Operating Systems Practical 2008/2009 – Documentation for assignment 1

Richard van Heuven van Staereling <richard@few.vu.nl>
Marc Hage <mrhage@few.vu.nl>

Adding the system calls

It was clear that we had to add things to the kernel. System calls are implemented by the system servers. It makes sense to add them in the process manager (the processes are being profiled), even though there is no technical reason to do so. The work that needs to be done must be placed in the kernel, so all that the system call needs to do is pass the call through to the kernel.

To invoke a system call, one runs the **_syscall()**-function with the appropriate system call number, but this is usually not done directly in the user program but in a library. This library is given in the assignment (**profstub.[ch]**). We've placed/edited the following files:

- **/usr/src/include/profstub.h** - contains data structure used to transfer profile info between user program and kernel, and prototypes for functions to invoke system calls. Placed here so any user-space program can access it with **#include <profstub.h>**
- **/usr/src/include/minix/callnr.h** - added the call numbers for our two new system calls
- **/usr/src/lib/other/profstub.c** - contains implementation of the library functions. Most system calls are in the **posix**-directory, but since our call is not defined by POSIX, we had to choose something else. Most other calls are found in **other**, so we added **profile()** and **getprof()** there
- **/usr/src/lib/other/Makefile** - added **profstub.c** to compilation targets

The system calls themselves are implemented in the process manager:

- **/usr/src/servers/pm/table.c** and **../fs/table.c** - changed to map the new system call numbers to the functions **do_profile()** and **do_getprof()** (FS had to be changed as well to take our new system calls into account, but no functionality is added, only null-mappings)
- **/usr/src/servers/pm/proto.h** - added prototypes for **do_profile()** and **do_getprof()**
- **/usr/src/servers/pm/param.h** - added synonyms for variables in a profile message (e.g., **#define prof_pid m1_i1**)
- **/usr/src/servers/pm/profile.c** - implementation of **do_profile()** and **do_getprof()** handlers. Before the kernel call is invoked, some minor checks are done that can be done here
- **/usr/src/servers/pm/Makefile** - added **profile.c** to compilation targets

We have tried to see if it is possible to add profiling code to the PM. However, it became quickly clear that everything we need is in the kernel, and not here. The kernel's process table contains the running processes' program counter, which we will need to increment the correct bin. The PM does not have access to such information. Furthermore, doing bin allocation and administration here creates a lot of overhead in the form of context switches. However, it is possible to do the calculation of the bin size and other such things in the PM. We could move this to the PM (we'll have to use a different message type) if we need to minimize code in the kernel.

Adding the kernel calls

Just like system calls, kernel call are not invoked directly by using the **taskcall()**-function, but by putting it in a library function and using that in your programs (in our case, the PM). The convention is to prefix the calls with **sys_**; our functions are **sys_profile()** and **sys_getprof()** and their only job is to do a task call to the system task:

- **/usr/src/include/minix/syslib.h** - contains prototypes for **sys_profile()** and **sys_getprof()**
- **/usr/src/include/minix/com.h** - added the call numbers and field names for our two new kernel calls
- **/usr/src/lib/syslib/sys_profile.c** and **/usr/src/lib/syslib/sys_getprof.c** - implementation of **sys_profile()** and **sys_getprof()**
- **/usr/src/lib/syslib/Makefile** - added **sys_profile.c** and **sys_getprof.c** to compilation targets

Implementation of kernel calls is in the following files. This is where the real work starts; everything we've described above is basically the context switch from user program to kernel.

- **/usr/src/kernel/system.c** - changed to map the new kernel call number to the functions **do_profile()** and **do_getprof()**
- **/usr/src/kernel/system.h** - added prototypes for **do_profile()** and **do_getprof()**
- **/usr/src/kernel/config.h** - added **USE_PROFILE** and **USE_GETPROF** definitions for consistency
- **/usr/src/kernel/system/do_profile.c** and **do_getprof.c** - implementation of **do_profile()** and **do_getprof()**
- **/usr/src/kernel/system/Makefile** - added **do_profile.c** and **do_getprof.c** to compilation targets

do_getprof() contains the retrieval of profiling information, respectively. Basically, this means copying the administration generated by a profile run from kernel to user memory. **do_profile()** is a bit more interesting. It contains the initialization of the profiler and sets up the data structures. But it is just that: initialization. Since this is just a routine that eventually ends, most likely before the profiled process does, we will have to find another place to put the actual profiling.

Adding the actual profiling code

We decided to add the actual profiling to the clock task. To be more precise, we added a couple of lines to the clock interrupt handler (**clock_handler()**); this function is called every time a clock interrupt is generated. One of the purposes of this function is to make sure the clock task is called only when it's needed, i.e., not on every clock tick. This means that the clock task itself is not called at a fixed rate. The clock interrupt handler, however, is; because clock interrupts are generated at a frequency that is unrelated to the system's activity, we must add our profile code here. Anywhere else, as we've seen, does not generate reliable results.

- **/usr/src/kernel/clock.c** – added initialization of certain profile variables in **init_clock()**, and bin increment code to **clock_handler()**. A function for this (**profile_inc()**) has been created for this.

This function could've been moved to a (currently non-existent) **profile.c** in the same directory but it is only needed here; this way we can keep it **PRIVATE**

- **/usr/src/kernel/profile.h** – variables that are used for the profiler (e.g. to see if the profiler is busy, or the bin allocation) are declared here. We could have put this in **glo.h** as well but this is neater
- **/usr/src/kernel/table.c** – included the **profile.h** file to allocate storage that is defined **EXTERN** there

A side effect of putting our code here is that it is executed even when we are not profiling anything. To minimize damage, it is first checked if the **prof_is_profiling** variable is set. In case it's not, all we waste per clock tick is this **if**-statement. If this is still too much, we could use the **USE_PROFILE** macro by putting an **#ifdef** around this **if**-statement, to enable it only when we compile the kernel with the macro enabled, but this would require kernel recompilation and a reboot, another unattractive solution.

We now have initialization of the profiler and modified the clock in such a way that our profiler actually profiles. To stop the profiler, we can invoke the **PROFILE** system call with pid 0, but we'll have to deal with the other way of stopping as well: once the profiled process stops. Fortunately, the **do_exit()**-routine in the kernel is called whenever a process terminates, no matter what's the reason. Since the exit code is compiled into the kernel binary it has access to the profile variables, allowing it to stop the profiling itself.

- **/usr/src/kernel/system/do_exit.c** - added a check there to stop profiling if the exited process is the profiled process

The user program

The user program is located in **/usr/src/extra** and consists of **profile.c**, **profile.h** and **Makefile**. Its main structure should be clear: fork, let the child **exec()** into the process to profile and let the parent call the profiler. One problem that we have to solve here is synchronization. It would be nice if MINIX had a synchronization server but alas. To emphasize the importance of synchronization, consider the following situation: if profile starts before the child's memory image is replaced, the wrong address range is profiled (namely the range of the profile program itself). The solution for this is found with **ptrace()**.

ptrace() allows you to control the execution of a child process. The parent waits for a notification by the child, indicating that it has started. At this point, the memory image is replaced, so it is safe to use profile. The child is then told to continue and the profiler is started. Whether the program starts before profile does or vice versa is no longer of real importance; only very, very, very small programs might generate a slightly inaccurate result. Because there is no reason to profile a short program, it is not needed to pay any further attention to this. Besides, most short programs can be rewritten to loop a large number of times, still generating meaningful results.