A1", R_OK) != 0) e(3);
if (access("A1", W_OK) != 0) e(4);
if (access("A1", X_OK) != 0) e(5);
if (access("A1", (R_OK|W_OK|X_OK)) != 0) e(6);

if (chmod("A1", 0400) != 0) e(7);
if (access("A1", R_OK) != 0) e(8);
if (access("A1", W_OK) != -1) e(9);
if (access("A1", X_OK) != -1) e(10);
if (access("A1", (R_OK|W_OK|X_OK)) != -1) e(11);

if (chmod("A1", 0077) != 0) e(12);
if (access("A1", R_OK) != -1) e(13);
if (access("A1", W_OK) != -1) e(14);
if (access("A1", X_OK) != -1) e(15);
if (access("A1", (R_OK|W_OK|X_OK)) != -1) e(16);
if (errno != EACCES) e(17);

if (access("", R_OK) != -1) e(18);
if (errno != ENOENT) e(19);
if (access("./A1/x", R_OK) != -1) e(20);
if (errno != ENOTDIR) e(21);

if (unlink("A1") != 0) e(22);
}

void test39i()
{
/* Test chmod. */

int fd, i;
struct stat stbuf;

subtest = 11;
system("rm -rf A1");
if ( (fd = creat("A1", 0777)) < 0) e(1);

for (i = 0; i < 511; i++) {
    if (chmod("A1", i) != 0) e(100+i);
    if (fstat(fd, &stbuf) != 0) e(200+i);
    if ( (stbuf.st_mode & (S_IRWXU|S_IRWXG|S_IRWXO)) != i) e(300+i);
}
if (close(fd) != 0) e(2);

if (chmod("A1/x", 0777) != -1) e(3);
if (errno != ENOTDIR) e(4);
if (chmod("Axxx", 0777) != -1) e(5);
if (errno != ENOENT) e(6);
errno = 0;
if (chmod("", 0777) != -1) e(7);
if (errno != ENOENT) e(8);

/* Now perform limited chown tests. These should work even as non su */
i = getuid();
/* DEBUG -- Not yet implemented
if (chown("A1", i, 0) != 0) e(9);
if (chown("A1", i, 1) != 0) e(10);
if (chown("A1", i, 2) != 0) e(11);
if (chown("A1", i, 3) != 0) e(12);
if (chown("A1", i, 4) != 0) e(13);
if (chown("A1", i, 0) != 0) e(14);
*/

if (unlink("A1") != 0) e(9);
}

void test39j()

```

```
{
/* Test utime. */

int fd;
time_t tloc;
struct utimbuf times;
struct stat stbuf;

subtest = 12;
if (system("rm -rf A2") != 0) e(1);
if ( (fd = creat("A2", 0666)) < 0) e(2);
times.modtime = 100;
if (utime("A2", &times) != 0) e(3);
if (stat("A2", &stbuf) != 0) e(4);
if (stbuf.st_mtime != 100) e(5);

tloc = time((time_t *)NULL);          /* get current time */
times.modtime = tloc;
if (utime("A2", &times) != 0) e(6);
if (stat("A2", &stbuf) != 0) e(7);
if (stbuf.st_mtime != tloc) e(8);
if (close(fd) != 0) e(9);
if (unlink("A2") != 0) e(10);
}

void e(n)
int n;
{
    int err_num = errno;              /* save errno in case printf clobbers it */

    printf("Subtest %d, error %d errno=%d ", subtest, n, errno);
    fflush(stdout);                  /* stdout and stderr are mixed horribly */
    errno = err_num;                 /* restore errno, just in case */
    perror("");
    if (errct++ > MAX_ERROR) {
        printf("Too many errors; test aborted\n");
        chdir("..");
        system("rm -rf DIR*");
        exit(1);
    }
}

void quit()
{
    chdir("..");
    system("rm -rf DIR*");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
        printf("%d errors\n", errct);
        exit(1);
    }
}
```

```

/* test 4 */

#include <sys/types.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

pid_t pid0, pid1, pid2, pid3;
int s, i, fd, nextb, errct = 0;
char *tempfile = "test4.temp";
char buf[1024];
extern int errno;

_PROTOTYPE(int main, (void));
_PROTOTYPE(void subr, (void));
_PROTOTYPE(void nofork, (void));
_PROTOTYPE(void quit, (void));

int main()
{
    int k;

    printf("Test 4 ");
    fflush(stdout);                /* have to flush for child's benefit */

    system("rm -rf DIR_04; mkdir DIR_04");
    chdir("DIR_04");

    creat(tempfile, 0777);
    for (k = 0; k < 20; k++) {
        subr();
    }
    unlink(tempfile);
    quit();
    return(-1);                  /* impossible */
}

void subr()
{
    if ( (pid0 = fork()) != 0 ) {
        /* Parent 0 */
        if (pid0 < 0) nofork();
        if ( (pid1 = fork()) != 0 ) {
            /* Parent 1 */
            if (pid1 < 0) nofork();
            if ( (pid2 = fork()) != 0 ) {
                /* Parent 2 */
                if (pid2 < 0) nofork();
                if ( (pid3 = fork()) != 0 ) {
                    /* Parent 3 */
                    if (pid3 < 0) nofork();
                    for (i = 0; i < 10000; i++)
                        kill(pid2, 9);
                    kill(pid1, 9);
                    kill(pid0, 9);
                    wait(&s);
                    wait(&s);
                    wait(&s);
                    wait(&s);
                } else {
                    fd = open(tempfile, O_RDONLY);
                    lseek(fd, 20480L * nextb, 0);
                    for (i = 0; i < 10; i++) read(fd, buf, 1024);
                    nextb++;
                    close(fd);
                    exit(0);
                }
            } else {
                while (1) getpid();
            }
        }
    }
}

```

```
        } else {
            while (1) getpid();
        }
    } else {
        while (1) getpid();
    }
}

void nofork()
{
    int e = errno;
    printf("Fork failed: %s (%d)\n", strerror(e), e);
    exit(1);
}

void quit()
{
    chdir("..");
    system("rm -rf DIR*");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
        printf("%d errors\n", errct);
        exit(1);
    }
}
```



```

/* test40: link() unlink()      Author: Jan-Mark Wams (jms@cs.vu.nl) */

/*
 * Not tested readonly file systems
 * Not tested fs full
 * Not tested unlinking busy files
 */

#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <limits.h>
#include <string.h>
#include <time.h>
#include <stdio.h>

int errct = 0;
int subtest = 1;
int superuser;
char MaxName[NAME_MAX + 1]; /* Name of maximum length */
char MaxPath[PATH_MAX]; /* Same for path */
char ToLongName[NAME_MAX + 2]; /* Name of maximum +1 length */
char ToLongPath[PATH_MAX + 1]; /* Same for path, both too long */

#define MAX_ERROR 4
#define ITERATIONS 2 /* LINK_MAX is high, so time consuming. */

#define System(cmd) if (system(cmd) != 0) printf("'%s' failed\n", cmd)
#define Chdir(dir) if (chdir(dir) != 0) printf("Can't goto %s\n", dir)
#define Stat(a,b) if (stat(a,b) != 0) printf("Can't stat %s\n", a)

_PROTOTYPE(void main, (int argc, char *argv[]));
_PROTOTYPE(void test40a, (void));
_PROTOTYPE(void test40b, (void));
_PROTOTYPE(void test40c, (void));
_PROTOTYPE(int stateq, (struct stat *stp1, struct stat *stp2));
_PROTOTYPE(void makelongnames, (void));
_PROTOTYPE(void e, (int __n));
_PROTOTYPE(void quit, (void));

void main(argc, argv)
int argc;
char *argv[];
{
    int i, m = 0xFFFF;

    sync();
    if (argc == 2) m = atoi(argv[1]);
    printf("Test 40 ");
    fflush(stdout);
    System("rm -rf DIR_40; mkdir DIR_40");
    Chdir("DIR_40");
    superuser = (getuid() == 0);
    makelongnames();

    for (i = 0; i < ITERATIONS; i++) {
        if (m & 0001) test40a();
        if (m & 0002) test40b();
        if (m & 0004) test40c();
    }
    quit();
}

void test40a()
{
    struct stat st1, st2, st3; /* Test normal operation. */
    time_t timel;

    subtest = 1;

    /* Clean up any residu. */

```

```

System("rm -rf ../DIR_40/*");

System("touch foo");          /* make source file */
Stat("foo", &st1);           /* get info of foo */
Stat(".", &st2);              /* and the cwd */
time(&timel);
while (timel >= time((time_t *)0))
    ;                          /* wait a sec */
if (link("foo", "bar") != 0) e(1); /* link foo to bar */
Stat("foo", &st3);           /* get new status */
if (st1.st_nlink + 1 != st3.st_nlink) e(2); /* link count foo up 1 */
#ifdef V1_FILESYSTEM
if (st1.st_ctime >= st3.st_ctime) e(3);      /* check stattime changed */
#endif
Stat(".", &st1);              /* get parend dir info */
if (st2.st_ctime >= st1.st_ctime) e(4);      /* ctime and mtime */
if (st2.st_mtime >= st1.st_mtime) e(5);     /* should be updated */
Stat("bar", &st2);            /* get info of bar */
if (st2.st_nlink != st3.st_nlink) e(6);     /* link count foo == bar */
if (st2.st_ino != st3.st_ino) e(7);         /* ino should be same */
if (st2.st_mode != st3.st_mode) e(8);      /* check mode same */
if (st2.st_uid != st3.st_uid) e(9);        /* check uid same */
if (st2.st_gid != st3.st_gid) e(10);       /* check gid same */
if (st2.st_size != st3.st_size) e(11);     /* check size */
if (st2.st_ctime != st3.st_ctime) e(12);   /* check ctime */
if (st2.st_atime != st3.st_atime) e(13);   /* check atime */
if (st2.st_mtime != st3.st_mtime) e(14);   /* check mtime */
Stat("foo", &st1);           /* get fooinfo */
Stat(".", &st2);            /* get dir info */
time(&timel);
while (timel >= time((time_t *)0))
    ;                          /* wait a sec */
if (unlink("bar") != 0) e(15); /* rm bar */
if (stat("bar", &st2) != -1) e(16); /* it's gone */
Stat("foo", &st3);           /* get foo again */
if (st1.st_nlink != st3.st_nlink + 1) e(17); /* link count back to normal */
#ifdef V1_FILESYSTEM
if (st1.st_ctime >= st3.st_ctime) e(18);    /* check ctime */
#endif
Stat(".", &st3);             /* get parend dir info */
if (st2.st_ctime >= st3.st_ctime) e(19);    /* ctime and mtime */
if (st2.st_mtime >= st3.st_mtime) e(20);   /* should be updated */
}

void test40b()
{
    register int nlink;
    char bar[30];
    struct stat st, st2;

    subtest = 2;

    /* Clean up any residu. */
    System("rm -rf ../DIR_40/*");

    /* Test what happens if we make LINK_MAX number of links. */
    System("touch foo");
    for (nlink = 2; nlink <= LINK_MAX; nlink++) {
        sprintf(bar, "bar.%d", nlink);
        if (link("foo", bar) != 0) e(2);
        Stat(bar, &st);
        if (st.st_nlink != nlink) e(3);
        Stat("foo", &st);
        if (st.st_nlink != nlink) e(4);
    }

    /* Check if we have LINK_MAX links that are all the same. */
    Stat("foo", &st);
    if (st.st_nlink != LINK_MAX) e(5);
    for (nlink = 2; nlink <= LINK_MAX; nlink++) {
        sprintf(bar, "bar.%d", nlink);
        Stat(bar, &st2);
        if (!stateq(&st, &st2)) e(6);
    }
}

```

```

/* Test no more links are possible. */
if (link("foo", "nono") != -1) e(7);
if (stat("nono", &st) != -1) e(8);
Stat("foo", &st);
if (st.st_nlink != LINK_MAX) e(9); /* recheck the number of links */

/* Now unlink() the bar.### files */
for (nlink = LINK_MAX; nlink >= 2; nlink--) {
    sprintf(bar, "bar.%d", nlink);
    Stat(bar, &st);
    if (st.st_nlink != nlink) e(10);
    Stat("foo", &st2);
    if (!stateq(&st, &st2)) e(11);
    if (unlink(bar) != 0) e(12);
}
Stat("foo", &st);
if (st.st_nlink != 1) e(13); /* number of links back to 1 */

/* Test max path ed. */
if (link("foo", MaxName) != 0) e(14); /* link to MaxName */
if (unlink(MaxName) != 0) e(15); /* and remove it */
MaxPath[strlen(MaxPath) - 2] = '/';
MaxPath[strlen(MaxPath) - 1] = 'a'; /* make ../.../a */
if (link("foo", MaxPath) != 0) e(16); /* it should be */
if (unlink(MaxPath) != 0) e(17); /* (un)linkable */

System("rm -f ../DIR_40/*"); /* clean cwd */
}

void test40c()
{
    subtest = 3;

    /* Clean up any residu. */
    System("rm -rf ../DIR_40/*");

    /* Check some simple things. */
    if (link("bar/nono", "nono") != -1) e(1); /* nonexistent */
    if (errno != ENOENT) e(2);
    Chdir("..");
    System("touch DIR_40/foo");
    System("chmod 677 DIR_40"); /* make inaccesable */
    if (!superuser) {
        if (unlink("DIR_40/foo") != -1) e(3);
        if (errno != EACCES) e(4);
    }
    if (link("DIR_40/bar/nono", "DIR_40/nono") != -1) e(5); /* nono no be */
    if (superuser) {
        if (errno != ENOENT) e(6); /* su has access */
    }
    if (!superuser) {
        if (errno != EACCES) e(7); /* we don't ;- ) */
    }
    System("chmod 577 DIR_40"); /* make unwritable */
    if (superuser) {
        if (link("DIR_40/foo", "DIR_40/nono") != 0) e(8);
        if (unlink("DIR_40/nono") != 0) e(9);
    }
    if (!superuser) {
        if (link("DIR_40/foo", "DIR_40/nono") != -1) e(10);
        if (errno != EACCES) e(11);
        if (unlink("DIR_40/foo") != -1) e(12); /* try to rm foo/foo */
        if (errno != EACCES) e(13);
    }
    System("chmod 755 DIR_40"); /* back to normal */
    Chdir("DIR_40");

    /* Too-long path and name test */
    ToLongPath[strlen(ToLongPath) - 2] = '/';
    ToLongPath[strlen(ToLongPath) - 1] = 'a'; /* make ../.../a */
    if (link("foo", ToLongPath) != -1) e(18); /* path is too long */
    if (errno != ENAMETOOLONG) e(19);
    if (unlink(ToLongPath) != -1) e(20); /* path is too long */
}

```

```
if (errno != ENAMETOOLONG) e(21);
if (link("foo", "foo") != -1) e(22);    /* try linking foo to foo */
if (errno != EEXIST) e(23);
if (link("foo", "bar") != 0) e(24);     /* make a link to bar */
if (link("foo", "bar") != -1) e(25);    /* try linking to bar again */
if (errno != EEXIST) e(26);
if (link("foo", "bar") != -1) e(27);    /* try linking to bar again */
if (errno != EEXIST) e(28);
if (unlink("nono") != -1) e(29);        /* try rm <not exist> */
if (errno != ENOENT) e(30);
if (unlink("") != -1) e(31);    /* try unlinking empty */
if (errno != ENOENT) e(32);
if (link("foo", "") != -1) e(33);    /* try linking to "" */
if (errno != ENOENT) e(34);
if (link("", "foo") != -1) e(35);    /* try linking "" */
if (errno != ENOENT) e(36);
if (link("", "") != -1) e(37); /* try linking "" to "" */
if (errno != ENOENT) e(38);
if (link("/foo/bar/foo", "a") != -1) e(39);    /* try no existing path */
if (errno != ENOENT) e(40);
if (link("foo", "/foo/bar/foo") != -1) e(41);    /* try no existing path */
if (errno != ENOENT) e(42);
if (link("/a/b/c", "/d/e/f") != -1) e(43);    /* try no existing path */
if (errno != ENOENT) e(44);
if (link("abc", "a") != -1) e(45);    /* try no existing file */
if (errno != ENOENT) e(46);
if (link("foo/bar", "bar") != -1) e(47);    /* foo is a file */
if (errno != ENOTDIR) e(48);
if (link("foo", "foo/bar") != -1) e(49);    /* foo is not a dir */
if (errno != ENOTDIR) e(50);
if (unlink("foo/bar") != -1) e(51);    /* foo still no dir */
if (errno != ENOTDIR) e(52);
if (!superuser) {
    if (link(".", "root") != -1) e(55);
    if (errno != EPERM) e(56);    /* noroot can't */
    if (unlink("root") != -1) e(57);
    if (errno != ENOENT) e(58);
}
if (mkdir("dir", 0777) != 0) e(59);
if (superuser) {
    if (rmdir("dir") != 0) e(63);
}
if (!superuser) {
    if (unlink("dir") != -1) e(64);
    if (errno != EPERM) e(65);    /* that ain't w'rkn */
    if (rmdir("dir") != 0) e(66);    /* that's the way to do it */
}
}

int stateq(stp1, stp2)
struct stat *stp1, *stp2;
{
    if (stp1->st_dev != stp2->st_dev) return 0;
    if (stp1->st_ino != stp2->st_ino) return 0;
    if (stp1->st_mode != stp2->st_mode) return 0;
    if (stp1->st_nlink != stp2->st_nlink) return 0;
    if (stp1->st_uid != stp2->st_uid) return 0;
    if (stp1->st_gid != stp2->st_gid) return 0;
    if (stp1->st_rdev != stp2->st_rdev) return 0;
    if (stp1->st_size != stp2->st_size) return 0;
    if (stp1->st_atime != stp2->st_atime) return 0;
    if (stp1->st_mtime != stp2->st_mtime) return 0;
    if (stp1->st_ctime != stp2->st_ctime) return 0;
    return 1;
}

void makelongnames()
{
    register int i;

    memset(MaxName, 'a', NAME_MAX);
    MaxName[NAME_MAX] = '\0';
    for (i = 0; i < PATH_MAX - 1; i++) {    /* idem path */
        MaxPath[i++] = '.';
    }
}
```

```
        MaxPath[i] = '/';
    }
    MaxPath[PATH_MAX - 1] = '\0';

    strcpy(ToLongName, MaxName); /* copy them Max to ToLong */
    strcpy(ToLongPath, MaxPath);

    ToLongName[NAME_MAX] = 'a';
    ToLongName[NAME_MAX + 1] = '\0'; /* extend ToLongName by one
                                       * too many */

    ToLongPath[PATH_MAX - 1] = '/';
    ToLongPath[PATH_MAX] = '\0'; /* inc ToLongPath by one */
}

void e(n)
int n;
{
    int err_num = errno; /* Save in case printf clobbers it. */

    printf("Subtest %d, error %d errno=%d: ", subtest, n, errno);
    errno = err_num;
    perror("");
    if (errct++ > MAX_ERROR) {
        printf("Too many errors; test aborted\n");
        chdir("..");
        system("rm -rf DIR*");
        exit(1);
    }
    errno = 0;
}

void quit()
{
    chdir("..");
    system("rm -rf DIR_40");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
        printf("%d errors\n", errct);
        exit(1);
    }
}
```

```
#include <stdio.h>
#include <minix/endpoint.h>
#include <minix/sys_config.h>

int main(int argc, char *argv[])
{
    int g, p;

    printf("Test 41 ");

    for(g = 0; g <= _ENDPOINT_MAX_GENERATION; g++) {
        for(p = -NR_TASKS; p < _NR_PROCS; p++) {
            int e, mg, mp;
            e = _ENDPOINT(g, p);
            mg = _ENDPOINT_G(e);
            mp = _ENDPOINT_P(e);
            if(mg != g || mp != p) {
                printf("%d != %d || %d != %d\n", mg, g, mp, p);
                return 1;
            }
            if(g == 0 && e != p) {
                printf("%d != %d and g=0\n", e, p);
                return 1;
            }
            if(e == ANY || e == SELF || e == NONE) {
                printf("endpoint (%d,%d) is %d; ANY, SELF or NONE\n",
                    g, p, e);
                return 1;
            }
        }
    }

    printf("ok\n");

    return 0;
}
```

```

/* test 5 */

#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <fcntl.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

#define ITERATIONS 2
#define MAX_ERROR 4

int errct;
int subtest;
int zero[1024];

int sigmap[5] = {9, 10, 11};

_PROTOTYPE(int main, (int argc, char *argv[]));
_PROTOTYPE(void test5a, (void));
_PROTOTYPE(void parent, (int childpid));
_PROTOTYPE(void child, (int parpid));
_PROTOTYPE(void func1, (int s));
_PROTOTYPE(void func8, (int s));
_PROTOTYPE(void func10, (int s));
_PROTOTYPE(void func11, (int s));
_PROTOTYPE(void test5b, (void));
_PROTOTYPE(void test5c, (void));
_PROTOTYPE(void test5d, (void));
_PROTOTYPE(void test5e, (void));
_PROTOTYPE(void test5f, (void));
_PROTOTYPE(void test5g, (void));
_PROTOTYPE(void funcalrm, (int s));
_PROTOTYPE(void test5h, (void));
_PROTOTYPE(void test5i, (void));
_PROTOTYPE(void ex, (void));
_PROTOTYPE(void e, (int n));
_PROTOTYPE(void quit, (void));

#ifdef _ANSI
void (*Signal(int _sig, void (*_func)(int))(int));
#define SIG_ZERO ((void (*)(int))0) /* default signal handling */
#else
sighandler_t Signal();
/* void (*Signal()) (); */
#define SIG_ZERO ((void (*)())0) /* default signal handling */
#endif

