From MEMOS to FIFOS

By Rich West, Boston University.

Overview

In this assignment you are required to take the protected mode code you developed from MemOS (notably `memos-2`) and develop a new version of the system, which we will call FIFOS (the First-In-First-Out System). FIFOS extends `memos-2` with capabilities to schedule threads in a FIFO (or FCFS) order. That is, any newly created or awoken threads are added to the back of a ready queue and when the scheduler is invoked, it picks the thread at the front of the ready queue for dispatching. This assignment requires you to build a standalone system rather than a solution on top of an existing system, so you cannot use pre-existing thread libraries such as Pthreads or a runtime environment such as a Java Virtual Machine (although you can implement your own versions of Pthreads or a JVM as part of your system if you want to give yourself a challenge!).

Details

Using code from `memos-2` that you developed for your primer assignment, you should start by adding functionality to support thread scheduling. For those who did not complete `memos-2` we will provide template source code. You should extend the code to allow threads to work within the default protection domain that is created when GRUB passes control to your system. For simplicity, you can assume a *pool* of N threads is created statically (where N is specified in a header or some boot parameter of your choosing. You can assume the thread pool exists at boot time, and no further threads need to be created dynamically.

For each thread, you should define a *thread control block* (TCB), which maintains appropriate state information. Example state information includes copies of machine register values such as the stack and instruction pointers (possibly others too), and a *thread ID* (TID). Each TID should be a unique integer value in the range [1,N] where N is the number of threads in your pool. The pool itself might simply be an array of pointers to TCBs, with some flag identifying whether or not a thread in the pool has been assigned work.

Initially, threads in your thread pool are all idle and they all have the same priority. You can then assign threads specific work by calling a function `thread_create()`. The function should look something like:

```
int thread_create (void *stack, void *func);
```

In the above, the thread creation function binds a thread in the pool to specific stack and function addresses. It then returns the TID of the thread associated with this call. For simplicity you can assume threads are taken from the pool in order from lowest to highest TID until all threads have been assigned work. Each thread assigned work is then marked as *busy* using a flag in the corresponding TCB. If all threads are busy, you cannot create any more threads and should deal with this case accordingly. If a thread returns from the function it is associated with, you can place it back into an *idle* state, so that it can be assigned new functionality via a subsequent `thread_create()` call.

Scheduling

You should define a *ready queue* for a scheduler that simply operates in FIFO order. That is, each time an idle thread becomes busy, it is
added to the FIFO queue in the order of calls to `thread_create()`. Any thread that later becomes idle is not put back in the ready queue.

First step

You should start by implementing threads that run to completion in a non-preemptive manner and are *stackless*. That is, they have no stack for local variables, arguments, or return addresses for function calls. They call a simple function that performs some task and then return to the idle state. To tackle this, you should look for information on programming ``protothreads'' or *coroutines*. Sample code is available on the course webpage but you are free to search the Internet for additional sources of information. Make sure, however, that any help you find is properly cited.

Second step

If you complete the first step, you should try to add support for threads with their own *stacks*, so they can operate on local variables and/or call other functions. Here, you need to create a region of memory to act as a stack for each thread, and then you need to set the stack pointer to reference the stack when setting up the execution context for the next thread to run.

Third step

If you get this far, congratulations! The next step is to add support for *preemption*. You can do this cooperatively, by implementing a `yield()` function, that suspends the calling thread and picks the next busy thread in the FIFO queue. Any yielded thread simply adds itself to the back of the FIFO queue.

Testing

For testing purposes, assume the functions invoked by your thread creation routing simply write a string to the screen, stating which thread is running. For example, "`Running thread <1>`" or just "`<1>`" will suffice for output from the execution of a thread with ID=1. If desired, you can model the execution time of a thread by a repeated loop that prints multiple text strings to the display. Alternatively, you can implement a function of your choosing to consume CPU time when a thread runs. Once a thread terminates, you should print a message such as "`Done <xxx>`" where "`xxx`" is replaced by the thread ID.

It is highly recommended that you develop your code using the Puppy Linux environment and QEMU. Specifically, you can test the execution of `fifos` using `qemu-system-i386` within Puppy Linux.

Adding Preemption

If you get this far, congratulations! You should now attempt to implement a second version of FIFOS, which supports a preemptive thread scheduler using a *timeout* mechanism. This way, we convert FIFO scheduling to a form of round-robin scheduling, where every thread has a pre-defined timeslice (say, 100 milliseconds). You need to program the 8253/4 programmable interval timer (PIT) chip to do this, along with setting up an interrupt descriptor table (IDT) for the 8259 programmable interrupt controller (PIC) interrupt. The timer interrupt handler should, on timeout, switch control to the thread at the head of the FIFO queue and place the current thread to the back of the queue.

To differentiate from the FCFS implementation, call the second version of FIFOS "`fifos-2`".

Style and Extra Credit

Extra credit will be given to elegant solutions, especially those that support "pluggable" scheduling policies. That is, if your code is well-written it should be easy to replace the scheduler function with a different policy.

