Communicating User Processes

Minix Laboratory set-up

Steffen van Bakel

Department of Computing, Imperial College London

Autumn 2013

Contents

1 Preliminary remarks

After a while you will be prompted to login (`login:`) and can login as root without a password. At all times, you will have four shells available: you can jump from one to the other by hitting `<Ctrl><Alt><→>` or `<Ctrl><Alt><←>`, giving you the possibility of working in four windows during your session. A bit stark, yes, but then again, who needs windows?

After start-up, youll find two files in the directory `/boot`: `boot` and `image-small`, as well as the directory `image`. You will add your new image to the directory `image` when you build your new version of the kernel (see below); MINIX will automatically boot the youngest version of the system, i.e. the image last added. You should not touch any of the existing files at all during the exercise: this is important because if you corrupt the backup image you may not be able to get back into MINIX. In case your version does not boot, or crashes, you can find in Section 1.4 how to boot the old one.

1.1 Man pages

For all commands in MINIX, there is a man page available that contains all (necessary) information needed to understand the working of the command. Typing

```
man command
```

will show the man page for command on the screen.

1.2 Saving and restoring the MINIX partition

To exit MINIX safely just type

```
shutdown
```

to leave the shell and make MINIX halt and sync (i.e. write all information kept in main memory to the disk). Do not just reboot or switch the machine off: this will give you an inconsistent file system that needs to be restored using `fsck`. This will take place during the systems installation phase, so after you reboot, is rather time consuming and may lead to lost files.

This takes us back to the boot monitor.

1.3 Recompiling MINIX

Each part of MINIX can be re-compiled separately. In the directory `/usr/src` you will find various directories and files; amongst them `boot`, `kernel`, `servers`, (with `pm` and `fs`), and `tools`. Each of these directories have a file `Makefile` that gives the dependencies between the `*.c` files and the `*.h` files, and describes which files are to be compiled into this part of the system. Typing

```
make part
```

in the directory part will bring a re-compilation of that part, but not more than that. Make sure that the contents of the file `Makefile` are correct. To obtain a new version of MINIX, it is necessary to re-compile the various parts of the system and link them all together. The `Makefile` that allows to do this all in one go, can be found in the directory `tools`. Typing

```
make install
```

there will visit the neighbouring directories, build the system parts, recompiling the changed files, and link the resulting files together into the new image file and place it in `/boot/image`. After typing

```
reboot
```

the system will shutdown and reboot, using the new image.

Although these two approaches will give the same result in case of a successful recompilation, while working on the `kernel` it could be wiser to recompile that part separately, and not try a link before the `kernel` compiles well. Alternatively, you can also re-compile a single file `filename.c` in one of the directories, by typing

```
make filename.o
```

For this to work it is of course essential that the `Makefile` knows how to build the `filename.o`. You can undo all the effect of a previous compilation by typing

```
make clean
```

in the directory `tools`.

1.4 Booting with an old image

As a result of recompiling the system, the new image is written to the directory `/boot/image`, ready to use as image at the start-up of the system. To reboot MINIX with your new image, just type

```
shutdown
```

to leave the shell and make MINIX halt and sync. At the prompt `fdo>` you can now select which image to boot; remember that the default is the youngest. Type

```
set
```

to see the current settings. Picking the old boot image is now achieved by typeing:

```
image = /boot/image/3.1.0
```

or any other file that is in the directory `/boot/image` will make sure that, during the next boot cycle, the original image will be used, not the youngest. (Remember you should not change the backup kernel `3.1.0` at all during the exercise!)

```
boot
```

will make the machine restart, using the new settings; these are saved, so in order to test your new system, you would have to choose

```
image = /boot/image
```

2 Some helpful advice

The implementation of the solution can become very complicated, with your code in many different parts of the system at once. The following advice should help you to keep track of what is where within the system and how the flow of control is operating.

- Keep your `Makefile`s up to date - whenever you add a file to the system, update the corresponding `Makefile` immediately. Otherwise you may have compilation errors or unexpected behaviour in your system.
- Add informative debug messages - it is likely that your system will require many debug messages, using `printf`, or `kprintf` in many different files. The C compiler supports a special `__FILE__` macro, which contains the filename of the current file surrounded by quotes. You can use this in the following way to easily keep track of which file a debug message originated from:

  ```
  printf(__FILE__": Message arrived");
  ```

  would print

  ```
  proc.c: Message arrived
  ```

  if used from `proc.c`.
- Explore similar MINIX code first before making any additions or alterations to the system, look at the existing MINIX code in the directory you are working in. Some pointers are given in the main part of the lab spec, but explore further to get a feel of how the system works.
- Use the tools available - MINIX includes `grep`, which will allow you to locate text within the source code. The MINIX version is slightly primitive, however it is still useful. Use `man grep` to find out more.