_VOLATILE int childsig, parsig, alarm;

int main(argc, argv)
int argc;
char *argv[];
{
    int i, m = 0x7777;

    printf("Test 5 ");
    fflush(stdout); /* have to flush for child's benefit */

    system("rm -rf DIR_05; mkdir DIR_05");
    chdir("DIR_05");

    for (i = 0; i < ITERATIONS; i++) {
        if (m & 0001) test5a();
        if (m & 0002) test5b();
        if (m & 0004) test5c();
        if (m & 0010) test5d();
        if (m & 0020) test5e();
        if (m & 0040) test5f();
        if (m & 0100) test5g();
        if (m & 0200) test5h();
    }
}

```

```
    if (m & 0400) test5i();
}
quit();
return(-1);          /* impossible */
}

void test5a()
{
    int parpid, childpid, flag, *zp;

    subtest = 0;
    flag = 0;
    for (zp = &zero[0]; zp < &zero[1024]; zp++)
        if (*zp != 0) flag = 1;
    if (flag) e(0);          /* check if bss is cleared to 0 */
    if (Signal(1, func1) == SIG_ERR) e(1);
    if (Signal(10, func10) < SIG_ZERO) e(2);
    parpid = getpid();
    if (childpid = fork()) {
        if (childpid < 0) ex();
        parent(childpid);
    } else {
        child(parpid);
    }
    if (Signal(1, SIG_DFL) < SIG_ZERO) e(4);
    if (Signal(10, SIG_DFL) < SIG_ZERO) e(5);
}

void parent(childpid)
int childpid;
{
    int i, pid;

    for (i = 0; i < 3; i++) {
        if (kill(childpid, 1) < 0) e(6);
        while (parsigs == 0);
        parsigs--;
    }
    if ( (pid = wait(&i)) < 0) e(7);
    if (i != 256 * 6) e(8);
}

void child(parpid)
int parpid;
{
    int i;

    for (i = 0; i < 3; i++) {
        while (childsigs == 0);
        childsigs--;
        if (kill(parpid, 10) < 0) e(9);
    }
    exit(6);
}

void func1(s)
int s;          /* for ANSI */
{
    if (Signal(1, func1) < SIG_ZERO) e(10);
    childsigs++;
}

void func8(s)
int s;
{
}

void func10(s)
int s;          /* for ANSI */
{
    if (Signal(10, func10) < SIG_ZERO) e(11);
    parsigs++;
}
```



```
void func11(s)
int s;                                /* for ANSI */
{
    e(38);
}

void test5b()
{
    int cpid, n, pid;

    subtest = 1;
    if ((pid = fork())) {
        if (pid < 0) ex();
        if ((pid = fork())) {
            if (pid < 0) ex();
            if (cpid = fork()) {
                if (cpid < 0) ex();
                if (kill(cpid, 9) < 0) e(12);
                if (wait(&n) < 0) e(13);
                if (wait(&n) < 0) e(14);
                if (wait(&n) < 0) e(15);
            } else {
                pause();
                while (1);
            }
        } else {
            exit(0);
        }
    } else {
        exit(0);
    }
}

void test5c()
{
    int n, i, pid, wpid;

    /* Test exit status codes for processes killed by signals. */
    subtest = 3;
    for (i = 0; i < 2; i++) {
        if (pid = fork()) {
            if (pid < 0) ex();
            sleep(2);          /* wait for child to pause */
            if (kill(pid, sigmap[i]) < 0) {
                e(20);
                exit(1);
            }
            if ((wpid = wait(&n)) < 0) e(21);
            if ((n & 077) != sigmap[i]) e(22);
            if (pid != wpid) e(23);
        } else {
            pause();
            exit(0);
        }
    }
}

void test5d()
{
    /* Test alarm */

    int i;

    subtest = 4;
    alarms = 0;
    for (i = 0; i < 8; i++) {
        Signal(SIGALRM, funcalarm);
        alarm(1);
        pause();
        if (alarms != i + 1) e(24);
    }
}
```

```
void test5e()
{
    /* When a signal knocks a processes out of WAIT or PAUSE, it is supposed to
     * get EINTR as error status. Check that.
     */
    int n, j;

    subtest = 5;
    if (Signal(8, func8) < SIG_ZERO) e(25);
    if (n = fork()) {
        /* Parent must delay to give child a chance to pause. */
        if (n < 0) ex();
        sleep(1);
        if (kill(n, 8) < 0) e(26);
        if (wait(&n) < 0) e(27);
        if (Signal(8, SIG_DFL) < SIG_ZERO) e(28);
    } else {
        j = pause();
        if (errno != EINTR && -errno != EINTR) e(29);
        exit(0);
    }
}

void test5f()
{
    int i, j, k, n;

    subtest = 6;
    if (getuid() != 0) return;
    n = fork();
    if (n < 0) ex();
    if (n) {
        wait(&i);
        i = (i >> 8) & 0377;
        if (i != (n & 0377)) e(30);
    } else {
        i = getgid();
        j = getegid();
        k = (i + j + 7) & 0377;
        if (setgid(k) < 0) e(31);
        if (getgid() != k) e(32);
        if (getegid() != k) e(33);
        i = getuid();
        j = geteuid();
        k = (i + j + 1) & 0377;
        if (setuid(k) < 0) e(34);
        if (getuid() != k) e(35);
        if (geteuid() != k) e(36);
        i = getpid() & 0377;
        if (wait(&j) != -1) e(37);
        exit(i);
    }
}

void test5g()
{
    int n;

    subtest = 7;
    Signal(11, func11);
    Signal(11, SIG_IGN);
    n = getpid();
    if (kill(n, 11) != 0) e(1);
    Signal(11, SIG_DFL);
}

void funcalrm(s)
int s;
/* for ANSI */
{
    alarms++;
}

void test5h()
{

```

```

/* When a signal knocks a processes out of PIPE, it is supposed to
 * get EINTR as error status. Check that.
 */
int n, j, fd[2];

subtest = 8;
unlink("XXX.test5");
if (Signal(8, func8) < SIG_ZERO) e(1);
pipe(fd);
if (n = fork()) {
    /* Parent must delay to give child a chance to pause. */
    if (n < 0) ex();
    while (access("XXX.test5", 0) != 0) /* just wait */ ;
    sleep(1);
    unlink("XXX.test5");
    if (kill(n, 8) < 0) e(2);
    if (wait(&n) < 0) e(3);
    if (Signal(8, SIG_DFL) < SIG_ZERO) e(4);
    if (close(fd[0]) != 0) e(5);
    if (close(fd[1]) != 0) e(6);
} else {
    if (creat("XXX.test5", 0777) < 0) e(7);
    j = read(fd[0], (char *) &n, 1);
    if (errno != EINTR) e(8);
    exit(0);
}
}

void test5i()
{
    int fd[2], pid, buf[10], n;

    subtest = 9;
    pipe(fd);
    unlink("XXXxxxXXX");

    if ( (pid = fork()) ) {
        /* Parent */
        /* Wait until child has started and has created the XXXxxxXXX file. */
        while (access("XXXxxxXXX", 0) != 0) /* loop */ ;
        sleep(1);
        if (kill(pid, SIGKILL) != 0) e(1);
        if (wait(&n) < 0) e(2);
        if (close(fd[0]) != 0) e(3);
        if (close(fd[1]) != 0) e(4);
    } else {
        if (creat("XXXxxxXXX", 0777) < 0) e(5);
        read(fd[0], (char *) buf, 1);
        e(5); /* should be killed by signal and not get here */
    }
    unlink("XXXxxxXXX");
}

void ex()
{
    int e = errno;
    printf("Fork failed: %s(%d)\n", strerror(e), e);
    exit(1);
}

void e(n)
int n;
{
    int err_num = errno; /* save errno in case printf clobbers it */

    printf("Subtest %d, error %d errno=%d ", subtest, n, errno);
    errno = err_num; /* restore errno, just in case */
    perror("");
    if (errct++ > MAX_ERROR) {
        printf("Too many errors; test aborted\n");
        chdir("..");
        system("rm -rf DIR*");
        exit(1);
    }
}

```

```
}

#ifdef _ANSI
void (*Signal(int a, void (*b)(int)))(int)
#else
sighandler_t Signal(a, b)
int a;
void (*b)();
#endif
{
    if (signal(a, (void (*)()) b) == (void (*)()) -1)
        return(SIG_ERR);
    else
        return(SIG_ZERO);
}

void quit()
{
    chdir("..");
    system("rm -rf DIR*");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
        printf("%d errors\n", errct);
        exit(1);
    }
}
```

```
/* test 6 */

#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
#include <limits.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

#define MAX_ERROR 4

int errct;
int subtest = 1;
int zilch[5000];
char curdir[PATH_MAX];

_PROTOTYPE(int main, (int argc, char *argv []));
_PROTOTYPE(void test6a, (void));
_PROTOTYPE(void test6b, (void));
_PROTOTYPE(void test6c, (void));
_PROTOTYPE(void e, (int n));
_PROTOTYPE(void quit, (void));

int main(argc, argv)
int argc;
char *argv[];
{
    int i, m = 0xFFFF;

    sync();
    if (geteuid() == 0 || getuid() == 0) {
        printf("Test 6 cannot run as root; test aborted\n");
        exit(1);
    }

    if (argc == 2) m = atoi(argv[1]);

    printf("Test 6 ");
    fflush(stdout);

    getcwd(curdir, PATH_MAX);
    system("rm -rf DIR_06; mkdir DIR_06");
    chdir("DIR_06");

    for (i = 0; i < 70; i++) {
        if (m & 00001) test6a();
        if (m & 00002) test6b();
        if (m & 00004) test6c();
    }

    quit();
    return(-1); /* impossible */
}

void test6a()
{
    /* Test sbrk() and brk(). */

    char *addr, *addr2, *addr3;
    int i, del, click, click2;

    subtest = 1;
    addr = sbrk(0);
    addr = sbrk(0); /* force break to a click boundary */
    for (i = 0; i < 10; i++) sbrk(7 * i);
    for (i = 0; i < 10; i++) sbrk(-7 * i);
    if (sbrk(0) != addr) e(1);
    sbrk(30);
    if (brk(addr) != 0) e(2);
    if (sbrk(0) != addr) e(3);
}
```

```
del = 0;
do {
    del++;
    brk(addr + del);
    addr2 = sbrk(0);
} while (addr2 == addr);
click = addr2 - addr;
sbrk(-1);
if (sbrk(0) != addr) e(4);
brk(addr);
if (sbrk(0) != addr) e(5);

del = 0;
do {
    del++;
    brk(addr - del);
    addr3 = sbrk(0);
} while (addr3 == addr);
click2 = addr - addr3;
sbrk(1);
if (sbrk(0) != addr) e(6);
brk(addr);
if (sbrk(0) != addr) e(8);
if (click != click2) e(9);

brk(addr + 2 * click);
if (sbrk(0) != addr + 2 * click) e(10);
sbrk(3 * click);
if (sbrk(0) != addr + 5 * click) e(11);
sbrk(-5 * click);
if (sbrk(0) != addr) e(12);
}

void test6b()
{
    int i, err;

    subtest = 2;
    signal(SIGQUIT, SIG_IGN);
    err = 0;
    for (i = 0; i < 5000; i++)
        if (zilch[i] != 0) err++;
    if (err > 0) e(1);
    kill(getpid(), SIGQUIT);
}

void test6c()
{
    /* Test mknod, chdir, chmod, chown, access. */

    int i, j;
    struct stat s;

    subtest = 3;
    if (getuid() != 0) return;
    for (j = 0; j < 2; j++) {
        umask(0);

        if (chdir("/") < 0) e(1);
        if (mknod("dir", 040700, 0) < 0) e(2);
        if (link("/", "/dir/..") < 0) e(3);
        if (mknod("T3a", 0777, 0) < 0) e(4);
        if (mknod("/dir/T3b", 0777, 0) < 0) e(5);
        if (mknod("dir/T3c", 0777, 0) < 0) e(6);
        if ((i = open("/dir/T3b", 0)) < 0) e(7);
        if (close(i) < 0) e(8);
        if ((i = open("dir/T3c", O_RDONLY)) < 0) e(9);
        if (close(i) < 0) e(10);
        if (chdir("dir") < 0) e(11);
        if ((i = open("T3b", 0)) < 0) e(12);
        if (close(i) < 0) e(13);
        if ((i = open("../T3a", O_RDONLY)) < 0) e(14);
        if (close(i) < 0) e(15);
        if ((i = open("../dir/./dir/./dir/./dir/./dir/T3c", O_RDONLY)) < 0)
```

```
        e(16);
    if (close(i) < 0) e(17);

    if (chmod("../dir../dir../dir../dir../T3a", 0123) < 0) e(18);
    if (stat("../dir../dir../dir../T3a", &s) < 0) e(19);
    if ((s.st_mode & 0777777) != 0123) e(20);
    if (chmod("../dir../dir../T3a", 0456) < 0) e(21);
    if (stat("../T3a", &s) < 0) e(22);
    if ((s.st_mode & 0777777) != 0456) e(23);
    if (chown("../dir../dir../T3a", 20, 30) < 0) e(24);
    if (stat("../T3a", &s) < 0) e(25);
    if (s.st_uid != 20) e(26);
    if (s.st_gid != 30) e(27);

    if ((i = open("/T3c", O_RDONLY)) >= 0) e(28);
    if ((i = open("/T3a", O_RDONLY)) < 0) e(29);
    if (close(i) < 0) e(30);

    if (access("/T3a", 4) < 0) e(31);
    if (access("/dir/T3b", 4) < 0) e(32);
    if (access("/dir/T3d", 4) >= 0) e(33);

    if (unlink("T3b") < 0) e(34);
    if (unlink("T3c") < 0) e(35);
    if (unlink("..") < 0) e(36);
    if (chdir("/") < 0) e(37);
    if (unlink("dir") < 0) e(38);
    if (unlink("/T3a") < 0) e(39);
}
}

void e(n)
int n;
{
    int err_num = errno;          /* save errno in case printf clobbers it */

    printf("Subtest %d, error %d errno=%d ", subtest, n, errno);
    errno = err_num;              /* restore errno, just in case */
    perror("");
    if (errct++ > MAX_ERROR) {
        printf("Too many errors; test aborted\n");
        chdir("..");
        system("rm -rf DIR*");
        exit(1);
    }
}

void quit()
{
    chdir("..");
    system("rm -rf DIR*");

    chdir(curdir);
    system("rm -rf DIR*");
    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
        printf("%d errors\n", errct);
        exit(1);
    }
}
```

```
/* POSIX test program (7). Author: Andy Tanenbaum */

/* The following POSIX calls are tested:
 *    pipe(), mkfifo(), fcntl()
 */

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <errno.h>
#include <fcntl.h>
#include <limits.h>
#include <signal.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <stdio.h>

#define ITERATIONS      4
#define MAX_ERROR 4
#define ITEMS    32
#define READ     10
#define WRITE    20
#define UNLOCK   30
#define U        70
#define L        80

char buf[ITEMS] = {0,1,2,3,4,5,6,7,8,9,8,7,6,5,4,3,2,1,0,1,2,3,4,5,6,7,8,9};

int subtest, errct, xfd;
int whence = SEEK_SET, func_code = F_SETLK;
extern char **environ;

_PROTOTYPE(int main, (int argc, char *argv []));
_PROTOTYPE(void test7a, (void));
_PROTOTYPE(void test7b, (void));
_PROTOTYPE(void test7c, (void));
_PROTOTYPE(void test7d, (void));
_PROTOTYPE(void test7e, (void));
_PROTOTYPE(void test7f, (void));
_PROTOTYPE(void test7g, (void));
_PROTOTYPE(void test7h, (void));
_PROTOTYPE(void test7i, (void));
_PROTOTYPE(void test7j, (void));
_PROTOTYPE(void cloexec_test, (void));
_PROTOTYPE(int set, (int how, int first, int last));
_PROTOTYPE(int locked, (int b));
_PROTOTYPE(void e, (int n));
_PROTOTYPE(void sigfunc, (int s));
_PROTOTYPE(void quit, (void));

int main(argc, argv)
int argc;
char *argv[];
{
    int i, m = 0xFFFF;

    sync();

    if (argc == 2) m = atoi(argv[1]);
    if (m == 0) cloexec_test(); /* important; do not remove this! */
    printf("Test 7 ");
    fflush(stdout);

    system("rm -rf DIR_07; mkdir DIR_07");
    chdir("DIR_07");

    for (i = 0; i < ITERATIONS; i++) {
        if (m & 00001) test7a();
        if (m & 00002) test7b();
        if (m & 00004) test7c();
        if (m & 00010) test7d();
        if (m & 00020) test7e();
    }
}
```



```
        if (m & 00040) test7f();
        if (m & 00100) test7g();
        if (m & 00200) test7h();
        if (m & 00400) test7i();
        if (m & 01000) test7j();
    }
    quit();
    return(-1);          /* impossible */
}

void test7a()
{
    /* Test pipe(). */

    int i, fd[2], ect;
    char buf2[ITEMS+1];

    /* Create a pipe, write on it, and read it back. */
    subtest = 1;
    if (pipe(fd) != 0) e(1);
    if (write(fd[1], buf, ITEMS) != ITEMS) e(2);
    buf2[0] = 0;
    if (read(fd[0], buf2, ITEMS+1) != ITEMS) e(3);
    ect = 0;
    for (i = 0; i < ITEMS; i++) if (buf[i] != buf2[i]) ect++;
    if (ect != 0) e(4);
    if (close(fd[0]) != 0) e(5);
    if (close(fd[1]) != 0) e(6);

    /* Error test. Keep opening pipes until it fails. Check error code. */
    errno = 0;
    while (1) {
        if (pipe(fd) < 0) break;
    }
    if (errno != EMFILE) e(7);

    /* Close all the pipes. */
    for (i = 3; i < OPEN_MAX; i++) close(i);
}

void test7b()
{
    /* Test mkfifo(). */

    int fdr, fdw, status;
    char buf2[ITEMS+1];
    int efork;

    /* Create a fifo, write on it, and read it back. */
    subtest = 2;
    if (mkfifo("T7.b", 0777) != 0) e(1);
    switch (fork()) {
    case -1:
        efork = errno;
        printf("Fork failed: %s(%d)\n", strerror(efork), efork);
        exit(1);
    case 0:
        /* Child reads from the fifo. */
        if ( (fdr = open("T7.b", O_RDONLY)) < 0) e(5);
        if (read(fdr, buf2, ITEMS+1) != ITEMS) e(6);
        if (strcmp(buf, buf2) != 0) e(7);
        if (close(fdr) != 0) e(8);
        exit(0);
    default:
        /* Parent writes on the fifo. */
        if ( (fdw = open("T7.b", O_WRONLY)) < 0) e(2);
        if (write(fdw, buf, ITEMS) != ITEMS) e(3);
        wait(&status);
        if (close(fdw) != 0) e(4);
    }

    /* Check some error conditions. */
    if (mkfifo("T7.b", 0777) != -1) e(9);
    errno = 0;
}
```

```

if (mkfifo("a/b/c", 0777) != -1) e(10);
if (errno != ENOENT) e(11);
errno = 0;
if (mkfifo("", 0777) != -1) e(12);
if (errno != ENOENT) e(13);
errno = 0;
if (mkfifo("T7.b/x", 0777) != -1) e(14);
if (errno != ENOTDIR) e(15);
if (unlink("T7.b") != 0) e(16);

/* Now check fifos and the O_NONBLOCK flag. */
if (mkfifo("T7.b", 0600) != 0) e(17);
errno = 0;
if (open("T7.b", O_WRONLY | O_NONBLOCK) != -1) e(18);
if (errno != ENXIO) e(19);
if ( (fdr = open("T7.b", O_RDONLY | O_NONBLOCK)) < 0) e(20);
if (fork()) {
    /* Parent reads from fdr. */
    wait(&status);          /* but first make sure writer has already run*/
    if ( ( (status>>8) & 0377) != 77) e(21);
    if (read(fdr, buf2, ITEMS+1) != ITEMS) e(22);
    if (strcmp(buf, buf2) != 0) e(23);
    if (close(fdr) != 0) e(24);
} else {
    /* Child opens the fifo for writing and writes to it. */
    if ( (fdw = open("T7.b", O_WRONLY | O_NONBLOCK)) < 0) e(25);
    if (write(fdw, buf, ITEMS) != ITEMS) e(26);
    if (close(fdw) != 0) e(27);
    exit(77);
}

if (unlink("T7.b") != 0) e(28);
}

void test7c()
{
/* Test fcntl(). */

int fd, m, s, newfd, newfd2;
struct stat stat1, stat2, stat3;

subtest = 3;
errno = -100;
if ( (fd = creat("T7.c", 0777)) < 0) e(1);

/* Turn the per-file-descriptor flags on and off. */
if (fcntl(fd, F_GETFD) != 0) e(2);    /* FD_CLOEXEC is initially off */
if (fcntl(fd, F_SETFD, FD_CLOEXEC) != 0) e(3); /* turn it on */
if (fcntl(fd, F_GETFD) != FD_CLOEXEC) e(4);    /* should be on now */
if (fcntl(fd, F_SETFD, 0) != 0) e(5);          /* turn it off */
if (fcntl(fd, F_GETFD) != 0) e(6);            /* should be off now */

/* Turn the open-file-description flags on and off. Start with O_APPEND. */
m = O_WRONLY;
if (fcntl(fd, F_GETFL) != m) e(7);    /* O_APPEND, O_NONBLOCK are off */
if (fcntl(fd, F_SETFL, O_APPEND) != 0) e(8); /* turn on O_APPEND */
if (fcntl(fd, F_GETFL) != (O_APPEND | m)) e(9); /* should be on now */
if (fcntl(fd, F_SETFL, 0) != 0) e(10); /* turn it off */
if (fcntl(fd, F_GETFL) != m) e(11);    /* should be off now */

