Practical 3 - Minix kernel calls

Prac 2 introduced system calls (among other things). This week we take a look at how to communicate information to and from the kernel itself, as well as provide some pointers to how the kernel handles processes.

Trapping (not the raccoons)

The kernel of the operating system runs in a different mode of the CPU, as compared to user applications (and drivers, in Minix 3). In order to enter kernel mode, the software is required to go through well-defined "call gates". The way that this works is that the user application places a call number in a particular place and then "traps" to the kernel, using a software interrupt instruction. This transfers control to the kernel, running in kernel mode. The kernel examines the call number and other information from the user application to determine if the operation is allowed, before executing the kernel call as required. The details of the exact differences between user mode and kernel mode under the x86 architecture are not important to us, but the process of entering kernel mode is. The following questions should guide you through the process of finding how the process works:

- Choose a system call to investigate (and examine how it enters the kernel). The following instructions apply to `/usr/src/lib/posix/_times.c` specifically.
- What is passed to the system call? Where does `_syscall` get implemented, and what does it do? (Don't follow the `_sendrec` for now - we'll follow it later)
- Find where the call is handled (by the PM). What does the PM do?
- Where is the `sys_times` function? What does it do?
- Where is `_taskcall` implemented? What does it do?
- At this point, try to find where the `_sendrec` is actually implemented. HINT: `_sendrec` performs the trap into the kernel. As such, it's hardware dependent. Have a look in `/lib` for some hardware dependent code.
- Once in the `kernel, kernel/mpx386.s` contains the code which jumps back out the C code - find where `sys_call` is called from.
- Within `kernel/proc.c:sys_call`, find where the messages actually get sent or received.

From this point, you should be able to trace how system calls (and kernel calls) use Minix 3's IPC mechanisms to pass information around the system.

Writing a kernel call

In order to better understand how the kernel call interface works, we will implement a new kernel call. The call is `sys_uidc_dump`. When the UID associated with a process changes, `sys_uidc_dump` asks the kernel to print the process num that changed its UID and the corresponding UID for that process to the terminal. Clearly this information could be gleamed from the PM, but this task will demonstrate how to pass parameters to the kernel from a server. The procedure is fairly similar to that for writing the system call as we covered in last week's prac.

First, we need to create a library function which may be called by the process manager code. In order to do this, you will need to do the following:

1. `include/minix/com.h` defines the kernel calls used by the system. You will need to add your call to the list, and increment `NR_SYS_CALLS` appropriately. Also in this file are definitions for the components of each message accepted by each kernel call. (You may like to insert the following (although this isn't strictly necessary): `#define PR_NR m1_i1`. This defines a meaningful name for the process number that will be passed to the kernel).
2. `include/minix/syslib.h` contains prototypes for the system calls. Add an appropriate line for the new call.
3. `lib/syslib/sys_uidc_dump.c` will be a new file which has the code for the new system call. Have a look at other files in the same directory for guidance (e.g. sys_nice).
4. `lib/syslib/Makefile.in` needs to be updated to include the new file.
5. `servers/pm/getset.c` is where the new call will actually be called from. This is where user ID's are changed - find where this happens and insert an appropriate call to the new kernel call.

Now, within the kernel itself, we need to implement the required functionality.

1. `kernel/config.h` defines which of the kernel calls are actually used. Add an appropriate line here.
2. `kernel/system/do_uidc_dump.c` will be a new file which implements the required functionality. Have a look at other files here (particularly `do_times.c`) for ideas.
3. `kernel/system/Makefile` needs to be updated to include the new file.
4. `kernel/system.c` provides the mapping of kernel calls to the functions that implement them. Add an appropriate line here.
5. `kernel/system.h` provides a prototype for the function. Add an appropriate line here.

Once all of these changes have been completed, the system will need to be rebuilt. Because of the changes to the libraries, you will need to "**`make libraries`**" before doing a "**`make install`**" (or alternatively do a **`make clean world`** from `/usr/src`; this will take a very long time to complete). Once this is done, reboot the system. You should see messages from your system call printed out. Write a small test program which manually calls `setuid` and check that your kernel call works correctly.

Summary

This prac has provided instructions for how to write a kernel call and access kernel data structures. It has not covered passing messages back to the user-mode processes - this is for you to work out on your own! The instructions provided here will however be useful for the assignments - consult the tutor if you have any difficulties (either writing the code, or understanding what the code is actually doing!)