Chapter 6
Interrupts

Revised: Feb. 14, 2011

6.1 INTRODUCTION

An important feature of almost all present-day machine architectures is the **interrupt** facility, which provides a mechanism to temporarily suspend one program and execute another one when some special event (detected by the hardware) has occurred. Interrupts were first introduced to provide the earliest type of concurrency in computer systems by allowing I/O transfers to proceed while the processor dealt with other tasks. The interrupt mechanism allows an I/O device to carry out a data transfer without the need for constant polling of the device by software. Instead, the processor can concentrate on other tasks, and the device will "interrupt" the processor when it requires attention. This mechanism also allows the software to efficiently monitor and respond to a variety of external events, as well as special situations that arise within the current program. In addition, interrupts enable the system to keep track of the time of day and to set alarms to time specific activities.

An understanding of interrupts, therefore, is important for dealing with issues that arise in time management, I/O device management, and many other resource management areas.

An analogy from the kitchen might aid in understanding interrupts. In cooking a roast, one could place a thermometer in the roast and constantly check it for the correct temperature. Some ovens provide for a temperature probe and an automatic reading of the temperature of the meat. The oven shuts off and perhaps generates an audible alarm when the meat has reached the desired temperature. You can go about other chores, knowing that the oven will inform you of the completed event. Another obvious alternative would be to set a timer that will produce an audible alarm at the time the roast should be done. You would then need to check a meat thermometer to be sure your meal is actually ready. Figure 6-1 compares the technique of continual checking, or polling, with the automatic signaling of event completion.

The "interrupts" generated by the oven in this example will likely call you away from another task, such as reading a book or writing a letter. One thing that is very important when leaving an interrupted task is to remember where you were in that task. For example, you may be reading a book when the oven begins signaling the completion of the event. If you close the book without marking the place where you were reading, it might be difficult to find your place

quickly. Even if you have used a bookmark, some thought will be required to relocate the exact sentence at which you stopped.

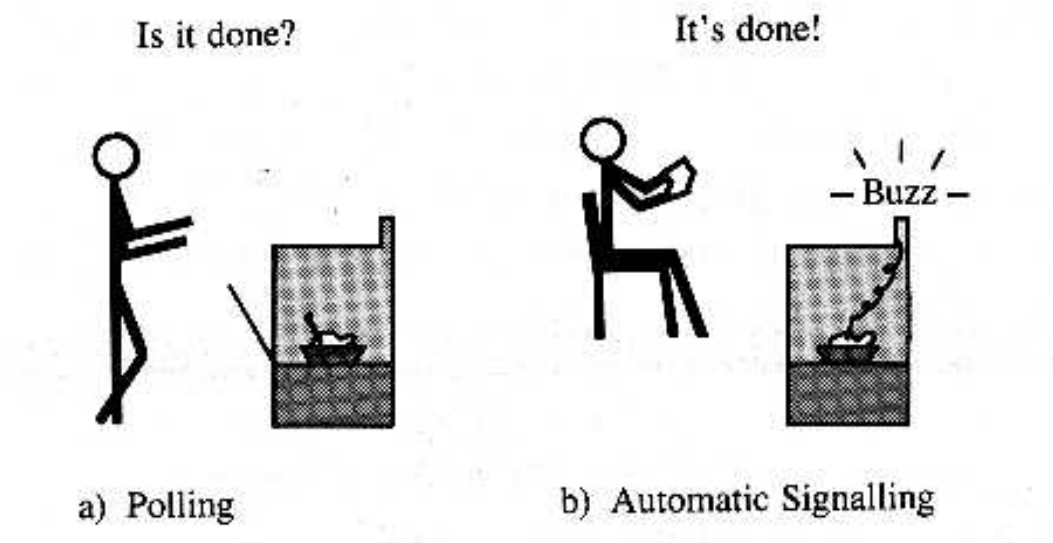

Figure 6-1: Polling and Interrupts Compared

As human beings, we often find it difficult to remember our place in what we were doing when more than one "interrupt" occurs in a short period of time (for instance, if the oven signal, telephone, and doorbell all ring within 15 seconds). This problem is illustrated by Figures 6-2 and 6-3.

