

How to Add a New System Call for Minix 3

Daniel Rodríguez García

Department of Electrical and Computer Engineering

Universidad de Puerto Rico, Mayagüez, Puerto Rico

In this laboratory, we are implementing and modify the System Calls inside the Minix operating system. The purpose of this is to learn how the system calls work and how they can be implemented and used within the system to achieve a certain function for the benefit of the user. A series of steps are going to be followed, gathered from the tutorial How to Add a System Call in Minix 3.1.8 to add and test a custom system call in Minix.

Introduction

System calls are primordial to every operating system. They are the way in which the user level can communicate with the kernel level, the way in which the level with the lowest privilege can access the functionality with the highest privilege. This communication is done through system calls. There are two basic model types of these calls, the shared memory model and the message passing model. The shared memory model uses a common region in memory between or more processes to communicate while the message passing model uses a mailbox as a receiver and bridge of communication between two processes.

In Minix we have a micro-kernel architecture which is the one who performs all the system calls related processes. The Micro-kernel has three basic calls: send, receive and sendrec. Minix has a system calls table for other functions, there are two of these, File System table and Memory Manager table.

Here we are going to implement a system call that prints a message, then we are going to modify it and add a parameter to see how it can be implemented. Afterwards we will explore how to extend the functionalities of a system call, which will include the way Minix gets the PID of a process and how it displays it to the user.

Methodology

In order to add a system-call we must follow a few steps.

1. Edit “/usr/src/include/minix/callnr.h” and add the name and number of your call in a non-used call number. For this exercise we’ll use MYCALL and number 35

```
#define GETGROUPS    34
#define MYCALL       35
#define SYNC         36
#define KILL         37
```

2. Then go to and edit “/usr/src/servers/pm/table.c”
Add the line do_{YOUR CALL NAME}, for this case the call name is **mycall** and comment the call number

```
do_get,          /* 34 = getgroups */
do_mycall,       /* 35 = (stty)  */
no_sys,         /* 36 = sync    */
```

3. Go to and edit “/usr/src/servers/pm/proto.h”
Under the section misc.c add the line
_PROTOTYPE(int do_mycall, (void));

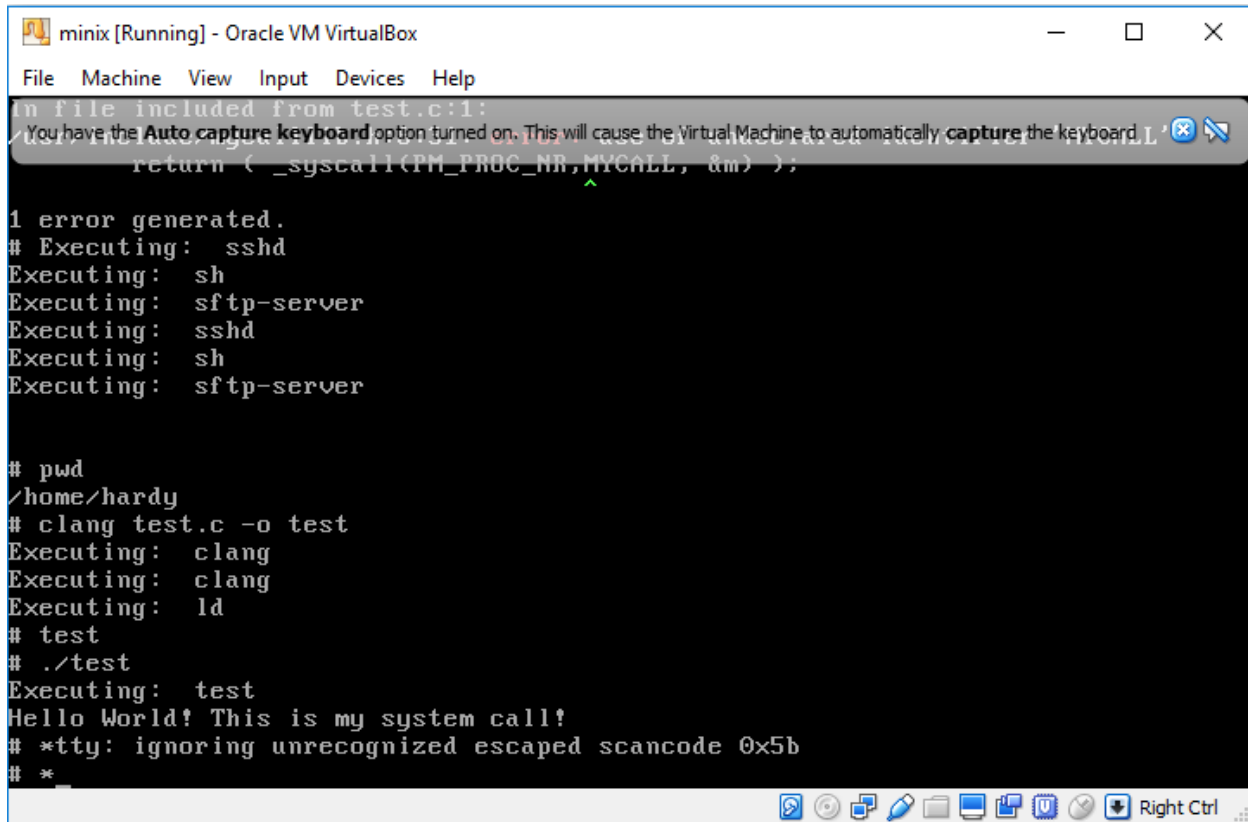
4. Now go to misc.c in the same directory and add:

```
/*=====
=====*
*                                do_mycall                                *
*
*=====
=====*/
PUBLIC int do_mycall()
{
    printf("Hello World! This is my system call!");
}
```