- Sometimes you KNOW it will crash - there will likely be times during testing when you know that some action you take will crash the system (when you have added debug information to find out why, for example). Before performing the action which will lead to a crash, run `sync` or `reboot` of `shutdown` to flush cached data to disk. This will reduce any file system problems resulting from the crash.

3 Extending the communication between user-processes and MINIX

We will now focus on a number of aspects related to extending MINIX by adding adding a system-call.

3.1 Adding a system-call

When a new system-call is to be added to the system, a number of changes need to be implemented:

- It should be possible for users to call this new system-call; hence, we need to extend the library that contains all system-calls.
- The memory manager `PM` (mapped to `PM_PROC_NR` in `usr/src/include/lib.h`) or the file system `FS` needs to be extended, so that the new system-call can be interpreted.
- The kernel will (very likely) need extensions.

The system-calls that are available for MINIX are, together with the other pre-defined library routines, taken by the C-compiler from the library files, like `libc.a`, in the directory `/usr/lib/i386`. In these library routines, messages are built that will be sent from the calling user process to a system task. Since in MINIX all communication between user processes and the system go through the memory manager `PM` or the file system `FS`, the sending of the message will (eventually) always have one of the following shapes:

- `sendrec(PM_PROC_NR, &message);`
- `sendrec(FS_PROC_NR, &message);`

where `PM_PROC_NR = 0`, `FS_PROC_NR = 1` and `message` is a variable of type `message` (you can find the definition of `message`s in `include/minix/ipc.h` and `include/minix/com.h`). The `sendrec`-routine consists of two parts: the sending of the message, followed immediately by receiving a message,

- `send(PROCESS, &message);`
- `receive(PROCESS, &message);`

which will generally mean that the process using the system-call will be blocked. The division into two separate functions enables `PM` or `FS` to accept the assignment, do the necessary work, or have it done by the appropriate task, and send a message (containing, for example, the requested data) as a reply to the request.

Of course the message that is constructed by executing the system-call routine (before the send) needs to contain all the information necessary for `PM` or `FS` to determine what work is to be done. This is accomplished by mentioning the number of the job, i.e. the number of the system-call, in the field `message.m_type` of the message. Obviously this number needs to correspond to the number of the intended system-call in `PM` or `FS`. Which numbers are in use can be found in the file `include/minix/callnr.h`. Also the library procedures get the value of the constants they use from this file, although through the file `include/lib.h`.

The other fields of the message can be used to transport other information from the user process to the operating system. An example of such use can be found in the library routine `_utime.c`, that you will find in the directory `src/lib/posix`.

3.2 Constructing a library routine

A library routine must be written as an ordinary C-program, with no procedure `main` defined, and consisting of a collection of procedures that need to be accessible for user programs. If there is to be more than one version of the system-call, for example by allowing the number of parameters to differ, you should put all the versions together in one file. The C-file should contain the include statement `#include <lib.h>`, so that constants and types are known inside this C-file. Now, you either want to add a new system-call by writing a new system-call file, or you want to add it to a file that already exists; in both cases you need to establish which is the subdirectory of the directory `src/lib` that you need to work in. Have a close look at, for example, the file `src/lib/posix/_utime.c`, and see how the various functions that are defined in that file are made `public` at the beginning of the file. If you create a new file, you will need to up-date the file `Makefile` in the directory where you have added a file. For both system-calls in new files, and those added to existing files, you will need to add an entry in the directory `src/lib/syscall`: just looking at, for example, `utime.s` will give you a clear idea of what it is you need to do for each new system-call. Creating a new system-call library that contains the new entries is relatively easy. Just change to the directory `src/lib` and type the command

```
make install
```

This will create all the new libraries that the C-compiler could need; it will overwrite the existing libraries, so you should have saved those (see below).

