

Notes on the Solution to Tutorial 9: Interrupt Handling

The aim of this tutorial is

1. To draw together your understanding of function calls, interrupts and context switches.
2. To help you gain clarity about how the stack is used in switching from one process to another in a multi-tasking OS.

Write down the exact sequence of events in switching from one CPU-process to another (caused by a timer interrupt, assuming no delays or semaphores).

In the following, text in italics is additional explanation that I do not necessarily expect you to know in detail.

1. The external clock causes a hardware interrupt.
There is an interrupt controller device which co-ordinates interrupts from the various processor-external hardware devices. The interrupt controller signals to the CPU that an interrupt has occurred by raising a pin. The CPU need not always be ready for handling an interrupt. Assuming it is, the interrupt controller communicates which external device caused the interrupt by writing the interrupt number (vector) to the CPU's pins.
The interrupt number (interrupt vector) for the clock device (which happens to be 32) is communicated to the CPU via its pins.
2. The hardware pushes the eflags register, the code segment register and the current program counter (eip register) onto the stack. The CPU also clears the IF flag in the eflags register, thus disabling any further interrupts for the time being.
3. The hardware finds the 8-byte interrupt descriptor corresponding to the 8-bit interrupt vector number that has been received. This contains a far pointer to the first-level interrupt handler routine for this interrupt.
The CPU finds the interrupt descriptor for interrupt vector N as the Nth element of the interrupt descriptor table (IDT), a 256-element array of 8-byte interrupt descriptors.
The IDT has to be initialised by the OS, and the OS informs the CPU of where the table is stored by loading a special register with the base address of the IDT at boot time.
4. The hardware jumps to and starts executing the first-level interrupt handler. This is an assembler routine, for which you have a listing in your lecture notes in Chapter 6.
If you read the Simple Kernel source code (file `int_asm.S`), you will notice that not all first-level interrupt handlers use exactly the assembler that is shown in your lecture notes. In fact, the clock interrupt (32) is handled by a slightly different routine. The reason for the differences is that for interrupts caused by external hardware devices, the first-level interrupt handler communicates with the interrupt controller, selectively masking (turning off) any further interrupts from the same device. This additional code does not affect what happens on the stack; it is what happens with the stack that I really want you to focus on.
5. The first-level interrupt handler pushes data segment registers (es and ds) and all general-purpose registers as well as the base pointer ebp onto the stack.
6. The first-level interrupt handler pushes the interrupt vector number (32 in this case, indicating the device was the clock) onto the stack. This is the parameter for the second-level interrupt handler (a routine written in C). The first-level interrupt handler calls the second-level interrupt handler.
7. The second-level interrupt handler looks up the function pointer for the correct third-level interrupt handler in an array of function pointers ("handlers") maintained by the OS. The second-level interrupt controller calls the third-level interrupt controller (in the case of the clock

interrupt, the function `tick()`, which was been installed in `initTime`). This is done via a normal C function call, with the stack being handled accordingly.

8. The third-level interrupt handler, `tick()`, calls the scheduler `dispatch()`; again, this is a normal C function call.
9. `dispatch()` selects the next process to run, then calls the assembler function `switch`, passing pointers to the PCB of the currently running (old) process and the new process to be run as parameters on the stack.
10. `switch` saves the full context of the currently running process, including `eflags` register, stack and instruction pointers, in the register-save area of the PCB for the old process.
11. `switch` then loads the full context of the new process, including `eflags`, stack pointer and program counter from the register-save area of the new process' PCB. Note that this would originally have been set up by exactly the same call sequence that has just resulted in the old process being de-scheduled. `switch` returns using an `iret` instruction.
12. Parameters to `switch` (two pointers to PCB) taken off stack.
13. Scheduler `dispatch()` returns.
14. Third-level interrupt handler `tick()` returns.
15. Second-level interrupt-handler `int_interrupt()` returns.
16. The first-level interrupt handler pops the parameter passed to `int_interrupt()`, the interrupt vector number, off the stack. The first-level interrupt handler then restores the base pointer `ebp`, all general-purpose registers and the data segment registers `es` and `ds`.
17. The first-level interrupt handler returns using the `iret` instruction. This restores the `eflags` register of the new running process, thus re-enabling interrupts.
18. The new process continues executing.

Draw a diagram showing both the stack and process control block of the two processes involved in the context switch caused by a timer interrupt described above, indicating each step from the sequence above.

Let the process we are switching from be process A and the new process be B.