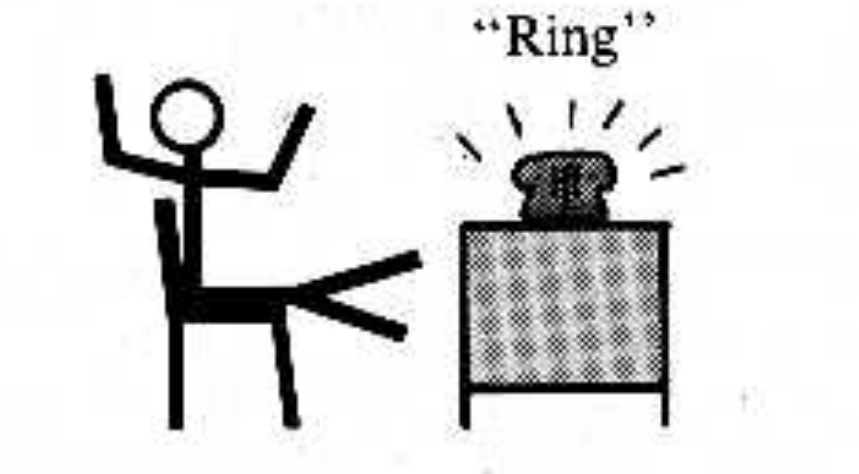

Figure 6-2: Human Interrupt Processing

Computer interrupts are similar to those demonstrated in the oven example. The automatic temperature probe mechanism is like a device interrupt that informs the CPU when a device has completed an operation. Some timer interrupts are also similar to the one demonstrated by the oven timer. In processing any interrupt, a computer that is interrupted must "remember" precisely what was executing and preserve the integrity of its environment.

Figure 6-3: Multiple Interrupt Handling

Unlike human beings, computers can handle thousands of interrupts per second. For every interrupt that occurs, however, the hardware and software must remember what was interrupted and guarantee that nothing is changed in the expected environment of the interrupted process.

With the analogy of everyday interrupts in mind, let us consider how interrupts work in a computer system.

6.2 CATEGORIES OF INTERRUPTS

The first question to consider is what type of events can bring about interrupts in a computer system. Most interrupts can be classified in the following categories: I/O, alert, timer, system request, program fault, and machine fault. Not all computer systems have all categories of interrupts. Almost all, however, have I/O, timer, and system request interrupts of some form or another.

I/O Interrupts. An I/O interrupt is generated by an I/O device (more precisely, by the device interface). This signal indicates that service is needed from a software module called an interrupt handler. Types of service that may be required include transferring another character to or from a

-

character-by-character device, servicing the completion of a total I/O request like a block of data being read or written, or handling an error condition that occurred during the I/O operation.

Alert Interrupts. An alert interrupt is a somewhat unexpected interrupt that generally results from conditions outside of the computer system. Some keyboards provide an interrupt key, which a user or operator can use to activate the command interpreter. This could be considered an alert interrupt. Another example occurs in a multiprocessor, when one computer signals another to signify some condition or request service from the other CPU.

Timer Interrupts. A timer interrupt could be considered a type of I/O completion interrupt, but since no physical I/O occurs, it is best to think of these interrupts in a category of their own. An interrupt generated from an interval timer means that the last unit of time placed into that timer (counter) has expired. An interrupt may also be generated at regular intervals by a hardware clock. Each interrupt means that another fixed fraction of time has expired, and that the software clocks should be updated accordingly.

Machine Fault Interrupts. A variety of interrupts may signal that a hardware error has occurred. These machine fault interrupts can indicate a memory fetch error, I/O device error, etc. Many operating systems simply report such errors and terminate operation. More sophisticated operating systems, with suitable hardware available, might attempt to isolate a failing component, lock it out to prevent further use, and keep running with alternative components.

The above categories of interrupts are all caused by events outside the running program. The final two categories are caused by the program itself.

System Request Interrupts. In many systems, processes request service from the operating system by a special machine instruction which generates a system request interrupt, also known as a supervisor request. The system call instruction, described in connection with the Program Interface, is an important example of this category. Other instructions, with names such as TRAP and IOT, may perform a similar action. These instructions cause a transfer of control to the operating system. A parameter block may be used to transfer arguments to the OS, identifying the specific type of service requested. The operating system will analyze the request and provide the requested service, possibly blocking the process requesting the service until the requested service has been completed (e.g., an I/O operation).

Program Fault Interrupts. Abnormal conditions that arise within a running program may result in a program fault interrupt. Possible causes of a program fault interrupt may include:

- an attempt to divide by zero
- overflow or underflow during an arithmetic operation
- an invalid memory reference
- an invalid or privileged instruction
- an attempt to access a protected resource
- an operation attempted on incorrectly formatted data

-

Users can specify how such errors are to be handled within their programs by specifying the address of a routine to execute when an error occurs. If they do not specify such an error routine, the operating system will normally terminate the execution of the process.

