MINIX3: Траектория программного вызова

Цилюрик О.И.

< olej@front.ru >

Редакция 1.09

от 18.12.2009

Оглавление

Стандартная библиотека.....1
Системные вызовы.....5
Вызовы ядра.....11
Заключение.....18
Источники информации.....18

Напишем простейшее приложение, с которого все начинают разговор, файл `hello.c`:

```
#include <stdio.h>
#include <stdlib.h>
int main( int argc, char **argv ) {
    printf( "Hello, World!\n" );
    return EXIT_SUCCESS;
}
```

Собираем из исходного кода приложение:

```
# cc hello.c -o hello
```

И можем его выполнить, как минимум, тремя разными способами:

```
# ./hello
Hello, World!
# nice -n -20 ./hello
Hello, World!
# service up /root/hello
Request to RS failed: unknown error (error 105)
```

(я не буду сейчас останавливаться на природе ошибки, и как запустить программу `hello` как сервис/драйвер, важен факт, что такой запуск возможен).

Собственно сама эта программа ничего не делает, всю работу в этой программе выполняют программные вызовы, которых в программе, как минимум, 2: явный `printf()` и неявный `exit()` при завершении. Моей задачей в этих заметках я вижу: отследить прохождение программных вызовов во всём их разнообразии сверху донизу, и какие сервисы операционной системы и на каком уровне их обслуживают. Вторичной задачей будет: прямая запись в программном коде некоторых из этих механизмов, минуя надстройки верхнего уровня.

Стандартная библиотека

Но рассматривать прохождение программных вызовов мы станем на чуть более развитой программе, файл `fork.c`:

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

```
int main( int argc, char **argv ) {
   pid_t pid = fork();
   if( pid < 0 ) {
      printf( "forking error!\n" );
      return EXIT_FAILURE;
   }
   else if( 0 == pid ) printf( "child:\tPID=%d <= %d\n", getpid(), getppid() );
   else printf( "parent:\tPID=%d => %d\n", getpid(), pid );
   return EXIT_SUCCESS;
}
```

Выполняем:

```
# cc hello.c -o hello
# ./fork
child:  PID=735 <= 734
parent: PID=734 => 735
```

Программа элементарно простая: процесс порождает дочерний процесс, и каждый из них извещает то, как он это наблюдает: родитель указывает PID порождённого, а дочерний — своего родителя. Все действия в программе выполняют программные вызовы из стандартной библиотеки C `/usr/lib/i386/libc.a` :

```
# ls /usr/lib/i386
as      end.a     libcurses.a  libedit.a  libm.a   libocm.a  libsysutil.a liby.a  prtso.o
cg      libbas.a  libd.a       libfl.a    libm2.a  libp.a    libtimers.a  libz.a
crtso.o libc.a    libe.a       libfp.a    libmq.a  libsys.a  libutil.a    m2rtso.o
```

Примечание: обратим внимание, что

– здесь же находятся стандартные библиотеки периода исполнения для MODULA-2 (`libm2.a`) и Pascal (`libp.a`);

– и стартовые преамбулы, для C программ это `crtso.o` который обеспечивает выполнение что-то вроде следующего:

```
                push    ecx         ! push envp
                push    edx         ! push argv
                push    eax         ! push argc
                . . .
                call    _main       ! main(argc, argv, envp)
                push    eax         ! push exit status
                call    _exit
```

– который, в принципе, тоже может быть подменён (исходный код находится в `/usr/src/lib/i386/rts/crtso.s`), например, чтобы выполнить некоторые действия прежде вызова `main()`.

Исходные коды реализации всех программных вызовов находятся в подкаталогах каталога `/usr/src/lib`. Простейшие программные вызовы стандартной библиотеки C начнём рассматривать с `/usr/src/lib/ansi`:

```
# ls /usr/src/lib/ansi
atof.c    exit.c     islower.c   malloc.c   qsort.c    strcpy.c     strstr.c
atoi.c    ext_comp.c isprint.c   mblen.c    raise.c    strcspn.c    strtok.c
...
```

Это всё в великом множестве знакомые функции стандартной библиотеки C. Смотрим реализацию отдельных из них[1]:

```
# cat /usr/src/lib/ansi/memcpy.c
void *memcpy(void *s1, const void *s2, register size_t n)
{
```

1 В множественных примерах реализаций я выбрасываю из кода комментарии, директивы препроцессора и весь вспомогательный код, который перегружает текст, но не способствует непосредственно пониманию.

```
        register char *p1 = s1;
        register const char *p2 = s2;


        if (n) {
                n++;
                while (--n > 0) {
                        *p1++ = *p2++;
                }
        }
        return s1;
}


# cat /usr/src/lib/ansi/strlen.c
size_t strlen(const char *org)
{
        register const char *s = org;

        while (*s++)
                /* EMPTY */ ;

        return --s - org;
}
```

Здесь всё предельно ясно: программа выполняет вызов функции, которая полностью обслуживается объектным кодом из библиотеки `libc.a`, как это показано на рисунке 1.

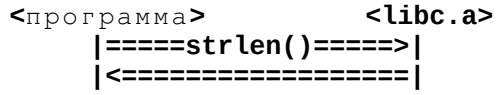

Рисунок 1. Обслуживание программного вызова библиотекой `/usr/lib/i386/libc.a` .

Таких (обслуживаемых по такой схеме, без привлечения служб операционной системы) программных вызовов (точек входа `libc.a`) — большинство. Но даже в подкаталоге `/usr/src/lib/ansi` (библиотека ANSI) есть неожиданности:

```
# cat /usr/src/lib/ansi/exit.c
#define NEXITS  32

void (*__functab[NEXITS])(void);
int __funccnt = 0;

extern void _exit(int);

/* only flush output buffers when necessary */
int (*_clean)(void) = NULL;

static void _calls(void)
{
        register int i = __funccnt;

        /* "Called in reversed order of their registration" */
        while (--i >= 0)
                (*__functab[i])();
}

void exit(int status)
{
        _calls();
        if (_clean) _clean();
        _exit(status) ;
}
```

Если кто увидел здесь освобождение стека процедур завершения — замечательно, но нас здесь будет интересовать другое: только один вызов `_exit()`, который мы пока возьмём на заметку.