/* Turn the open-file-description flags on and off. Now try O_NONBLOCK. */
if (fcntl(fd, F_SETFL, O_NONBLOCK) != 0) e(12); /* turn on O_NONBLOCK */
if (fcntl(fd, F_GETFL) != (O_NONBLOCK | m)) e(13); /* should be on now */
if (fcntl(fd, F_SETFL, 0) != 0) e(14); /* turn it off */
if (fcntl(fd, F_GETFL) != m) e(15);    /* should be off now */

/* Now both at once. */
if (fcntl(fd, F_SETFL, O_APPEND|O_NONBLOCK) != 0) e(16);
if (fcntl(fd, F_GETFL) != (O_NONBLOCK | O_APPEND | m)) e(17);
if (fcntl(fd, F_SETFL, 0) != 0) e(18);
if (fcntl(fd, F_GETFL) != m) e(19);

/* Now test F_DUPFD. */
if ( (newfd = fcntl(fd, F_DUPFD, 0)) != 4) e(20);    /* 0-4 open */

```

```

if ( (newfd2 = fcntl(fd, F_DUPFD, 0)) != 5) e(21);      /* 0-5 open */
if (close(newfd) != 0) e(22);                          /* 0-3, 5 open */
if ( (newfd = fcntl(fd, F_DUPFD, 0)) != 4) e(23);      /* 0-5 open */
if (close(newfd) != 0) e(24);                          /* 0-3, 5 open */
if ( (newfd = fcntl(fd, F_DUPFD, 5)) != 6) e(25);      /* 0-3, 5, 6 open */
if (close(newfd2) != 0) e(26);                        /* 0-3, 6 open */

/* O_APPEND should be inherited, but FD_CLOEXEC should be cleared. Check. */
if (fcntl(fd, F_SETFD, FD_CLOEXEC) != 0) e(26); /* turn FD_CLOEXEC on */
if (fcntl(fd, F_SETFL, O_APPEND) != 0) e(27); /* turn O_APPEND on */
if ( (newfd2 = fcntl(fd, F_DUPFD, 10)) != 10) e(28); /* 0-3, 6, 10 open */
if (fcntl(newfd2, F_GETFD) != 0) e(29); /* FD_CLOEXEC must be 0 */
if (fcntl(newfd2, F_GETFL) != (O_APPEND | m)) e(30); /* O_APPEND set */
if (fcntl(fd, F_SETFD, 0) != 0) e(31); /* turn FD_CLOEXEC off */

/* Check if newfd and newfd2 are the same inode. */
if (fstat(fd, &stat1) != 0) e(32);
if (fstat(fd, &stat2) != 0) e(33);
if (fstat(fd, &stat3) != 0) e(34);
if (stat1.st_dev != stat2.st_dev) e(35);
if (stat1.st_dev != stat3.st_dev) e(36);
if (stat1.st_ino != stat2.st_ino) e(37);
if (stat1.st_ino != stat3.st_ino) e(38);

/* Now check on the FD_CLOEXEC flag. Set it for fd (3) and newfd2 (10) */
if (fd != 3 || newfd2 != 10 || newfd != 6) e(39);
if (fcntl(fd, F_SETFD, FD_CLOEXEC) != 0) e(40); /* close 3 on exec */
if (fcntl(newfd2, F_SETFD, FD_CLOEXEC) != 0) e(41); /* close 10 on exec */
if (fcntl(newfd, F_SETFD, 0) != 0) e(42); /* don't close 6 */
if (fork()) {
    wait(&s); /* parent just waits */
    if (WEXITSTATUS(s) != 0) e(43);
} else {
    execl("./test7", "test7", "0", (char *) 0, environ);
    exit(1); /* the impossible never happens, right? */
}

/* Finally, close all the files. */
if (fcntl(fd, F_SETFD, 0) != 0) e(44); /* FD_CLOEXEC off */
if (fcntl(newfd2, F_SETFD, 0) != 0) e(45); /* FD_CLOEXEC off */
if (close(fd) != 0) e(46);
if (close(newfd) != 0) e(47);
if (close(newfd2) != 0) e(48);
}

void test7d()
{
/* Test file locking. */

    subtest = 4;

    if ( (xfile = creat("T7.d", 0777)) != 3) e(1);
    close(xfile);
    if ( (xfile = open("T7.d", O_RDWR)) < 0) e(2);
    if (write(xfile, buf, ITEMS) != ITEMS) e(3);
    if (set(WRITE, 0, 3) != 0) e(4);
    if (set(WRITE, 5, 9) != 0) e(5);
    if (set(UNLOCK, 0, 3) != 0) e(6);
    if (set(UNLOCK, 4, 9) != 0) e(7);

    if (set(READ, 1, 4) != 0) e(8);
    if (set(READ, 4, 7) != 0) e(9);
    if (set(UNLOCK, 4, 7) != 0) e(10);
    if (set(UNLOCK, 1, 4) != 0) e(11);

    if (set(WRITE, 0, 3) != 0) e(12);
    if (set(WRITE, 5, 7) != 0) e(13);
    if (set(WRITE, 9, 10) != 0) e(14);
    if (set(UNLOCK, 0, 4) != 0) e(15);
    if (set(UNLOCK, 0, 7) != 0) e(16);
    if (set(UNLOCK, 0, 2000) != 0) e(17);

    if (set(WRITE, 0, 3) != 0) e(18);
    if (set(WRITE, 5, 7) != 0) e(19);

```

```
if (set(WRITE, 9, 10) != 0) e(20);
if (set(UNLOCK, 0, 100) != 0) e(21);

if (set(WRITE, 0, 9) != 0) e(22);
if (set(UNLOCK, 8, 9) != 0) e(23);
if (set(UNLOCK, 0, 2) != 0) e(24);
if (set(UNLOCK, 5, 5) != 0) e(25);
if (set(UNLOCK, 4, 6) != 0) e(26);
if (set(UNLOCK, 3, 3) != 0) e(27);
if (set(UNLOCK, 7, 7) != 0) e(28);

if (set(WRITE, 0, 10) != 0) e(29);
if (set(UNLOCK, 0, 1000) != 0) e(30);

/* Up until now, all locks have been disjoint. Now try conflicts. */
if (set(WRITE, 0, 4) != 0) e(31);
if (set(WRITE, 4, 7) != 0) e(32);      /* owner may lock same byte twice */
if (set(WRITE, 5, 10) != 0) e(33);
if (set(UNLOCK, 0, 11) != 0) e(34);

/* File is now unlocked. Length 0 means whole file. */
if (set(WRITE, 2, 1) != 0) e(35);      /* this locks whole file */
if (set(WRITE, 9, 10) != 0) e(36);      /* a process can relock its file */
if (set(WRITE, 3, 3) != 0) e(37);
if (set(UNLOCK, 0, -1) != 0) e(38);      /* file is now unlocked. */

/* Test F_GETLK. */
if (set(WRITE, 2, 3) != 0) e(39);
if (locked(1) != U) e(40);
if (locked(2) != L) e(41);
if (locked(3) != L) e(42);
if (locked(4) != U) e(43);
if (set(UNLOCK, 2, 3) != 0) e(44);
if (locked(2) != U) e(45);
if (locked(3) != U) e(46);

close(xfd);
}

void test7e()
{
/* Test to see if SETLKW blocks as it should. */

int pid, s;

subtest = 5;

if ( ( xfd = creat("T7.e", 0777) ) != 3) e(1);
if (close(xfd) != 0) e(2);
if ( ( xfd = open("T7.e", O_RDWR) ) < 0) e(3);
if (write(xfd, buf, ITEMS) != ITEMS) e(4);
if (set(WRITE, 0, 3) != 0) e(5);

if ( (pid = fork()) ) {
    /* Parent waits until child has started before signaling it. */
    while (access("T7.e1", 0) != 0) ;
    unlink("T7.e1");
    sleep(1);
    if (kill(pid, SIGKILL) < 0) e(6);
    if (wait(&s) != pid) e(7);
} else {
    /* Child tries to lock and should block. */
    if (creat("T7.e1", 0777) < 0) e(8);
    func_code = F_SETLKW;
    if (set(WRITE, 0, 3) != 0) e(9);      /* should block */
    errno = -1000;
    e(10);
    exit(0);
}
close(xfd);
}

void test7f()
{
```

```
/* Test to see if SETLKW gives EINTR when interrupted. */

int pid, s;

subtest = 6;

if ( (xfd = creat("T7.f", 0777)) != 3) e(1);
if (close(xfd) != 0) e(2);
if ( (xfd = open("T7.f", O_RDWR)) < 0) e(3);
if (write(xfd, buf, ITEMS) != ITEMS) e(4);
if (set(WRITE, 0, 3) != 0) e(5);

if ( (pid = fork()) ) {
    /* Parent waits until child has started before signaling it. */
    while (access("T7.fl", 0) != 0) ;
    unlink("T7.fl");
    sleep(1);
    if (kill(pid, SIGTERM) < 0) e(6);
    if (wait(&s) != pid) e(7);
    if ( (s >> 8) != 19) e(8);
} else {
    /* Child tries to lock and should block.
     * 'signal(SIGTERM, sigfunc);' to set the signal handler is inadequate
     * because on systems like BSD the sigaction flags for signal include
     * 'SA_RESTART' so syscalls are restarted after they have been
     * interrupted by a signal.
     */
    struct sigaction sa, osa;

    sa.sa_handler = sigfunc;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    if (sigaction(SIGTERM, &sa, &osa) < 0) e(999);
    if (creat("T7.fl", 0777) < 0) e(9);
    func_code = F_SETLKW;
    if (set(WRITE, 0, 3) != -1) e(10); /* should block */
    if (errno != EINTR) e(11); /* signal should release it */
    exit(19);
}
close(xfd);
}

void test7g()
{
    /* Test to see if SETLKW unlocks when the needed lock becomes available. */

    int pid, s;

    subtest = 7;

    if ( (xfd = creat("T7.g", 0777)) != 3) e(1);
    if (close(xfd) != 0) e(2);
    if ( (xfd = open("T7.g", O_RDWR)) < 0) e(3);
    if (write(xfd, buf, ITEMS) != ITEMS) e(4);
    if (set(WRITE, 0, 3) != 0) e(5); /* bytes 0 to 3 are now locked */

    if ( (pid = fork()) ) {
        /* Parent waits for child to start. */
        while (access("T7.gl", 0) != 0) ;
        unlink("T7.gl");
        sleep(1);
        if (set(UNLOCK, 0, 3) != 0) e(5);
        if (wait(&s) != pid) e(6);
        if ( (s >> 8) != 29) e(7);
    } else {
        /* Child tells parent it is alive, then tries to lock and is blocked.*/
        func_code = F_SETLKW;
        if (creat("T7.gl", 0777) < 0) e(8);
        if (set(WRITE, 3, 3) != 0) e(9); /* process must block now */
        if (set(UNLOCK, 3, 3) != 0) e(10);
        exit(29);
    }
    close(xfd);
}
```

```

void test7h()
{
    /* Test to see what happens if two processes block on the same lock. */

    int pid, pid2, s, w;

    subtest = 8;

    if ( (xfd = creat("T7.h", 0777)) != 3) e(1);
    if (close(xfd) != 0) e(2);
    if ( (xfd = open("T7.h", O_RDWR)) < 0) e(3);
    if (write(xfd, buf, ITEMS) != ITEMS) e(4);
    if (set(WRITE, 0, 3) != 0) e(5);      /* bytes 0 to 3 are now locked */

    if ( (pid = fork()) ) {
        if ( (pid2 = fork()) ) {
            /* Parent waits for child to start. */
            while (access("T7.h1", 0) != 0) ;
            while (access("T7.h2", 0) != 0) ;
            unlink("T7.h1");
            unlink("T7.h2");
            sleep(1);
            if (set(UNLOCK, 0, 3) != 0) e(6);
            w = wait(&s);
            if (w != pid && w != pid2) e(7);
            s = s >> 8;
            if (s != 39 && s != 49) e(8);
            w = wait(&s);
            if (w != pid && w != pid2) e(9);
            s = s >> 8;
            if (s != 39 && s != 49) e(10);
        } else {
            func_code = F_SETLK;
            if (creat("T7.h1", 0777) < 0) e(11);
            if (set(WRITE, 0, 0) != 0) e(12);      /* block now */
            if (set(UNLOCK, 0, 0) != 0) e(13);
            exit(39);
        }
    } else {
        /* Child tells parent it is alive, then tries to lock and is blocked.*/
        func_code = F_SETLK;
        if (creat("T7.h2", 0777) < 0) e(14);
        if (set(WRITE, 0, 1) != 0) e(15);      /* process must block now */
        if (set(UNLOCK, 0, 1) != 0) e(16);
        exit(49);
    }
    close(xfd);
}

void test7i()
{
    /* Check error conditions for fcntl(). */

    int tfd, i;

    subtest = 9;

    errno = 0;
    if ( (xfd = creat("T7.i", 0777)) != 3) e(1);
    if (close(xfd) != 0) e(2);
    if ( (xfd = open("T7.i", O_RDWR)) < 0) e(3);
    if (write(xfd, buf, ITEMS) != ITEMS) e(4);
    if (set(WRITE, 0, 3) != 0) e(5);      /* bytes 0 to 3 are now locked */
    if (set(WRITE, 0, 0) != 0) e(6);
    if (errno != 0) e(7);
    errno = 0;
    if (set(WRITE, 3, 3) != 0) e(8);
    if (errno != 0) e(9);
    tfd = xfd;      /* hold good value */
    xfd = -99;
    errno = 0;
    if (set(WRITE, 0, 0) != -1) e(10);
    if (errno != EBADF) e(11);
}

```

```

errno = 0;
if ( (xfd = open("T7.i", O_WRONLY)) < 0) e(12);
if (set(READ, 0, 0) != -1) e(13);
if (errno != EBADF) e(14);
if (close(xfd) != 0) e(15);

errno = 0;
if ( (xfd = open("T7.i", O_RDONLY)) < 0) e(16);
if (set(WRITE, 0, 0) != -1) e(17);
if (errno != EBADF) e(18);
if (close(xfd) != 0) e(19);
xfd = tfd;                                /* restore legal xfd value */

/* Check for EINVAL. */
errno = 0;
if (fcntl(xfd, F_DUPFD, OPEN_MAX) != -1) e(20);
if (errno != EINVAL) e(21);
errno = 0;
if (fcntl(xfd, F_DUPFD, -1) != -1) e(22);
if (errno != EINVAL) e(23);

xfd = 0;                                /* stdin does not support locking */
errno = 0;
if (set(READ, 0, 0) != -1) e(24);
if (errno != EINVAL) e(25);
xfd = tfd;

/* Check ENOLCK. */
for (i = 0; i < ITEMS; i++) {
    if (set(WRITE, i, i) == 0) continue;
    if (errno != ENOLCK) {
        e(26);
        break;
    }
}

/* Check EMFILE. */
for (i = xfd + 1; i < OPEN_MAX; i++) open("T7.i", 0); /* use up all fds */
errno = 0;
if (fcntl(xfd, F_DUPFD, 0) != -1) e(27);          /* No fds left */
if (errno != EMFILE) e(28);

for (i = xfd; i < OPEN_MAX; i++) if (close(i) != 0) e(29);
}

void test7j()
{
    /* Test file locking with two processes. */

    int s;

    subtest = 10;

    if ( (xfd = creat("T7.j", 0777)) != 3) e(1);
    close(xfd);
    if ( (xfd = open("T7.j", O_RDWR)) < 0) e(2);
    if (write(xfd, buf, ITEMS) != ITEMS) e(3);
    if (set(WRITE, 0, 4) != 0) e(4);          /* lock belongs to parent */
    if (set(READ, 10, 16) != 0) e(5);        /* lock belongs to parent */

    /* Up until now, all locks have been disjoint. Now try conflicts. */
    if (fork()) {
        /* Parent just waits for child to finish. */
        wait(&s);
    } else {
        /* Child does the testing. */
        errno = -100;
        if (set(WRITE, 5, 7) < 0) e(6); /* should work */
        if (set(WRITE, 4, 7) >= 0) e(7);          /* child may not lock byte 4 */
        if (errno != EACCES && errno != EAGAIN) e(8);
        if (set(WRITE, 5, 9) != 0) e(9);
        if (set(UNLOCK, 5, 9) != 0) e(10);
        if (set(READ, 9, 17) < 0) e(11);          /* shared READ lock is ok */
    }
}

```

```
        exit(0);
    }
    close(xfd);
}

void cloexec_test()
{
    /* To test whether the FD_CLOEXEC flag actually causes files to be
     * closed upon exec, we have to exec something. The test is carried
     * out by forking, and then having the child exec test7 itself, but
     * with argument 0. This is detected, and control comes here.
     * File descriptors 3 and 10 should be closed here, and 10 open.
     */

    if (close(3) == 0) e(1001); /* close should fail; it was closed on exec */
    if (close(6) != 0) e(1002); /* close should succeed */
    if (close(10) == 0) e(1003); /* close should fail */
    fflush(stdout);
    exit(0);
}

int set(how, first, last)
int how, first, last;
{
    int r;
    struct flock flock;

    if (how == READ) flock.l_type = F_RDLCK;
    if (how == WRITE) flock.l_type = F_WRLCK;
    if (how == UNLOCK) flock.l_type = F_UNLCK;
    flock.l_whence = whence;
    flock.l_start = (long) first;
    flock.l_len = (long) last - (long) first + 1;
    r = fcntl(xfd, func_code, &flock);
    if (r != -1)
        return(0);
    else
        return(-1);
}

int locked(b)
int b;
/* Test to see if byte b is locked. Return L or U */
{
    struct flock flock;
    pid_t pid;
    int status;

    flock.l_type = F_WRLCK;
    flock.l_whence = whence;
    flock.l_start = (long) b;
    flock.l_len = 1;

    /* Process' own locks are invisible to F_GETLK, so fork a child to test. */
    pid = fork();
    if (pid == 0) {
        if (fcntl(xfd, F_GETLK, &flock) != 0) e(2000);
        exit(flock.l_type == F_UNLCK ? U : L);
    }
    if (pid == -1) e(2001);
    if (fcntl(xfd, F_GETLK, &flock) != 0) e(2002);
    if (flock.l_type != F_UNLCK) e(2003);
    if (wait(&status) != pid) e(2004);
    if (!WIFEXITED(status)) e(2005);
    return(WEXITSTATUS(status));
}

void e(n)
int n;
{
    int err_num = errno; /* save errno in case printf clobbers it */

    printf("Subtest %d, error %d errno=%d ", subtest, n, errno);
    fflush(stdout);
}
```



```
    errno = err_num;                /* restore errno, just in case */
    perror("");
    if (errct++ > MAX_ERROR) {
        printf("Too many errors; test aborted\n");
        chdir("..");
        system("rm -rf DIR*");
        exit(1);
    }
}

void sigfunc(s)
int s;                /* for ANSI */
{
}

void quit()
{
    chdir("..");
    system("rm -rf DIR*");

    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
        printf("%d errors\n", errct);
        exit(1);
    }
}
```

```
/* test8: pipe() Author: Jan-Mark Wams (jms@cs.vu.nl) */

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <errno.h>
#include <time.h>
#include <stdio.h>

#define MAX_ERROR 4
#define ITERATIONS 60

#define Fstat(a,b) if (fstat(a,b) != 0) printf("Can't fstat %d\n", a)
#define Time(t) if (time(t) == (time_t)-1) printf("Time error\n")

int errct = 0;
int subtest = 1;
char MaxName[NAME_MAX + 1]; /* Name of maximum length */
char MaxPath[PATH_MAX]; /* Same for path */
char ToLongName[NAME_MAX + 2]; /* Name of maximum +1 length */
char ToLongPath[PATH_MAX + 1]; /* Same for path, both too long */

_PROTOTYPE(void main, (int argc, char *argv[]));
_PROTOTYPE(void test8a, (void));
_PROTOTYPE(void test8b, (void));
_PROTOTYPE(void e, (int number));
_PROTOTYPE(void quit, (void));

void main(argc, argv)
int argc;
char *argv[];
{
    int i, m = 0xFFFF;

    sync();
    if (argc == 2) m = atoi(argv[1]);
    printf("Test 8 ");
    fflush(stdout);

    for (i = 0; i < ITERATIONS; i++) {
        if (m & 0001) test8a();
        if (m & 0002) test8b();
    }
    quit();
}

void test8a()
{
    /* Test fcntl flags. */
    int tube[2], t1[2], t2[2], t3[2];
    time_t time1, time2;
    char buf[128];
    struct stat st1, st2;
    int stat_loc, flags;

    subtest = 1;

    /* Check if lowest fds are returned. */
    if (pipe(tube) != 0) e(1);
    if (tube[0] != 3 && tube[1] != 3) e(2);
    if (tube[1] != 4 && tube[0] != 4) e(3);
    if (tube[1] == tube[0]) e(4);
    if (pipe(t1) != 0) e(5);
    if (t1[0] != 5 && t1[1] != 5) e(6);
    if (t1[1] != 6 && t1[0] != 6) e(7);
    if (t1[1] == t1[0]) e(8);
    if (close(t1[0]) != 0) e(9);
    if (close(tube[0]) != 0) e(10);
    if (pipe(t2) != 0) e(11);
    if (t2[0] != tube[0] && t2[1] != tube[0]) e(12);
```

```
if (t2[1] != t1[0] && t2[0] != t1[0]) e(13);
if (t2[1] == t2[0]) e(14);
if (pipe(t3) != 0) e(15);
if (t3[0] != 7 && t3[1] != 7) e(16);
if (t3[1] != 8 && t3[0] != 8) e(17);
if (t3[1] == t3[0]) e(18);
if (close(tube[1]) != 0) e(19);
if (close(t1[1]) != 0) e(20);
if (close(t2[0]) != 0) e(21);
if (close(t2[1]) != 0) e(22);
if (close(t3[0]) != 0) e(23);
if (close(t3[1]) != 0) e(24);