System request and program fault interrupts are classified as **synchronous interrupts**. Since they are caused by the program directly, there is a precise point within the program at which the transfer of control should occur. Most other interrupts are **asynchronous interrupts**. Although they may require service as soon as possible, the exact point at which the program should be interrupted is not predetermined.

6.3 INTERRUPT GENERATION

Most interrupts begin as physical signals sent to the processor when certain events occur. This is most clearly seen for I/O interrupts. If a printer is attached to your computer, for example, the printer may be set up to generate an interrupt when it has finished printing a line or a page. In addition, the printer may generate an interrupt of a different type if it encounters an error such as a paper jam.

Interrupt signals of other types are generated within the computer itself for a variety of reasons. Examples include: if a hardware error is detected, if the user pushes an "attention" button, or on every "tick" of the computer's clock. Finally, some conditions that occur within the program itself may cause an interrupt signal (just as your microwave may warn you if you push an invalid combination of buttons).

Typically, each interrupt signal is recorded when it occurs by setting a flag in a special processor register. The number of separate flags can vary from one to a dozen or more. However, it is usually much smaller than the number of possible interrupt types, so many interrupts share a single flag. Interrupts are thus grouped into a small number of classes. This grouping is discussed further in Section 6.5.

The processor can be set by special instructions to ignore (most) interrupts, to accept all interrupts, or to accept some and ignore others. If an interrupt is initially ignored it may be possible to service it later. However, if there is a long delay, the condition that caused the interrupt may no longer exist.

If an interrupt signal is accepted, the processor will "handle" the interrupt as soon as possible, which usually means immediately after the current machine instruction. Individual machine instructions are rarely interrupted, but interrupts may occur at almost any point between instructions.

If multiple interrupts occur, they must be handled one at a time according to some ranking. It is often possible, however, for one interrupt to occur during the handling of another.

-

6.4 INTERRUPT PROCESSING

As long as interrupts do not occur (or are not accepted), execution proceeds sequentially from one instruction to the next until a branch instruction causes a transfer of control to another memory location. When interrupts occur, transfer of control can also occur automatically when an interrupt condition is signaled. This automatic transfer mechanism, like a call instruction, must provide the capability to save the address for the next instruction that would have been executed had the interrupt not occurred. The problem is actually more difficult, because when the interruption is complete, the exact state of every CPU register and status flag must be restored just as if the interrupt had never occurred. While different computer architectures provide different mechanisms for saving this information, the basic concept is the same. After each instruction is executed, the hardware performs the following algorithm or its equivalent:

```
IF interrupt THEN
     save execution address (PC)
     save status information
     transfer control to interrupt handler
ENDIF
```

The precise control transfer mechanism is dependent on the computer's architecture. There is no branch or call instruction, so the handler address is not explicitly specified. In a few architectures, the device or element generating each interrupt actually delivers a handler address to the processor along with the interrupt signal. In the more usual mechanism, each interrupt is associated with the address of a small data structure in low memory called an interrupt vector. This vector, in turn, specifies the handler address, and may also provide a place to store the return address and status.

Most computers today are **fully vectored** (or nearly so) which means they provide a distinct interrupt vector for each interrupt. The algorithm performed by the hardware in such a system then becomes:

```
IF interrupt THEN
     save execution address (Program Counter)
     save status (Program Status Word)

     CASE interrupt IS

       WHEN device1: transfer control to
          device1 interrupt handler
```

```
            WHEN device2: transfer control to
               device2 interrupt handler

            WHEN device3: transfer control to
               device3 interrupt handler

            WHEN device4: transfer control to
               device4 interrupt handler

        ENDCASE
    ENDIF
```

With this method, the cause of the interrupt can be determined without further program execution.

Other systems define a limited set of categories of interrupts, and provide only a small set of interrupt vectors (perhaps only one). For example, in the IBM S/370, all I/O interrupts caused control to pass to the same **first-level interrupt handler (FLIH)**. This handler then tested a special register to determine which I/O device caused the interrupt. The FLIH then branches to the appropriate **second-level interrupt handler (SLIH)** to process the interrupt. The logic of the FLIH is:

```
    FLIH:save registers
         read interrupt_status_register
         CASE interrupt_status_register IS
           WHEN device1: call device1_SLIH
           WHEN device2: call device2_SLIH
           WHEN device3: call device3_SLIH
         ENDCASE
         restore registers
         RETURN from interrupt
```

Although current systems hava distinct vector for most interrupts, first and second level interrupt handling still occurs in some special cases.