Следующий каталог исходных кодов библиотеки, из числа очень знакомых, `/usr/src/lib/stdio`:

```
# ls /usr/src/lib/stdio
data.c   fflush.c  fopen.c   fseek.c   icompute.c  puts.c   sscanf.c  vsprintf.c
doprnt.c fgetc.c   fprintf.c fsetpos.c loc_incl.h  remove.c tmpfile.c vsscanf.c
...
fclose.c fileno.c  fread.c   getc.c    printf.c    setbuf.c vfprintf.c
...
```

Здесь у нас есть старый знакомый `printf.c`, который использовался в наших программах `hello.c/fork.c`:

```
# cat /usr/src/lib/stdio/printf.c
int printf(const char *format, ...)
{
        va_list ap;
        int retval;
        va_start(ap, format);
        retval = _doprnt(format, ap, stdout);
        va_end(ap);
        return retval;
}
```

Но перейдём к более сложным случаям, когда для выполнения вызова библиотека должна обращаться к сервисам операционной системы. В системах с монолитным ядром в этом случае производится прерывание ядра: INT 21h в MS-DOS, INT в 80h Linux. Будем следить, как это происходит в микроядерном MINIX3. Первоначально на примере используемого нашими программами вызова `getpid()`:

```
# cat /usr/src/lib/posix/_getpid.c
#include <lib.h>
#define getpid  _getpid
#include <unistd.h>

PUBLIC pid_t getpid()
{
  message m;
  return(_syscall(MM, MINIX_GETPID, &m));
}
```

Библиотека делает **системный вызов** к сервисам операционной системы, посылая сообщение микроядра. Вот этим мы детально и займёмся в следующем разделе.

А сейчас, до конца этого раздела, остановимся на маленькой детали, которая поможет вам находить самостоятельно файлы нужных вам реализаций. Почему в показанном выше коде точка входа названа `_getpid`, а не `getpid` ? Заглянем:

```
# ls /usr/src/lib/syscall
close.s      fstatfs.s     getuid.s      pipe.s      sigfillset.s   time.s
closedir.s   getcwd.s      ioctl.s       ptrace.s    sigismember.s  times.s
cprofile.s   getdents.s    isatty.s      read.s      sigpending.s   truncate.s
creat.s      getdma.s      kill.s        readdir.s   sigprocmask.s  umask.s
deldma.s     getegid.s     killpg.s      readlink.s  sigreturn.s    umount.s
devctl.s     geteuid.s     link.s        reboot.s    sigsuspend.s   uname.s
...
_exit.s       dup.s        getgid.s      lseek.s     rename.s     sleep.s     unlink.s
...
cfsetispeed.s fchmod.s     getpid.s      mount.s     setsid.s    tcdrain.s    vm_unmap.s
cfsetospeed.s fchown.s     getppid.s     nanosleep.s setuid.s    tcflow.s     wait.s
chdir.s       fcntl.s      getpprocnr.s  open.s      sigaction.s tcflush.s    waitpid.s
chmod.s       fork.s       getprocnr.s   opendir.s   sigaddset.s tcgetattr.s  write.s
...
```

Вот здесь — все интересные нам для дальнейшего рассмотрения вызовы (из стандарта POSIX): `getpid()`, `fork()`, `open()`, уже встречавшийся нам выше `_exit` , и теперь понятно, почему он переадресовался в имя с подчёркиванием. Смотрим несколько из образцов:

```
# cat /usr/src/lib/syscall/getpid.s
.sect .text
.extern __getpid
.define _getpid
```

```
.align 2

_getpid:
        jmp     __getpid


# cat /usr/src/lib/syscall/fork.s
.sect .text
.extern __fork
.define _fork
.align 2

_fork:
        jmp     __fork


# cat /usr/src/lib/syscall/_exit.s
.sect .text
.extern ___exit
.define __exit
.align 2

__exit:
        jmp     ___exit
```

Это только перетрансляция вызовов! А сами коды реализации находим здесь :

```
# ls /usr/src/lib/posix
_chroot.c     _fpathconf.c  _kill.c       _read.c       _sigsuspend.c    _uname.c
_close.c      _fstat.c      _killpg.c     _readdir.c    _sleep.c         _unlink.c
_closedir.c   _fstatfs.c    _link.c       _readlink.c   _stat.c          _utime.c
_creat.c      _fsync.c      _lseek.c      _rename.c     _stime.c         _wait.c
_dup.c        _getcwd.c     _lstat.c      _rewinddir.c  _symlink.c       _waitpid.c
_dup2.c       _getegid.c    _mkdir.c      _rmdir.c      _sync.c          _write.c
__exit.c      _execl.c      _geteuid.c    _mkfifo.c     _select.c        _tcdrain.c
_access.c     _execle.c     _getgid.c     _mknod.c      _setgid.c        _tcflow.c
_alarm.c      _execlp.c     _getgroups.c  _mmap.c       _setitimer.c     _tcflush.c
_cfgetispeed.c _execv.c     _getitimer.c  _nanosleep.c  _setsid.c        _tcgetattr.c
_cfgetospeed.c _execve.c    _getpgrp.c    _open.c       _setuid.c        _tcsendbreak.c
_cfsetispeed.c _execvp.c    _getpid.c     _opendir.c    _sigaction.c     _tcsetattr.c
_cfsetospeed.c _fchmod.c    _getppid.c    _pathconf.c   _sigpending.c    _time.c
_chdir.c      _fchown.c     _getuid.c     _pause.c      _sigprocmask.c   _times.c
_chmod.c      _fcntl.c      _ioctl.c      _pipe.c       _sigreturn.c     _truncate.c
_chown.c      _fork.c       _isatty.c     _ptrace.c     _sigset.c        _umask.c
```

Вот теперь всё стало на свои места, и можем переходить к системным вызовам.

Системные вызовы

Повторю код реализации программного вызова `getpid()`, на котором мы остановились:

```
# cat /usr/src/lib/posix/_getpid.c
#include <lib.h>
#define getpid  _getpid
#include <unistd.h>

PUBLIC pid_t getpid()
{
  message m;
  return(_syscall(MM, MINIX_GETPID, &m));
}
```

Убеждаемся, что `_syscall()`— это ни что иное, как посылка сообщения микроядра:

```
# cat /usr/src/lib/other/syscall.c
#include <lib.h>

PUBLIC int _syscall(who, syscallnr, msgptr)
int who;
int syscallnr;
register message *msgptr;
{
```

```
  int status;

  msgptr->m_type = syscallnr;
  status = _sendrec(who, msgptr);
  if (status != 0) {
        /* 'sendrec' itself failed. */
        /* XXX - strerror doesn't know all the codes */
        msgptr->m_type = status;
  }
  if (msgptr->m_type < 0) {
        errno = -msgptr->m_type;
        return(-1);
  }
  return(msgptr->m_type);
}
```

Все реализации POSIX программных вызовов (через `_syscall()`), да и сама показанная выше реализация `_syscall()` ссылаются на `/usr/include/lib.h`, привожу его полностью:

`# cat /usr/include/lib.h`

```
/* The <lib.h> header is the master header used by the library.
 * All the C files in the lib subdirectories include it.
 */


#ifndef _LIB_H
#define _LIB_H


/* First come the defines. */
#define _POSIX_SOURCE      1    /* tell headers to include POSIX stuff */
#define _MINIX             1    /* tell headers to include MINIX stuff */


/* The following are so basic, all the lib files get them automatically. */
#include <minix/config.h>       /* must be first */
#include <sys/types.h>
#include <limits.h>
#include <errno.h>
#include <ansi.h>


#include <minix/const.h>
#include <minix/com.h>
#include <minix/type.h>
#include <minix/callnr.h>


#include <minix/ipc.h>


#define MM                 PM_PROC_NR
#define FS                 FS_PROC_NR


_PROTOTYPE( int __execve, (const char *_path, char *const _argv[],
                        char *const _envp[], int _nargs, int _nenvps)   );
_PROTOTYPE( int _syscall, (int _who, int _syscallnr, message *_msgptr)  );
_PROTOTYPE( void _loadname, (const char *_name, message *_msgptr)       );
```

```
_PROTOTYPE( int _len, (const char *_s)                                  );
_PROTOTYPE( void _begsig, (int _dummy)                                  );


#endif /* _LIB_H */
```

- здесь определяются для нас получатели, которым могут направляться сообщения-запросы, а сами константы адресатов определены в `/usr/src/include/minix/com.h`, это такой огромный файл определений, к которому мы ещё будем возвращаться:

```
# cat /usr/src/include/minix/com.h
...
/* User-space processes, that is, device drivers, servers, and INIT. */
#define PM_PROC_NR        0     /* process manager */
#define FS_PROC_NR        1     /* file system */
...
```

Вот теперь всё становится на свои места, и самое время вернуться к рассмотрению ещё нескольких, подобных `getpid()` запросов, каждый из них — это всё так же: подготовка и отсылка сообщения микроядра (`_syscall()`). Сосредоточимся теперь на деталях того, «куда» и «какого сообщения»:

`# cat /usr/src/lib/posix/_fork.c`

```
#include <lib.h>
#define fork    _fork
#include <unistd.h>

PUBLIC pid_t fork()
{
  message m;
  return(_syscall(MM, FORK, &m));
}
```

А вот операция определённо из области файловых манипуляций:

`# cat /usr/src/lib/posix/_open.c`

```
#include <lib.h>
#define open    _open
#include <fcntl.h>
#include <stdarg.h>
#include <string.h>

#if _ANSI
PUBLIC int open(const char *name, int flags, ...)
#else
PUBLIC int open(name, flags)
_CONST char *name;
int flags;
#endif
{
  va_list argp;
  message m;

  va_start(argp, flags);
  if (flags & O_CREAT) {
        m.m1_i1 = strlen(name) + 1;
        m.m1_i2 = flags;
        m.m1_i3 = va_arg(argp, _mnx_Mode_t);
        m.m1_p1 = (char *) name;
  } else {
        _loadname(name, &m);
        m.m3_i2 = flags;
  }
  va_end(argp);
  return (_syscall(FS, OPEN, &m));
}
```

- и сообщения направляются теперь не к MM (менеджер процессов, он же менеджер памяти), а к FS (менеджер файловых систем).

Вот наш многострадальный `exit()`, в его рассмотрении мы дошли до самого низа, до отправки сообщения нескольких подобных, представляющих интерес, каждый из них — это подготовка и отсылка сообщения `EXIT`:

`# cat /usr/src/lib/posix/__exit.c`

```
#define _exit   __exit
#include <lib.h>
#include <unistd.h>

PUBLIC void _exit(status)
int status;
{
  void (*suicide)(void);
  message m;

  m.m1_i1 = status;
  _syscall(MM, EXIT, &m);

  /* If exiting nicely through PM fails for some reason, try to
   * commit suicide. E.g., message to PM might fail due to deadlock.
   */
  suicide = (void (*)(void)) -1;
  suicide();

  /* If committing suicide fails for some reason, hang. */
  for(;;) { }
}
```

Здесь же, в каталоге, интересный файл `priority.c`, с реализацией функций манипуляции приоритетами, на который я обращаю отдельное внимание:

```
# cat /usr/src/lib/posix/priority.c
int getpriority(int which, int who)
{
        int v;
        message m;

        m.m1_i1 = which;
        m.m1_i2 = who;

        if((v = _syscall(MM, GETPRIORITY, &m)) < 0) {
                return v;
        }

        return v + PRIO_MIN;
}

int setpriority(int which, int who, int prio)
{
        message m;

        m.m1_i1 = which;
        m.m1_i2 = who;
        m.m1_i3 = prio;

        return _syscall(MM, SETPRIORITY, &m);
}
```

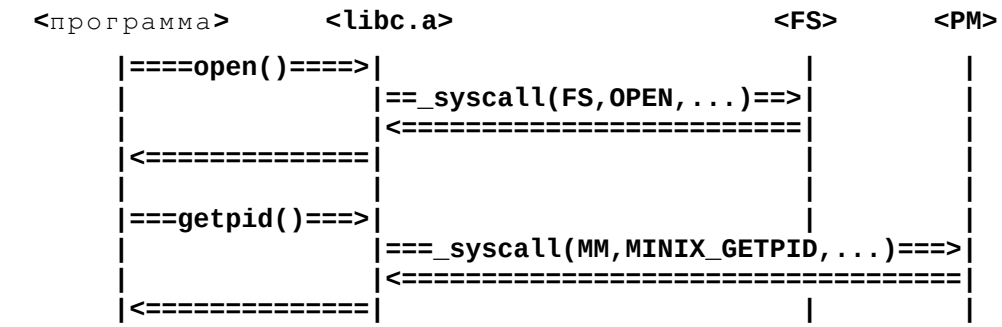

Рисунок 2. Обслуживание программных вызовов, когда они ретранслируются библиотекой `libc.a` в системные вызовы.