3.3 Changes in `PM` or `FS`

After adding new routines to the library, where new messages are sent to either `PM` or `FS`, the servers (`PM` and `FS`) need to be extended with code that correctly interprets the new system-calls. Also, the servers need to be extended with code that guarantees that the right server routine is called, depending on the value of the type field in the message sent. The principle used is as simple as elegant. For both servers there is a file `table.c` (so in both the directories `servers/pm` and `servers/fs`), each containing an array of possible system-calls. These arrays are the size of the maximum number of possible system-calls, as defined in `include/minix/callnr.h`. The n-th position in this array contains the name of the routine that needs to be called when a system-call with number n arrives. If a specific system-call gets served by the other server, the entry in the array will be a dummy name `no_sys`. Choose for your new system-call an entry in the table that is not used in both arrays. Insert the name of the procedure that is to be called in the array of the appropriate server, and mark the entry in the array for the other server as used, using comments. You will also need to up-date the file `include/minix/callnr.h`, `#define`ing the name of the new system-call.

4 The exercise: Letting user processes send and receive messages

The purpose of the laboratory is to extend the communication system of MINIX by allowing user processes to send to and receive messages from other user processes. The sources and destinations of messages will be the processes `pid`s. Since the actual sending of messages is a kernel privilege, user processes can only request the system to send a message, which implies doing a system call; in total, two new system calls will be added to the system.

System calls in MINIX are implemented using `sendrec`, so we are going to find ourselves in the - apparent conflicting - situation of using message passing to ask to pass a message. But there is no real conflict. When asking the system to send a message, a request goes out, via a message, to `PM`. In fact, the type of the message (`USER_SEND`) will indicate that the message has to be passed on to its actual destination which is mentioned in the message (stored in the field `DST_PROC_NR`). `PM` receives this message and passes it on to `SYSTASK`, after adding the `pid` of the sending user process to the message field `SRC_PROC_NR`. Although the sending process `id` is stored in the message when it arrives in `PM`, sending it on the `SYSTASK` will overwrite that value (`PM` sends a message, and that will cause the field `m_source` to be overwritten with `PM`s id), so we need to store it. `SYSTAKS` will, upon receiving the message, invoke `user_send`. Before it can do that, it has to extract the process numbers for the user processes involved, because it uses those rather than their `pid`s. As for receiving, the above holds as well, but for the use of `USER_REC`, `SRC_PROC_NR`, `DST_PROC_NR` and `user_receive` (in that order). Once the message is sent, we should not reply to the original system call; the process should act as if it is just sending or receiving, so, in particular sending a reply message by `PM` would seriously confuse the system.

The solution sketched below is very easy and straightforward, perhaps not the best solution that one could imagine, but that is of course also not the scope of this laboratory. Moreover, it is, because of its intended simplicity, feasible to implement in the available time. We require that you follow as closely as possible the solution sketched below. In this way, the various solutions will be compatible, centralised help will be available, and part of the necessary solution is already present. It is however compulsory that your solution respects the organisation of MINIX. In particular, code needs to be extremely readable, with as much comments as possible to explain procedures. Also, any form of communication, be it either through sub-routine call or by passing data, from one driver to another, not using the message passing mechanism, is strictly forbidden.

4.1 Before we start

The files that need to be updated or created follow are given in Figure 1; some are already supplied. It is up to you to decide what needs to be changed in those files, although we will give you hints.

Before you start up the virtual machine, it is better to save all the relevant files in `/usr` inside MINIX; move to the directory where the virtual disk `MINIX.vdi` is located, make directories `Disk` and `BCK`, and type

```
mount_minix Disk
```

This will mount the directory `/usr` on the virtual disk on the directory `Disk`. This is now a file system that you can access under linux, but you are strongly advised to not do more than read or write to it: the linux file system is different in detail, and you risk corrupting the virtual disk, with possible loss of data. For example, do not mount the disk when the virtual machine is running. You can now type

```
cp -pR Disk BCK
```

which will copy the entire directory `/usr` to `BCK`. This allows you to use any linux editor to have a look through the files, rather than just using the 23-line MINIX window.

You can run this command also inside MINIX itself, creating a back-up on the virtual disk.