/* All time fields should be marked for update. */
Time(&time1);
if (pipe(tube) != 0) e(25);
Fstat(tube[0], &st1);
Fstat(tube[1], &st2);
Time(&time2);
if (st1.st_atime < time1) e(26);
if (st1.st_ctime < time1) e(27);
if (st1.st_mtime < time1) e(28);
if (st1.st_atime > time2) e(29);
if (st1.st_ctime > time2) e(30);
if (st1.st_mtime > time2) e(31);
if (st2.st_atime < time1) e(32);
if (st2.st_ctime < time1) e(33);
if (st2.st_mtime < time1) e(34);
if (st2.st_atime > time2) e(35);
if (st2.st_ctime > time2) e(36);
if (st2.st_mtime > time2) e(37);

/* Check the file characteristics. */
if ((flags = fcntl(tube[0], F_GETFD)) != 0) e(38);
if ((flags & FD_CLOEXEC) != 0) e(39);
if ((flags = fcntl(tube[0], F_GETFL)) != 0) e(40);
if ((flags & O_RDONLY) != O_RDONLY) e(41);
if ((flags & O_NONBLOCK) != 0) e(42);
if ((flags & O_RDWR) != 0) e(43);
if ((flags & O_WRONLY) != 0) e(44);

if ((flags = fcntl(tube[1], F_GETFD)) != 0) e(45);
if ((flags & FD_CLOEXEC) != 0) e(46);
if ((flags = fcntl(tube[1], F_GETFL)) == -1) e(47);
if ((flags & O_WRONLY) != O_WRONLY) e(48);
if ((flags & O_NONBLOCK) != 0) e(49);
if ((flags & O_RDWR) != 0) e(50);
if ((flags & O_RDONLY) != 0) e(51);

/* Check if we can read and write. */
switch (fork()) {
    case -1: printf("Can't fork\n"); break;
    case 0:
        alarm(20);
        if (close(tube[0]) != 0) e(52);
        if (write(tube[1], "Hello", 6) != 6) e(53);
        if (close(tube[1]) != 0) e(54);
        exit(0);
    default:
        if (read(tube[0], buf, sizeof(buf)) != 6) e(55);
        if (strncmp(buf, "Hello", 6) != 0) e(56);
        wait(&stat_loc);
        if (stat_loc != 0) e(57); /* Alarm? */
}
if (close(tube[0]) != 0) e(58);
if (close(tube[1]) != 0) e(59);
}

void test8b()
{
    int tube[2], child2parent[2], parent2child[2];
    int i, nchild = 0, nopen = 3, stat_loc;
    int fd;
    int forkfailed = 0;
```

```
char c;

subtest = 2;

/* Take all the pipes we can get. */
while (nopen < OPEN_MAX - 2) {
    if (pipe(tube) != 0) {
        /* We have not reached OPEN_MAX yet, so we have ENFILE. */
        if (errno != ENFILE) e(1);
        sleep(2);          /* Wait for others to (maybe) closefiles. */
        break;
    }
    nopen += 2;
}

if (nopen < OPEN_MAX - 2) {
    if (pipe(tube) != -1) e(2);
    switch (errno) {
        case EMFILE:          /* Errno value is ok. */
            break;
        case ENFILE:          /* No process can open files any more. */
            switch (fork()) {
                case -1:
                    printf("Can't fork\n");
                    break;
                case 0:
                    alarm(20);
                    if (open("/", O_RDONLY) != -1) e(3);
                    if (errno != ENFILE) e(4);
                    exit(0);
                default:
                    wait(&stat_loc);
                    if (stat_loc != 0) e(5);          /* Alarm? */
            }
            break;
        default:               /* Wrong value for errno. */
            e(6);
    }
}

/* Close all but stdin,out,err. */
for (i = 3; i < OPEN_MAX; i++) (void) close(i);

/* ENFILE test. Have children each grab OPEN_MAX fds. */
if (pipe(child2parent) != 0) e(7);
if (pipe(parent2child) != 0) e(8);
while (!forkfailed && (fd = open("/", O_RDONLY)) != -1) {
    close(fd);
    switch (fork()) {
        case -1:
            forkfailed = 1;
            break;
        case 0:
            alarm(60);

            /* Grab all the fds. */
            while (pipe(tube) != -1);
            while (open("/", O_RDONLY) != -1);

            /* Signal parent OPEN_MAX fds gone. */
            if (write(child2parent[1], "*", 1) != 1) e(9);

            /* Wait for parent befor freeing all the fds. */
            if (read(parent2child[0], &c, 1) != 1) e(10);
            exit(0);
        default:
            /* Wait for child to grab OPEN_MAX fds. */
            if (read(child2parent[0], &c, 1) != 1) e(11);
            nchild++;
            break;
    }
}
```

```
if (!forkfailed) {
    if (pipe(tube) != -1) e(12);
    if (errno != ENFILE) e(13);
}

/* Signal children to die and wait for it. */
while (nchild-- > 0) {
    if (write(parent2child[1], "*", 1) != 1) e(14);
    wait(&stat_loc);
    if (stat_loc != 0) e(15);      /* Alarm? */
}

/* Close all but stdin,out,err. */
for (i = 3; i < OPEN_MAX; i++) (void) close(i);
}

void e(n)
int n;
{
    int err_num = errno;          /* Save in case printf clobbers it. */

    printf("Subtest %d, error %d errno=%d: ", subtest, n, errno);
    errno = err_num;
    perror("");
    if (errct++ > MAX_ERROR) {
        printf("Too many errors; test aborted\n");
        chdir("..");
        system("rm -rf DIR*");
        exit(1);
    }
    errno = 0;
}

void quit()
{
    if (errct == 0) {
        printf("ok\n");
        exit(0);
    } else {
        printf("%d errors\n", errct);
        exit(1);
    }
}
```

```

/* Test 9 setjmp with register variables.          Author: Cerial Jacobs */

#include <sys/types.h>
#include <setjmp.h>
#include <signal.h>

#define MAX_ERROR 4

#include "common.c"

char *tmpa;

_PROTOTYPE(int main, (int argc, char *argv []));
_PROTOTYPE(void test9a, (void));
_PROTOTYPE(void test9b, (void));
_PROTOTYPE(void test9c, (void));
_PROTOTYPE(void test9d, (void));
_PROTOTYPE(void test9e, (void));
_PROTOTYPE(void test9f, (void));
_PROTOTYPE(char *addr, (void));
_PROTOTYPE(void garbage, (void));
_PROTOTYPE(void levell, (void));
_PROTOTYPE(void level2, (void));
_PROTOTYPE(void dolev, (void));
_PROTOTYPE(void catch, (int s));
_PROTOTYPE(void hard, (void));

int main(argc, argv)
int argc;
char *argv[];
{
    jmp_buf envm;
    int i, j, m = 0xFFFF;

    start(9);
    if (argc == 2) m = atoi(argv[1]);
    for (j = 0; j < 100; j++) {
        if (m & 00001) test9a();
        if (m & 00002) test9b();
        if (m & 00004) test9c();
        if (m & 00010) test9d();
        if (m & 00020) test9e();
        if (m & 00040) test9f();
    }
    if (errct) quit();
    i = 1;
    if (setjmp(envm) == 0) {
        i = 2;
        longjmp(envm, 1);
    } else {
        if (i == 2) {
            /* Correct */
        } else if (i == 1) {
            printf("WARNING: The setjmp/longjmp of this machine restore register variables\n\nto the value they had at the time of the Setjmp\n");
        } else {
            printf("Aha, I just found one last error\n");
            return 1;
        }
    }
    quit();
    return(-1);
}

void test9a()
{
    register p;

    subtest = 1;
    p = 200;
    garbage();
    if (p != 200) e(1);
}

```

```
void test9b()
{
    register p, q;

    subtest = 2;
    p = 200;
    q = 300;
    garbage();
    if (p != 200) e(1);
    if (q != 300) e(2);
}

void test9c()
{
    register p, q, r;

    subtest = 3;
    p = 200;
    q = 300;
    r = 400;
    garbage();
    if (p != 200) e(1);
    if (q != 300) e(2);
    if (r != 400) e(3);
}

char buf[512];

void test9d()
{
    register char *p;

    subtest = 4;
    p = &buf[100];
    garbage();
    if (p != &buf[100]) e(1);
}

void test9e()
{
    register char *p, *q;

    subtest = 5;
    p = &buf[100];
    q = &buf[200];
    garbage();
    if (p != &buf[100]) e(1);
    if (q != &buf[200]) e(2);
}

void test9f()
{
    register char *p, *q, *r;

    subtest = 6;
    p = &buf[100];
    q = &buf[200];
    r = &buf[300];
    garbage();
    if (p != &buf[100]) e(1);
    if (q != &buf[200]) e(2);
    if (r != &buf[300]) e(3);
}

jmp_buf env;

/*      return address of local variable.
   This way we can check that the stack is not polluted.
*/
char *
addr()
{
    char a;
```

```
    return &a;
}

void garbage()
{
    register i, j, k;
    register char *p, *q, *r;
    char *a;

    p = &buf[300];
    q = &buf[400];
    r = &buf[500];
    i = 10;
    j = 20;
    k = 30;
    switch (setjmp(env)) {
        case 0:
            a = addr();
#ifdef __GNUC__
            /*
             * to defeat the smartness of the GNU C optimizer we pretend we
             * use 'a'. Otherwise the optimizer will not detect the looping
             * effectuated by setjmp/longjmp, so that it thinks it can get
             * rid of the assignment to 'a'.
             */
            srand((unsigned)&a);
#endif
            longjmp(env, 1);
            break;
        case 1:
            if (i != 10) e(11);
            if (j != 20) e(12);
            if (k != 30) e(13);
            if (p != &buf[300]) e(14);
            if (q != &buf[400]) e(15);
            if (r != &buf[500]) e(16);
            tmpa = addr();
            if (a != tmpa) e(17);
            level1();
            break;
        case 2:
            if (i != 10) e(21);
            if (j != 20) e(22);
            if (k != 30) e(23);
            if (p != &buf[300]) e(24);
            if (q != &buf[400]) e(25);
            if (r != &buf[500]) e(26);
            tmpa = addr();
            if (a != tmpa) e(27);
            level2();
            break;
        case 3:
            if (i != 10) e(31);
            if (j != 20) e(32);
            if (k != 30) e(33);
            if (p != &buf[300]) e(34);
            if (q != &buf[400]) e(35);
            if (r != &buf[500]) e(36);
            tmpa = addr();
            if (a != tmpa) e(37);
            hard();
        case 4:
            if (i != 10) e(41);
            if (j != 20) e(42);
            if (k != 30) e(43);
            if (p != &buf[300]) e(44);
            if (q != &buf[400]) e(45);
            if (r != &buf[500]) e(46);
            tmpa = addr();
            if (a != tmpa) e(47);
            return;
            break;
        default:
            e(100);
    }
}
```



```
e(200);
}

void level1()
{
    register char *p;
    register i;

    i = 1000;
    p = &buf[10];
    i = 200;
    p = &buf[20];
    longjmp(env, 2);
}

void level2()
{
    register char *p;
    register i;

    i = 0200;
    p = &buf[2];
    *p = i;
    dolev();
}

void dolev()
{
    register char *p;
    register i;

    i = 010;
    p = &buf[3];
    *p = i;
    longjmp(env, 3);
}

void catch(s)
int s;
{
    longjmp(env, 4);
}

void hard()
{
    register char *p;

    signal(SIGHUP, catch);
    for (p = buf; p <= &buf[511]; p++) *p = 025;
    kill(getpid(), SIGHUP);
}
```

```
#!/bin/sh

# Shell script used to test MINIX.

PATH=:/bin:/usr/bin
export PATH

echo -n "Shell test 1 "
rm -rf DIR_SH1
mkdir DIR_SH1
cd DIR_SH1

f=../test1.c
if test -r $f; then : ; else echo sh1 cannot read $f; exit 1; fi

#Initial setup
echo "abcdefghijklmnopqrstuvwxyz" >alpha
echo "ABCDEFGHIJKLMNOPQRSTUVWXYZ" >ALPHA
echo "0123456789" >num
echo "!@#%$^&*()_+=-{}[];<>?/., " >special
cp /etc/rc rc
cp /etc/passwd passwd
cat alpha ALPHA num rc passwd special >tmp
cat tmp tmp tmp >f1

#Test cp
mkdir foo
cp /etc/rc /etc/passwd foo
if cmp -s foo/rc /etc/rc ; then : ; else echo Error on cp test 1; fi
if cmp -s foo/passwd /etc/passwd ; then : ; else echo Error on cp test 2; fi
rm -rf foo

#Test cat
cat num num num num num >y
wc -c y >x1
echo " 55y" >x2
if cmp -s x1 x2; then : ; else echo Error on cat test 1; fi
cat <y >z
if cmp -s y z; then : ; else echo Error on cat test 2; fi

#Test ar
cat passwd >p
cp passwd q
if cmp -s p q; then : ; else echo Error on ar test 1; fi
date >r
ar r x.a p q r 2>/dev/null
ar r x.a /usr/bin/cp
ar r x.a /usr/bin/cat
rm p q
mv r R
ar x x.a
if cmp -s p /etc/passwd; then : ; else Error on ar test 2; fi
if cmp -s q /etc/passwd; then : ; else Error on ar test 3; fi
if cmp -s r R; then : ; else Error on ar test 4; fi
if cmp -s cp /usr/bin/cp; then : ; else Error on ar test 5; fi
if cmp -s cat /usr/bin/cat; then : ; else Error on ar test 6; fi
rm cp cat p q r
ar d x.a r >/dev/null
ar x x.a
if test -r r; then echo Error on ar test 7; fi
rm -rf p q r R

#Test basename
if test `basename /usr/ast/foo.c .c` != 'foo'
then echo Error on basename test 1
fi

if test `basename a/b/c/d` != 'd'; then Error on basename test 2; fi

#Test cdiff, sed, and patch
cp $f x.c # x.c is a copy $f
echo "/a/s/#####g" >s # create sed script
sed -f s <x.c >y.c # y.c is new version of x.c
```

```
cdiff x.c y.c >y                # y is cdiff listing
patch x.c y 2>/dev/null         # z should be y.c
if cmp -s x.c y.c; then : ; else echo Error in cdiff test; fi
rm x.c* y.c s y

#Test comm, grep -v
ls /etc >x                      # x = list of /etc
grep -v "passwd" x >y           # y = x except for /etc/passwd
comm -3 x y >z                  # should only be 1 line, /etc/passwd
echo "passwd" >w
if cmp -s w z; then : else echo Error on comm test 1; fi

comm -13 x y >z                  # should be empty
if test -s z; then echo Error on comm test 2; fi
rm -rf w x y z

#Test compress
compress -fc $f >x.c.Z          # compress the test file
compress -cd x.c.Z >y           # uncompress it
if cmp -s $f y; then : else echo Error in compress test 1; fi
rm -rf x.c.Z y

#Test ed
cp $f x                          # copy $f to x
cat >y <<END
g/a/s//#####/g
g/b/s//@@@@@/g
g/c/s//!!!!!!/g
w
q
END
ed x <y >/dev/null
cat >y <<END
g/#####/s//a/g
g/@@@@@/s//b/g
g/!!!!!!/s//c/g
w
q
END
ed x <y >/dev/null
if cmp -s x $f; then : ; else echo Error in ed test 1; fi
rm x y

#Test expr
if test `expr 1 + 1` != 2; then echo Error on expr test 1; fi
if test `expr 10000 - 1` != 9999; then echo Error on expr test 2; fi
if test `expr 100 '*' 50` != 5000; then echo Error on expr test 3; fi
if test `expr 120 / 5` != 24; then echo Error on expr test 4; fi
if test `expr 143 % 7` != 3; then echo Error on expr test 5; fi
a=100
a=`expr $a + 1`
if test $a != `101`; then echo Error on expr test 6; fi

#Test fgrep
fgrep "abc" alpha >z
if cmp -s alpha z ; then : else echo Error on fgrep test 1; fi
fgrep "abc" num >z
if test -s z; then echo Error on fgrep test 2; fi
cat alpha num >z
fgrep "012" z >w
if cmp -s w num; then : ; else echo Error fgrep test 3; fi

#Test find
date >Rabbit
echo "Rabbit" >y
find . -name Rabbit -print >z
if cmp -s y z; then : else echo Error on find test 1; fi
find . -name Bunny -print >z
if test -s z; then echo Error on find test 2; fi
rm Rabbit y z

#Test grep
grep "a" alpha >x
```

```
if cmp -s x alpha; then : ; else echo Error on grep test 1; fi
grep "a" ALPHA >x
if test -s x; then echo Error on grep test 2; fi
grep -v "0" alpha >x
if cmp -s x alpha; then : ; else echo Error on grep test 3; fi
grep -s "a" alpha >x
if test -s x; then echo Error on grep test 4; fi
if grep -s "a" alpha >x; then : else echo Error on grep test 5; fi
if grep -s "a" ALPHA >x; then echo Error on grep test 6; fi

#Test head
head -1 f1 >x
if cmp -s x alpha; then : else echo Error on head test 1; fi
head -2 f1 >x
cat alpha ALPHA >y
if cmp -s x y; then : else echo Error on head test 2; fi

#Test ls
mkdir FOO
cp passwd FOO/z
cp alpha FOO/x
cp ALPHA FOO/y
cd FOO
ls >w
cat >w1 <<END
w
x
y
z
END
if cmp -s w w1; then : ; else echo Error on ls test 1; fi
rm *
cd ..
rmdir FOO

#Test mkdir/rmdir
mkdir Foo Bar
if test -d Foo; then : ; else echo Error on mkdir test 1; fi
if test -d Bar; then : ; else echo Error on mkdir test 2; fi
rmdir Foo Bar
if test -d Foo; then echo Error on mkdir test 3; fi
if test -d Foo; then echo Error on rmdir test 4; fi

#Test mv
mkdir MVDIR
cp $f x
mv x y
mv y z
if cmp -s $f z; then : ; else echo Error on mv test 1; fi
cp $f x
mv x MVDIR/y
if cmp -s $f MVDIR/y; then : ; else echo Error on mv test 2; fi

#Test rev
rev <f1 | head -1 >ahpla
echo "zyxwvutsrqponmlkjihgfedcba" >x
if cmp -s x ahpla; then : ; else echo Error on rev test 1; fi
rev <$f >x
rev <x >y
if cmp -s $f x; then echo Error on rev test 2; fi
if cmp -s $f y; then : ; else echo error on rev test 3; fi

#Test shar
cp $f w
cp alpha x
cp ALPHA y
cp num z
shar w x y z >x1
rm w x y z
sh <x1 >/dev/null
if cmp -s w $f; then : ; else echo Error on shar test 1; fi
if cmp -s x alpha; then : ; else echo Error on shar test 2; fi
if cmp -s y ALPHA; then : ; else echo Error on shar test 3; fi
if cmp -s z num; then : ; else echo Error on shar test 4; fi
```

```
#Test sort
sort <$f >x
wc <$f >x1
wc <x >x2
if cmp -s x1 x2; then : ; else echo Error on sort test 1; fi
cat >x <<END
demit 10
epitonic 40
apoop 20
bibelot 3
comate 4
END
cat >y <<END
apoop 20
bibelot 3
comate 4
demit 10
epitonic 40
END
cat >z <<END
epitonic 40
demit 10
comate 4
bibelot 3
apoop 20
END
sort <x >x1
if cmp -s y x1; then : ; else echo Error on sort test 2; fi
sort -r <x1 >x2
if cmp -s x2 z; then : ; else echo Error on sort test 3; fi
sort +1 -n <x |head -1 >y
echo "bibelot3" >z
if cmp -s y z; then : ; else echo Error on sort test 4; fi

#Test tail
tail -1 f1 >x
if cmp -s x special; then : ; else echo Error on tail test 1; fi

#Test tsort
cat >x <<END
a d
b e
c f
a c
p z
k p
a k
a b
b c
c d
d e
e f
f k
END
cat >answer <<END
a
b
c
d
e
f
k
p
z
END
tsort <x >y
if cmp -s y answer; then : ; else echo Error on tsort test 1; fi

#Test uue/uud
cp $f x
uue x
if test -s x.uue; then : ; else echo Error on uue/uud test 1; fi
rm x
```

```
uud x.uue
if cmp -s x $f; then : ; else echo Error on uue/uud test 2; fi

compress -fc x >x.Z 2>/dev/null
uue x.Z
rm x x.Z
uud x.uue
compress -cd x.Z >x
if cmp -s x $f; then : ; else echo Error on uue/uud test 3; fi

cd ..
rm -rf DIR_SH1

echo ok
```

```
#!/bin/sh

# Shell script #2 used to test MINIX.

PATH=:/bin:/usr/bin
export PATH

# CC="exec cc -wo -F"          # nonstandard flags for ACK :-(
CC=cc

ARCH='arch'

echo -n "Shell test 2 "
rm -rf DIR_SH2
mkdir DIR_SH2          # all files are created here
cd DIR_SH2

cat >file <<END
The time has come the walrus said to talk of many things
Of shoes and ships and sealing wax of cabbages and kings
Of why the sea is boiling hot and whether pigs have wings
END
f=file          # scratch file

cat >makefile <<END          # create a makefile
all:      x.c
          @$CC x.c >/dev/null 2>&1
END
cat >x.c <<END          # create a C program
#include <stdio.h>
char s[] = {"MS-DOS: Just say no"};          /* used by strings later */
main()
{
    int i;
    for (i = 15; i < 18; i++) printf("%d\\n",i*i);
}
END

cat >answer <<END          # C program should produce these results
225
256
289
END

make
if test -f a.out; then : ; else echo Compilation failed; fi
a.out >x
if test -f x; then : ; else echo No compiler output; fi
if cmp -s x answer; then : ; else echo Error in cc test 1; fi