Ну и, наконец, для полной ясности: а что же относительно второго `_syscall()` параметра, который являются

кодом запрашиваемой операции? Так вот же они все (список великоват, но он стоит того, чтобы его привести полностью, и не рыскать позже по файлам):

 # cat /usr/include/minix/callnr.h

```
#define NCALLS           111    /* number of system calls allowed */


#define EXIT              1
#define FORK              2
#define READ              3
#define WRITE             4
#define OPEN              5
#define CLOSE             6
#define WAIT              7
#define CREAT             8
#define LINK              9
#define UNLINK           10
#define WAITPID          11
#define CHDIR            12
#define TIME             13
#define MKNOD            14
#define CHMOD            15
#define CHOWN            16
#define BRK              17
#define STAT             18
#define LSEEK            19
#define MINIX_GETPID     20
#define MOUNT            21
#define UMOUNT           22
#define SETUID           23
#define GETUID           24
#define STIME            25
#define PTRACE           26
#define ALARM            27
#define FSTAT            28
#define PAUSE            29
#define UTIME            30
#define ACCESS           33
#define SYNC             36
#define KILL             37
#define RENAME           38
#define MKDIR            39
#define RMDIR            40
#define DUP              41
#define PIPE             42
#define TIMES            43
#define SYMLINK          45
```

```
#define SETGID            46
#define GETGID            47
#define SIGNAL            48
#define RDLNK             49
#define LSTAT             50
#define IOCTL             54
#define FCNTL             55
#define FS_READY          57
#define EXEC              59
#define UMASK             60
#define CHROOT            61
#define SETSID            62
#define GETPGRP           63
#define ITIMER            64

/* Posix signal handling. */
#define SIGACTION         71
#define SIGSUSPEND        72
#define SIGPENDING        73
#define SIGPROCMASK       74
#define SIGRETURN         75

#define REBOOT            76
#define SVRCTL            77
#define SYSUNAME          78
#define GETSYSINFO        79    /* to PM or FS */
#define GETDENTS          80    /* to FS */
#define LLSEEK            81    /* to FS */
#define FSTATFS           82    /* to FS */
#define SELECT            85    /* to FS */
#define FCHDIR            86    /* to FS */
#define FSYNC             87    /* to FS */
#define GETPRIORITY       88    /* to PM */
#define SETPRIORITY       89    /* to PM */
#define GETTIMEOFDAY      90    /* to PM */
#define SETEUID           91    /* to PM */
#define SETEGID           92    /* to PM */
#define TRUNCATE          93    /* to FS */
#define FTRUNCATE         94    /* to FS */
#define FCHMOD            95    /* to FS */
#define FCHOWN            96    /* to FS */
#define GETSYSINFO_UP     97    /* to PM or FS */
#define SPROF             98    /* to PM */
#define CPROF             99    /* to PM */
```

```
/* Calls provided by PM and FS that are not part of the API */
#define EXEC_NEWMEM     100     /* from FS or RS to PM: new memory map for
                                 * exec
                                 */
#define FORK_NB         101     /* to PM: special fork call for RS */
#define EXEC_RESTART    102     /* to PM: final part of exec for RS */
#define PROCSTAT        103     /* to PM */
#define GETPROCNR       104     /* to PM */
#define ALLOCMEM        105     /* to PM */
#if 0
#define FREEMEM         106     /* to PM, not used, not implemented */
#endif
#define GETPUID         107     /* to PM: get the uid of a process (endpoint) */
#define ADDDMA          108     /* to PM: inform PM about a region of memory
                                 * that is used for bus-master DMA
                                 */
#define DELDMA          109     /* to PM: inform PM that a region of memory
                                 * that is no longer used for bus-master DMA
                                 */
#define GETDMA          110     /* to PM: ask PM for a region of memory
                                 * that should not be used for bus-master DMA
                                 * any longer
                                 */

#define DEVCTL          120     /* to FS, map or unmap a device */
#define TASK_REPLY      121     /* to FS: reply code from drivers, not
                                 * really a standalone call.
                                 */
#define MAPDRIVER       122     /* to FS, map a device */
```

Подавляющее большинство программных вызовов (библиотеки C) мы покрыли уже рассмотренными механизмами. Но не все. В некоторых случаях, для разрешения полученного ими запроса, MM или FS будут нуждаться в информации ядра системы (таблиц ядра). Одним из характерных вызовов из такого числа является `fork()`, который мы активно рассматриваем. Но они не могут (и не имеют права) пользоваться такой информацией. Поэтому они направляют сообщение-запрос (вызов ядра) к задаче ядра `system`:

```
# ps -axl | head -n 7
  F S UID   PID  PPID  PGRP     SZ         RECV TTY  TIME CMD
  0 R   0  (-4)     0     0      0                ?  8h13 idle
 10 W   0  (-3)     0     0      0          ANY   ?  0:20 clock
  0 R   0  (-2)     0     0      0                ? 28:25 system
  2 W   0  (-1)     0     0      0                ?  0:00 kernel
 10 W   0     0     0   104      0       system   ?  1:03 pm
 10 W   0     4     0     0      0          ANY   ?  3:47 vfs
```

Вызовы ядра

Утверждается [2], что:

Действительные реализации вызовов ядра определены в одной из задач ядра. В противоположность MINIX2, задача `CLOCK` более не принимает системных вызовов. Вместо этого, все вызовы теперь направляются к задаче `SYSTEM`. Предполагается, что программа делает `sys_call()` системный вызов. Согласно соглашениям, этот вызов трансформируется в сообщение запроса с типом `SYS_CALL`, которое посылается

задаче ядра SYSTEM. Задача SYSTEM обслуживает запрос в функции с именем do_call() и возвращает результат.

Константы-определения системных задач находим среди основного файла определений /usr/src/include/minix/com.h, который мы уже упоминали:

```
# cat /usr/src/include/minix/com.h
...

/*===========================================================================*
 *              Process numbers of processes in the system image             *
 *===========================================================================*/

/* The values of several task numbers depend on whether they or other tasks
 * are enabled. They are defined as (PREVIOUS_TASK - ENABLE_TASK) in general.
 * ENABLE_TASK is either 0 or 1, so a task either gets a new number, or gets
 * the same number as the previous task and is further unused. Note that the
 * order should correspond to the order in the task table defined in table.c.
 */

/* Kernel tasks. These all run in the same address space. */
#define IDLE             -4     /* runs when no one else can run */
#define CLOCK            -3     /* alarms and other clock functions */
#define SYSTEM           -2     /* request system functionality */
#define KERNEL           -1     /* pseudo-process for IPC and scheduling */
#define HARDWARE     KERNEL     /* for hardware interrupt handlers */

/* Number of tasks. Note that NR_PROCS is defined in <minix/config.h>. */
#define MAX_NR_TASKS    1023
#define NR_TASKS          4

/* User-space processes, that is, device drivers, servers, and INIT. */
#define PM_PROC_NR        0     /* process manager */
#define FS_PROC_NR        1     /* file system */
#define VFS_PROC_NR       FS_PROC_NR /* FS has been renamed to VFS. */
#define RS_PROC_NR        2     /* memory driver (RAM disk, null, etc.) */
#define MEM_PROC_NR       3     /* memory driver (RAM disk, null, etc.) */
#define LOG_PROC_NR       4     /* log device driver */
#define TTY_PROC_NR       5     /* terminal (TTY) driver */
#define DS_PROC_NR        6     /* data store server */
#define MFS_PROC_NR       7     /* minix root filesystem */
#define VM_PROC_NR        8     /* memory server */
#define INIT_PROC_NR      9     /* init -- goes multiuser */

/* Number of processes contained in the system image. */
#define NR_BOOT_PROCS   (NR_TASKS + INIT_PROC_NR + 1)
...
```