Just how can the execution address and the status be saved? Two basic techniques are prevalent:

1. The information may be saved on a stack.

2. The information may be stored in a special memory location, usually part of the interrupt vector.

Let us consider an example from the PDP-11. Each device on the PDP-11 has assigned to it a unique address (by hardware configuration), in low memory, called an interrupt vector. The

-

assignment of a vector location for a particular device depends on the hardware configuration. Each interrupt vector helps determine where to branch as well as the new status word to use upon the occurrence of an interrupt caused by that device. Each vector consists of two 16-bit words. The first is the address of the interrupt handler. The contents of this word are placed into the PC after the necessary information about the interrupted program has been saved on the stack. The second word is a new status word to be transferred to the PSW before control is given to the interrupt handler. The contents of the previous PC and PSW are saved on a stack. The algorithm for the hardware interrupt facility invoked after each instruction on the PDP-11 is:

```
IF interrupt THEN
     push PS
     push PC
     IF cause=device1 THEN
          PC := vector(device1)
          PS := vector(device1)+2
     ENDIF
     IF cause=device2 THEN
          PC := vector(device2)
          PS := vector(device2)+2
     ENDIF
     .
     .
ENDIF
```

A difficult timing problem can arise during interrupt processing with high-performance processors. Because these systems use pipelining and similar techniques to overlap instruction execution, they sometimes cannot service interrupts at the end of the exact instruction during which they occurred, even if that instruction caused the interrupt. These processors are said to have **imprecise interrupt handling**. This can be a problem for correct handling of synchronous interrupts.

6.5 INTERRUPT ORDERING AND MASKING

Systems that generate interrupts usually assign priorities to various interrupt types. This means if two interrupt signals occur simultaneously, one has priority over the other. Once the high-priority interrupt has been serviced, the lower-priority interrupt will be generated. It is possible for interrupt servicing to be nested; in other words, a low-priority interrupt handler's execution can be interrupted by a higher-priority interrupt, thus invoking another interrupt handler. Once the higher-priority interrupt is serviced, a return from interrupt instruction is issued and the lower-priority interrupt handler regains control to complete processing.

Sometimes such interrupting of an interrupt handler is undesirable. This can be controlled in most systems by the use of interrupt masking. This technique allows certain interrupts to be blocked, so they will not be accepted by the CPU. Interrupt masking is performed by setting bits

-

in a CPU register, called an interrupt mask register, which correspond to specific interrupts or classes of interrupts to be blocked. An alternative method is used by the PDP-11: each interrupt is assigned a priority level, and a priority value in the program status word may be set greater than the priority of certain interrupts. In this case, only interrupts having a priority greater than the priority in the PSW will be accepted. Such blocked interrupts remain pending until the processor priority is reduced to a value less than the signaling priority of the previously blocked interrupt.

It should be obvious that masking or blocking of interrupts in a real-time system must be minimized. Rules for how long interrupt handlers can run with interrupts blocked are established and strictly observed in implementing real-time operating system interrupt handlers.

Certain interrupts are immediate and predictable; they occur as the result of a machine instruction, such as a system call or trap instruction. Such interrupts generally cannot be masked. Program fault interrupts also occur immediately, although they cannot be predicted because they result from the program environment (e.g., dividing by a register whose contents is 0). Similarly, machine fault interrupts occur immediately when the condition is detected.

6.6 STRUCTURE OF AN INTERRUPT HANDLER

There are several ways in which the structure and design of an interrupt handler must differ from that of an ordinary procedure. In general the following issues must be considered:

1. A linkage must be created between the physical interrupt and the handler, so that the handler will be invoked when the interrupt occurs.

2. Since interrupts may occur at unpredictable points in program execution, the complete system state must be preserved during interrupt handling. This means that after the interrupt the content of all machine registers must be exactly as before; moreover, all memory used by the running process, including "hidden" areas such as a stack, must be undisturbed.

3. Parameters cannot be passed to an interrupt handler in the usual way.

4. Because interrupt handling is a temporary deviation from the normal execution of a running process, the handler may be severely restricted in the resources it may use, and in the subprograms it may call.

5. Interrupt handlers must allow for the appropriate handling of other interrupts while the handler is in progress.

6. Interrupt handlers must complete their work in the shortest possible time, leaving more complex followup tasks to other system or application software.

7. Interrupt handling is subject to errors that are hard to reproduce, and cannot often be debugged by normal means.

We will consider each of these points in more detail.