#Test chmod
echo Hi there folks >x
if test -r x; then : ; else echo Error on chmod test 1; fi
chmod 377 x
if test -r x; then test -w / || echo Error on chmod test 2; fi
chmod 700 x
if test -r x; then : ; else echo Error on chmod test 3; fi

#Test cut
cat >x <<END          # x is a test file with 3 columns
1 white bunny
2 gray rabbits
3 brown hares
4 black conies
END

cat >answer <<END          # after cutting out cols 3-7, we get this
white
gray
brown
black
END

cut -c 3-7 x >y          # extract columns 3-7
```

```
if cmp -s y answer; then : ; else echo Error in cut test 1; fi

#Test dd
dd if=$f of=x bs=12 count=1 2>/dev/null          # x = bytes 0-11
dd if=$f of=y bs=6 count=4 skip=2 2>/dev/null    # y = bytes 11-35
cat x y >z                                         # z = bytes 0-35
dd if=$f of=answer bs=9 count=4 2>/dev/null      # answer = bytes 0-35
if cmp -s z answer; then : ; else echo Error in dd test 1; fi

#Test df                                           # hard to make a sensible Test here
rm ?
df >x
if test -r x; then : ; else echo Error in df Test 1; fi

#Test du                                           # see df
rm ?
du >x
if test -r x; then : ; else echo Error in du Test 1; fi

#Test od
head -1 $f |od >x                                # see if od converts ascii to octal ok
if [ $ARCH = i86 -o $ARCH = i386 ]
then
cat >answer <<END
0000000 064124 020145 064564 062555 064040 071541 061440 066557
0000020 020145 064164 020145 060567 071154 071565 071440 064541
0000040 020144 067564 072040 066141 020153 063157 066440 067141
0000060 020171 064164 067151 071547 000012
0000071
END
else
cat >answer <<END
0000000 052150 062440 072151 066545 020150 060563 020143 067555
0000020 062440 072150 062440 073541 066162 072563 020163 060551
0000040 062040 072157 020164 060554 065440 067546 020155 060556
0000060 074440 072150 064556 063563 005000
0000071
END
fi

if cmp -s x answer; then : ; else echo Error in od test 1; fi

head -1 $f |od -d >x                             # see if od converts ascii to decimal ok
if [ $ARCH = i86 -o $ARCH = i386 ]
then
cat >answer <<END
0000000 26708 08293 26996 25965 26656 29537 25376 28015
0000020 08293 26740 08293 24951 29292 29557 29472 26977
0000040 08292 28532 29728 27745 08299 26223 27936 28257
0000060 08313 26740 28265 29543 00010
0000071
END
else
cat >answer <<END
0000000 21608 25888 29801 28005 08296 24947 08291 28525
0000020 25888 29800 25888 30561 27762 30067 08307 24937
0000040 25632 29807 08308 24940 27424 28518 08301 24942
0000060 31008 29800 26990 26483 02560
0000071
END
fi

if cmp -s x answer; then : ; else echo Error in od test 2; fi

#Test paste
cat >x <<END
red
green
blue
blue
END

cat >y <<END
rood
groen
```



```
blauw
END
cat >answer <<END
red      rood
green    groen
blue     blauw
END

paste x y >z
if cmp -s z answer; then : ; else echo Error in paste test 1; fi

#Test prep
echo >x <<END
"Hi," said Carol, laughing, "How's life?"
END

echo >answer <<END
hi
said
carol
laughing
how's
life
END

if cmp -s x answer; then : ; else echo Error in prep test 1; fi

#Test printenv
printenv >x
if grep HOME x >/dev/null; then : ; else echo Error in printenv test 1; fi
if grep PATH x >/dev/null; then : ; else echo Error in printenv test 2; fi
if grep SHELL x >/dev/null; then : ; else echo Error in printenv test 3; fi
if grep USER x >/dev/null; then : ; else echo Error in printenv test 4; fi

#Test pwd
pwd >Pwd_file
cd 'pwd'
pwd >x
if test -s Pwd_file; then : ; else echo Error in pwd test 1; fi
if cmp -s Pwd_file x; then : ; else echo Error in pwd test 2; fi

#Test strings
strings a.out | grep "MS-DOS" >x
cat >answer <<END
MS-DOS: Just say no
END

if cmp -s x answer; then : ; else echo Error in strings test 1; fi

#Test sum
sum $f >x
cat >answer <<END
29904 1
END

if cmp -s x answer; then : ; else echo Error in sum test 1; fi

#Test tee
cat $f | tee x >/dev/null
if cmp -s x $f; then : ; else echo Error in tee test 1; fi

#Test true
if true ; then : ; else echo Error in true test 1; fi

#Test uniq
cat >x <<END
100
200
200
300
END

cat >answer <<END
100
```

```
200
300
END
```

```
uniq <x >y
if cmp -s y answer; then : ; else echo Error in uniq test 1; fi
```

```
#Test pipelines
cat >x <<END
the big black dog
the little white cat
the big white sheep
the little black cat
END
```

```
cat >answer <<END
1 dog
1 sheep
2 big
2 black
2 cat
2 little
2 white
4 the
END
```

```
prep x | sort | uniq -c >y1
sort +1 <y1 >y
if cmp -s y answer; then : ; else echo Error in pipeline test 1; fi
```

```
cat $f $f $f | sort | uniq >x
sort <$f >y
if cmp -s x y; then : ; else echo Error in pipeline test 2; fi
```

```
cd ..
rm -rf DIR_SH2
```

```
echo ok
```

```
# Makefile for the tests
```

```
CC = exec cc
```

```
CFLAGS = -Wall -D_MINIX -D_POSIX_SOURCE
```

```
PROG = speed test00 test01 test02 test03 test04_srv test04_cli test05_srv \  
      test05_cli test06_srv test06_cli test07_srv test07_cli test08_srv \  
      test08_cli test09 test10 test11 test12 test13a test13b test14
```

```
all: $(PROG)
```

```
$(PROG):
```

```
    $(CC) $(CFLAGS) -o $@ $@.c -lutil
```

```
clean:
```

```
    /usr/bin/rm -f *.o $(PROG)
```

```
speed: speed.c
```

```
test00: test00.c
```

```
test01: test01.c
```

```
test02: test02.c
```

```
test03: test03.c
```

```
test04_cli: test04_cli.c
```

```
test04_srv: test04_srv.c
```

```
test05_cli: test05_cli.c
```

```
test05_srv: test05_srv.c
```

```
test06_cli: test06_cli.c
```

```
test06_srv: test06_srv.c
```

```
test07_cli: test07_cli.c
```

```
test07_srv: test07_srv.c
```

```
test08_cli: test08_cli.c
```

```
test08_srv: test08_srv.c
```

```
test09: test09.c
```

```
test10: test10.c
```

```
test11: test11.c
```

```
test12: test12.c
```

```
test13a: test13a.c
```

```
test13b: test13b.c
```

```
test14: test14.c
```

```
/*
 * Test name: speed.c
 *
 * Objective: Test the time it takes for select to run.
 *
 * Description: This tests creates a number of udp connections and performs
 * a select call waiting on them for reading with timeout of 0.
 * This is done 10,000 thousands of times and then the average time it takes
 * is computed
 *
 * Jose M. Gomez
 */

#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/select.h>
#include <sys/asynchio.h>
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <net/netlib.h>
#include <time.h>

#define NUMBER 12

void main(void) {
    char *udp_device;
    int fd[NUMBER];
    fd_set fds_write;
    struct timeval timeout;
    time_t start_time, end_time;
    int i;

    FD_ZERO(&fds_write);
    for (i = 0; i < NUMBER; i++) {
        fd[i] = open("/dev/tty", O_RDWR);
        if (fd[i] < 0) {
            fprintf(stderr, "Error opening tty %d\n", i);
            exit(-1);
        }
        FD_SET(fd[i], &fds_write);
    }

    printf("Select will send 1 msg to terminal and %d to inet: \n", NUMBER);
    timeout.tv_sec = 0;
    timeout.tv_usec = 0;
    /* get initial time */
    start_time = time(NULL);
    for (i = 0; i < 32000; i++) {
        select(NUMBER + 4, NULL, &fds_write, NULL, &timeout);
    }
    /* get final time */
    end_time = time(NULL);
    printf("The select call took on average: %f\n", (float)(end_time - start_time) / 32000.0);
    for (i = 0; i < NUMBER; i++) {
        close(fd[i]);
    }
}
```

```
/*
 * Test name: test00.c
 *
 * Objective: The purpose of this test is to make sure that the bitmap
 * manipulation macros work without problems.
 *
 * Description: This tests first fills a fd_set bit by bit, and shows it, then
 * it clears the fd_set bit by bit as well.
 *
 * Jose M. Gomez
 */

#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/select.h>
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

int main(int argc, char *argv[]) {
    fd_set fds;
    int i,j;

    FD_ZERO(&fds);
    for (i=0;i<FD_SETSIZE;i++) {
        /* see if SET works */
        FD_SET(i, &fds);
        if(!FD_ISSET(i, &fds))
            return 1;
    }
    FD_ZERO(&fds);
    for (i=0;i<FD_SETSIZE;i++) {
        /* see if ZERO works */
        if(FD_ISSET(i, &fds))
            return 1;
    }
    for (i=0;i<FD_SETSIZE;i++) {
        FD_SET(i, &fds);
        for(j = 0; j <= i; j++)
            if(!FD_ISSET(j, &fds))
                return 1;
        for(; j < FD_SETSIZE; j++)
            if(FD_ISSET(j, &fds))
                return 1;
    }
    for (i=0;i<FD_SETSIZE;i++) {
        FD_CLR(i, &fds);
        for(j = 0; j <= i; j++)
            if(FD_ISSET(j, &fds))
                return 1;
        for(; j < FD_SETSIZE; j++)
            if(!FD_ISSET(j, &fds))
                return 1;
    }
    printf("ok\n");
    return 0;
}
```

```
/*
 * Test name: test01.c
 *
 * Objective: The purpose of this test is to make sure that the timeout mechanisms
 * work without errors.
 *
 * Description: Executes a select as if it was a sleep and compares the time it
 * has been actually sleeping against the specified time in the select call.
 * Three cases are tested: first, a timeout specified in seconds, second, a timeout in
 * microseconds, and third, a timeout with more precision than seconds.
 *
 * Jose M. Gomez
 */

#include <sys/types.h>
#include <sys/select.h>
#include <sys/time.h>
#include <time.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>

#define SECONDS 3
#define USECONDS 3000000L

int main(void) {
    int r;
    time_t start, end;      /* variables for timing */
    struct timeval timeout; /* timeout structure */

    /* Set timeout for 3 seconds */
    timeout.tv_sec = SECONDS;
    timeout.tv_usec = 0;
    printf("Sleeping now for %d seconds...\n", SECONDS);
    /* Record time before starting */
    start = time(NULL);
    r = select(0, NULL, NULL, NULL, &timeout);
    printf("select return code: %d error: %s\n",
           r, strerror(errno));
    end = time(NULL);
    printf("For a timeout with select of %d seconds, it took %d actual seconds\n",
           SECONDS, end-start);

    /* Set timeout for 3 seconds , but specified in microseconds */

    timeout.tv_sec = 0;
    timeout.tv_usec = USECONDS;
    printf("\n*****\n");
    printf("Sleeping now for %ld microseconds...\n", USECONDS);
    /* Record time before starting */
    start = time(NULL);
    r = select(0, NULL, NULL, NULL, &timeout);
    printf("select return code: %d error: %s\n",
           r, strerror(errno));
    end = time(NULL);
    printf("For a timeout with select of %ld useconds, it took %d actual seconds\n",
           USECONDS, end-start);

    /* Set timeout for 1.5 seconds, but specified in microseconds */

    timeout.tv_sec = 0;
    timeout.tv_usec = USECONDS/2;
    printf("\n*****\n");
    printf("Sleeping now for %ld microseconds...\n", USECONDS/2);
    /* Record time before starting */
    start = time(NULL);
    r = select(0, NULL, NULL, NULL, &timeout);
    printf("select return code: %d error: %s\n",
           r, strerror(errno));
    end = time(NULL);
    printf("For a timeout with select of %ld useconds, it took %d actual seconds\n",
           USECONDS/2, end-start);

    return 0;
}
```

```
/*
 * Test name: test02.c
 *
 * Objective: The purpose of this test is to make sure that select works
 * when working with files.
 *
 * Description: This tests first creates six dummy files with different
 * modes and performs select calls on them evaluating the input and resulting
 * bitmaps.
 *
 * Jose M. Gomez
 */

#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/select.h>
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

void dump_fdset(fd_set *set) {
    int i;
    for (i = 0; i < OPEN_MAX; i++)
        if (FD_ISSET(i, set))
            printf(" %d->1 ", i);
        else
            printf(" %d->0 ", i);
    printf("\n");
}

void main(void) {
    int fd1, fd2, fd3, fd4, fd5, fd6;          /* file descriptors of files */
    fd_set fds_read, fds_write;
    int retval;

    /* Creates the dummy files with different modes */
    fd1 = open("dummy1.txt", O_CREAT | O_RDONLY);
    if (fd1 < 0) {
        perror("Error opening file");
        exit(-1);
    }

    fd2 = open("dummy2.txt", O_CREAT | O_RDONLY);
    if (fd2 < 0) {
        perror("Error opening file");
        exit(-1);
    }

    fd3 = open("dummy3.txt", O_CREAT | O_WRONLY);
    if (fd3 < 0) {
        perror("Error opening file");
        exit(-1);
    }

    fd4 = open("dummy4.txt", O_CREAT | O_WRONLY);
    if (fd4 < 0) {
        perror("Error opening file");
        exit(-1);
    }

    fd5 = open("dummy5.txt", O_CREAT | O_RDWR);
    if (fd5 < 0) {
        perror("Error opening file");
        exit(-1);
    }

    fd6 = open("dummy6.txt", O_CREAT | O_RDWR);
    if (fd6 < 0) {
        perror("Error opening file");
        exit(-1);
    }

    /* Create the fd_set structures */
}
```

```
FD_ZERO(&fds_read);
FD_ZERO(&fds_write);
FD_SET(fd1, &fds_read);          /* fd1 => O_RDONLY */
FD_SET(fd2, &fds_read);          /* fd2 => O_RDONLY */
FD_SET(fd3, &fds_write);         /* fd3 => O_WRONLY */
FD_SET(fd4, &fds_write);         /* fd4 => O_WRONLY */
FD_SET(fd5, &fds_read);          /* fd5 => O_RDWR */
FD_SET(fd5, &fds_write);         /* fd5 => O_RDWR */
FD_SET(fd6, &fds_read);          /* fd6 => O_RDWR */
FD_SET(fd6, &fds_write);         /* fd6 => O_RDWR */

printf(" * Dump INPUT fds_read:\n" );
dump_fdset(&fds_read);
printf(" * Dump INPUT fds_write:\n" );
dump_fdset(&fds_write);

retval=select(9, &fds_read, &fds_write, NULL, NULL);
printf("\n*****\n");
printf("After select: \n");
printf("Return value: %d\n", retval);
printf(" * Dump RESULTING fds_read:\n");
dump_fdset(&fds_read);
printf(" * Dump RESULTING fds_write:\n");
dump_fdset(&fds_write);
/* close and delete dummy files */
close(fd1);
close(fd2);
close(fd3);
close(fd4);
close(fd5);
close(fd6);
unlink("dummy1.txt");
unlink("dummy2.txt");
unlink("dummy3.txt");
unlink("dummy4.txt");
unlink("dummy5.txt");
unlink("dummy6.txt");
```

```
}
```



```
/*
 * Test name: test02.c
 *
 * Objective: The purpose of this test is to make sure that select works
 * when working with files.
 *
 * Description: This test shows more cases than in test02.c, where every
 * descriptor is ready. Here in one select call only half of the fd's will
 * be ready and in the next one none of them will be ready.
 *
 * Jose M. Gomez
 */

#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/select.h>
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <sys/time.h>

void dump_fdset(fd_set *set) {
    int i;
    for (i = 0; i < OPEN_MAX; i++)
        if (FD_ISSET(i, set))
            printf(" %d ", i);
    printf("\n");
}

void main(void) {
    int fd1, fd2, fd3, fd4, fd5, fd6;          /* file descriptors of files */
    fd_set fds_read, fds_write;                /* bit maps */
    struct timeval timeout;                     /* timeout */
    int retval;                                 /* ret value */

    /* Creates the dummy files with different modes */
    fd1 = open("dummy1.txt", O_CREAT | O_RDONLY);
    if (fd1 < 0) {
        perror("Error opening file");
        exit(-1);
    }

    fd2 = open("dummy2.txt", O_CREAT | O_RDONLY);
    if (fd2 < 0) {
        perror("Error opening file");
        exit(-1);
    }

    fd3 = open("dummy3.txt", O_CREAT | O_WRONLY);
    if (fd3 < 0) {
        perror("Error opening file");
        exit(-1);
    }

    fd4 = open("dummy4.txt", O_CREAT | O_WRONLY);
    if (fd4 < 0) {
        perror("Error opening file");
        exit(-1);
    }

    fd5 = open("dummy5.txt", O_CREAT | O_RDWR);
    if (fd5 < 0) {
        perror("Error opening file");
        exit(-1);
    }

    fd6 = open("dummy6.txt", O_CREAT | O_RDWR);
    if (fd6 < 0) {
        perror("Error opening file");
        exit(-1);
    }

    /* Create the fd_set structures */
}
```

```
FD_ZERO(&fds_read);
FD_ZERO(&fds_write);
FD_SET(fd1, &fds_write);          /* fd1 => O_RDONLY */
FD_SET(fd2, &fds_write);          /* fd2 => O_RDONLY */
FD_SET(fd3, &fds_read);           /* fd3 => O_WRONLY */
FD_SET(fd4, &fds_read);           /* fd4 => O_WRONLY */
FD_SET(fd5, &fds_read);           /* fd5 => O_RDWR */
FD_SET(fd5, &fds_write);          /* fd5 => O_RDWR */
FD_SET(fd6, &fds_read);           /* fd6 => O_RDWR */
FD_SET(fd6, &fds_write);          /* fd6 => O_RDWR */

printf(" * Dump INPUT fds_read:\n" );
dump_fdset(&fds_read);
printf(" * Dump INPUT fds_write:\n" );
dump_fdset(&fds_write);

retval=select(9, &fds_read, &fds_write, NULL, NULL);
printf("\n*****\n");
printf("After select: \n");
printf("Return value: %d\n", retval);
printf(" * Dump RESULTING fds_read:\n" );
dump_fdset(&fds_read);
printf(" * Dump RESULTING fds_write:\n" );
dump_fdset(&fds_write);

/* make a select call where none of them are ready (don't use fd5 and fd6) */

FD_ZERO(&fds_read);
FD_ZERO(&fds_write);
FD_SET(fd1, &fds_write);          /* fd1 => O_RDONLY */
FD_SET(fd2, &fds_write);          /* fd2 => O_RDONLY */
FD_SET(fd3, &fds_read);           /* fd3 => O_WRONLY */
FD_SET(fd4, &fds_read);           /* fd4 => O_WRONLY */

/* make a select call where none of them are ready (don't use fd5 and fd6) */
/* create a timeout as well */
timeout.tv_sec = 5;
timeout.tv_usec = 0;
retval=select(7, &fds_read, &fds_write, NULL, NULL);
printf("\n*****\n");
printf("After select: \n");
printf("Return value: %d\n", retval);
printf(" * Dump RESULTING fds_read:\n" );
dump_fdset(&fds_read);
printf(" * Dump RESULTING fds_write:\n" );
dump_fdset(&fds_write);

/* close and delete dummy files */
close(fd1);
close(fd2);
close(fd3);
close(fd4);
close(fd5);
close(fd6);
unlink("dummy1.txt");
unlink("dummy2.txt");
unlink("dummy3.txt");
unlink("dummy4.txt");
unlink("dummy5.txt");
unlink("dummy6.txt");
}
```

```
/*
 * Test name: test04_cli.c
 *
 * Objective: Test a simple UDP client
 *
 * Description: Implements a simple echo client using the UDP protocol. First
 * it waits until it is possible to write (which is always).
 */

#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/select.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdarg.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <net/netlib.h>
#include <net/gen/netdb.h>
#include <net/gen/in.h>
#include <net/gen/udp.h>
#include <net/gen/udp_hdr.h>
#include <net/gen/udp_io.h>
#include <net/hton.h>

#define PORT 6000L

/* Type for received data */
typedef struct
{
    udp_io_hdr_t header;
    char data[1024];
} udp_buffer_t;

int udp_conf(char *host, long port, udp_io_hdr_t *header)
{
    /* configures UDP connection */
    char *udp_device;
    struct hostent *hp;
    int netfd;
    nwio_udpopt_t udpopt;
    ipaddr_t dirhost;
    int result;

    /* get host address */
    if ((hp = gethostbyname(host)) == (struct hostent*) NULL)
    {
        fprintf(stderr, "Unknown host\n");
        return(-1);
    }
    memcpy((char *)&dirhost, (char *)hp->h_addr, hp->h_length);

    /* Get UDP device */
    if ((udp_device = getenv("UDP_DEVICE")) == NULL)
        udp_device = UDP_DEVICE;

    /* Get UDP connection */
    if ((netfd = open(udp_device, O_RDWR)) < 0)
    {
        fprintf(stderr, "Error opening UDP device\n");
        return -1;
    }