Add the file `usersendrec.h`

```
/* send and receive at user level */
_PROTOTYPE( int userreceive, (int process, message *mess)                );
_PROTOTYPE( int usersend, (int process, message *mess)                   );
```

File name	Phase:	0	1	2	3	4	5
communicate.c		×					
include/sys/usersendrec.h		×					
include/minix/callnr.h			×				
include/minix/com.h			×		×		
lib/posix/Makefile			×				
lib/posix/_userreceive.c			×				
lib/posix/_usersend.c			×				
lib/syscall/Makefile			×				
lib/syscall/userreceive.s			×				
lib/syscall/usersend.s			×				
servers/pm/proto.h				×			
servers/pm/main.c				×		×	×
servers/pm/misc.c				×			
servers/pm/table.c				×			
servers/fs/table.c				×			
kernel/proto.h						×	
kernel/system.h					×		
kernel/proc.c						×	×
kernel/system.c					×		
kernel/system/do_usrrec.c					×	×	×
kernel/system/do_usrsnd.c					×	×	×

Figure 1: The affected files

to the directory `usr/src/include/sys`; it supplies `PROTOTYPES` for the new system-calls, so that the compiler treats the calls correctly in terms of types; if you forget to include it when you need to, you'll get the complaint of an 'old-fashioned definition' from the C-compiler. Copy the file `communcations.c` over to the virtual disk.

Now start the virtual machine.

4.2 Stage one: installing system calls

The directory `/usr/src` contains a directory `include` that contains the header files for the C-compiler, but the files of the system are not compiled against these, but rather against the copy of them kept in the folder `/usr/include`; the various `Makefile`s copy the directory `/usr/src/include` to `/usr/include`.

The first thing to do is to allow for conditional compilation, and allowing all code of the various phases to be present in your final solution. We achieve this easily be adding

```
#define PHASEONE
#define PHASETWO
#define PHASETHREE
#define PHASEFOUR
#define PHASEFIVE
```

in the file `const.h`, and surrounding, for example, the code added in `.c` and `.h` files with

```
#ifdef  PHASEONE


#endif
```

If in a later phase you need to change the added code, keep the original part as well, but change the first line of the current code dealing with that event into

```
#ifndef PHASEXXX
```

effectively excluding that part from the compiled system. Then add the new code for the next phase.

(From now on we will not mention the surrounding `#ifdef` ... `#endif` statement.)

Add

```
/* Field names for user process communication */
#define USER_PID                m2_i1
#define DST_PROC_NR             m2_i1
#define SRC_PROC_NR             m2_i2
#define MESS_ADDR               m2_p1
```

to the file `include/minix/com.h`.

Choose two values from the file `callnr.h` for the three new system calls

```
USER_SEND
USER_REC
```

any values will do, as long as they are not already in use. Remember these values, you are going to need them later.

In the directories `src/lib/posix` and `src/lib/syscall` you need to create the system-call files mentioned above (for `userreceive` and `usersend`). These should send a message to `PM`, with the type field set to `USER_REC` and `USER_SEND`, respectively (this is accomplished by passing it as second argument to the call to `_syscall`); look for some inspiration at some of the other files in the directories `posix` and `syscall`, in particular those for `utime`.

Notice, however, that we are to use two messages here. The first is the one sent or received between user processes, the second the vehicle to inform the operating system of our request. We implement the sending and receiving of messages at user level via system-calls, that are communicated to the system via messages. Make sure that the pointer to the message sent or received is passed on using the field `MESS_ADDR`, and put the passed argument in the field `USER_PID`. Update the two `Makefiles` and run

```
make
```

in each of the directories. If both compile nicely, recompile the system: move to the directory `src/tools`, and type

```
make install
```

and once that has finished, type

```
reboot
```

You might get error messages in the compilation, which you will need to fix before rebooting.

`PM` needs no changes at this stage, it suffices that the command `communications.c` compiles; it is best to place this in `/root`, where you are placed directly after logging in.

4.3 Stage two: involving `PM`

Add the first versions of the `PUBLIC` system-call procedures `do_usersend` and `do_userreceive` in `server/pm/misc.c`; follow the set-up of the other procedures defined there. Change the file `proto.h`, adding to the section for `misc` the added procedures.

For the moment, let both print some information on the screen, like process numbers that are involved in the communication, and reply using

```
return(OK);
```

This value will be returned to the user process doing the calls, so at least they should not fail now when you run communications. Change the file table.c in pm to reflect the new calls; notice that you need to define here what should be the routine that gets invoked when a call with this m_type arrives, so be precise; remember the choices made in callnr.h. For the sake of balance, mark the entries as used in fs/table.c; for this exercise, this is all that needs to be done in FS.

