Operating Systems – Project Report
PART - I

Team number - 19
Members:
1. Shreyas G - 1PI13CS153
2. Raghavendra G - 1PI13CS113
3. Sharath KP - 1PI13CS141
4. Shashwath S Bharadwaj - 1PI13CS144

Minix Version: 3.2.1

System Call:

Implementation:

We implemented a new system call in minix to print a simple message. The steps followed were as follows:

1. in /usr/src/servers/pm/table.c ,the pm server maintains a list of the system calls and it's numbers.We added our system call and it's number to it.
2. in /usr/src/include/minix/callnr.h a call number definition for our new system call added in step 1 was defined.
3. in /usr/src/servers/pm/proto.h the new system call's function prototype was added.
4. in /usr/src/servers/pm a C file was created that contains the implementation of our system defined in proto.h

The C file had the following code

```
#include <stdio.h>
int do_printmsg() {
   printf("System call do_printmsg called\n");
   return 0;
        }
```

5. in /usr/src/servers/pm/Makefile the name of the C file was added to the list of SRCS so that our implementation would be included in the compiled server.
6. We then compiled the system call and included it in the boot image by executing the following commands:

```
make services
make install
```

The system was rebooted after this.

7. We used the following program to test our new system call:

```
#include <lib.h>    // provides _syscall and message
#include <stdio.h>

int main(void) {
   message m;
```

```
    _syscall(PM_PROC_NR, PRINTMSG, &m);

}
```

The above program was compiled and run. The message ”System call do_printmsg called” was printed ,thereby indicating the success of our implementation of a system call

Understanding execution:

The function linked into user programs to send a message to an operating system process is:
int _syscall(int destination, int fncNum, message *msg);
where
destination: endpoint number of the receiving operating system process
fncNum: an integer associated with the operating system call to be executed.
msg: the address of a message structure

In our program,
_syscall(PM_PROC_NR, PRINTMSG, &m);

1. _syscall leads to the system server identified by PM_PRO_NR (the PM server process) invoking the function identified by call number PRINTMSG with parameters in the message copied to address &m.
2. The call number PRINTMSG is used to map to the corresponding entry in /usr/src/servers/pm/table.c i.e do_printmsg.
3. Next ,the do_printmsg function is called and executes.The result of this is put into a reply message structure .
4. Finally ,the reply message structure is transferred back to the user process.The reply is copied back by the kernel process into the message strucutre parameter m that the user process passed to _syscall.

Top 5 insights:

1. Understanding of what a system call means and how it manifests inside the machine. In minix, a system call refers to call made by a process in layer 4(consisting of user processes) to layers 2 and 3 (consisting of device drivers and servers respectively)
2. Understanding of how a user defined system call is mapped to its function definition and the subsequent execution of the same.
3. Understanding of how to create a user defined library to include functions required by the specific user.
4. Found that the layered architecture and simple UI of minix provides easy methods to modify and understand the OS source code in comparison to environments like Windows where these are pretty much transparent to the user.

Kernel Call:

Difference between System call and Kernel call:

1.A System Call in Minix is how a program requests a service from a server or a driver.
2.A Kernel Call in Minix is how a server or a driver request a service from the kernel.
3.Kernel call is a layer below the System call. System call can invoke the kernel call.

Implementation:

1. The SYSTASK request type of the kernel call was defined in include/minix/com.h and the number of allowed kernel calls was modified to accommodate the new kernel call
2. The prototype of the Kernel call handler was included in usr/src/minix/kernel/system.h
3. The handler's definition/implementation was added to usr/src/minix/kernel/system
4. The request type was bound to the handler by adding the following statement in usr/src/minix/kernel/system.c:

```
map(SYS_MYCALL, do_mycall);
```

5. A new file containing the call was added to usr/src/minix/lib/libsys so that servers and drivers can make the call.
6. The file callnr.h was edited to increase the number of allowed system calls so as to accommodate the system call to the handler
7. A system call handler for this system call was written whose prototype was included in usr/src/minix/servers/pm/proto.h
8. The call was implemented by including a function containing the definition of the system call in usr/src/minix/kernel/system/misc.c
9. The call was mapped to the handler by including the following line at line no 95(96th line) in usr/src/minix/servers/pm/table.c:

```
do_myscall;
```

Understanding Code Flow:

Since user processes cannot directly call functions in the kernel layer, a mechanism has to be provided for the call to be relayed through layers 3 and 2 in order for the call to be executed.

1. A kernel call handler is defined which performs the required task. This is done in 2 parts:
   a. by defining the prototype and
   b. by specifying the implementation of the handler.
2. The call request is mapped to the handler so that the call vector can be properly initialised i.e the request type is bound to the handler.
3. The call defined above when added to the system library can be accessed by servers and drivers of layers 3 and 2 to make the kernel call.
4. At this stage, the kernel call is only defined and made accessible to the other layers.

However, to actually execute it, a system call is to be made.

5. To make a system call, a similar procedure is followed i.e a system call handler is defined by defining its prototype and definition. This is then mapped to the request type so that when the system call is made by a user process, the appropriate kernel call can be executed.

Insights:

1. A clear understanding of the difference between a system call and kernel call leading to appreciation of the layered architecture of minix.
2. Understanding of how access to various services are provided to the user by the minix architecture. Eg: a kernel layer service is not directly provided to the user but is provided through the implementation of a handler which acts like an interface between the different layers.
3. Understanding of how a call is mapped to its definition and how the call is relayed between the different layers i.e how a system call made by the user eventually leads to the execution of a kernel call through appropriate mapping of handlers.
4. A clearer understanding of the directory structure in minix in terms of visibility, accessibility and functionality.
5. Understanding of how a particular service can be very easily made available through an OS by the user by adding a few simple lines of code and recompiling the OS each time. Appreciation of the fact that ultimately, the OS is just a few lines of code.