Chapter 14

Multitasking

How do you make multiple processes appear to run at the same time? Today, this question has two answers:

- With the availability of multi-core processors, or on system with multiple processors, two processes can actually run at the same time by running two processes on different cores or processors.
- Fake it. That is, switch rapidly (faster than a human can notice) between the processes. At any given moment there is only one process executing, but the rapid switching gives the impression that they are running "at the same time".

Since the operating system created in this book does not support multi-core processors or multiple processors the only option is to fake it. The part of the operating system responsible for rapidly switching between the processes is called the *scheduling algorithm*.

Creating New Processes

Creating new processes is usually done with two different system calls: `fork` and `exec`. `fork` creates an exact copy of the currently running process, while `exec` replaces the current process with one that is specified by a path to the location of a program in the file system. Of these two we recommend that you start implementing `exec`, since this system call will do almost exactly the same steps as described in the section "Setting up for user mode" in the chapter "User Mode".

Cooperative Scheduling with Yielding

The easiest way to achieve rapid switching between processes is if the processes themselves are responsible for the switching. The processes run for a while and then tell the OS (via a system call) that it can now switch to another process. Giving up the control of CPU to another process is called *yielding* and when the processes themselves are responsible for the scheduling it's called *cooperative scheduling*, since all the processes must cooperate with each other.

When a process yields the process' entire state must be saved (all the registers), preferably on the kernel heap in a structure that represents a process. When changing to a new process all the registers must be restored from the saved values.

Scheduling can be implemented by keeping a list of which processes are running. The system call `yield` should then run the next process in the list and put the current one last (other schemes are possible, but this is a simple one).

The transfer of control to the new process is done via the `iret` assembly code instruction in exactly the same way as explained in the section "Entering user mode" in the chapter "User Mode".

We **strongly** recommend that you start to implement support for multiple processes by implementing cooperative scheduling. We further recommend that you have a working solution for both `exec`, `fork` and `yield` before implementing preemptive scheduling. Since cooperative scheduling is deterministic, it is much easier to debug than preemptive scheduling.

Preemptive Scheduling with Interrupts

Instead of letting the processes themselves manage when to change to another process the OS can switch processes automatically after a short period of time. The OS can set up the *programmable interval timer* (PIT) to raise an interrupt after a short period of time, for example 20 ms. In the interrupt handler for the PIT interrupt the OS will change the running process to a new one. This way the processes themselves don't need to worry about scheduling. This kind of scheduling is called *preemptive scheduling.*

Programmable Interval Timer

To be able to do preemptive scheduling the PIT must first be configured to raise interrupts every x milliseconds, where x should be configurable.

The configuration of the PIT is very similar to the configuration of other hardware devices: a byte is sent to an I/O port. The command port of the PIT is `0x43`. To read about all the configuration options, see the article about the PIT on OSDev [39]. We use the following options:

- Raise interrupts (use channel 0)
- Send the divider as low byte then high byte (see next section for an explanation)
- Use a square wave
- Use binary mode

This results in the configuration byte `00110110`.

Setting the interval for how often interrupts are to be raised is done via a *divider*, the same way as for the serial port. Instead of sending the PIT a value (e.g. in milliseconds) that says how often an interrupt should be raised you send the divider. The PIT operates at 1193182 Hz as default. Sending the divider 10 results in the PIT running at `1193182 / 10 = 119318` Hz. The divider can only be 16 bits, so it is only possible to configure the timer's frequency between 1193182 Hz and `1193182 / 65535 = 18.2` Hz. We recommend that you create a function that takes an interval in milliseconds and converts it to the correct divider.

The divider is sent to the channel 0 data I/O port of the PIT, but since only one byte can be sent at at a time, the lowest 8 bits of the divider has to sent first, then the highest 8 bits of the divider can be sent. The channel 0 data I/O port is located at `0x40`. Again, see the article on OSDev [39] for more details.

Separate Kernel Stacks for Processes

If all processes uses the same kernel stack (the stack exposed by the TSS) there will be trouble if a process is interrupted while still in kernel mode. The process that is being switched to will now use the same kernel stack and will overwrite what the previous process have written on the stack (remember that TSS data structure points to the *beginning* of the stack).

To solve this problem every process should have it's own kernel stack, the same way that each process have their own user mode stack. When switching process the TSS must be updated to point to the new process' kernel stack.

Difficulties with Preemptive Scheduling

When using preemptive scheduling one problem arises that doesn't exist with cooperative scheduling. With cooperative scheduling every time a process yields, it must be in user mode (privilege level 3), since yield is a system call. With preemptive scheduling, the processes can be interrupted in either user mode or kernel mode (privilege level 0), since the process itself does not control when it gets interrupted.

Interrupting a process in kernel mode is a little bit different than interrupting a process in user mode, due to the way the CPU sets up the stack at interrupts. If a privilege level change occurred (the process was interrupted in user mode) the CPU will push the value of the process `ss` and `esp` register on the stack. If *no* privilege level change occurs (the process was interrupted in kernel mode) the CPU won't push the `esp` register on the stack. Furthermore, if there was no privilege level change, the CPU won't change stack to the one defined it the TSS.

