Group 11 - Report 1

Introduction to Operating System (IOS)

01FB15ECS213	Praveen C Naik
01FB15ECS202	Parashara Ramesh
01FB15ECS209	Prajwal B
01FB15ECS224	Rahul Pillai

Minix Version : 3.2.1

http://www.minix3.org

System Call

Specific experimentation we did :

Reference links : http://homepages.cs.ncl.ac.uk/nick.cook/csc2025/minix/syscall-exercise1.html

Experiment : We tried to pass a parameter to a system call using “messages” (because processes in minix generally communicate with each other via messages)

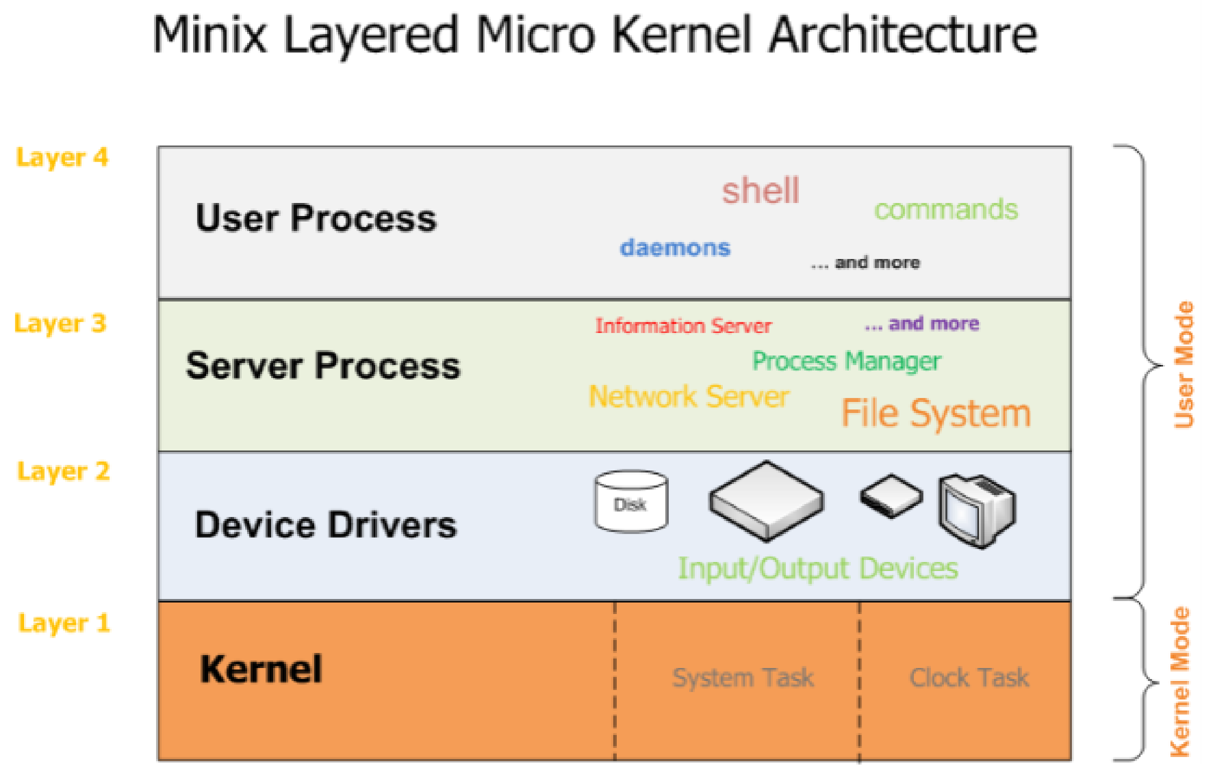
Approach:

1. We used the “message type 1 ie mess_1” as that has member fields that allow is to pass upto 3 integer and 3 char* pointers.
2. In our test code (step 8 of system call) we pass an address of a “message struct “ to the _syscall() function .so we set the value of message.m1_i1 to the value we want to pass.
3. In the doprintmsg.c file (where we have the implementation of the system call) we access that m1_i1 value with the global variable “m_in.m1_i1” which is a global variable set to PM’s incoming message.
4. We compile and execute this test program and in the system call we printed the value of the parameter passed.So this showed the expected output.

Kernel Call

Difference between system call and kernel call:

- Kernel calls are low-level functions provided by the kernel to the server layer and device layer to use services that only the kernel has access to.
- System call are called from the processes in layer 4(user processes) to layer 2 or layer 3.



