System Call Tutorial for Minix (3.2.1)

Ryan Cobb

Important Notes:

- All code or configuration changes must be performed under **/usr/src/** in Minix.
- This tutorial is written with respect to the *fork* system call in Process Manager (PM) of Minix, but the same steps can apply to other services as they conform to a similar architecture.
- When trying to find references within the source base you can use the following command:
  - `grep -R <keyword> <location> | more`
  - To get more details on grep, refer to its manual page with `man grep`.

Declaring the Library Call:

In order for a user program to invoke some system call, we must first declare a library function that accepts as arguments, invokes the appropriate system call and potentially returns a result. This function declaration must be declared in some system library header that is visible to user programs. These headers are located within **/usr/src/include/,** and fork's declaration is contained within **unistd.h**.

/usr/src/include/unistd.h(line 111):

```
pid_t  fork(void);
```

In the case of fork, it requires no arguments, but will return the process id, of type pid_t, to the calling process (parent and child). Similarly, for your own system call, you can either add it to an appropriate preexisting header file or create your own header file in **/usr/src/include/**.

Defining the Library Call:

Once we have declared what our library call's prototype, we must define the function body. This function will actually be responsible for calling down onto the appropriate service (PM in this case) to perform the requested action (fork). This system call invocation is performed using *_syscall*. This function is responsible for performing the Inter-Process Communication(IPC) necessary to communicate the requested action to the appropriate service.

/usr/src/lib/libc/sys-minix/fork.c:

```
pid_t fork()
{
  message m;

  return(_syscall(PM_PROC_NR, FORK, &m));
}
```

/usr/src/include/lib.h(line 33):

```
int _syscall(endpoint_t _who, int _syscallnr, message *_msgptr);
```

/usr/src/include/minix/callnr.h(line 5):

```
#define FORK              2
```

Dissecting the above _*syscall* parameters, we can see that it is divided into three sections: who, system call number and a message pointer. The first parameter, who, is the process target we are trying to communicate with. In the case of *fork*, we are trying to request the service from PM, indicated by *PM_PROC_NR*. The second parameter, system call number, is a unique number indicating what service we want PM to perform (*FORK*). Lastly, the message pointer is a genericized struct used to pass parameters associated with the system call. Since FORK does not require any parameters, the message object is left uninitialized, but other system calls (such as chmod) do require parameters. Refer to **/usr/src/include/ipc.h** for more details on the *message* struct.

For the purpose of creating your own library function, you must follow a similar process to that of fork. You will need to create your own source file(.c) in **/usr/src/lib/libc/sys-minix/** to define the body of your library call (e.g., myCall() in myCall.c). This new source file must be added into **Makefile.inc** in the same directory so it will be built into the library. The body of this call should resemble the body of fork, but with alterations made to the _syscall parameters. In order to fill out the second parameter, we must choose an **unused** call number in **/usr/src/include/minix/callnr.h** and define your call number there (E.g, FORK). Some example unused numbers would be 35, 69, 70 or any other number in the listing that is not currently defined (but less than NCALLS). Once these steps have been done, whenever your new library function is called, a message will be passed to the service requesting that new call number you've reserved.

Processing a System Call

Once the system call has been requested, the service (PM in this case) must be able to recognize and process that call number to perform the requested action (fork). All of the code pertinent to services within Minix are located within **/usr/src/servers/** and, in particular, PM existing in the **pm/** subdirectory. In order for PM to appropriately process your new system call number, code must be modified within three specific locations. For *fork* the code as follows are the modifications made to those locations:

/usr/src/servers/pm/proto.h(line 26):

```
int do_fork(void);
```

/usr/src/servers/pm/table.c (line 16):

```
do_fork,    /*  2 = fork    */
```

/usr/src/servers/pm/forkexit.c (line 44):

```
int do_fork()
{
        …
}
```

The first two files, **proto.h** and **table.c** are responsible for registering the appropriate function with the system call number you've created. As we can see for the *fork* system call, first you must define the function that will handle the call, *do_fork*. This function must be declared with no parameters(void) and return an integer. Once the function was defined, it was registered into the call vector in the third array position (zero based). This corresponds with the FORK define in **callnr.h** that associated FORK with the system call number 2. With these two steps completed, the function is registered in the call table so PM can process it, but the *do_fork* function body must be declared for it to correctly operate. The body of *do_fork* is declared in the **forkexit.c** file and handles all of the fork related operations necessary for PM to properly handle it.

For your own system call you will need to declare your function in **proto.h** in the same fashion as the other system calls, `int do_myCall(void);` Once this function has been defined, it needs to be registered into the call vector in **table.c**. Like *do_fork*, you must also register your new function *do_myCall* into the table at the appropriate number you reserved in **callnr.h**. If you've correctly selected an available system call number, the comment in the table next to that location should state it is **unused** and the function registered is *no_sys*. Once both of those changes have been performed, the function must be declared. You can either create a new **.c** file to contain your function or add it into an existing **.c** file in the PM directory (i.e., **misc.c**). If you do choose to make a new **.c** file, you must add it to the **Makefile** in PM in order for the build system to compile and include your changes.

Compiling and Testing your changes

The Minix source build system is currently setup to utilize makefiles throughout in order to build and link all components to create your new Minix revision. The makefiles are constructed in a tiered approach so you can compile and link locally (such as in PM) or at a global level to create a new version of Minix. Compiling the local Makefile(PM's) is useful to quickly check if your code changes will compile correctly without going through the entire build process of the top level Makefile. The top level Makefile can be found under **/usr/src/releasetools**. To actually deploy your Minix OS changes, you must use the top level makefile. This makefile can simply be used as *make install*. The first time this command is run, it may take a while to compile all of the objects depending on the speed of your computer, but should be faster on subsequent runs due to object file caching. Once it has finished, you can simply *reboot* to restart Minix.

The boot loader at startup allows you to pick either your newest changes (**2**) or other options. By default it will load your newest build of Minix. However, if your changes corrupted Minix such that you can no longer use it, the first option in the boot loader is a fallback allowing you to use an unaltered version of Minix to fix your changes and try again. Once you've loaded into your new version of Minix, you can create a simple program to test your new library/system call. For testing purposes it is recommended to use *printf*s profusely so you can easily track and debug any changes you've made throughout the system. The simple program can #include the header file that defines your library call (**#include <unistd.h>** in the case of *fork*) and in main call the library call (*fork()*). You can then compile your program using *clang*.

Sample Main using Fork:

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

int main(int argc, char** args) {
     pid_t id = fork();
     printf("Process ID: %d\n", id);

     return 0;
}
```

Compile:

```
      clang –o testFork test.c
```

Run:

```
      ./testFork
```