Здесь же находим и полный перечень системных вызовов, реализованных в текущей редакции MINIX3:

```
# cat /usr/src/include/minix/com.h
...

/*===========================================================================*
 *                  SYSTASK request types and field names                    *
 *===========================================================================*/

/* System library calls are dispatched via a call vector, so be careful when
 * modifying the system call numbers. The numbers here determine which call
 * is made from the call vector.
 */
#define KERNEL_CALL     0x600   /* base for kernel calls to SYSTEM */

#  define SYS_FORK       (KERNEL_CALL + 0)      /* sys_fork() */
#  define SYS_EXEC       (KERNEL_CALL + 1)      /* sys_exec() */
#  define SYS_EXIT       (KERNEL_CALL + 2)      /* sys_exit() */
#  define SYS_NICE       (KERNEL_CALL + 3)      /* sys_nice() */
#  define SYS_PRIVCTL    (KERNEL_CALL + 4)      /* sys_privctl() */
#  define SYS_TRACE      (KERNEL_CALL + 5)      /* sys_trace() */
#  define SYS_KILL       (KERNEL_CALL + 6)      /* sys_kill() */
```

```
#  define SYS_GETKSIG    (KERNEL_CALL + 7)      /* sys_getsig() */
#  define SYS_ENDKSIG    (KERNEL_CALL + 8)      /* sys_endsig() */
#  define SYS_SIGSEND    (KERNEL_CALL + 9)      /* sys_sigsend() */
#  define SYS_SIGRETURN  (KERNEL_CALL + 10)     /* sys_sigreturn() */

#  define SYS_NEWMAP     (KERNEL_CALL + 11)     /* sys_newmap() */
#  define SYS_SEGCTL     (KERNEL_CALL + 12)     /* sys_segctl() */
#  define SYS_MEMSET     (KERNEL_CALL + 13)     /* sys_memset() */

#  define SYS_UMAP       (KERNEL_CALL + 14)     /* sys_umap() */
#  define SYS_VIRCOPY    (KERNEL_CALL + 15)     /* sys_vircopy() */
#  define SYS_PHYSCOPY   (KERNEL_CALL + 16)     /* sys_physcopy() */
#  define SYS_VIRVCOPY   (KERNEL_CALL + 17)     /* sys_virvcopy() */
#  define SYS_PHYSVCOPY  (KERNEL_CALL + 18)     /* sys_physvcopy() */

#  define SYS_IRQCTL     (KERNEL_CALL + 19)     /* sys_irqctl() */
#  define SYS_INT86      (KERNEL_CALL + 20)     /* sys_int86() */
#  define SYS_DEVIO      (KERNEL_CALL + 21)     /* sys_devio() */
#  define SYS_SDEVIO     (KERNEL_CALL + 22)     /* sys_sdevio() */
#  define SYS_VDEVIO     (KERNEL_CALL + 23)     /* sys_vdevio() */

#  define SYS_SETALARM   (KERNEL_CALL + 24)     /* sys_setalarm() */
#  define SYS_TIMES      (KERNEL_CALL + 25)     /* sys_times() */
#  define SYS_GETINFO    (KERNEL_CALL + 26)     /* sys_getinfo() */
#  define SYS_ABORT      (KERNEL_CALL + 27)     /* sys_abort() */
#  define SYS_IOPENABLE  (KERNEL_CALL + 28)     /* sys_enable_iop() */
#  define SYS_SAFECOPYFROM (KERNEL_CALL + 31)   /* sys_safecopyfrom() */
#  define SYS_SAFECOPYTO   (KERNEL_CALL + 32)   /* sys_safecopyto() */
#  define SYS_VSAFECOPY  (KERNEL_CALL + 33)     /* sys_vsafecopy() */
#  define SYS_SETGRANT   (KERNEL_CALL + 34)     /* sys_setgrant() */
#  define SYS_READBIOS   (KERNEL_CALL + 35)     /* sys_readbios() */

#  define SYS_SPROF      (KERNEL_CALL + 36)     /* sys_sprof() */
#  define SYS_CPROF      (KERNEL_CALL + 37)     /* sys_cprof() */
#  define SYS_PROFBUF    (KERNEL_CALL + 38)     /* sys_profbuf() */

#  define SYS_STIME     (KERNEL_CALL + 39)      /* sys_stime() */

#  define SYS_MAPDMA    (KERNEL_CALL + 42)      /* sys_mapdma() */
#  define SYS_VMCTL     (KERNEL_CALL + 43)      /* sys_vmctl() */
#  define SYS_SYSCTL    (KERNEL_CALL + 44)      /* sys_sysctl() */

#  define SYS_VTIMER     (KERNEL_CALL + 45)     /* sys_vtimer() */
#  define SYS_RUNCTL     (KERNEL_CALL + 46)     /* sys_runctl() */

#define NR_SYS_CALLS    47      /* number of system calls */
...
```

Каждый системный вызов определяется как реализующая функция вида `sys_*()`, на самом же деле, каждая такая функция делает вызов функции отправки сообщения запроса `_taskcall()` с соответствующими параметрами (которая практически не отличается от функции `_syscall()`, которую мы её уже видели выше[2]). Тип запроса `_taskcall()` при этом (2-й параметр) будет соответствовать запрошенной операции (например SYS_FORK), а адресатом (получателем) является SYSTASK - системная задача `system`. Но прежде чем детально рассматривать процесс отправки сообщения, задержимся на реализующих вызовы ядра функциях вида `sys_*()`. Прототипы всех функций определены в файле `/usr/src/include/minix/syslib.h`:

```
# cat /usr/src/include/minix/syslib.h
/* Prototypes for system library functions. */
...
#define SYSTASK SYSTEM

/*===========================================================================*
 * Minix system library.                                                     *
 *===========================================================================*/
_PROTOTYPE( int _taskcall, (endpoint_t who, int syscallnr, message *msgptr));
```