As a test of the ability to implement a different scheduler, bonus will be given to anyone who implements "processor capacity reserves". A processor capacity reserve is a constraint (Ci,Ti) for a thread \tau_i, such that τi is limited to a budget of at most Ci units of CPU time every period Ti. If the thread blocks before the end of its period, having consumed Ai actual time (where Ai<Ci), then it is replenished Ai time units Ti-Ai time units in the future after it blocked. To test your processor capacity reserve implementation, you should output a running schedule including timestamps and thread IDs, to show that each thread is limited to Ci time every period Ti. You should test cases where different threads have different budgets and periods, and some threads are forced to yield before consuming their full budget. In a real system, a thread would be forced to yield when it blocks on I/O or access to a shared resource that is currently not available. In this assignment, you can implement a test routine that emulates this arbitrary blocking behavior, to ensure your tasks aren't completely CPU-bound.

If you implement this version of your code, please call it `fifos-pcr`.

Resources

You should look at Volume 3 of the Intel Software Developer's Manual for help on architectural specifics relating to systems programming. A link to the manual is available from the main webpage for the course. This will prove to be particularly useful for the timeout preemption part of the assignment. As with the MEMOS assignment, www.osdev.org is a useful site for information about PIT and PIC programming.

Hints

To help, we will provide some helper files from `memos-2`. To extract, type at the shell prompt: `$tar xvf memos-2.tar`

You can then issue `make` in the `memos-2` directory, followed by:
`$qemu-system-i386 -kernel memos-2 -m XXX` (where `XXX` is some configured amount of machine memory)

*** NOTE: Please do not share class code on public website or repositories ***

You can assume that all the code written for this assignment works in the kernel protection domain, is 32-bit (for both code and data), and the memory model follow a basic "flat" memory layout covering a 32-bit logical space up to 4GB (although, in reality, you will not be able to access 4GB of physical memory if you do not have that much RAM). You do *not* need to enable paging to tackle this assignment. To set up a basic flat memory model you should look at Volume 3 of the Intel Software Developer's Manual (Chapter 3, Section 3.2.1). You will need to set up a basic global descriptor table (GDT) defining a null segment of memory, followed by two other segments: one for kernel code and another for kernel data. The code segment is where your instructions are mapped, while the data area is used for things such as stacks and variables.

Context switching: regardless of whether or not you support timeout preemption, you still need to be able to yield, and hence, switch from one thread to another. Doing this requires saving (or discarding) the state of the thread to be suspended (or terminated), while loading the machine state of the next thread to run. The IA32 architecture supports both hardware- and software-based context (a.k.a. task) switching. For this assignment, you can use either technique but you are strongly recommended to read Volume 3 of the Intel Software Developer's Manual for help. If you use hardware task switching, you should read Chapter 7, otherwise you do not need to use information in that chapter if you use a software task switching technique. For software-based context switching, you should look in Volume 2 of the Developer's Manual, specifically at the `pusha` and `popa` instructions, as they automatically save and restore many of the machine registers you need to manage for each thread. Be aware that you should save `ss, ds, es, fs` and `gs` segment selectors (and set these all to your kernel data segment selector value) on context switching, just to be safe. Although in all likelihood you will not change these selector values in your code once they are initialized, it is good practice to be aware that you are managing state correctly by saving and restoring at the appropriate times, rather than assuming machine state is correct when context switching.

The `flags` register: The flags register (specifically, `eflags` in in the IA32 32-bit x86 architecture) stores some important flag values. For the non-preemptive part of this assignment, you do not need to set up an interrupt descriptor table (IDT) and you can assume interrupts are disabled. However, for the preemption timeout functionality, you need to enable interrupts. The `eflags` register has an interrupt flag field (IF). If set, interrupts are enabled and if cleared, interrupts are disabled. You can use `cli` and `sti` machine instructions to enable and disable interrupts. Note, it is useful on a uniprocessor to use `cli/sti` pairs to manage interrupts when updating code in critical sections, to avoid arbitrary interrupts that could preempt and interfere with machine or program state (leading to data races!). Moreover, for the preemption part of the assignment, you need to enable and disable interrupts at the correct times, to implement a critical section for updating the state of the scheduler ready queue.

The GDT and IDT: For the non-preemptive part of the assignment, you only need to setup a basic GDT. To do this, you should use the machine instruction `lgdt`. For the IDT, you need to read Chapter 6 on Interrupt and Exception Handling (in the Developer's Manual, Vol 3). When enabling interrupts, you need to setup a minimal interrupt descriptor (or vector) table, sufficient to cover the interrupts you care about (notably a timer interrupt). You will need to also setup the PIT and PIC accordingly. Details how to do that are available on www.osdev.org. For each 8-byte IDT entry (a.k.a. descriptor), you can assume they take the form of an Interrupt Gate Descriptor. The layout of these descriptors, which you have to create, as as described in the Developer's Manual for IDT Gate Descriptors.

Submission

You should use `gsubmit` to submit files to a `fifos` directory. You should include all source files, including any Makefiles and build script, as well as a `README` file that documents how to run and test your code. Make sure you reference any sources of information, including code snippets on the Internet that you use to tackle this assignment.

YOU CAN WORK IN GROUPS OF UP TO TWO PEOPLE (RECOMMENDED=2) BUT EACH GROUP MEMBER MUST PROVIDE EVIDENCE OF THEIR CONTRIBUTIONS EITHER IN THE SUBMITTED README FILE OR VIA A PRIVATE EMAIL TO THE INSTRUCTOR.

Happy Programming!