    /* Configure UDP connection */
    udpopt.nwuo_flags = NWUO_COPY | NWUO_LP_SEL | NWUO_EN_LOC | NWUO_DI_BROAD
                      | NWUO_RP_SET | NWUO_RA_SET | NWUO_RWDATALL | NWUO_DI_IPOPT;
    udpopt.nwuo_remaddr = dirhost;
    udpopt.nwuo_rempport = (udpport_t) htons(port);

    if ((result = ioctl(netfd, NWIOSUDPOPT, &udpopt)) < 0)
    {
        fprintf(stderr, "Error establishing communication\n");
    }
}
```

```
    printf("Error: %d\n", result);
    close(netfd);
    return -1;
}

/* Get configuration for UDP comm */
if ((result = ioctl(netfd, NWIOGUDPOPT, &udpopt) ) < 0)
{
    fprintf(stderr, "Error getting configuration\n");
    printf("Error: %d\n", result);
    close(netfd);
    return -1;
}

header->uih_src_addr = udpopt.nwuo_locaddr;
header->uih_dst_addr = udpopt.nwuo_remaddr;
header->uih_src_port = udpopt.nwuo_locport;
header->uih_dst_port = udpopt.nwuo_rempport;

return netfd;
}

int main(int argc, char *argv[]) {
    int fd;
    ssize_t data_read;
    udp_buffer_t buffer_send, buffer_rec;
    fd_set fds_write;
    int ret;

    /* Check parameters */
    if (argc != 2) {
        fprintf(stderr, "Usage: %s host\n", argv[0]);
        exit(-1);
    }

    if ((fd = udp_conf(argv[1], PORT, &buffer_send.header) ) < 0)
        exit(-1);

    /* init fd_set */
    FD_ZERO(&fds_write);
    FD_SET(fd, &fds_write);

    while (1)
    {
        /* Wait until it is possible to write with select */

        ret = select(4, NULL, &fds_write, NULL, NULL);
        if (ret < 0) {
            fprintf(stderr, "Error on select waiting for write: %d\n", errno);
            exit(-1);
        }
        if (!FD_ISSET(fd, &fds_write)) {
            fprintf(stderr, "Error: The net connection is not ready for writing (?)\n");
            exit(-1);
        }

        /* Get a string and send it */
        printf("Ready to write...\n");
        printf("Send data: ");
        gets(buffer_send.data);
        write(fd, &buffer_send, sizeof(udp_buffer_t));

        /* If data sent is exit then break */
        if (!strcmp(buffer_send.data, "exit"))
            break;

        /* Get server response */
        data_read = read(fd, &buffer_rec, sizeof(udp_buffer_t));
        printf("Received: %s\n\n", buffer_rec.data);
    }

    /* Close UDP communication */
    close(fd);
}
```

```
/*
 * Test name: test04_srv.c
 *
 * Objective: Test a simple UDP server
 *
 * Description: Implements a simple echo server using the UDP protocol. Instead
 * of blocking on read(), it performs a select call first blocking there
 * until there is data to be read
 */

#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/select.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdarg.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <net/netlib.h>
#include <net/gen/netdb.h>
#include <net/gen/in.h>
#include <net/gen/udp.h>
#include <net/gen/udp_hdr.h>
#include <net/gen/udp_io.h>
#include <net/hton.h>

#define PORT 6000L

/* type for received data */
typedef struct
{
    udp_io_hdr_t header;
    char data[1024];
} udp_buffer_t;

int udp_conf(long port) {

    char *udp_device;
    int netfd;
    nwio_udpopt_t udpopt;

    /* Get default UDP device */
    if ((udp_device = getenv("UDP_DEVICE")) == NULL)
        udp_device = UDP_DEVICE;

    /* Open UDP connection */
    if ((netfd = open(udp_device, O_RDWR)) < 0)
    {
        fprintf(stderr, "Error opening UDP connection\n");
        return -1;
    }

    /* Configure UDP connection */
    udpopt.nwuo_flags = NWUO_COPY | NWUO_LP_SET | NWUO_EN_LOC | NWUO_DI_BROAD
        | NWUO_RP_ANY | NWUO_RA_ANY | NWUO_RWDATALL | NWUO_DI_IPOPT;

    udpopt.nwuo_locport = (udpport_t) htons(port);

    if ((ioctl(netfd, NWIOSUDPOPT, &udpopt)) < 0)
    {
        fprintf(stderr, "Error configuring the connection\n");
        close(netfd);
        return -1;
    }

    /* Get conf options */
    if ((ioctl(netfd, NWIOGUDPOPT, &udpopt)) < 0)
    {
        fprintf(stderr, "Error getting the conf\n");
        close(netfd);
        return -1;
    }
}
```

```
    return netfd;
}

int main(int argc, char *argv[]) {
    int fd;
    ssize_t data_read;
    udp_buffer_t buffer;
    ipaddr_t tmp_addr;
    udpport_t tmp_port;
    int ret;
    fd_set fds_read;

    if ((fd = udp_conf(PORT)) < 0) {
        fprintf(stderr, "Error configuring UDP connection\n");
        exit(-1);
    }
    printf("Waiting for messages on port: %ld\n", PORT);
    fflush(stdout);
    /* Initialize fd_set */
    FD_ZERO(&fds_read);
    FD_SET(fd, &fds_read);

    while (1)
    {
        /* Wait for data available to be read (no timeout) */
        ret = select(4, &fds_read, NULL, NULL, NULL);
        if (ret < 0) {
            fprintf(stderr, "Error on select: %d", errno);
            exit(-1);
        }
        if (!FD_ISSET(fd, &fds_read)) {
            printf("Error: network fd is not ready (?)\n");
            exit(-1);
        }
        printf("Ready to receive...\n");
        /* Read received data */
        data_read = read(fd, &buffer, sizeof(udp_buffer_t));
        printf("Received data: %s\n", buffer.data);

        /* Can exit if the received string == exit */
        if (!strcmp(buffer.data, "exit"))
            break;

        /* Send data back, swap addresses */
        tmp_addr = buffer.header.uih_src_addr;
        buffer.header.uih_src_addr = buffer.header.uih_dst_addr;
        buffer.header.uih_dst_addr = tmp_addr;

        /* Swap ports */
        tmp_port = buffer.header.uih_src_port;
        buffer.header.uih_src_port = buffer.header.uih_dst_port;
        buffer.header.uih_dst_port = tmp_port;

        /* Write the same back */
        write(fd, &buffer, data_read);
    }

    close(fd);
}
```

```
/*
 * Test name: test05_cli.c
 *
 * Objective: Test a impatient UDP client with timeout and incoming data from
 * network and terminal.
 *
 * Description: Implements a echo client using the UDP protocol. It is
 * based on test04_cli, but the difference is that it uses timeout and waits
 * for data both from terminal (stdin) and network connection.
 */

#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/select.h>
#include <sys/asynchio.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdarg.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <net/netlib.h>
#include <net/gen/netdb.h>
#include <net/gen/in.h>
#include <net/gen/udp.h>
#include <net/gen/udp_hdr.h>
#include <net/gen/udp_io.h>
#include <net/hton.h>

#define PORT 6000L

/* Type for received data */
typedef struct
{
    udp_io_hdr_t header;
    char data[1024];
} udp_buffer_t;

int udp_conf(char *host, long port, udp_io_hdr_t *header)
{
    /* configures UDP connection */
    char *udp_device;
    struct hostent *hp;
    int netfd;
    nwio_udpopt_t udpopt;
    ipaddr_t dirhost;
    int result;

    /* get host address */
    if ((hp = gethostbyname(host)) == (struct hostent*) NULL)
    {
        fprintf(stderr, "Unknown host\n");
        return(-1);
    }
    memcpy((char *)&dirhost, (char *)hp->h_addr, hp->h_length);

    /* Get UDP device */
    if ((udp_device = getenv("UDP_DEVICE")) == NULL)
        udp_device = UDP_DEVICE;

    /* Get UDP connection */
    if ((netfd = open(udp_device, O_RDWR)) < 0)
    {
        fprintf(stderr, "Error opening UDP device\n");
        return -1;
    }

    /* Configure UDP connection */
    udpopt.nwuo_flags = NWUO_COPY | NWUO_LP_SEL | NWUO_EN_LOC | NWUO_DI_BROAD
        | NWUO_RP_SET | NWUO_RA_SET | NWUO_RWDATALL | NWUO_DI_IPOPT;
    udpopt.nwuo_remaddr = dirhost;
    udpopt.nwuo_rempport = (udpport_t) htons(port);
}
```

```
if ((result = ioctl(netfd, NWIOSUDPOPT, &udpopt) ) < 0)
{
    fprintf(stderr, "Error establishing communication\n");
    printf("Error: %d\n", result);
    close(netfd);
    return -1;
}

/* Get configuration for UDP comm */
if ((result = ioctl(netfd, NWIOGUDPOPT, &udpopt) ) < 0)
{
    fprintf(stderr, "Error getting configuration\n");
    printf("Error: %d\n", result);
    close(netfd);
    return -1;
}

header->uih_src_addr = udpopt.nwuo_locaddr;
header->uih_dst_addr = udpopt.nwuo_remaddr;
header->uih_src_port = udpopt.nwuo_locport;
header->uih_dst_port = udpopt.nwuo_rempport;

return netfd;
}

int main(int argc, char *argv[]) {
    int fd;
    ssize_t data_read;
    udp_buffer_t buffer_send, buffer_rec;
    fd_set fds_read;
    int ret;
    struct timeval timeout;

    /* Check parameters */
    if (argc != 2) {
        fprintf(stderr, "Usage: %s host\n", argv[0]);
        exit(-1);
    }

    if ((fd = udp_conf(argv[1], PORT, &buffer_send.header) ) < 0)
        exit(-1);

    while (1)
    {

        /* init fd_set */
        FD_ZERO(&fds_read);
        FD_SET(0, &fds_read);
        FD_SET(fd, &fds_read);

        /* set timeval */
        timeout.tv_sec = 5;
        timeout.tv_usec = 0;
        printf("Send data: ");
        fflush(stdout);
        /* Wait until it is possible to write with select */
        ret = select(4, &fds_read, NULL, NULL, &timeout);
        if (ret < 0) {
            fprintf(stderr, "Error on select waiting for read: %d\n", errno);
            exit(-1);
        }
        if (ret == 0) {
            printf("\nClient says: Hey! I want to send data!!\n");
            fflush(stdout);
            continue;
        }
        /* if got message from server */
        if (FD_ISSET(fd, &fds_read)) {
            data_read = read(fd, &buffer_rec, sizeof(udp_buffer_t));
            printf("Server says: %s\n\n", buffer_rec.data);
            fflush(stdout);
        }
        /* if got data from terminal */
        if (FD_ISSET(0, &fds_read)) {
```



```
    /* Get a string and send it */
    gets(buffer_send.data);
    write(fd, &buffer_send, sizeof(udp_buffer_t));

    /* If data sent is exit then break */
    if (!strcmp(buffer_send.data, "exit"))
        break;
    /* Get server response */
    data_read = read(fd, &buffer_rec, sizeof(udp_buffer_t));
    printf("Received: %s\n\n", buffer_rec.data);
    fflush(stdout);
}

/* Close UDP communication */
close(fd);
}
```

```
/*
 * Test name: test05_srv.c
 *
 * Objective: Test an impatient UDP server with timeouts
 *
 * Description: Implements an echo server using the UDP protocol. It is
 * based on test04_srv, but it has a timeout value.
 */

#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/select.h>
#include <sys/asynchio.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdarg.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <net/netlib.h>
#include <net/gen/netdb.h>
#include <net/gen/in.h>
#include <net/gen/udp.h>
#include <net/gen/udp_hdr.h>
#include <net/gen/udp_io.h>
#include <net/hton.h>

#define PORT 6000L

/* type for received data */
typedef struct
{
    udp_io_hdr_t header;
    char data[1024];
} udp_buffer_t;

int udp_conf(long port) {

    char *udp_device;
    int netfd;
    nwio_udpopt_t udpopt;

    /* Get default UDP device */
    if ((udp_device = getenv("UDP_DEVICE")) == NULL)
        udp_device = UDP_DEVICE;

    /* Open UDP connection */
    if ((netfd = open(udp_device, O_RDWR)) < 0)
    {
        fprintf(stderr, "Error opening UDP connection\n");
        return -1;
    }

    /* Configure UDP connection */
    udpopt.nwuo_flags = NWUO_COPY | NWUO_LP_SET | NWUO_EN_LOC | NWUO_DI_BROAD
        | NWUO_RP_ANY | NWUO_RA_ANY | NWUO_RWDATALL | NWUO_DI_IPOPT;

    udpopt.nwuo_locport = (udpport_t) htons(port);

    if ((ioctl(netfd, NWIOSUDPOPT, &udpopt)) < 0)
    {
        fprintf(stderr, "Error configuring the connection\n");
        close(netfd);
        return -1;
    }

    /* Get conf options */
    if ((ioctl(netfd, NWIOGUDPOPT, &udpopt)) < 0)
    {
        fprintf(stderr, "Error getting the conf\n");
        close(netfd);
        return -1;
    }
}
```

```
    return netfd;
}

int main(int argc, char *argv[]) {
    int fd;
    ssize_t data_read;
    udp_buffer_t buffer;
    ipaddr_t tmp_addr;
    udpport_t tmp_port;
    int ret;
    fd_set fds_read;
    struct timeval timeout;
    ipaddr_t client_addr, this_addr;
    udpport_t client_port;

    if ((fd = udp_conf(PORT)) < 0) {
        fprintf(stderr, "Error configuring UDP connection\n");
        exit(-1);
    }
    printf("Waiting for messages on port: %ld\n", PORT);
    fflush(stdout);

    /* get a first message so we know who is the client and we can harass it
       afterwards */

    /* Initialize fd_set */
    FD_ZERO(&fds_read);
    FD_SET(fd, &fds_read);
    /* Set timeout structure */
    timeout.tv_sec = 3;
    timeout.tv_usec = 0;
    ret = select(4, &fds_read, NULL, NULL, NULL);

    if (ret < 0) {
        fprintf(stderr, "Error in select\n");
        exit(-1);
    }
    if (!FD_ISSET(fd, &fds_read)) {
        fprintf(stderr, "Error: Should be receiving some data from network(?)\n");
        exit(-1);
    }
    printf("Ready to receive...\n");
    /* Read received data */
    data_read = read(fd, &buffer, sizeof(udp_buffer_t));
    printf("Received data: %s\n", buffer.data);

    /* Can exit if the received string == exit */
    if (!strcmp(buffer.data, "exit"))
        exit(0);

    /* Send data back, swap addresses */
    tmp_addr = buffer.header.uih_src_addr;
    buffer.header.uih_src_addr = buffer.header.uih_dst_addr;
    buffer.header.uih_dst_addr = tmp_addr;
    /* save address of both ends */
    client_addr = tmp_addr;
    this_addr = buffer.header.uih_src_addr;

    /* Swap ports */
    tmp_port = buffer.header.uih_src_port;
    buffer.header.uih_src_port = buffer.header.uih_dst_port;
    buffer.header.uih_dst_port = tmp_port;
    /* save client port */
    client_port = tmp_port;

    /* Write the same back */
    write(fd, &buffer, data_read);

    while (1)
    {
        /* Initialize fd_set */
        FD_ZERO(&fds_read);
```

```
FD_SET(fd, &fds_read);
/* Set timeout structure */
timeout.tv_sec = 3;
timeout.tv_usec = 0;
/* Wait for data available to be read (timeout) */
ret = select(4, &fds_read, NULL, NULL, &timeout);
if (ret < 0) {
    fprintf(stderr, "Error on select: %d", errno);
    exit(-1);
}
/* if timeout */
if (ret == 0) {
    /* Send angry msg to client asking for more */
    printf("Tired of waiting, send client an angry message\n");
    buffer.header.uih_src_addr = this_addr;
    buffer.header.uih_dst_addr = client_addr;
    buffer.header.uih_src_port = PORT;
    buffer.header.uih_dst_port = client_port;
    strcpy(buffer.data, "Hey! I want to receive some data!\n");
    write(fd, &buffer, sizeof(udp_buffer_t));
}
/* If receive data from network */
if (FD_ISSET(fd, &fds_read)) {
    printf("Ready to receive...\n");
    /* Read received data */
    data_read = read(fd, &buffer, sizeof(udp_buffer_t));
    printf("Received data: %s\n", buffer.data);

    /* Can exit if the received string == exit */
    if (!strcmp(buffer.data, "exit"))
        break;

    /* Send data back, swap addresses */
    tmp_addr = buffer.header.uih_src_addr;
    buffer.header.uih_src_addr = buffer.header.uih_dst_addr;
    buffer.header.uih_dst_addr = tmp_addr;

    /* Swap ports */
    tmp_port = buffer.header.uih_src_port;
    buffer.header.uih_src_port = buffer.header.uih_dst_port;
    buffer.header.uih_dst_port = tmp_port;

    /* Write the same back */
    write(fd, &buffer, data_read);
}
}
close(fd);
}
```

```
/*
 * Test name: test06_cli.c
 *
 * Objective: Test a simple TCP client
 *
 * Description: Implements a simple echo client using the TCP protocol. First
 * it waits until it is possible to write (which is always).
 */

#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/select.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdarg.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <net/netlib.h>
#include <net/gen/netdb.h>
#include <net/gen/in.h>
#include <net/gen/tcp.h>
#include <net/gen/tcp_io.h>
#include <net/hton.h>

#define PORT 6060L

int tcp_connect(char *host, long port)
{
    /* creates a tcp connection with specified host and port */
    char *tcp_device;
    struct hostent *hp;
    int netfd;
    nwio_tcpcl_t tcpcopt;
    nwio_tcpconf_t tcpconf;
    ipaddr_t dirhost;
    int tries;
    int result;

    /* get host address */
    if ((hp = gethostbyname(host)) == (struct hostent*) NULL)
    {
        fprintf(stderr, "Unknown host\n");
        return(-1);
    }
    memcpy((char *)&dirhost, (char *)hp->h_addr, hp->h_length);

    /* Get default TCP device */
    if (( tcp_device = getenv("TCP_DEVICE") ) == NULL)
        tcp_device = TCP_DEVICE;

    /* Establish TCP connection */
    if ((netfd = open(tcp_device, O_RDWR)) < 0)
    {
        fprintf(stderr, "Error opening TCP device\n");
        return -1;
    }

    /* Configure TCP connection */
    tcpconf.nwtc_flags=NWTC_LP_SEL | NWTC_SET_RA | NWTC_SET_RP;
    tcpconf.nwtc_remaddr = dirhost;
    tcpconf.nwtc_rempport = (tcpport_t) htons(port);

    if ((result = ioctl(netfd, NWIOSTCPCONF, &tcpconf) ) < 0)
    {
        fprintf(stderr, "Error establishing communication\n");
        printf("Error: %d\n", result);
        close(netfd);
        return -1;
    }

    /* Get configuration for TCP comm */
    if ((result = ioctl(netfd, NWIOGTCPCONF, &tcpconf) ) < 0)
```

```
{
    fprintf(stderr, "Error getting configuration\n");
    printf("Error: %d\n", result);
    close(netfd);
    return -1;
}

/* Establish connection */
tcpcopt.nwtcl_flags = 0;
tries = 0;
while (tries < 10) {
    if ( (result = ioctl(netfd, NWIOTCPCONN, &tcpcopt)) < 0 ) {
        if (errno != EAGAIN)
        {
            fprintf(stderr, "Server is not listening\n");
            close(netfd);
            return(-1);
        }
        fprintf(stderr, "Unable to connect\n");
        sleep(1);
        tries++;
    }
    else
        break; /* Connection */
}
/* Check result value */
if (result < 0) {
    fprintf(stderr, "Error connecting\n");
    fprintf(stderr, "Error: %d\n", result);
    printf("Number of tries: %d\n", tries);
    printf("Error: %d\n", errno);
    close(netfd);
    return -1;
}
return netfd;
}

int main(int argc, char *argv[]) {
    int fd;
    ssize_t data_read;
    char send_buf[1024];
    char recv_buf[1024];
    fd_set fds_write;
    int ret;

    /* Check parameters */
    if (argc != 2) {
        fprintf(stderr, "Usage: %s host\n", argv[0]);
        exit(-1);
    }

    if ((fd = tcp_connect(argv[1], PORT) ) < 0)
        exit(-1);
    printf("Connected to server\n");
    /* init fd_set */
    FD_ZERO(&fds_write);
    FD_SET(fd, &fds_write);
    while (1)
    {
        /* Wait until it is possible to write with select */
        ret = select(4, NULL, &fds_write, NULL, NULL);
        if (ret < 0) {
            fprintf(stderr, "Error on select waiting for write: %d\n", errno);
            exit(-1);
        }
        if (!FD_ISSET(fd, &fds_write)) {
            fprintf(stderr, "Error: The net connection is not ready for writing (?)\n");
            exit(-1);
        }

        /* Get a string and send it */
        printf("Ready to write...\n");
        printf("Send data: ");
        gets(send_buf);
    }
}
```

```
write(fd, &send_buf, strlen(send_buf)+1);

/* If data sent is exit then break */
if (!strcmp(send_buf, "exit"))
    break;

/* Get server response */
data_read = read(fd, &recv_buf, 1024);
printf("Received: %s\n\n", recv_buf);
}

/* Close TCP communication */
close(fd);
}
```

```
/*
 * Test name: test06_srv.c
 *
 * Objective: Test a simple TCP server
 *
 * Description: Implements a simple echo server using the TCP protocol. Instead
 * of blocking on read(), it performs a select call first blocking there
 * until there is data to be read
 */

#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/select.h>
#include <sys/wait.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdarg.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <net/netlib.h>
#include <net/gen/netdb.h>
#include <net/gen/in.h>
#include <net/gen/tcp.h>
#include <net/gen/tcp_io.h>
#include <net/hton.h>
#include <net/gen/inet.h>