2 Здесь в цитируемом выше фрагменте [2] содержится неточность, вызов ядра производится через вызов `_taskcall()`, а не через `sys_call()`; близкое подобие кода — не означает их идентичность.

```
_PROTOTYPE( int sys_abort, (int how, ...));
_PROTOTYPE( int sys_enable_iop, (endpoint_t proc_ep));
_PROTOTYPE( int sys_exec, (endpoint_t proc_ep, char *ptr,
                                char *aout, vir_bytes initpc));
_PROTOTYPE( int sys_fork, (endpoint_t parent, endpoint_t child, endpoint_t *,
        struct mem_map *ptr, u32_t vm, vir_bytes *));
_PROTOTYPE( int sys_newmap, (endpoint_t proc_ep, struct mem_map *ptr));
_PROTOTYPE( int sys_exit, (endpoint_t proc_ep));
_PROTOTYPE( int sys_trace, (int req, endpoint_t proc_ep, long addr, long *data_p));

/* Shorthands for sys_runctl() system call. */
#define sys_stop(proc_ep) sys_runctl(proc_ep, RC_STOP, 0)
#define sys_delay_stop(proc_ep) sys_runctl(proc_ep, RC_STOP, RC_DELAY)
#define sys_resume(proc_ep) sys_runctl(proc_ep, RC_RESUME, 0)
_PROTOTYPE( int sys_runctl, (endpoint_t proc_ep, int action, int flags));

_PROTOTYPE( int sys_privctl, (endpoint_t proc_ep, int req, void *p));
_PROTOTYPE( int sys_setgrant, (cp_grant_t *grants, int ngrants));
_PROTOTYPE( int sys_nice, (endpoint_t proc_ep, int priority));

_PROTOTYPE( int sys_int86, (struct reg86u *reg86p));
_PROTOTYPE( int sys_vm_setbuf, (phys_bytes base, phys_bytes size,
                                                        phys_bytes high));
_PROTOTYPE( int sys_vm_map, (endpoint_t proc_ep, int do_map,
        phys_bytes base, phys_bytes size, phys_bytes offset));
_PROTOTYPE( int sys_vmctl, (endpoint_t who, int param, u32_t value));
_PROTOTYPE( int sys_vmctl_get_pagefault_i386, (endpoint_t *who, u32_t *cr2, u32_t *err));
_PROTOTYPE( int sys_vmctl_get_cr3_i386, (endpoint_t who, u32_t *cr3)  );
_PROTOTYPE( int sys_vmctl_get_memreq, (endpoint_t *who, vir_bytes *mem,
        vir_bytes *len, int *wrflag, endpoint_t *) );
_PROTOTYPE( int sys_vmctl_enable_paging, (struct mem_map *));

_PROTOTYPE( int sys_readbios, (phys_bytes address, void *buf, size_t size));
_PROTOTYPE( int sys_stime, (time_t boottime));
_PROTOTYPE( int sys_sysctl, (int ctl, char *arg1, int arg2));
_PROTOTYPE( int sys_sysctl_stacktrace, (endpoint_t who));

/* Shorthands for sys_sdevio() system call. */
#define sys_insb(port, proc_ep, buffer, count) \
  sys_sdevio(DIO_INPUT_BYTE, port, proc_ep, buffer, count, 0)
#define sys_insw(port, proc_ep, buffer, count) \
  sys_sdevio(DIO_INPUT_WORD, port, proc_ep, buffer, count, 0)
#define sys_outsb(port, proc_ep, buffer, count) \
  sys_sdevio(DIO_OUTPUT_BYTE, port, proc_ep, buffer, count, 0)
#define sys_outsw(port, proc_ep, buffer, count) \
  sys_sdevio(DIO_OUTPUT_WORD, port, proc_ep, buffer, count, 0)
#define sys_safe_insb(port, ept, grant, offset, count) \
  sys_sdevio(DIO_SAFE_INPUT_BYTE, port, ept, (void*)grant, count, offset)
#define sys_safe_outsb(port, ept, grant, offset, count) \
  sys_sdevio(DIO_SAFE_OUTPUT_BYTE, port, ept, (void*)grant, count, offset)
#define sys_safe_insw(port, ept, grant, offset, count) \
  sys_sdevio(DIO_SAFE_INPUT_WORD, port, ept, (void*)grant, count, offset)
#define sys_safe_outsw(port, ept, grant, offset, count) \
  sys_sdevio(DIO_SAFE_OUTPUT_WORD, port, ept, (void*)grant, count, offset)
_PROTOTYPE( int sys_sdevio, (int req, long port, endpoint_t proc_ep,
        void *buffer, int count, vir_bytes offset));
_PROTOTYPE(void *alloc_contig, (size_t len, int flags, phys_bytes *phys));
#define AC_ALIGN4K      0x01
#define AC_LOWER16M     0x02
#define AC_ALIGN64K     0x04
#define AC_LOWER1M      0x08

/* Clock functionality: get system times, (un)schedule an alarm call, or
 * retrieve/set a process-virtual timer.
 */
_PROTOTYPE( int sys_times, (endpoint_t proc_ep, clock_t *user_time,
        clock_t *sys_time, clock_t *uptime));
_PROTOTYPE(int sys_setalarm, (clock_t exp_time, int abs_time));
_PROTOTYPE( int sys_vtimer, (endpoint_t proc_nr, int which, clock_t *newval,
        clock_t *oldval));
```

```
/* Shorthands for sys_irqctl() system call. */
#define sys_irqdisable(hook_id) \
    sys_irqctl(IRQ_DISABLE, 0, 0, hook_id)
#define sys_irqenable(hook_id) \
    sys_irqctl(IRQ_ENABLE, 0, 0, hook_id)
#define sys_irqsetpolicy(irq_vec, policy, hook_id) \
    sys_irqctl(IRQ_SETPOLICY, irq_vec, policy, hook_id)
#define sys_irqrmpolicy(hook_id) \
    sys_irqctl(IRQ_RMPOLICY, 0, 0, hook_id)
_PROTOTYPE ( int sys_irqctl, (int request, int irq_vec, int policy,
    int *irq_hook_id) );

/* Shorthands for sys_vircopy() and sys_physcopy() system calls. */
#define sys_biosin(bios_vir, dst_vir, bytes) \
        sys_vircopy(SELF, BIOS_SEG, bios_vir, SELF, D, dst_vir, bytes)
#define sys_biosout(src_vir, bios_vir, bytes) \
        sys_vircopy(SELF, D, src_vir, SELF, BIOS_SEG, bios_vir, bytes)
#define sys_datacopy(src_proc, src_vir, dst_proc, dst_vir, bytes) \
        sys_vircopy(src_proc, D, src_vir, dst_proc, D, dst_vir, bytes)
#define sys_textcopy(src_proc, src_vir, dst_proc, dst_vir, bytes) \
        sys_vircopy(src_proc, T, src_vir, dst_proc, T, dst_vir, bytes)
#define sys_stackcopy(src_proc, src_vir, dst_proc, dst_vir, bytes) \
        sys_vircopy(src_proc, S, src_vir, dst_proc, S, dst_vir, bytes)
_PROTOTYPE(int sys_vircopy, (endpoint_t src_proc, int src_s, vir_bytes src_v,
        endpoint_t dst_proc, int dst_seg, vir_bytes dst_vir, phys_bytes bytes));

#define sys_abscopy(src_phys, dst_phys, bytes) \
        sys_physcopy(NONE, PHYS_SEG, src_phys, NONE, PHYS_SEG, dst_phys, bytes)
_PROTOTYPE(int sys_physcopy, (endpoint_t src_proc, int src_seg, vir_bytes src_vir,
        endpoint_t dst_proc, int dst_seg, vir_bytes dst_vir, phys_bytes bytes));


/* Grant-based copy functions. */
_PROTOTYPE(int sys_safecopyfrom, (endpoint_t source, cp_grant_id_t grant,
        vir_bytes grant_offset, vir_bytes my_address, size_t bytes, int my_seg));
_PROTOTYPE(int sys_safecopyto, (endpoint_t dest, cp_grant_id_t grant,
        vir_bytes grant_offset, vir_bytes my_address, size_t bytes, int my_seg));
_PROTOTYPE(int sys_vsafecopy, (struct vscp_vec *copyvec, int elements));

_PROTOTYPE(int sys_memset, (unsigned long pattern,
                phys_bytes base, phys_bytes bytes));


/* Vectored virtual / physical copy calls. */
#if DEAD_CODE           /* library part not yet implemented */
_PROTOTYPE(int sys_virvcopy, (phys_cp_req *vec_ptr,int vec_size,int *nr_ok));
_PROTOTYPE(int sys_physvcopy, (phys_cp_req *vec_ptr,int vec_size,int *nr_ok));
#endif

_PROTOTYPE(int sys_umap, (endpoint_t proc_ep, int seg, vir_bytes vir_addr,
         vir_bytes bytes, phys_bytes *phys_addr));
_PROTOTYPE(int sys_umap_data_fb, (endpoint_t proc_ep, vir_bytes vir_addr,
         vir_bytes bytes, phys_bytes *phys_addr));
_PROTOTYPE(int sys_segctl, (int *index, u16_t *seg, vir_bytes *off,
        phys_bytes phys, vir_bytes size));

/* Shorthands for sys_getinfo() system call. */
#define sys_getkmessages(dst)   sys_getinfo(GET_KMESSAGES, dst, 0,0,0)
#define sys_getkinfo(dst)       sys_getinfo(GET_KINFO, dst, 0,0,0)
#define sys_getloadinfo(dst)    sys_getinfo(GET_LOADINFO, dst, 0,0,0)
#define sys_getmachine(dst)     sys_getinfo(GET_MACHINE, dst, 0,0,0)
#define sys_getproctab(dst)     sys_getinfo(GET_PROCTAB, dst, 0,0,0)
#define sys_getprivtab(dst)     sys_getinfo(GET_PRIVTAB, dst, 0,0,0)
#define sys_getproc(dst,nr)     sys_getinfo(GET_PROC, dst, 0,0, nr)
#define sys_getrandomness(dst)  sys_getinfo(GET_RANDOMNESS, dst, 0,0,0)
#define sys_getrandom_bin(d,b)  sys_getinfo(GET_RANDOMNESS_BIN, d, 0,0,b)
#define sys_getimage(dst)       sys_getinfo(GET_IMAGE, dst, 0,0,0)
#define sys_getirqhooks(dst)    sys_getinfo(GET_IRQHOOKS, dst, 0,0,0)
#define sys_getirqactids(dst)   sys_getinfo(GET_IRQACTIDS, dst, 0,0,0)
#define sys_getmonparams(v,vl)  sys_getinfo(GET_MONPARAMS, v,vl, 0,0)
#define sys_getschedinfo(v1,v2) sys_getinfo(GET_SCHEDINFO, v1,0, v2,0)
```

```
#define sys_getlocktimings(dst) sys_getinfo(GET_LOCKTIMING, dst, 0,0,0)
#define sys_getprivid(nr)       sys_getinfo(GET_PRIVID, 0, 0,0, nr)
_PROTOTYPE(int sys_getinfo, (int request, void *val_ptr, int val_len,
                                void *val_ptr2, int val_len2)          );
_PROTOTYPE(int sys_whoami, (endpoint_t *ep, char *name, int namelen));

/* Signal control. */
_PROTOTYPE(int sys_kill, (endpoint_t proc_ep, int sig) );
_PROTOTYPE(int sys_sigsend, (endpoint_t proc_ep, struct sigmsg *sig_ctxt) );
_PROTOTYPE(int sys_sigreturn, (endpoint_t proc_ep, struct sigmsg *sig_ctxt) );
_PROTOTYPE(int sys_getksig, (endpoint_t *proc_ep, sigset_t *k_sig_map) );
_PROTOTYPE(int sys_endksig, (endpoint_t proc_ep) );

/* NOTE: two different approaches were used to distinguish the device I/O
 * types 'byte', 'word', 'long': the latter uses #define and results in a
 * smaller implementation, but looses the static type checking.
 */
_PROTOTYPE(int sys_voutb, (pvb_pair_t *pvb_pairs, int nr_ports)         );
_PROTOTYPE(int sys_voutw, (pvw_pair_t *pvw_pairs, int nr_ports)         );
_PROTOTYPE(int sys_voutl, (pvl_pair_t *pvl_pairs, int nr_ports)         );
_PROTOTYPE(int sys_vinb, (pvb_pair_t *pvb_pairs, int nr_ports)          );
_PROTOTYPE(int sys_vinw, (pvw_pair_t *pvw_pairs, int nr_ports)          );
_PROTOTYPE(int sys_vinl, (pvl_pair_t *pvl_pairs, int nr_ports)          );

/* Shorthands for sys_out() system call. */
#define sys_outb(p,v)   sys_out((p), (unsigned long) (v), _DIO_BYTE)
#define sys_outw(p,v)   sys_out((p), (unsigned long) (v), _DIO_WORD)
#define sys_outl(p,v)   sys_out((p), (unsigned long) (v), _DIO_LONG)
_PROTOTYPE(int sys_out, (int port, unsigned long value, int type)       );

/* Shorthands for sys_in() system call. */
#define sys_inb(p,v)    sys_in((p), (v), _DIO_BYTE)
#define sys_inw(p,v)    sys_in((p), (v), _DIO_WORD)
#define sys_inl(p,v)    sys_in((p), (v), _DIO_LONG)
_PROTOTYPE(int sys_in, (int port, unsigned long *value, int type)       );
...
```

