Chapter VII
Exceptions and processor management

VII.1 Exceptions

The control transfer instructions are used for jumps and branches within a user program. Computer processors also need to support another kind of control transfer. For example, a network interface in the computer may receive data from the attached network, and this may require operating system code to run to deal with the incoming data and perhaps set up a response. We don't want every user program to have to keep checking all the I/O systems to see if any of them needs attention, so we need some way for a user's program to be interrupted so that operating system code can run when needed.

An unusual event which causes a break in the normal flow of execution of a user's program is called an *exception*. CPUs provide a hardware exception mechanism which allows the running code to be interrupted, and operating system code to run instead. Once the cause of the exception has been dealt with, control can return to the interrupted code. Note this is not at all the same thing as what happens when a Java program throws what Java calls an exception. In Java, an exception is a software mechanism supported by the Java virtual machine. The exception mechanism we are discussing here is a hardware mechanism in the CPU.

There are two distinguishable kinds of hardware exception, *external* exceptions, which are generated externally to the CPU/memory system, in the I/O systems, and *internal* exceptions, generated inside the CPU or memory system.

Internal exceptions

Internal exceptions are generated internally to the CPU. Internal exceptions are sometimes called *traps*, and they indicate some kind of error or fault. Examples include:

- The program tried to execute an illegal or unknown instruction.
- Arithmetic overflow occurred.

External exceptions

The causes of external exceptions, often called *interrupts*, include:

- An I/O subsystem needs the processor's attention.
- The timeslice timer has timed out — this is used in a multitasking system to interrupt a user program to allow other user programs to run for a while.

In all these cases, we need to interrupt the running program. In some cases the exception is fatal to the program — there is no point in returning to a program that has tried to execute an illegal instruction, for example — and the operating system will terminate the program. In the case of arithmetic overflow, we may want to resume the interrupted program, and in the case of a memory fault, the operating system may be able to fix the problem and then return to the interrupted program. (More on this later in the course.)

VII.1.1 Exception handling

A signal from the I/O subsystem, timer, memory system etc., called an *interrupt request*, indicates that attention from the operating system is needed. This signal is checked by the processor control at the end of each instruction, and if it is asserted, then instead of fetching and executing the next instruction in the normal way, the processor executes exception processing steps. These involve saving the Program Counter in a safe place, so we can return to the interrupted code once the interrupt has been dealt with, and then setting the Program Counter to a known address – the address of the entry point of the *interrupt handler* code, which is part of the operating system.

When an exception occurs in a MIPS processor, the PC is saved in a special processor register called the *Exception Program Counter (EPC)*, and the PC is then loaded with the value 0x80000180[1], which is the address where the interrupt handler starts. At the end of the interrupt handler code, we jump to the address in the EPC, to return to the interrupted program where we left off.

The interrupt handler will undoubtedly want to use some of the processor registers while it is dealing with the interrupt, but because an interrupt can happen between any two instructions in a user's program, it is vital that none of the registers have been changed when we return from the interrupt handler to the user's code. Therefore the first thing the interrupt handler does is to save the contents of any registers it will be using into a safe place in memory, and the last thing it does before returning is to restore those register values. This process is quite similar to a function call, but no registers can be modified.

Cause register

The interrupt handler needs some way to find out what caused the interrupt: was it the disk controller, or the timer, for example? The MIPS processor actually has six interrupt request signals — asserting any of them will cause the processor to deal with the interrupt as described above, and each of these signals can be supplied by a different I/O controller. The interrupt handler can find out which interrupt request signal caused the interrupt by looking in another special processor register, called the *Cause* register.

[1]Note that SPIM uses 0x80000080.

Interrupt masking

After an interrupt is requested, and the CPU starts fetching instructions at the entry point of the interrupt handler, the interrupt request signal from the I/O controller will still be asserted — the I/O controller does not yet know that the CPU has started dealing with the interrupt. What we don't want is for the CPU to recognise the interrupt request again after executing the first instruction of the interrupt handler, and to interrupt the interrupt handler. To avoid this, the interrupt request signal is automatically *masked* when the interrupt is recognised. This involves setting a bit in another special CPU register, the *Status* register, known as the *Exception Level (EXL)* bit. While this bit is set (1), the interrupt request signal has no effect.

Status register

By the end of the interrupt handler code, the interrupting I/O controller will have de-asserted its interrupt request signal, having had its needs dealt with. When control returns to the interrupted program therefore, the EXL bit is cleared again, so that subsequent interrupt request signals will be recognised.

The three extra processor registers introduced, the EPC, Cause and Status registers, can be accessed using instructions that allow their contents to be transferred to or from the 31 general purpose MIPS registers. The return from the interrupt handler to the interrupted program is implemented by transferring the EPC contents to the PC. A special MIPS instruction, `eret` (exception return), handles the return to the interrupted program.