This problem is solved by calculating what the value of `esp` was *before* the interrupt. Since you know that the CPU pushes 3 things on the stack when no privilege change happens and you know how much you have pushed on the stack, you can calculate what the value of `esp` was at the time of the interrupt. This is possible since the CPU won't change stacks if there is no privilege level change, so the content of `esp` will be the same as at the time of the interrupt.

To further complicate things, one must think of how to handle case when switching to a new process that should be running in kernel mode. Since `iret` is being used without a privilege level change the CPU won't update the value of `esp` with the one placed on the stack - you must update `esp` yourself.

Further Reading

- For more information about different scheduling algorithms, see http://wiki.osdev.org/Scheduling_Algorithms

References

[1] Andrew Tanenbaum, 2007. *Modern operating systems, 3rd edition.* Prentice Hall, Inc.,

[2] *The royal institute of technology*, http://www.kth.se,

[3] Wikipedia, *Hexadecimal*, http://en.wikipedia.org/wiki/Hexadecimal,

[4] OSDev, *OSDev*, http://wiki.osdev.org/Main_Page,

[5] James Molloy, *James m's kernel development tutorial*, http://www.jamesmolloy.co.uk/tutorial_html/,

[6] Canonical Ltd, *Ubuntu*, http://www.ubuntu.com/,

[7] Oracle, *Oracle vM virtualBox*, http://www.virtualbox.org/,

[8] Dennis M. Ritchie Brian W. Kernighan, 1988. *The c programming language, second edition.* Prentice Hall, Inc.,

[9] Wikipedia, *C (programming language)*, http://en.wikipedia.org/wiki/C_(programming_language),

[10] Free Software Foundation, *GCC, the gNU compiler collection*, http://gcc.gnu.org/,

[11] NASM, *NASM: The netwide assembler*, http://www.nasm.us/,

[12] Wikipedia, *Bash*, http://en.wikipedia.org/wiki/Bash_%28Unix_shell%29,

[13] Free Software Foundation, *GNU make*, http://www.gnu.org/software/make/,

[14] Volker Ruppert, *bochs: The open souce iA-32 emulation project*, http://bochs.sourceforge.net/,

[15] QEMU, *QEMU*, http://wiki.qemu.org/Main_Page,

[16] Wikipedia, *BIOS*, https://en.wikipedia.org/wiki/BIOS,

[17] Free Software Foundation, *GNU gRUB*, http://www.gnu.org/software/grub/,

[18] Wikipedia, *Executable and linkable format*, http://en.wikipedia.org/wiki/Executable_and_Linkable_Format,

[19] Free Software Foundation, *Multiboot specification version 0.6.96*, http://www.gnu.org/software/grub/manual/multiboot/multiboot.html,

[20] GNU, *GNU binutils*, http://www.gnu.org/software/binutils/,

[21] Lars Nodeen, *Bug #426419: configure: error: GRUB requires a working absolute objcopy*, https://bugs.launchpad.net/ubuntu/+source/grub/+bug/426419,

[22] Wikipedia, *ISO image*, http://en.wikipedia.org/wiki/ISO_image,

[23] Bochs, *bochsrc*, http://bochs.sourceforge.net/doc/docbook/user/bochsrc.html,

[24] NASM, *RESB and friends: Declaring uninitialized data*, http://www.nasm.us/doc/nasmdoc3.htm,

[25] Wikipedia, *x86 calling conventions*, http://en.wikipedia.org/wiki/X86_calling_conventions,

[26] Wikipedia, *Framebuffer*, http://en.wikipedia.org/wiki/Framebuffer,

[27] Wikipedia, *VGA-compatible text mode*, http://en.wikipedia.org/wiki/VGA-compatible_text_mode,

[28] Wikipedia, *ASCII*, https://en.wikipedia.org/wiki/Ascii,

[29] OSDev, *VGA hardware*, http://wiki.osdev.org/VGA_Hardware,

[30] Wikipedia, *Serial port*, http://en.wikipedia.org/wiki/Serial_port,

[31] OSDev, *Serial ports*, http://wiki.osdev.org/Serial_ports,

[32] WikiBooks, *Serial programming/8250 uART programming*, http://en.wikibooks.org/wiki/Serial_Programming/8250_UART_Programming,

[33] Intel, *Intel 64 and iA-32 architectures software developer's manual vol. 3A*, http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html/,

[34] NASM, *Multi-line macros*, http://www.nasm.us/doc/nasmdoc4.html#section-4.3,

[35] SIGOPS, *i386 interrupt handling*, http://www.acm.uiuc.edu/sigops/roll_your_own/i386/irq.html,

[36] Andries Brouwer, *Keyboard scancodes*, http://www.win.tue.nl/,

[37] Steve Chamberlain, *Using ld, the gNU linker*, http://www.math.utah.edu/docs/info/ld_toc.html,

[38] OSDev, *Page frame allocation*, http://wiki.osdev.org/Page_Frame_Allocation,

[39] OSDev, *Programmable interval timer*, http://wiki.osdev.org/Programmable_Interval_Timer,