Файлы реализации системных вызовов (а точнее код трансформации вызовов функций вида `sys_*()` в вызов `_taskcall()`) находим в каталоге `/usr/src/lib/syslib` :

```
# ls /usr/src/lib/syslib
...
sys_abort.c     sys_newmap.c      sys_sprof.c       sys_vtimer.c
sys_cprof.c     sys_nice.c        sys_stime.c       syslib.h
sys_endsig.c    sys_out.c         sys_sysctl.c      taskcall.c
sys_eniop.c     sys_physcopy.c    sys_times.c
sys_exec.c      sys_privctl.c     sys_trace.c
sys_exit.c      sys_profbuf.c     sys_umap.c
sys_fork.c      sys_readbios.c    sys_vinb.c
sys_getinfo.c   sys_runctl.c      sys_vinl.c
sys_getsig.c    sys_safecopy.c    sys_vinw.c
sys_in.c        sys_sdevio.c      sys_vircopy.c
sys_int86.c     sys_segctl.c      sys_vmctl.c
sys_irqctl.c    sys_setalarm.c    sys_voutb.c
sys_kill.c      sys_setgrant.c    sys_voutl.c
sys_mapdma.c    sys_sigreturn.c   sys_voutw.c
sys_memset.c    sys_sigsend.c     sys_vsafecopy.c
...
```