5. Do a “make” on the same directory
6. Now go to /usr/include and add a file called mycalllib.h
Add the following:

```
#include<lib.h>
#include<unistd.h>
#include<stdlib.h>
PUBLIC int mycall ()
{
    message m;
    return ( _syscall(PM_PROC_NR,MYCALL, &m) );
}
```

7. Now cd to /usr/src/tools
8. make hdbboot
9. sync
10. reboot
11. Create a C file, include the library mycalllib.h and call the system call by using mycall()
12. Compile it with clang using the following command:
clang test.c -o test
Run the program with ./test
where test is the name of the C file
Should look like the following image

A screenshot of a virtual machine window titled "minix [Running] - Oracle VM VirtualBox". The window has a menu bar with "File", "Machine", "View", "Input", "Devices", and "Help". A tooltip is visible over the "Input" menu, stating: "You have the Auto capture keyboard option turned on. This will cause the Virtual Machine to automatically capture the keyboard." The terminal window shows the following output:

```
in file included from test.c:1:
return ( _syscall(PM_PROC_NR,MYCALL, &m) );

1 error generated.
# Executing: sshd
Executing: sh
Executing: sftp-server
Executing: sshd
Executing: sh
Executing: sftp-server

# pwd
/home/hardy
# clang test.c -o test
Executing: clang
Executing: clang
Executing: ld
# test
# ./test
Executing: test
Hello World! This is my system call!
# *tty: ignoring unrecognized escaped scancode 0x5b
# *
```

13. A message should now appear in the display saying: Hello World! This is my system call!

14. Now if we want to add a parameter we must use the global array described in the /usr/src/include/minix.ipc.h file.

15. We use for example m_in.m4_l1

16. Add this to the end of the expression in do_mycall in misc.c we add in the printf a number parameter for a long variable and the variable being m_in.m4_l1

17. We now modify mycalllib.h with the following code

```
#include<lib.h>
#include<unistd.h>
#include<stdlib.h>
PUBLIC int mycall (long i)
{

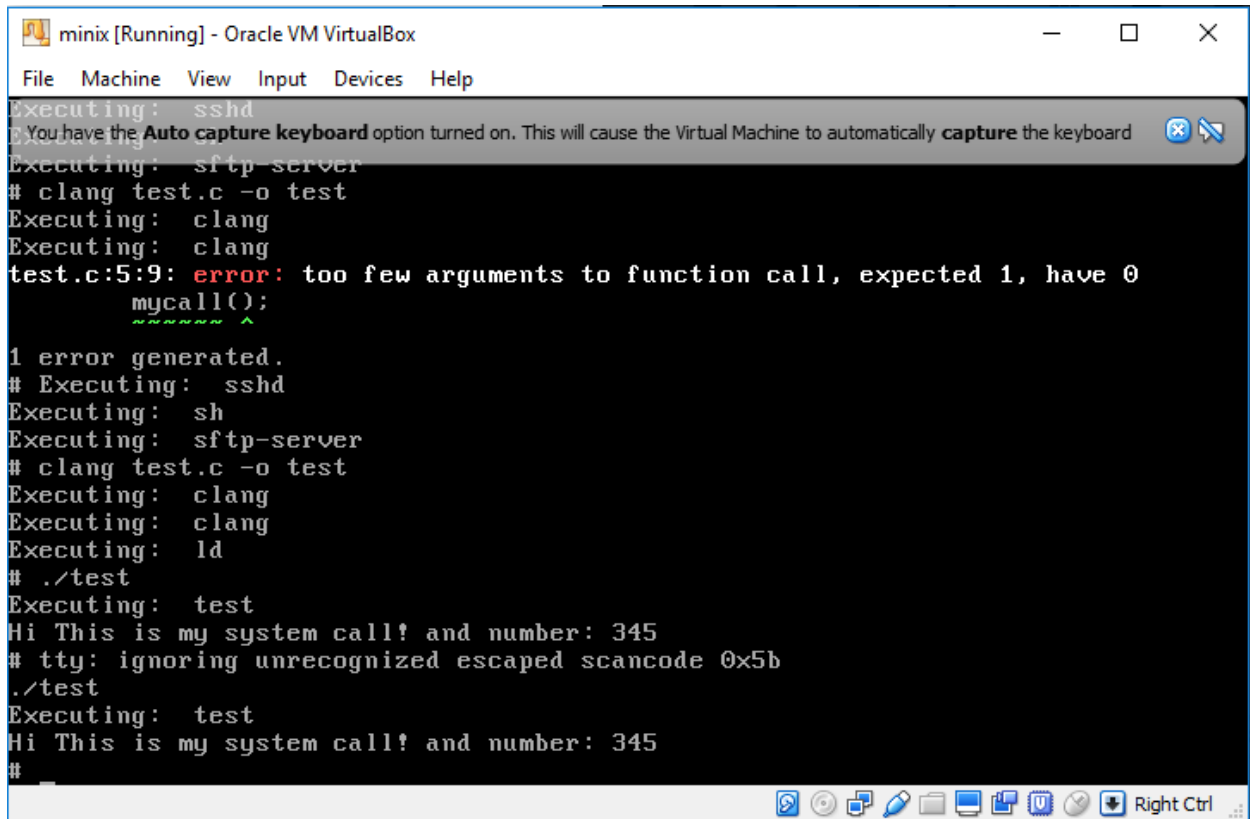
    message m;
    m.m4_l1 = i;
    return ( _syscall(PM_PROC_NR,MYCALL, &m) );
}
```