#define PORT 6060L

int listen(long port) {

    char *tcp_device;
    int netfd;
    nwio_tcpconf_t tcpconf;
    nwio_tcpcl_t tcplistenopt;

    /* Get default UDP device */
    if ((tcp_device = getenv("TCP_DEVICE")) == NULL)
        tcp_device = TCP_DEVICE;

    /* Open TCP connection */
    if ((netfd = open(tcp_device, O_RDWR)) < 0)
    {
        fprintf(stderr, "Error opening TCP connection\n");
        return -1;
    }

    /* Configure TCP connection */
    tcpconf.nwtc_flags = NWTC_LP_SET | NWTC_UNSET_RA | NWTC_UNSET_RP;
    tcpconf.nwtc_locport = (tcpport_t) htons(port);

    if ((ioctl(netfd, NWIOSTCPCONF, &tcpconf)) < 0)
    {
        fprintf(stderr, "Error configuring the connection\n");
        close(netfd);
        return -1;
    }

    /* Get communication options */
    if ((ioctl(netfd, NWIOGTCPCONF, &tcpconf)) < 0) {
        fprintf(stderr, "Error getting configuration\n");
        close(netfd);
        return -1;
    }

    /* Set conf options */
    tcplistenopt.nwtcl_flags = 0;
    printf("Waiting for connections...\n");
    while ((ioctl(netfd, NWIOTCPLISTEN, &tcplistenopt)) == -1)
    {
        if (errno != EAGAIN)
        {

```



```
        fprintf(stderr, "Unable to listen for connections\n" );
        close(netfd);
    }
    sleep(-1);
}
return netfd;
}

int main(int argc, char *argv[]) {
    int fd;
    ssize_t data_read;
    char buffer[1024];
    int ret;
    fd_set fds_read;

    if ((fd = listen(PORT)) < 0) {
        exit(-1);
    }
    printf("Waiting for messages on port: %d\n", PORT);
    fflush(stdout);
    /* Initialize fd_set */
    FD_ZERO(&fds_read);
    FD_SET(fd, &fds_read);

    while (1)
    {
        /* Wait for data available to be read (no timeout) */

        ret = select(4, &fds_read, NULL, NULL, NULL);
        if (ret < 0) {
            fprintf(stderr, "Error on select: %d\n", errno);
            exit(-1);
        }
        if (!FD_ISSET(fd, &fds_read)) {
            printf("Error: network fd is not ready (?)\n");
            exit(-1);
        }

        printf("Ready to receive...\n");
        /* Read received data */
        data_read = read(fd, &buffer, 1024);
        printf("Received data: %s\n", buffer);

        /* Can exit if the received string == exit */
        if (!strcmp(buffer, "exit"))
            break;

        /* Write the same back */
        write(fd, &buffer, data_read);
    }
    printf("Connection finished\n");
    close(fd);
}
```

```
/*
 * Test name: test07_cli.c
 *
 * Objective: Test an impatient TCP client with timeout which waits input both
 * from the terminal and from the network connection.
 *
 * Description: Implements a echo client using the TCP protocol with a timeout
 * value. It waits for data on both the terminal and the network connection and
 * prints an angry message asking for more data if there is a timeout.
 */

#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/select.h>
#include <sys/asynchio.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdarg.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <net/netlib.h>
#include <net/gen/netdb.h>
#include <net/gen/in.h>
#include <net/gen/tcp.h>
#include <net/gen/tcp_io.h>
#include <net/hton.h>

#define PORT 6060L

int tcp_connect(char *host, long port)
{
    /* creates a tcp connection with specified host and port */
    char *tcp_device;
    struct hostent *hp;
    int netfd;
    nwio_tcpcl_t tcpcopt;
    nwio_tcpconf_t tcpconf;
    ipaddr_t dirhost;
    int tries;
    int result;

    /* get host address */
    if ((hp = gethostbyname(host)) == (struct hostent*) NULL)
    {
        fprintf(stderr, "Unknown host\n");
        return(-1);
    }
    memcpy((char *)&dirhost, (char *)hp->h_addr, hp->h_length);

    /* Get default TCP device */
    if (( tcp_device = getenv("TCP_DEVICE") ) == NULL)
        tcp_device = TCP_DEVICE;

    /* Establish TCP connection */
    if ((netfd = open(tcp_device, O_RDWR)) < 0)
    {
        fprintf(stderr, "Error opening TCP device\n");
        return -1;
    }

    /* Configure TCP connection */
    tcpconf.nwtc_flags=NWTC_LP_SEL | NWTC_SET_RA | NWTC_SET_RP;
    tcpconf.nwtc_remaddr = dirhost;
    tcpconf.nwtc_rempport = (tcpport_t) htons(port);

    if ((result = ioctl(netfd, NWIOSTCPCONF, &tcpconf) ) <0)
    {
        fprintf(stderr, "Error establishing communication\n");
        printf("Error: %d\n",result);
        close(netfd);
        return -1;
    }
}
```

```
}

/* Get configuration for TCP comm */
if ((result = ioctl(netfd, NWIOGTCPCONF, &tcpconf) ) < 0)
{
    fprintf(stderr, "Error getting configuration\n");
    printf("Error: %d\n", result);
    close(netfd);
    return -1;
}

/* Establish connection */
tcpcopt.nwtcl_flags = 0;
tries = 0;
while (tries < 10) {
    if ( (result = ioctl(netfd, NWIOTCPCONN, &tcpcopt)) < 0 ) {
        if (errno != EAGAIN)
        {
            fprintf(stderr, "Server is not listening\n");
            close(netfd);
            return(-1);
        }
        fprintf(stderr, "Unable to connect\n");
        sleep(1);
        tries++;
    }
    else
        break; /* Connection */
}
/* Check result value */
if (result < 0) {
    fprintf(stderr, "Error connecting\n");
    fprintf(stderr, "Error: %d\n", result);
    printf("Number of tries: %d\n", tries);
    printf("Error: %d\n", errno);
    close(netfd);
    return -1;
}
return netfd;
}

int main(int argc, char *argv[]) {
    int fd;
    ssize_t data_read;
    char send_buf[1024];
    char recv_buf[1024];
    fd_set fds_read;
    int ret;
    struct timeval timeout;

    /* Check parameters */
    if (argc != 2) {
        fprintf(stderr, "Usage: %s host\n", argv[0]);
        exit(-1);
    }

    if ((fd = tcp_connect(argv[1], PORT) ) < 0)
        exit(-1);
    printf("Connected to server\n");

    while (1)
    {
        /* init fd_set */
        FD_ZERO(&fds_read);
        FD_SET(0, &fds_read); /* stdin */
        FD_SET(fd, &fds_read);
        /* set timeout */
        timeout.tv_sec = 3;
        timeout.tv_usec = 0;

        printf("Send data: ");
        fflush(stdout);
        /* Wait until it is possible to read with select */
    }
}
```

```
ret = select(4, &fds_read, NULL, NULL, &timeout);
if (ret < 0) {
    fprintf(stderr, "Error on select waiting for write: %d\n", errno);
    exit(-1);
}
/* timeout */
if (ret == 0) {
    printf("\nClient says: Hey! I want to send some data!!\n");
    continue;
}
/* handle data from network */
if (FD_ISSET(fd, &fds_read)) {
    data_read = read(fd, &recv_buf, 1024);
    printf("Server says: %s\n\n", recv_buf);
}
/* handle data from terminal */
if (FD_ISSET(0, &fds_read)) {
    /* Get a string and send it */
    gets(send_buf);
    write(fd, &send_buf, strlen(send_buf)+1);

    /* If data sent is exit then break */
    if (!strcmp(send_buf, "exit"))
        break;

    /* Get server response */
    data_read = read(fd, &recv_buf, 1024);
    printf("Received: %s\n\n", recv_buf);
}
}

/* Close TCP communication */
close(fd);
}
```

```
/*
 * Test name: test07_srv.c
 *
 * Objective: Test an impatient TCP server with a timeout.
 *
 * Description: Implements an echo server using the TCP protocol. It is
 * based on test06_srv.c but has a timeout for receiving data from a client
 * and if the time is up it sends an angry message to the client requesting
 * for more information.
 */

#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/select.h>
#include <sys/wait.h>
#include <sys/asynchio.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdarg.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <net/netlib.h>
#include <net/gen/netdb.h>
#include <net/gen/in.h>
#include <net/gen/tcp.h>
#include <net/gen/tcp_io.h>
#include <net/hton.h>
#include <net/gen/inet.h>

#define PORT 6060L

int listen(long port) {

    char *tcp_device;
    int netfd;
    nwio_tcpconf_t tcpconf;
    nwio_tcpcl_t tcplistenopt;

    /* Get default UDP device */
    if ((tcp_device = getenv("TCP_DEVICE")) == NULL)
        tcp_device = TCP_DEVICE;

    /* Open TCP connection */
    if ((netfd = open(tcp_device, O_RDWR)) < 0)
    {
        fprintf(stderr, "Error opening TCP connection\n");
        return -1;
    }

    /* Configure TCP connection */
    tcpconf.nwtc_flags = NWTC_LP_SET | NWTC_UNSET_RA | NWTC_UNSET_RP;
    tcpconf.nwtc_locport = (tcpport_t) htons(port);

    if ((ioctl(netfd, NWIOSTCPCONF, &tcpconf)) < 0)
    {
        fprintf(stderr, "Error configuring the connection\n");
        close(netfd);
        return -1;
    }

    /* Get communication options */
    if ((ioctl(netfd, NWIOGTCPCONF, &tcpconf)) < 0) {
        fprintf(stderr, "Error getting configuration\n");
        close(netfd);
        return -1;
    }

    /* Set conf options */
    tcplistenopt.nwtcl_flags = 0;
    printf("Waiting for connections...\n");
    while ((ioctl(netfd, NWIOTCPLISTEN, &tcplistenopt)) == -1)
    {
```

```
    if (errno != EAGAIN)
    {
        fprintf(stderr, "Unable to listen for connections\n" );
        close(netfd);
    }
    sleep(-1);
}
return netfd;
}

int main(int argc, char *argv[]) {
    int fd;
    ssize_t data_read;
    char buffer[1024];
    int ret;
    fd_set fds_read;
    struct timeval timeout;

    if ((fd = listen(PORT)) < 0) {
        exit(-1);
    }
    printf("Waiting for messages on port: %ld\n", PORT);
    fflush(stdout);
    while (1)
    {
        /* Initialize fd_set */
        FD_ZERO(&fds_read);
        FD_SET(fd, &fds_read);
        /* set timeout */
        timeout.tv_sec = 3;
        timeout.tv_usec = 0;
        /* Wait for data available to be read (no timeout) */
        ret = select(4, &fds_read, NULL, NULL, &timeout);
        if (ret < 0) {
            fprintf(stderr, "Error on select: %d\n", errno);
            exit(-1);
        }
        /* timeout */
        if (ret == 0) {
            strcpy(buffer, "I want to get some data!!\n");
            write(fd, &buffer, 1024);
            continue;
        }

        /* data received from client */
        if (FD_ISSET(fd, &fds_read)) {
            printf("Ready to receive...\n");
            /* Read received data */
            data_read = read(fd, &buffer, 1024);
            printf("Received data: %s\n", buffer);

            /* Can exit if the received string == exit */
            if (!strcmp(buffer, "exit"))
                break;

            /* Write the same back */
            write(fd, &buffer, data_read);
        }
    }
    printf("Connection finished\n");
    close(fd);
}
```

```
/*
 * Test name: test08_cli.c
 *
 * Objective: Test select on urgent data. TCP client.
 *
 * Description: It is based on test06_cli but sends urgent data.
 */

#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/select.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdarg.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <net/netlib.h>
#include <net/gen/netdb.h>
#include <net/gen/in.h>
#include <net/gen/tcp.h>
#include <net/gen/tcp_io.h>
#include <net/hton.h>

#define PORT 6060L

int tcp_connect(char *host, long port)
{
    /* creates a tcp connection with specified host and port */
    char *tcp_device;
    struct hostent *hp;
    int netfd;
    nwio_tcpcl_t tcpcl;
    nwio_tcpopt_t tcpopt;
    nwio_tcpconf_t tcpconf;
    ipaddr_t dirhost;
    int tries;
    int result;

    /* get host address */
    if ((hp = gethostbyname(host)) == (struct hostent*) NULL)
    {
        fprintf(stderr, "Unknown host\n");
        return(-1);
    }
    memcpy((char *)&dirhost, (char *)hp->h_addr, hp->h_length);

    /* Get default TCP device */
    if ((tcp_device = getenv("TCP_DEVICE")) == NULL)
        tcp_device = "TCP_DEVICE";

    /* Establish TCP connection */
    if ((netfd = open(tcp_device, O_RDWR)) < 0)
    {
        fprintf(stderr, "Error opening TCP device\n");
        return -1;
    }

    /* Configure TCP connection */
    tcpconf.nwtc_flags=NWTC_LP_SEL | NWTC_SET_RA | NWTC_SET_RP;
    tcpconf.nwtc_remaddr = dirhost;
    tcpconf.nwtc_rempport = (tcpport_t) htons(port);

    if ((result = ioctl(netfd, NWIOSTCPCONF, &tcpconf)) < 0)
    {
        fprintf(stderr, "Error establishing communication\n");
        printf("Error: %d\n", result);
        close(netfd);
        return -1;
    }

    /* Get configuration for TCP comm */
}
```

```
if ((result = ioctl(netfd, NWIOGTCPCONF, &tcpconf) ) < 0)
{
    fprintf(stderr, "Error getting configuration\n");
    printf("Error: %d\n", result);
    close(netfd);
    return -1;
}

/* Establish connection options (send URG data) */
tcpopt.nwto_flags = NWTO_SND_URG_MASK;

/* Set options for TCP comm */
if ((result = ioctl(netfd, NWIOSTCPOPT, &tcpopt) ) < 0)
{
    fprintf(stderr, "Error getting configuration\n");
    printf("Error: %d\n", result);
    close(netfd);
    return -1;
}

/* Get configuration for TCP comm */
if ((result = ioctl(netfd, NWIOGTCPOPT, &tcpopt) ) < 0)
{
    fprintf(stderr, "Error getting options\n");
    printf("Error: %d\n", result);
    close(netfd);
    return -1;
}

tcpcl.nwtcl_flags = 0;
tries = 0;
while (tries < 10) {
    if ( (result = ioctl(netfd, NWIOTCPCONN, &tcpcl)) < 0 ) {
        if (errno != EAGAIN)
        {
            fprintf(stderr, "Server is not listening\n");
            close(netfd);
            return(-1);
        }
        fprintf(stderr, "Unable to connect\n");
        sleep(1);
        tries++;
    }
    else
        break; /* Connection */
}
/* Check result value */
if (result < 0) {
    fprintf(stderr, "Error connecting\n");
    fprintf(stderr, "Error: %d\n", result);
    printf("Number of tries: %d\n", tries);
    printf("Error: %d\n", errno);
    close(netfd);
    return -1;
}
return netfd;
}

int main(int argc, char *argv[]) {
    int fd;
    ssize_t data_read;
    char send_buf[1024];
    char recv_buf[1024];
    fd_set fds_write;
    int ret;

    /* Check parameters */
    if (argc != 2) {
        fprintf(stderr, "Usage: %s host\n", argv[0]);
        exit(-1);
    }

    if ((fd = tcp_connect(argv[1], PORT) ) < 0)
        exit(-1);
}
```



```
printf("Connected to server\n");
/* init fd_set */
FD_ZERO(&fds_write);
FD_SET(fd, &fds_write);
while (1)
{
    /* Wait until it is possible to write with select */
    ret = select(4, NULL, &fds_write, NULL, NULL);
    if (ret < 0) {
        fprintf(stderr, "Error on select waiting for write: %d\n", errno);
        exit(-1);
    }
    if (!FD_ISSET(fd, &fds_write)) {
        fprintf(stderr, "Error: The net connection is not ready for writing (?)\n");
        exit(-1);
    }

    /* Get a string and send it */
    printf("Ready to write...\n");
    printf("Send data: ");
    gets(send_buf);
    write(fd, &send_buf, strlen(send_buf)+1);

    /* If data sent is exit then break */
    if (!strcmp(send_buf, "exit"))
        break;

    /* Get server response */
    data_read = read(fd, &recv_buf, 1024);
    printf("Received: %s\n\n", recv_buf);
}

/* Close UDP communication */
close(fd);
}
```

```
/*
 * Test name: test08_srv.c
 *
 * Objective: Test a simple TCP server waiting for urgent data.
 *
 * Description: Implements a echo TCP server as in test06_srv but waits
 * for urgent data, using select on exception.
 */

#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/select.h>
#include <sys/wait.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdarg.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <net/netlib.h>
#include <net/gen/netdb.h>
#include <net/gen/in.h>
#include <net/gen/tcp.h>
#include <net/gen/tcp_io.h>
#include <net/hton.h>
#include <net/gen/inet.h>

#define PORT 6060L

int listen(long port) {

    char *tcp_device;
    int netfd;
    nwio_tcpconf_t tcpconf;
    nwio_tcpcl_t tcpcl;
    nwio_tcpopt_t tcpopt;

    /* Get default UDP device */
    if ((tcp_device = getenv("TCP_DEVICE")) == NULL)
        tcp_device = TCP_DEVICE;

    /* Open TCP connection */
    if ((netfd = open(tcp_device, O_RDWR)) < 0)
    {
        fprintf(stderr, "Error opening TCP connection\n");
        return -1;
    }

    /* Configure TCP connection */
    tcpconf.nwtc_flags = NWTC_LP_SET | NWTC_UNSET_RA | NWTC_UNSET_RP;
    tcpconf.nwtc_locport = (tcpport_t) htons(port);

    if ((ioctl(netfd, NWIOSTCPCONF, &tcpconf)) < 0)
    {
        fprintf(stderr, "Error configuring the connection\n");
        close(netfd);
        return -1;
    }

    /* Get communication conf */
    if ((ioctl(netfd, NWIOGTCPCONF, &tcpconf)) < 0) {
        fprintf(stderr, "Error getting configuration\n");
        close(netfd);
        return -1;
    }

    /* Set comm options */
    tcpopt.nwto_flags = NWTO_RCV_URG;

    if ((ioctl(netfd, NWIOSTCPOPT, &tcpopt)) < 0)
    {
        fprintf(stderr, "Error configuring the connection\n");
    }
}
```

```
    close(netfd);
    return -1;
}

/* Get communication opt*/
if ((ioctl(netfd, NWIOGTCPOPT, &tcptopt)) < 0) {
    fprintf(stderr, "Error getting options\n");
    close(netfd);
    return -1;
}

/* Set conn options */
tcpcl.nwtcl_flags = 0;
printf("Waiting for connections...\n");
while ((ioctl(netfd, NWIOTCPLISTEN, &tcpcl)) == -1)
{
    if (errno != EAGAIN)
    {
        fprintf(stderr, "Unable to listen for connections\n");
        close(netfd);
    }
    sleep(-1);
}
return netfd;
}

int main(int argc, char *argv[]) {
    int fd;
    ssize_t data_read;
    char buffer[1024];
    int ret;
    fd_set fds_excep;

    if ((fd = listen(PORT)) < 0) {
        exit(-1);
    }
    printf("Waiting for messages on port: %d\n", PORT);
    fflush(stdout);
    /* Initialize fd_set */
    FD_ZERO(&fds_excep);
    FD_SET(fd, &fds_excep);

    while (1)
    {
        /* Wait for data available to be read (no timeout) */

        ret = select(4, NULL, NULL, &fds_excep, NULL);
        if (ret < 0) {
            fprintf(stderr, "Error on select: %d\n", errno);
            exit(-1);
        }
        if (!FD_ISSET(fd, &fds_excep)) {
            printf("Error: no URG data received (?)\n");
            exit(-1);
        }

        printf("Ready to receive...\n");
        /* Read received data */
        data_read = read(fd, &buffer, 1024);
        printf("Received data: %s\n", buffer);

        /* Can exit if the received string == exit */
        if (!strcmp(buffer, "exit"))
            break;

        /* Write the same back */
        write(fd, &buffer, data_read);
    }
    printf("Connection finished\n");
    close(fd);
}
```

```
/*
 * Test name: test09.c
 *
 * Objective: The purpose of this test is to make sure that select works
 * when working with the terminal.
 *
 * Description: This tests wait entry from stdin using select and displays
 * it again in stdout when it is ready to write (which is always)
 *
 * Jose M. Gomez
 */

#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/select.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>

void main(void) {
    fd_set fds_read, fds_write;
    int retval;
    char data[1024];

    FD_ZERO(&fds_read);
    FD_ZERO(&fds_write);
    FD_SET(0, &fds_read);          /* stdin */
    FD_SET(1, &fds_write);         /* stdout */

    while(1) {
        printf("Input some data: ");
        fflush(stdout);
        retval=select(3, &fds_read, NULL, NULL, NULL);
        if (retval < 0) {
            fprintf(stderr, "Error while executing select\n");
            exit(-1);
        }
        printf("select retval: %d\n", retval);
        if (!FD_ISSET(0, &fds_read)) {
            fprintf(stderr, "Error: stdin not ready (?)\n");
            exit(-1);
        }
        printf("gets..\n");
        gets(data);
        printf("gets done..\n");
        if (!strcmp(data, "exit"))
            exit(0);
        printf("Try to write it back\n");
        retval=select(3, NULL, &fds_write, NULL, NULL);
        if (retval < 0) {
            fprintf(stderr, "Error while executing select\n");
            exit(-1);
        }
        if (!FD_ISSET(1, &fds_write)) {
            fprintf(stderr, "Error: stdout not ready (?)\n");
            exit(-1);
        }
        printf("Data: %s\n", data);
    }
}
```

