How to Add a New System Call for Minix 3

Karthick Jayaraman
Department of Electrical Engineering & Computer Science
Syracuse University, Syracuse, New York

1. Introduction

Minix3 has the micro-kernel architecture. The micro-kernel handles interrupts, provides basic mechanisms for process management, implements inter-process communication, and performs process scheduling. Filesystem, process management, networking, and other user-services are available from separate servers outside the micro-kernel. The system calls handled by these services are now processed outside the kernel. The kernel supports a few system-calls and these are called system-tasks. Systems-tasks are more like a hardware abstraction.

In Minix3, the servers handle system calls. Adding a new system call consists of two steps: writing a system-call handler and writing a user library. System-call handler is a function that is called in response to a user requesting a system call. Each system call has one handler. A user library packages the parameters for the system call and calls the handler on the appropriate server. A user always invokes a system call using the library.

The system-call handler should be placed in an appropriate server, which in turn would process a user request by invoking the matching handler. It is important to choose the correct server for the system-call. For instance, if the system call should update filesystem or fproc data-structures, then the system-call handler should be placed in the FS (filesystem) server.

This document illustrates the method for adding a new system call for Minix3 using an example. We would implement a system-call handler do_printmessage() in the FS server that would simply print a message “I am a system call”. However, the method described could be used for adding the handler to any server. We would also add a user-library to call the handler.

2. Creating a System-call Handler

The source code for all servers are located at /usr/src/servers. Each server has a separate directory. Filesystem (FS) is located at /usr/src/servers/fs. Each of the server source directories contain two files: table.c and proto.h. Table.c contains definition for the call_vec table. The call_vec table is an array of function pointers that is indexed by the system-call number. In each line, the address of a system-call handler function is assigned to one entry in the table and the index of the entry is the system-call number.

```
PUBLIC _PROTOTYPE (int (*call_vec[]), (void) ) = {
        no_sys,         /*  0 = unused      */
        do_exit,        /*  1 = exit        */
        do_fork,        /*  2 = fork        */
        do_read,        /*  3 = read        */
        do_write,       /*  4 = write       */
        do_open,        /*  5 = open        */
        do_close,       /*  6 = close       */
        no_sys,         /*  7 = wait        */
        do_creat,       /*  8 = creat       */
```

Figure 1: Some entries from /usr/src/servers/fs/table.c

Figure 1 contains a few entries from /usr/src/servers/fs/table.c. The second line in the table assigns the address of function do_exit to the second entry in the table. The index of the second entry, which is the number 2, is the system-call number for calling the handler do_exit.

```
        do_unpause,     /* 65 = UNPAUSE*/
        no_sys,         /* 66 = unused  */
        do_revive,      /* 67 = REVIVE    */
        no_sys,         /* 68 = TASK_REPLY      */
        no_sys,         /* 69 = unused */
        no_sys,         /* 70 = unused */
        no_sys,         /* 71 = si */
        no_sys,         /* 72 = sigsuspend */
        no_sys,         /* 73 = sigpending */
        no_sys,         /* 74 = sigprocmask */
```

Figure 2: Unused entries

There are a few unused entries. For adding a new system call, we need to identify one unused entry. For instance, index 69 contains an unused entry. We could use slot number 69 for our system-call handler do_printmessage(). To use entry 69, we replace no_sys with do_printmessage().

```
        do_revive,      /* 67 = REVIVE*/
        no_sys,         /* 68 = TASK_REPLY*/
do_printmessage(),      /* 69 = unused */
        no_sys,         /* 70 = unused */
        no_sys,         /* 71 = si */
```

Figure 3: Using entry 69

The next step is to declare a prototype of the system-call handler in file /usr/src/servers/fs/proto.h. This file contains the prototypes of all system-call handler functions. Figure 4 contains a few prototype declarations from /usr/src/servers/fs/proto.h. We should add the prototype for the system-call handler to the proto.h file.

```
_PROTOTYPE( int do_printmessage, (void) );
```

```
/* open.c */
_PROTOTYPE( int do_close, (void)                    );
_PROTOTYPE( int do_creat, (void)                    );
_PROTOTYPE( int do_lseek, (void)                    );
_PROTOTYPE( int do_mknod, (void)                    );
_PROTOTYPE( int do_mkdir, (void)                    );
_PROTOTYPE( int do_open, (void)                     );
```

Figure 4: /usr/src/servers/fs/proto.h

```
Int do_printmessage()
{
        printf("\I am a system call \n");
        return (OK);
}
```

Figure 5: Our system-call handler

A few files like misc.c, stadir.c, write.c, and read.c contain the definitions for the system-call handler functions. We could either add our system-call handler to one of these files or have it in a separate file. If we choose to add it in a separate file, we have to make changes in the /usr/src/servers/fs/Makefile accordingly. For our example, we will add the definition of function do_printmessage() to /usr/src/servers/fs/misc.c. After implementing the system-call handler, we can compile the FS server to ensure that our new system-call handler does not contain any errors.

2.1. Compiling the FS Server

Steps for compiling the servers:

- Go to directory /usr/src/servers/
- Issue "make image"
- Issue "make install"

2.2. Calling the System-call Handler Function Directly

Our system-call handler function do_printmessage() has the system-call number 69. We could call the system-handler function directly using the system call _syscall. _syscall takes three parameters: the recipient process, system-call number, and pointer to a message structure.

