CS 321 Lecture Notes

Orion Sky Lawlor
Department of Computer Science, University of Alaska at Fairbanks
http://lawlor.cs.uaf.edu/ olawlor olawlor@acm.org

1 2005/01/31 Lecture: Interrupt Processing

1.1 Interrupt Processing

In addition to executing straight-line code, every processor also has (at least one) special wire headed in that can start a special execution mode called an `interrupt`. On a PC, these interrupt request lines are called IRQs, and are connected to any hardware device that might require immediate processing—for example, the I/O cards (disk, network, serial port, etc.) always have an IRQ so they can interrupt the processor.

When the processor's interrupt line receives a signal, the processor saves what it was doing (saves the processor registers) and executes a special "interupt handler". This interrupt handler then checks the relevant hardware device to see what happened, responds to it appropriately, and resumes processing. From the point of view of the program, it seems as though nothing has happened (except, perhaps, a short delay).

External interrupts can be used, for example, by:

- Network cards to notify the processor that a packet of data just arrived from the network. The processor would then figure out what to do with the packet in the interrupt routine—for example, it might wake up the server program listening for that packet.
- The disk controller to notify the processor that a block of data has finished being written to disk. The procesor could then pass the controller the next block of data to write.

Interrupts are used to notify the processor of all sorts of external hardware (e.g., I/O) events and the passage of time via a timer interrupt. On most machines, calls from ordinary programs and libraries into the operating system ("system calls") are a special kind of interrupt. All sorts of error conditions (e.g., the dreaded Windows "general protection fault" or UNIX "segfault") are reported to the processor via interrupts.

Interrupts are always received by the operating system—regular programs can't normally get access to interrupts. We looked at the interrupt processing sequence for a slightly simpler-than-x86 processor, the PowerPC. Like virtually all machines, when the PowerPC receives an interrupt it jumps to a piece of code at a fixed location in memory–the location is called the "interrupt vector", which the kernel installs in memory.

1.2 Signals

Signals can be seen as a standardized interface for delivering interrupts to user programs. To receive a signal ("add a signal handler"), you call an operating system routine like `signal`, passing in the name of

the signal you want to receive and a function to execute when the signal is received.

Signals used for error handling by all POSIX operating systems (including UNIX and Windows) include:

- `SIGSEGV`, segmentation fault, is delivered when your program accesses an out-of-bounds memory address. If you manipulate the memory map, you can actually resume from this signal.
- `SIGFPE`, floating-point (or arithmetic) exception, is delivered when you divide by zero or encounter a problem with floating-point.
- `SIGILL`, illegal instruction, is delivered when your program hits an invalid instruction, usually caused by overwriting your own code or jumping to a bad function pointer.

Signals can also be used to indicate that I/O is ready (SIGIO, enabled using "fcntl"), that a timer has expired (SIGALRM, SIGPROF, or SIGVPROF, enabled using "setitimer"), that the operating system wants you to shut down (SIGTERM, SIGQUIT, SIGKILL, all UNIX-specific), that various events have happened on the terminal (SIGHUP, SIGWINCH, SIGPIPE, SIGTTIN, SIGTTOU, all UNIX-specific), or for application-defined purposes (SIGUSR1/SIGUSR2, which must be sent explicitly).

Signals, like interrupts, are hence a generic "catch-all" notification mechanism, used for a variety of tasks.