Recompile the system, and test.

4.4 Stage three: passing control to SYSTASK

We will now let do_usersend and do_userreceive send a message to SYSTASK, effectively passing on the request that came in from the user process; we shall let these procedures still reply with OK, so are only simulating the forwarding of the message. We will call on MINIX' sending and receiving services later, for which we need process numbers rather than process identifiers (i.e. indexes in the process array rather than the unique numbers). Mark that the user command communications.c only knows the unique process identifiers, so we need a private procedure

```
find_proc_nr(process_pid)
```

that finds the PROC_NR for a process of which we have the pid (see servers/pm/mproc.h); take special care of ANY. Add a _PROTOTYPE for this definition at the beginning of the file proc.c.

We now need to send the two process numbers of the processes involved in the user communication to SYSTASK in the file misc.c. Mark that the *calling* process is the user that sent the message m_in to PM, called who there.

Define a new message m_out in both procedures, and, for do_userreceive, fill the field DST_PROC_NR with the user processs PROC_NR and copy the fields SRC_PROC_NR and MESS_ADDR (note that these fields are set in the system-call) lib/posix/_userreceive.c. For do_usersend, fill the field SRC_PROC_NR with the user processs pid, and copy the other two fields.

Now extend the interface for SYSTASK. Add entries for SYS_USRREC and SYS_USRSND to the file minix/com.h, and increase the value of NR_SYS_CALLS with two. Set in pm/misc.c the field m_type field to the appropriate value and pass the message on using sendrec.

The messages will arrive in SYSTASK, as specified in kernel/system.c. It uses a function map that maps system-call numbers to procedures. Add the cases for SYS_USRREC and SYS_USRSND at the bottom of the definitions for map, and insert calls to the appropriate procedures do_usrrec and do_usrsnd; do not forget to insert their PROTOTYPE in system.h as well. SYSTASK will send a reply back, so make sure you receive that.

The two procedures do_usrrec and do_usrsnd need to be defined in the directory kernel/system in separate files; let them print text on the screen and return OK; add entries in kernel/system/Makefile.

Recompile the system, and test.

4.5 Stage four: involving process management

We will now add two functions to the system, that are called from do_usrrec and do_usrsnd in kernel/system, but are logically part of kernel/proc.c, so we will place them there. In order to have our conditional compilation work correctly, change the order of the include files to

```
#include "kernel.h"
#include <minix/com.h>
#include <minix/callnr.h>
#include "proc.h"
```

otherwise our definitions in com.h are not seen (do *not* surround this change with #ifdef, but rather comment out the original includes.

Add two PUBLIC procedures to kernel/proc.c, called user_send and user_receive; do not forget kernel/proto.h. Specify those procedures as just printing text on the screen, and returning OK.

Recompile and test.

4.6 Stage five: doing the actual sending and receiving

We will now take care of the actual sending and receiving of messages between user processes. We would like to call mini_send and mini_receive directly, but these are PRIVATE in proc.c, and also demand different type parameters. We will rather implement the routines user_send and user_receive copying mini_send and mini_receive; that way any change you make will not affect the normal message passing. Note that you will have to convert the values you have found to the right parameters (see proc.h). The main change you will have to make is to return SUSPEND rather than OK when you are blocking the process trying to send or receive.

Since we now have actually sent or received a message, we should no longer simulate the result, but make sure that the correct return value is passed on: save the result of any call in a local variable, that you pass on with return; do the same in system/do_usrrec.c and system/do_usrsnd.c.

We also no longer need to reply to the processes directly, but only after the sending or receiving has actually taken place; they are still waiting for PM to reply, so that's the process that needs to answer. The only tricky bit is now in pm: SYSTASK has placed the return value for sendred in the field m_type (the answers arrives in the same message), and you need to test if this is SUSPEND; if this is the case, return that value as a result to the calls do_usersend and do_userrec. Instead, if the m_type coming back from SYSTAKS is *not* SUSPEND, you then prepare the actual replies going to the user process by setreply(proc_nr, OK) (where proc_nr should be the correct value), and return the result of sendred itself. PM will store the two wake-up calls, and execute them for you.

Then in pm/main.c, check if the call made was either USER_REC or USER_SEND and the result SUSPEND, if so, continue: you then do not answer to the call.