Exceptions are in reality much more complicated than presented above. Section 7 of appendix A of the course textbook (P&H) provides some more details, but for course examination purposes the above simplified procedure suffices.

Vectored interrupts

Some processors use an alternative way to handle exceptions compared to the one described above. When an exception happens, the processor starts fetching instructions from an address which specifically corresponds to the type of exception. This method is called vectored interrupts and a Cause register is not needed since there are separate exception handlers for each exception.

VII.1.2 Software exceptions

The exceptions described above are *hardware* exceptions. However, most processors also allow the exception mechanism to be invoked by an instruction. In the MIPS, the `syscall` instruction, short for *system call*, causes exactly the same sequence of steps as a hardware exception. The interrupt handler can tell that the exception was caused by a `syscall` instruction by looking at the Cause register.

The name of the `syscall` instruction gives away its purpose – it is the way that user programs make calls to operating system functions. There is a whole range of functions that the operating system provides for user programs, for example: performing I/O, allocating more memory to the program, communicating with another program, terminating this program etc., but there is only one syscall instruction. The user program indicates which system service it is requesting by placing system call parameters in specified general purpose processor registers before executing the syscall instruction. The interrupt handler discovers from the Cause register that the exception was caused by a system call, and then uses the supplied parameters to determine which operating system function is being invoked. After executing that function, returning from the interrupt handler in the usual way returns control to to the user program at the instruction after the syscall.

VII.2 Kernel and user mode protection mechanism

Why access operating system functions via the exception mechanism, instead of calling them in the same way that a program calls one of its own methods, i.e. with a jump and link instruction? The answer lies in the *dual mode protection* mechanism that the all modern processors provide. The CPU has two distinct modes of operation: *user* mode, which is the mode in which user programs execute, and *kernel* mode (also called system or supervisor mode), which is the mode in which the operating system executes. Which of these modes the CPU is in is determined by a bit in the CPU Status register.

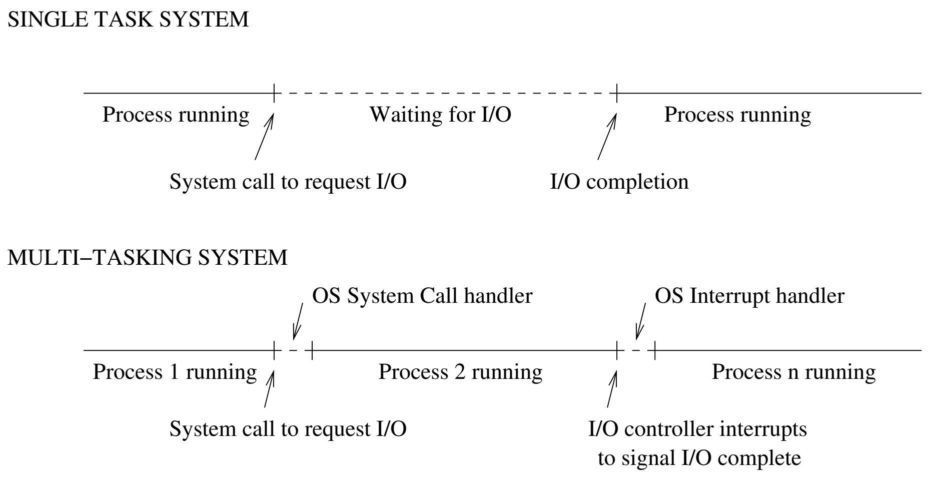
Some CPU instructions are *privileged*, and will only work in kernel mode. These include instructions for accessing I/O systems, and instructions that alter the processor Status register, including the kernel/user mode bit. Any attempt to execute a privileged instruction while in user mode will cause an internal exception and thus call the operating system, which can terminate the offending program.

Since the exception mechanism involves control passing to part of the operating system, the kernel/user mode bit is automatically set to kernel mode when an exception is recognised, so that the interrupt handler runs in kernel mode. The eret instruction executed at the end of the interrupt handler switches the mode back to what it was before the exception. An exception is the only way that the mode can switch from user to kernel mode, and this is why a user program calls operating system functions using a syscall instruction, not an ordinary jump and link instruction, which would leave the processor in user mode.

Entering the operating system via a syscall instruction also has the advantage that the operating system can only be entered from user programs at one fixed entry point: the interrupt handler entry point (0x80000180 in the MIPS). User programs cannot jump into any other part of the operating system. This can be enforced by the memory protection mechanism described later in the course.

VII.3 Processor management

Multi-tasking systems share the processor between several processes resident in memory at the same time, by switching execution between the *processes*[2], under control of the operating system. This idea was originally developed to keep the processor busy while a process is unable to proceed because it is waiting for input or output from/to a relatively slow I/O device:



Process 1 requests an I/O operation by executing a system call, which is handled by a part of the operating system. Rather than waiting for the I/O to complete, then returning to process 1 (upper line in diagram), the OS starts another process running (lower line). When the I/O completes, the I/O controller hardware generates an interrupt, which causes an interrupt handler in the OS to run, to deal with the completion of the I/O operation. Process 1 may now be restarted, or, alternatively, process 2 or some other process resident in the memory of the machine may run next.