```
/*
 * Test name: test10.c
 *
 * Objective: The purpose of this test is to make sure that select works
 * when working with the terminal with timeouts
 *
 * Description: This tests wait entry from stdin using select and displays
 * it again in stdout when it is ready to write (which is always). It has
 * a timeout value as well.
 *
 * Jose M. Gomez
 */

#include <time.h>
#include <sys/types.h>
#include <sys/asynchio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/select.h>
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <string.h>

void main(void) {
    fd_set fds_read, fds_write;
    int retval;
    char data[1024];
    struct timeval timeout;

    while(1) {
        timeout.tv_sec = 3;
        timeout.tv_usec = 0;
        FD_ZERO(&fds_read);
        FD_ZERO(&fds_write);
        FD_SET(0, &fds_read);
        FD_SET(1, &fds_write);
        printf("Input some data: ");
        fflush(stdout);
        retval=select(3, &fds_read, NULL, NULL, &timeout);
        if (retval < 0) {
            fprintf(stderr, "Error while executing select\n");
            exit(-1);
        }
        if (retval == 0) {
            printf("\n Hey! Feed me some data!\n");
            fflush(stdout);
            continue;
        }
        if (!FD_ISSET(0, &fds_read)) {
            fprintf(stderr, "Error: stdin not ready (?)\n");
            exit(-1);
        }
        gets(data);
        if (!strcmp(data, "exit"))
            exit(0);
        printf("Try to write it back\n");
        retval=select(3, NULL, &fds_write, NULL, NULL);
        if (retval < 0) {
            fprintf(stderr, "Error while executing select\n");
            exit(-1);
        }
        if (!FD_ISSET(1, &fds_write)) {
            fprintf(stderr, "Error: stdout not ready (?)\n");
            exit(-1);
        }
        printf("Data: %s\n", data);
    }
}
```

```
/*
 * Test name: test11.c
 *
 * Objective: The purpose of this test is to make sure that select works
 * with pipes.
 *
 * Description: The select checks are divided in checks on writing for the
 * parent process, which has the writing end of the pipe, and checks on reading
 * and exception on the child process, which has the reading end of pipe. So
 * when the first process is ready to write to the pipe it will request a string
 * from the terminal and send it through the pipe. If the string is 'exit' then
 * the pipe is closed. The child process is blocked in a select checking for read
 * and exception. If there is data to be read then it will perform the read and
 * prints the read data. If the pipe is closed (user typed 'exit'), the child
 * process finishes.
 *
 * Jose M. Gomez
 */

#include <time.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/asynchio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/select.h>
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <string.h>
#include <signal.h>

void pipehandler(int sig)
{
}

void do_child(int data_pipe[])
{
    /* reads from pipe and prints out the data */
    char data[2048];
    int retval;
    fd_set fds_read;
    fd_set fds_exception;
    struct timeval timeout;

    signal(SIGPIPE, pipehandler);
    signal(SIGUSR1, pipehandler);

    /* first, close the write part of the pipe, since it is not needed */
    close(data_pipe[1]);

    while(1) {
        FD_ZERO(&fds_read);
        FD_ZERO(&fds_exception);
        FD_SET(data_pipe[0], &fds_read);
        FD_SET(data_pipe[0], &fds_exception);
        timeout.tv_sec = 5;
        timeout.tv_usec = 0;
        retval = select(data_pipe[0]+1, &fds_read, NULL, &fds_exception, &timeou
t);

        if (retval == -1) {
            perror("select");
            fprintf(stderr, "child: Error in select\n");
            continue;
        } else printf("child select: %d\n", retval);
        if (FD_ISSET(data_pipe[0], &fds_exception)) {
            printf("child: exception fd set. quitting.\n");
            break;
        }
        if (FD_ISSET(data_pipe[0], &fds_read)) {
            printf("child: read fd set. reading.\n");
            if ((retval = read(data_pipe[0], data, sizeof(data))) < 0) {
                perror("read");
            }
        }
    }
}
```

```

        fprintf(stderr, "child: couldn't read from pipe\n" );
        exit(-1);
    }
    if(retval == 0) {
        fprintf(stderr, "child: eof on pipe\n");
        break;
    }
    data[retval] = '\0';
    printf("pid %d Pipe reads (%d): %s\n", getpid(), retval, data);
} else printf("child: no fd set\n");
}

/* probably pipe was broken, or got EOF via the pipe. */
exit(0);
}

void do_parent(int data_pipe[])
{
    char data[1024];
    int retval;
    fd_set fds_write;

    signal(SIGPIPE, pipehandler);
    signal(SIGUSR1, pipehandler);

    /* first, close the read part of pipe, since it is not needed */
    close(data_pipe[0]);

    /* now enter a loop of read user input, and writing it to the pipe */
    while (1) {
        FD_ZERO(&fds_write);
        FD_SET(data_pipe[1], &fds_write);
        printf("pid %d Waiting for pipe ready to write...\n", getpid());
        retval = select(data_pipe[1]+1, NULL, &fds_write, NULL, NULL);
        if (retval == -1) {
            perror("select");
            fprintf(stderr, "Parent: Error in select\n");
            exit(-1);
        }

        printf("Input data: ");
        if(!gets(data)) {
            printf("parent: eof; exiting\n");
            break;
        }
        if (!strcmp(data, "exit"))
            break;
        if (!FD_ISSET(data_pipe[1], &fds_write)) {
            fprintf(stderr, "parent: write fd not set?! retrying\n");
            continue;
        }
        retval = write(data_pipe[1], &data, 1024);
        if (retval == -1) {
            perror("write");
            fprintf(stderr, "Error writing on pipe\n");
            exit(-1);
        }
    }

    /* got exit from user */
    close(data_pipe[1]); /* close pipe, let child know we're done */
    wait(&retval);
    printf("Child exited with status: %d\n", retval);
    exit(0);
}

void main(void) {
    int pipes[2];
    int retval;
    int pid;

    /* create the pipe */
    retval = pipe(pipes);
    if (retval == -1) {

```

```
        perror("pipe");
        fprintf(stderr, "Error creating the pipe\n");
        exit(-1);
    }
    pid = fork();
    if (pid == -1) {
        fprintf(stderr, "Error forking\n");
        exit(-1);
    }

    if (pid == 0) /* child proc */
        do_child(pipes);
    else
        do_parent(pipes);
}
```



```
/*
 * Test name: test12.c
 *
 * Objective: The purpose of this check the behaviour when a signal is received
 * and there is a handler.
 *
 * Description: The program handles SIGHUP and expects the user to actually send
 * the signal from other terminal with 'kill -1 pid'
 *
 * Jose M. Gomez
 */

#include <sys/types.h>
#include <sys/select.h>
#include <unistd.h>
#include <time.h>
#include <stdio.h>
#include <signal.h>
#include <errno.h>

#define SECONDS 15

void catch_hup(int sig_num)
{
    /* don't need to reset signal handler */
    printf("Received a SIGHUP, inside the handler now\n");
}

int main(void) {
    int ret;                                /* return value */

    /* set the HUP signal handler to 'catch_hup' */
    signal(SIGHUP, catch_hup);
    /* Get proc_id and print it */
    printf("Send a signal from other terminal with: kill -1 %d\n", getpid());
    printf("Blocking now on select...\n", SECONDS);
    ret = select(0, NULL, NULL, NULL, NULL);
    printf("Errno: %d\n", errno);
}
```

```
/*
 * Test name: test13a.c
 *
 * Objective: The purpose of this tests is to show how a select can
 * get into a race condition when dealing with signals.
 *
 * Description: The program waits for SIGHUP or input in the terminal
 */

#include <sys/types.h>
#include <sys/select.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <stdio.h>
#include <signal.h>
#include <errno.h>

int got_sighup = 0;

void catch_hup(int sig_num)
{
    printf("Received a SIGHUP, set global vble\n");
    got_sighup = 1;
}

int main(void) {
    int ret;                /* return value */
    fd_set read_fds;
    char data[1024];

    /* Init read fd_set */
    FD_ZERO(&read_fds);
    FD_SET(0, &read_fds);

    /* set the HUP signal handler to 'catch_hup' */
    signal(SIGHUP, catch_hup);

    /* Get proc_id and print it */
    printf("Send a signal from other terminal with: kill -1 %d\n", getpid());
    printf("Going to sleep for 5 seconds, if the signal arrives meanwhile\n");
    printf("the process will be blocked until there is input in the keyboard\n");
    printf("if the signal arrives after the timeout and while in select, it will\n");
    printf("behave as it should.\n");
    printf("Sleeping for 5 secs\n");
    sleep(5);

    printf("Blocking now on select...\n");
    ret = select(1, &read_fds, NULL, NULL, NULL);
    if (got_sighup) {
        printf("We have a sighup signal so exit the program\n");
        exit(0);
    }
    gets(data);
    printf("Got entry for terminal then, bye\n");
}
```

```
/*
 * Test name: test13b.c
 *
 * Objective: The purpose of this tests is to show how pselect()
 * solves the situation shown in test13a.c
 *
 * Description: The program waits for SIGHUP or input in the terminal
 */

#include <sys/types.h>
#include <sys/select.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <stdio.h>
#include <signal.h>
#include <errno.h>

int got_sighup = 0;

void catch_hup(int sig_num)
{
    printf("Received a SIGHUP, set global vble\n");
    got_sighup = 1;
}

int main(void) {
    int ret; /* return value */
    fd_set read_fds;
    sigset_t sigmask, orig_sigmask;
    char data[1024];

    /* Init read fd_set */
    FD_ZERO(&read_fds);
    FD_SET(0, &read_fds);

    /* set the signal masks */
    sigemptyset(&sigmask);
    sigaddset(&sigmask, SIGHUP);
    sigprocmask( SIG_BLOCK, &sigmask, &orig_sigmask);

    /* set the HUP signal handler to 'catch_hup' */
    signal(SIGHUP, catch_hup);

    /* Get proc_id and print it */
    printf("Send a signal from other terminal with: kill -1 %d\n", getpid());
    printf("Going to sleep for 5 seconds, if the signal arrives meanwhile\n");
    printf("the process will be blocked until there is input in the keyboard\n");
    printf("if the signal arrives after the timeout and while in select, it will\n");
    printf("behave as it should.\n");
    printf("Sleeping for 5 secs\n");
    sleep(5);

    printf("Blocking now on select...\n");
    #if 0
    ret = pselect(1, &read_fds, NULL, NULL, NULL, &orig_sigmask);
    #else
    ret = -1;
    #endif
    if (got_sighup) {
        printf("We have a sighup signal so exit the program\n");
        exit(0);
    }
    gets(data);
    printf("Got entry for terminal then, bye\n");
}
```

```

/*
 * Test name: test14.c
 *
 * Objective: The purpose of this test is to make sure that select works
 * with ptys.
 *
 * Adapted from test11.c (pipe test).
 *
 * Ben Gras
 */
#include <time.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/asynchio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/select.h>
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <string.h>
#include <signal.h>
#include <libutil.h>
char name[100];
void pipehandler(int sig)
{
}
#define CHILDFD 1
#define PARENTFD 0
void do_child(int pty_fds[])
{
    /* reads from pipe and prints out the data */
    char data[2048];
    int retval;
    fd_set fds_read;
    fd_set fds_exception;
    struct timeval timeout;
    signal(SIGPIPE, pipehandler);
    signal(SIGUSR1, pipehandler);
    /* first, close the write part, since it is not needed */
    close(pty_fds[PARENTFD]);

    while(1) {
        FD_ZERO(&fds_read);
        FD_ZERO(&fds_exception);
        FD_SET(pty_fds[CHILDFD], &fds_read);
        FD_SET(pty_fds[CHILDFD], &fds_exception);
        timeout.tv_sec = 5;
        timeout.tv_usec = 0;
        retval = select(pty_fds[CHILDFD]+2, &fds_read, NULL, &fds_exception, &timeout);

        if (retval == -1) {
            perror("select");
            fprintf(stderr, "child: Error in select\n");
            continue;
        } else printf("child select: %d\n", retval);
        if (FD_ISSET(pty_fds[CHILDFD], &fds_exception)) {
            printf("child: exception fd set. quitting.\n");
            break;
        }
        if (FD_ISSET(pty_fds[CHILDFD], &fds_read)) {
            printf("child: read fd set. reading.\n");
            if ((retval = read(pty_fds[CHILDFD], data, sizeof(data))) < 0) {
                perror("read");
                fprintf(stderr, "child: couldn't read from pty\n");
                exit(-1);
            }
            if(retval == 0) {
                fprintf(stderr, "child: eof on pty\n");
                break;
            }
            data[retval] = '\0';
            printf("pid %d pty reads (%d): %s\n", getpid(), retval, data);
        } else printf("child: no fd set\n");
    }
}

```

```

    }

    exit(0);
}

void do_parent(int pty_fds[])
{
    char data[1024];
    int retval;
    fd_set fds_write;
    signal(SIGPIPE, pipehandler);
    signal(SIGUSR1, pipehandler);
    /* first, close the read part of pty, since it is not needed */
    close(pty_fds[CHILDFD]);
    /* now enter a loop of read user input, and writing it to the pty */
    while (1) {
        FD_ZERO(&fds_write);
        FD_SET(pty_fds[PARENTFD], &fds_write);
        printf("pid %d Waiting for pty ready to write on %s...\n",
            getpid(), name);
        retval = select(pty_fds[PARENTFD]+2, NULL, &fds_write, NULL, NULL);
        if (retval == -1) {
            perror("select");
            fprintf(stderr, "Parent: Error in select\n");
            exit(-1);
        }
        printf("Input data: ");
        if(!gets(data)) {
            printf("parent: eof; exiting\n");
            break;
        }
        if (!strcmp(data, "exit"))
            break;
        if (!FD_ISSET(pty_fds[PARENTFD], &fds_write)) {
            fprintf(stderr, "parent: write fd not set?! retrying\n");
            continue;
        }
        retval = write(pty_fds[PARENTFD], data, 1024);
        if (retval == -1) {
            perror("write");
            fprintf(stderr, "Error writing on pty\n");
            exit(-1);
        } else printf("wrote %d\n", retval);
    }
    /* got exit from user */
    close(pty_fds[PARENTFD]); /* close pty, let child know we're done */
    wait(&retval);
    printf("Child exited with status: %d\n", retval);
    exit(0);
}

int main(int argc, char *argv[])
{
    int ptys[2];
    int retval;
    int pid;
    if(openpty(&ptys[0], &ptys[1], name, NULL, NULL) < 0) {
        perror("openpty");
        return 1;
    }
    printf("Using %s\n", name);
    pid = fork();
    if (pid == -1) {
        fprintf(stderr, "Error forking\n");
        exit(-1);
    }
    if (pid == 0) /* child proc */
        do_child(ptys);
    else
        do_parent(ptys);
    /* not reached */
    return 0;
}

```

Table of Contents

1	<i>Makefile</i>	sheets	1 to	2 (2)	pages	1- 2	79 lines
2	<i>common.c</i>	sheets	3 to	4 (2)	pages	3- 4	104 lines
3	<i>run</i>	sheets	5 to	5 (1)	pages	5- 5	41 lines
4	<i>t10a.c</i>	sheets	6 to	6 (1)	pages	6- 6	9 lines
5	<i>t11a.c</i>	sheets	7 to	7 (1)	pages	7- 7	71 lines
6	<i>t11b.c</i>	sheets	8 to	8 (1)	pages	8- 8	56 lines
7	<i>test1.c</i>	sheets	9 to	10 (2)	pages	9- 10	149 lines
8	<i>test10.c</i>	sheets	11 to	13 (3)	pages	11- 13	153 lines
9	<i>test11.c</i>	sheets	14 to	17 (4)	pages	14- 17	225 lines
10	<i>test12.c</i>	sheets	18 to	18 (1)	pages	18- 18	56 lines
11	<i>test13.c</i>	sheets	19 to	19 (1)	pages	19- 19	73 lines
12	<i>test14.c</i>	sheets	20 to	21 (2)	pages	20- 21	80 lines
13	<i>test15.c</i>	sheets	22 to	31 (10)	pages	22- 31	705 lines
14	<i>test16.c</i>	sheets	32 to	35 (4)	pages	32- 35	252 lines
15	<i>test17.c</i>	sheets	36 to	55 (20)	pages	36- 55	1159 lines
16	<i>test18.c</i>	sheets	56 to	73 (18)	pages	56- 73	1280 lines
17	<i>test19.c</i>	sheets	74 to	81 (8)	pages	74- 81	522 lines
18	<i>test2.c</i>	sheets	82 to	87 (6)	pages	82- 87	410 lines
19	<i>test20.c</i>	sheets	88 to	93 (6)	pages	88- 93	399 lines
20	<i>test21.c</i>	sheets	94 to	103 (10)	pages	94-103	686 lines
21	<i>test22.c</i>	sheets	104 to	106 (3)	pages	104-106	202 lines
22	<i>test23.c</i>	sheets	107 to	112 (6)	pages	107-112	416 lines
23	<i>test24.c</i>	sheets	113 to	118 (6)	pages	113-118	400 lines
24	<i>test25.c</i>	sheets	119 to	128 (10)	pages	119-128	727 lines
25	<i>test26.c</i>	sheets	129 to	132 (4)	pages	129-132	277 lines
26	<i>test27.c</i>	sheets	133 to	137 (5)	pages	133-137	349 lines
27	<i>test28.c</i>	sheets	138 to	143 (6)	pages	138-143	439 lines
28	<i>test29.c</i>	sheets	144 to	147 (4)	pages	144-147	247 lines
29	<i>test3.c</i>	sheets	148 to	151 (4)	pages	148-151	255 lines
30	<i>test30.c</i>	sheets	152 to	156 (5)	pages	152-156	343 lines
31	<i>test31.c</i>	sheets	157 to	160 (4)	pages	157-160	272 lines
32	<i>test32.c</i>	sheets	161 to	165 (5)	pages	161-165	368 lines
33	<i>test33.c</i>	sheets	166 to	174 (9)	pages	166-174	653 lines
34	<i>test34.c</i>	sheets	175 to	184 (10)	pages	175-184	669 lines
35	<i>test35.c</i>	sheets	185 to	190 (6)	pages	185-190	415 lines
36	<i>test36.c</i>	sheets	191 to	194 (4)	pages	191-194	245 lines
37	<i>test37.c</i>	sheets	195 to	209 (15)	pages	195-209	1050 lines
38	<i>test38.c</i>	sheets	210 to	220 (11)	pages	210-220	771 lines
39	<i>test39.c</i>	sheets	221 to	230 (10)	pages	221-230	723 lines
40	<i>test4.c</i>	sheets	231 to	232 (2)	pages	231-232	104 lines
41	<i>test40.c</i>	sheets	233 to	237 (5)	pages	233-237	341 lines
42	<i>test41.c</i>	sheets	238 to	238 (1)	pages	238-238	39 lines
43	<i>test5.c</i>	sheets	239 to	244 (6)	pages	239-244	401 lines
44	<i>test6.c</i>	sheets	245 to	247 (3)	pages	245-247	214 lines
45	<i>test7.c</i>	sheets	248 to	257 (10)	pages	248-257	696 lines
46	<i>test8.c</i>	sheets	258 to	261 (4)	pages	258-261	266 lines
47	<i>test9.c</i>	sheets	262 to	265 (4)	pages	262-265	274 lines
48	<i>testsh1.sh</i>	sheets	266 to	270 (5)	pages	266-270	311 lines
49	<i>testsh2.sh</i>	sheets	271 to	274 (4)	pages	271-274	261 lines
50	<i>Makefile</i>	sheets	275 to	275 (1)	pages	275-275	40 lines
51	<i>speed.c</i>	sheets	276 to	276 (1)	pages	276-276	60 lines
52	<i>test00.c</i>	sheets	277 to	277 (1)	pages	277-277	59 lines
53	<i>test01.c</i>	sheets	278 to	278 (1)	pages	278-278	75 lines
54	<i>test02.c</i>	sheets	279 to	280 (2)	pages	279-280	113 lines
55	<i>test03.c</i>	sheets	281 to	282 (2)	pages	281-282	136 lines
56	<i>test04_cli.c</i>	sheets	283 to	284 (2)	pages	283-284	149 lines
57	<i>test04_srv.c</i>	sheets	285 to	286 (2)	pages	285-286	135 lines
58	<i>test05_cli.c</i>	sheets	287 to	289 (3)	pages	287-289	166 lines
59	<i>test05_srv.c</i>	sheets	290 to	292 (3)	pages	290-292	197 lines
60	<i>test06_cli.c</i>	sheets	293 to	295 (3)	pages	293-295	163 lines
61	<i>test06_srv.c</i>	sheets	296 to	297 (2)	pages	296-297	128 lines
62	<i>test07_cli.c</i>	sheets	298 to	300 (3)	pages	298-300	183 lines
63	<i>test07_srv.c</i>	sheets	301 to	302 (2)	pages	301-302	138 lines
64	<i>test08_cli.c</i>	sheets	303 to	305 (3)	pages	303-305	184 lines
65	<i>test08_srv.c</i>	sheets	306 to	307 (2)	pages	306-307	146 lines
66	<i>test09.c</i>	sheets	308 to	308 (1)	pages	308-308	63 lines
67	<i>test10.c</i>	sheets	309 to	309 (1)	pages	309-309	70 lines
68	<i>test11.c</i>	sheets	310 to	312 (3)	pages	310-312	163 lines
69	<i>test12.c</i>	sheets	313 to	313 (1)	pages	313-313	40 lines
70	<i>test13a.c</i>	sheets	314 to	314 (1)	pages	314-314	57 lines
71	<i>test13b.c</i>	sheets	315 to	315 (1)	pages	315-315	68 lines
72	<i>test14.c</i>	sheets	316 to	317 (2)	pages	316-317	145 lines