To pass the long variable through the message variable m.

18. Now in the tester create a long variable and assign the number you want to pass through the parameter and assign the variable to the parameter in mycall.

19. Compile and Run the tester again

20. An output like this one should appear:



The screenshot shows a window titled "minix [Running] - Oracle VM VirtualBox". The window has a menu bar with "File", "Machine", "View", "Input", "Devices", and "Help". A message bar at the top states: "You have the Auto capture keyboard option turned on. This will cause the Virtual Machine to automatically capture the keyboard". The terminal output is as follows:

```
Executing: sshd
Executing: sftp-server
# clang test.c -o test
Executing: clang
Executing: clang
test.c:5:9: error: too few arguments to function call, expected 1, have 0
      mycall();
      ^
1 error generated.
# Executing: sshd
Executing: sh
Executing: sftp-server
# clang test.c -o test
Executing: clang
Executing: clang
Executing: ld
# ./test
Executing: test
Hi This is my system call! and number: 345
# tty: ignoring unrecognized escaped scancode 0x5b
./test
Executing: test
Hi This is my system call! and number: 345
#
```

The bottom of the window shows a taskbar with various icons and the text "Right Ctrl".

4. Some of the processes attributes are:

1. Supplemental groups
2. Signal handling information
3. Child user and system times. Accounting done on child exit
4. Scheduling priority
5. User space scheduling

5. The flag name to know if a process is in use is called: IN_USE

6.A system call that returns the parent or children pid in terms of the received parameter:

```
PUBLIC int do_getProcessIDCall()
{
    int cond = m_in.m4_l1;
    pid_t pid;
    if(cond)
    {
        pid = mproc[who_p].mp_pid;
        printf("PID: %d", &pid);
    }
    else
    {
        pid = mproc[mproc[who_p].mp_parent].mp_pid;
    }
    return pid;
}
```

7. To find file getpid.c type “find / -name getpid.c -print

getpid.c:

```
#include <sys/cdefs.h>
#include "namespace.h"
#include <lib.h>

#include <unistd.h>

#ifdef __weak_alias
__weak_alias(getpid, _getpid)
#endif

PUBLIC pid_t getpid()
{
    message m;

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

As we can appreciate from the code, MINIX_GETPID is used to pass the information.

```
getppid.c:
#include <sys/cdefs.h>
#include "namespace.h"
#include <lib.h>

#include <unistd.h>

#ifdef __weak_alias
__weak_alias(getppid, _getppid)
#endif

PUBLIC pid_t getppid()
{
    message m;

    /* POSIX says that this function is always successful and that
    no
    * return value is reserved to indicate an error.  Minix
    syscalls
    * are not always successful and Minix returns the reserved
    value
    * (pid_t) -1 when there is an error.
    */
    if (_syscall(PM_PROC_NR, MINIX_GETPID, &m) < 0) return (
(pid_t) -1);
    return( (pid_t) m.m2_i1);
}
```

Here the call works the same way but in this case if something unexpected occurs it returns pid-1, otherwise it returns m.m2_i1.

Results and discussion

In this laboratory, we demonstrated how to implement a system call, how to pass parameters through it and how to modify the system to use the system call as its own. Personally, I could understand in a deeper way how a system call works and how to code it to use it and include in ordinary code. And for running code I learnt that Minix has no gcc, you have to use clang but it works in a similar way when compiling. Also, learnt about how the pid is sent through the system when requested and how the Minix micro-kernel communicates within the system and interacts with the user.