Здесь же находится и файл `taskcall.c`, содержащий собственно код `_taskcall()`:

```
# cat /usr/src/lib/syslib/taskcall.c
PUBLIC int _taskcall(who, syscallnr, msgptr)
endpoint_t who;
int syscallnr;
register message *msgptr;
{
  int status;

  msgptr->m_type = syscallnr;
  status = _sendrec(who, msgptr);
```

```
  if (status != 0) return(status);
  return(msgptr->m_type);
}
```

Смотрим реализации вызовов ядра для некоторых, знакомых нам уже по предыдущему рассмотрению, функций:

```
# cat /usr/src/lib/syslib/sys_fork.c
PUBLIC int sys_fork(parent, child, child_endpoint, map_ptr, flags, msgaddr)
endpoint_t parent;              /* process doing the fork */
endpoint_t child;               /* which proc has been created by the fork */
endpoint_t *child_endpoint;
struct mem_map *map_ptr;
u32_t flags;
vir_bytes *msgaddr;
{
/* A process has forked.  Tell the kernel. */

  message m;
  int r;

  m.PR_ENDPT = parent;
  m.PR_SLOT = child;
  m.PR_MEM_PTR = (char *) map_ptr;
  m.PR_FORK_FLAGS = flags;
  r = _taskcall(SYSTASK, SYS_FORK, &m);
  *child_endpoint = m.PR_ENDPT;
  *msgaddr = (vir_bytes) m.PR_FORK_MSGADDR;
  return r;
}

# cat /usr/src/lib/syslib/sys_exit.c
/*==========================================================================*
 *                                sys_exit                                  *
 *==========================================================================*/
PUBLIC int sys_exit(proc_ep)
endpoint_t proc_ep;                     /* which process has exited */
{
/* A process has exited. PM tells the kernel. In addition this call can be
 * used by system processes to directly exit without passing through the
 * PM. This should be used with care to prevent inconsistent PM tables.
 */
  message m;

  m.PR_ENDPT = proc_ep;
  return(_taskcall(SYSTASK, SYS_EXIT, &m));
}

# cat /usr/src/lib/syslib/sys_nice.c
/*==========================================================================*
 *                                sys_nice                                  *
 *==========================================================================*/
PUBLIC int sys_nice(endpoint_t proc_ep, int prio)
{
  message m;

  m.PR_ENDPT = proc_ep;
  m.PR_PRIORITY = prio;
  return(_taskcall(SYSTASK, SYS_NICE, &m));
}
```

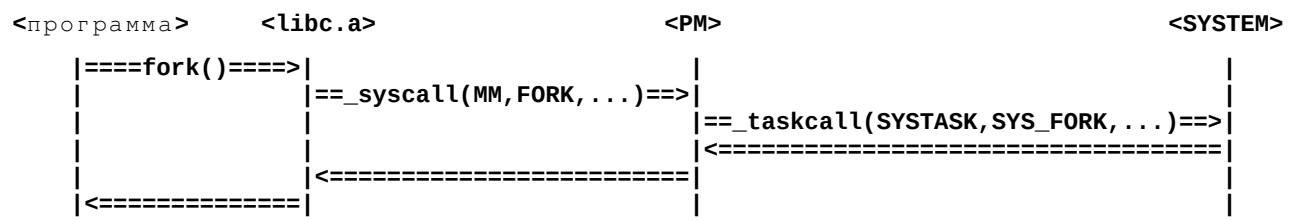

Рисунок 3. Программный вызов, когда он ретранслируется библиотекой `libc.a` в вызов системный, а тот, в свою очередь, производит вызов ядра.

Заключение

Изложенное выше прохождение программных вызовов в MINIX3, восстановлено только исходя из перечисленных далее нескольких публикаций, и анализа файлов исходного кода MINIX3, особенно комментариев в них. Никакой иной дополнительной информацией автор не обладал. Поэтому вполне могли возникнуть те или иные неточности в толковании происходящих процессов. Но на общую изложенную картину вряд ли эти неточности могут наложить существенный отпечаток.

Не проработанными, в настоящей редакции, являются вопросы:

– проведен анализ прохождения программных вызовов, но не проделан синтез: практические примеры программного кода, который использует описываемые механизмы напрямую, минуя некоторые промежуточные интерфейсы с верхнего уровня.

– вызовы ядра рассмотрены в том виде, как их осуществляют PM и FS; совершенно вероятно, что их непосредственно могут выполнять и ординарные программы, возможно, запросив и получив некоторые дополнительные привилегии, как это происходит при запуске сервиса командой: `service <programm_path>`. Хотелось бы на примерах кода рассмотреть такую возможность.

Я оставляю эти вопросы на проработку себе в последующих редакциях текста.

Источники информации

1. Pablo Andrés Pessolani , «MINIX 3.X: Local Bindery Service»

http://sites.google.com/site/minix3projects/MINIX3-BIND-REPORT.pdf

Перевод: http://www.minix3.ru/docs/LocalBindery-05.pdf

2. Jorrit N. Herder , «MINIX 3 Kernel API»

http://www.minix3.ru/docs/kernel-api.pdf

Перевод: http://www.minix3.ru/docs/KernelAPI-07.pdf

3. Niek Linnenbank , «Programming Device Drivers in Minix»

http://wiki.minix3.org/en/DevelopersGuide/DriverProgramming

Перевод: http://www.minix3.ru/docs/DriverProgramming-05.pdf

4. Matej Košík «Faun: Device Driver which Deals with PCI»

http://www.altair.sk/mediawiki/upload/d/d1/Faun.pdf

Перевод: http://www.minix3.ru/docs/DriverPCI-02.pdf