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

In our example, the recipient process is FS, the system-call number is 69, and we do not pass any parameters. Still, we should pass a pointer to an empty message for the third parameter when calling the handler function. We call the handler as show below.

```
message m;
_syscall(FS,69,&m);
```

When the system-call handler needs to receive some parameters, we pass the parameters using the message structure. The message structure is described in figure 6. To use the message structure, the header file "lib.h" should be used. The header file contains some "#define"'s that makes using the message structure simple.

```
typedef struct {int m1i1, m1i2, m1i3; char *m1p1, *m1p2, *m1p3;} mess_1;
typedef struct {int m2i1, m2i2, m2i3; long m2l1, m2l2; char *m2p1;} mess_2;
typedef struct {int m3i1, m3i2; char *m3p1; char m3ca1[M3_STRING];} mess_3;
typedef struct {long m4l1, m4l2, m4l3, m4l4, m4l5;} mess_4;
typedef struct {short m5c1, m5c2; int m5i1, m5i2; long m5l1, m5l2, m5l3;}mess_5;
typedef struct {int m7i1, m7i2, m7i3, m7i4; char *m7p1, *m7p2;} mess_7;
typedef struct {int m8i1, m8i2; char *m8p1, *m8p2, *m8p3, *m8p4;} mess_8;

typedef struct {
  int m_source;                         /* who sent the message */
  int m_type;                           /* what kind of message is it */
  union {
            mess_1 m_m1;
            mess_2 m_m2;
            mess_3 m_m3;
            mess_4 m_m4;
            mess_5 m_m5;
            mess_7 m_m7;
            mess_8 m_m8;
  } m_u;
} message;

/* The following defines provide names for useful members. */
#define m1_i1  m_u.m_m1.m1i1
#define m1_i2  m_u.m_m1.m1i2
#define m1_i3  m_u.m_m1.m1i3
#define m1_p1  m_u.m_m1.m1p1
#define m1_p2  m_u.m_m1.m1p2
#define m1_p3  m_u.m_m1.m1p3

#define m2_i1  m_u.m_m2.m2i1
#define m2_i2  m_u.m_m2.m2i2
#define m2_i3  m_u.m_m2.m2i3
#define m2_l1  m_u.m_m2.m2l1
#define m2_l2  m_u.m_m2.m2l2
#define m2_p1  m_u.m_m2.m2p1

#define m3_i1  m_u.m_m3.m3i1
#define m3_i2  m_u.m_m3.m3i2
#define m3_p1  m_u.m_m3.m3p1
#define m3_ca1 m_u.m_m3.m3ca1

#define m4_l1  m_u.m_m4.m4l1
#define m4_l2  m_u.m_m4.m4l2
#define m4_l3  m_u.m_m4.m4l3
#define m4_l4  m_u.m_m4.m4l4
#define m4_l5  m_u.m_m4.m4l5

#define m5_c1  m_u.m_m5.m5c1
#define m5_c2  m_u.m_m5.m5c2
#define m5_i1  m_u.m_m5.m5i1
#define m5_i2  m_u.m_m5.m5i2
```

Figure 6: Message structure

Say a system-call handler do_managecap needs to receive three integer parameters. The system-call number of do_managecap is 58. We need to initialize the three parameters in the message structure, and call the system call handler using the message structure as shown in figure 7.

```
message m;

m.m1_i1=45;
m.m1_i2=55;
m.m1_i3=65;

_syscall(FS,58,&m);
```

Figure 7: Passing Parameter Using the Message Structure

The FS server has a global variable named **"m_in"**, which is a message structure. Whenever a system-call arrives at the FS server, **m_in** would contain the message structure pointed to by the third parameter in the call. We retrieve the three parameters from the **m_in** message structure in the system-call handler function.

```
PUBLIC int do_manageusercap(void)
{
        int user_id = m_in.m1_i1;
        int what = m_in.m1_i2;
        int cap_to_process=m_in.m1_i3;
```

Figure 8: Retrieving Parameters From the Message Structure

3. Creating a User Library Function

A user library function would package the parameters for the system-call handler in the message structure and would call the handler function. First, we should use #define to map the system-call number of the handler function to an identifier in the file /usr/src/include/minix/callnr.h and /usr/include/minix/callnr.h.

```
#define PRINTMESSAGE 69
```

We implement the library function for the do_printmessage system call in a separate file named _printmessage.c. This file should be placed in the directory /usr/src/lib/posix/.

```
#include <lib.h>
#include <unistd.h>

PUBLIC int printmessage(void)
{
        message m;
        return(_syscall(FS,PRINTMESSAGE,&m));
}
```

Figure 9: Library Function Implementation

3.1. Compiling the Library

Steps to compile the new library

- Go to the directory /usr/src/lib/posix/
- Add the name of the file in the /usr/src/lib/posix/Makefile.in
- Issue the command "make Makefile". (This command will generate a new makefile with the rules for the new file included.)
- Go to directory /usr/src/
- Issue command "make libraries"
- All these steps will compile and install the updated posix library.

3.2. Creating a New Boot-Image Using the Updated Servers and Library

We already compiled and created the binaries for the servers, and now we have the fresh libraries compiled and installed. Now, we need to merge them the updated binaries and create a new boot image.

Steps for creating the boot-image:

- Go to directory /usr/src/tools
- Issue command "make hdboot"
- Issue command "make install"

These steps would create a new boot-image in the directory /boot/image/. Note down the name of the new boot-image. When we shutdown and reboot, we should select the new boot-image.

In the boot prompt, we could setup the new image for booting using the command "image =/boot/image/<name of boot-image>". Then we could issue the command boot to startup using the new boot-image.

Now, the new system call is ready to use.

4. Using the New System Call

```
#include <stdio.h>

int main()
{

        printmessage();
}
```