What we did :

Reference links :
http://wiki.minix3.org/doku.php?id=developersguide:newkernelcall
http://wiki.minix3.org/doku.php?id=developersguide:driverprogramming

A dummy kernel call was done using the official documentation and was tested with the hello driver (which is located in the drivers layer).

Specific experimentation we did beyond the document:

Experiment 1 : Called a kernel call through a system call.
Approach :

- In /usr/src/servers/pm/doprintmsg.c(ie our system call) we added the call “sys_sample()”(which is the kernel call)defined in syslib.h.
- we then tested this with the test.c which is in the root directory(refer system call step 8)
- The prints “Hello world !” (which is printed inside the system call) followed by another print statement which was inside the kernel call ie “inside kernel call!”

Experiment 2 : We tried to pass a parameter to a kernel call through the system call.
Approach:

- We used the message field m1_p1 of type 1 i.e mess_1
- In our test code (step 8 of system call) we pass an address of a “message struct” to the _syscall() function .so we set the value of message.m1_p1 to the value we want to pass.
- In the doprintmsg.c file (where we have the implementation of the system call) we access that m1_i1 value with the global variable “m_in.m1_p1” which is a global variable set to PM’s incoming message after which we create a new message structure with another variable name (ex:mm) and then set that message’s m1_p1 as the global variable’s m1_p1 .This message is passed to the _kernel_call function.
- In the kernel call, we print m1_p1 by accessing it from the message pointer *m_ptr.
- We compile and execute this test program and we printed the value of the parameter passed in both the system and kernel call. So this showed the expected output .

Code flow execution :(Both system call and kernel call where kernel call is called from system call)

- So firstly in our test program we call the system call using the function "_syscall(PM_PROC_NR,PRINTMSG,&m) " _syscall leads to the system server identified by PM_PROC_NR (the PM server process) invoking the function identified by call number PRINTMSG with parameters in the message copied to address &m (where PRINTMSG is #defined to 69 in /usr/src/include/callnr.h)
- It maps " PRINTMSG" to the corresponding call number 69 to the entry for do_printmsg in /usr/src/servers/pm/table.c inside the call vector table(where do_printmsg() is the system call)
- The control transfers to "do_printmsg()" (defined in /usr/src/servers/pm/doprintmsg.c)
- Here we print the message "Hello world!inside system call!" and then we call the kernel call using the function "_kernel_call(SYS_SAMPLE,&m)" (where SYS_SAMPLE was defined in /usr/src/minix/include/minix/com.h like so "#define SYS_SAMPLE (KERNEL_CALL + 40)" where 40 was an unused slot").
- The above task maps to the do_sample() (which is our kernel call) in the file "/usr/src/minix/kernel/system.c" like so "map(SYS_SAMPLE,do_sample)"
- This SYS_SAMPLE request message is handled in the function called do_sample().
- do_sample() returns OK .(since it's a dummy call ,we are just returning okay).

Insights learnt

- We understood the architecture of minix and the difference between a monolithic kernel and a microkernel.Such as Implementing a Monolithic Kernel system call in a Microkernel requires multiple system calls and context switches...as a result there is Performance loss (One of the disadvantages of Microkernel w.r.t system calls)
- we got a first hand glimpse of the file directory structure and observed that each layer has its own directory in the file system.
- we observed that when booting the minix os provides us with multiple options which includes booting the latest version of the os (which might be the case after compiling a new system call or kernel call) along with loading previous versions of the minix os(in case of breakdown of the system so that we can use the previous stable version).
- How does the compiler compile a system call ? - When a compiler compiles a system call it will use the number of the system call rather than it's name.

Reference links :

https://stackoverflow.com/questions/6241627/how-do-system-calls-work https://stackoverflow.com/questions/3546760/how-does-compiler-know-that-the-function-you-used-is-a-system-call

- Only in microkernel systems we have the concept of a “kernel call” whereas in monolithic kernels we have just a “system call” to transfer the control from the user level to the “kernel call”.
- When we tried calling the kernel call via a system call we observed that using the kernel call in a process in the user level is not possible.
- We found out that printf with format specifiers like %d or %f can’t be used inside the kernel call nor inside the system servers as they don’t have a standard output and also we can print only text using printf to the console and log. In kernel , a different kind of printf(defined in libsys called vprintf) is used or we could write our own print function.(we faced this issue when trying an experiment of passing message parameters to a kernel call)
- The processes in layer 2 have the most privileges, those in layer 3 have some privileges, and those in layer 4 have no special privileges. For example, processes in layer 2, called device drivers, are allowed to request that the system task read data from or write data to I/O ports on their behalf.

Part 2 overview :

We would like trace the following system calls:

- Read
- alarm, getsysinfo