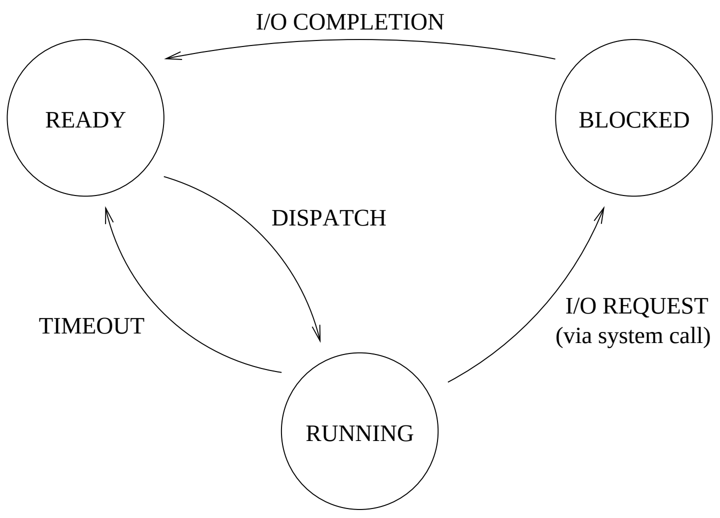
While it is waiting for its I/O request to complete, process 1 is said to be *blocked* — it cannot run further for the time being. A process which is not currently running, but which is able to run immediately, is said to be *ready*. The operating system maintains lists of processes which are ready, and of those that are blocked, with an indication of the event (e.g. I/O completion) which each of the latter is waiting for.

The operating system is always entered via some kind of exception (a system call, an I/O interrupt or an internal exception) — remember that this sets the processor into kernel mode. It then runs for a short time to deal with the

[2]A process is a program in execution.

exception, and then starts one of the ready processes running. The part of the operating system which decides which ready process to run next is often called the *dispatcher*, and it must ensure a reasonably fair division of processor time.

Each process moves between three different states: ready, blocked and running:



In order to prevent a process which makes no I/O requests and does not generate any kind of internal exception, from hogging the processor, a timer is set up to interrupt the processor after a process has run for a certain length of time (its *timeslice*), so that another process may be dispatched. Operating systems which support this feature are called *pre-emptive* multi-tasking systems and their dispatcher includes the state transition labelled *timeout* in the figure.

The switching of execution from one process to another, for any reason, is known as a *context switch* – the context of a process is its complete state (i.e. the contents of its data areas and of the processor registers, including the program counter, while it is running). When an interrupt occurs, the context of the interrupted process must be saved, so that the process can be dispatched again later. The process's data in memory will not get altered between the interruption of the process and its next dispatch, as we will assume that no other process will access this process's memory space. However the processor registers will obviously be used by other processes in the meantime, so their contents must be saved. The interrupt handler must copy them into a save area (unique to the interrupted process) in system memory, so that the dispatcher may dispatch a different process by loading *its* saved register contents into the processor registers and executing a return from exception. The registers are saved in an area of system memory known as the *process control block* (PCB).

There is one PCB per process in the computer, and it holds all the information the OS requires to manage the process, including: process ID, process state information (blocked, ready, etc.), the saved processor register contents, information on the process's priority, information about which parts of the machine's memory the process is using, information about the I/O resources allocated to the process (e.g. serial port, tape drive, etc.)

VII.3.1 Creating and Destroying Processes

When a multi-tasking system is started, a number of processes are created, including for example a login process, which outputs a login prompt on the screen and waits for a username to be typed. Processes may, by using an operating system call, request the creation of new processes. For example, when the user runs a program in a shell window, the shell process must not be lost, as the user will want to resume using the shell when the new program terminates. The shell process requests the creation of a new process in which the new program will be run. If the OS decides there is 'room' in the machine, it creates a new PCB, and allocates resources to the new process.

The OS keeps track of the relationship between the new process and the process which requested its creation – they are known respectively as *child* and *parent*. The creation of a child process is often called *spawning* a new process. Processes may be destroyed at their own request or at the request of some other process, and this involves freeing the process's resources and deleting its PCB.

VII.3.2 The operating system kernel

The parts of the operating system which execute as a result of exceptions and system calls need to be short and efficient (so that user processes get to execute as much as possible of the time, and interrupts are not disabled for too long). These sections of code reside in a reserved memory space, and are known as the *kernel* or *nucleus* of the operating system.

Typical kernel functions include:

- Interrupt handling.
- Process creation and destruction.
- Process state switching.
- Memory management (covered later in the course).
- Inter-process communication and synchronisation.
- I/O support.