Linking to the Interrupt

Each (first level) interrupt handler is linked to the appropriate physical interrupt by storing its starting address in the associated interrupt vector location. These locations must be set up by the OS during system initialization. When the interrupt occurs, hardware mechanisms access the contents of the vector and cause a transfer to the specified handler.

Saving the State

When an interrupt occurs, it is generally necessary to save all registers that could be modified by the interrupt handler on a stack or in a private memory area. The hardware itself may save some critical registers such as a status register, along with the return address. The handler software is then expected to immediately save all other registers that might be used.

When the interrupt handler terminates, the software must restore all the registers originally saved by software, and the hardware restores those saved by hardware. For this reason a special return from interrupt machine instruction is normally used rather than an ordinary return from procedure.

One problem that arises on a stack-based machine such as the IBM-PC is caused by the fact that an interrupt cannot avoid having some effect on the current stack. When an interrupt occurs, the hardware automatically pushes some status information onto the stack. No matter when the interrupt occurs, the stack pointer register must be pointing to a valid stack with sufficient room for these values. If you ever set this pointer to an invalid stack, however briefly, you must first disable interrupts.

Handling Parameters

Since interrupt handlers are not called by the usual procedure call mechanism, parameters cannot be passed in the conventional way. If a handler requires input information such as the identity of the exact cause of the interrupt, this information must be preloaded in a standard place (such as special registers) by the interrupt hardware.

Resource Use and Procedure Calls

In general, it is dangerous and inappropriate for an interrupt handler to make use of resources that could also be in use by the current process. An interrupt handler should not ordinarily access files or I/O devices, nor should it allocate memory or use any memory other than its own. Except for debugging purposes, an interrupt handler should not display messages on the terminal screen.

-

Caution must also be used by an interrupt handler in calling other functions or procedures. It is important to be sure that these procedures do not in turn use inappropriate resources or violate any of the interrupt handler disciplines. In addition, if a stack mechanism is used, each function call adds further data to the stack, which can lead to overflow if the available stack capacity is unknown or limited.

Another problem occurs whenever an interrupt handler calls a subprogram which could also be called by the interrupted programs. This could lead to an attempt to re-execute the subprogram before a previous execution is finished. Trouble will occur unless the design of the subprogram allows this type of reuse. A program which can be re-entered in this fashion is called **reentrant.** Any program that accesses global (static) data is not reentrant. If two instances of such a program are active, they will both try to manipulate the same global data structure.

Handling Other Interrupts

During normal operation, a typical computer must respond to a variety of interrupts, some at a high rate. In particular, clock interrupts may need to be handled many times per second, or else the system clock will become inaccurate. When an interrupt occurs, the hardware automatically disables other interrupts; this is necessary to avoid confusion during the initial saving of the system state. However, it is necessary for each interrupt handler to re-enable interrupts as soon as possible to avoid problems.

If an interrupt handler is very short, interrupts may be left disabled until the handler terminates. Otherwise, the possibility exists that the handler itself may be interrupted. Special care must be taken in this case. For example, the following questions may need to be considered: Is an adequate stack available for the secondary interrupts to use? Will this interrupt handler still be "short enough," even if it is interrupted? Are this handler and all procedures it calls fully re-entrant, in case a secondary interrupt requires use of the same handler?

Keeping the Handler Short

The final concern is that an interrupt handler should complete its work and terminate in the shortest possible time. Reasons for this have been discussed in the previous subsection. In many cases an interrupt handler should do little more than set an event flag to indicate that a certain event has occurred. Other software should then examine that flag, and carry out the additional work that the event may require.

Debugging Issues

The timing of interrupts is often not repeatable. The systems programmer will often be asked to solve a problem that seems to be random, but ultimately is a result of timing differences in the occurrence of interrupts. Only in certain orderings of interrupts do such problems occur, and so these problems can be extremely difficult to solve because the exact ordering of interrupts cannot be easily observed or repeated.

Interactive debuggers are valuable tools for identifying and correcting problems in most programs, but they may break down when interrupt handlers are involved. The debugger itself

-

relies on the integrity of some critical system facilities, including the interrupt system. Efforts to suspend or monitor execution of an interrupt handler are likely to cause the entire system to fail.

6.7 EXCEPTIONS

The interrupt mechanisms described here are normally managed by the OS, and hidden from most application programs. There is a higher-level mechanism that may be presented to programs for their own use. This mechanism is commonly known as **exceptions**. A program which makes use of this mechanism may define certain external or program-generated conditions as exceptions, and define exception handlers to be executed when these conditions arise.

A program can effectively make use of exceptions only if support is provided by both the programming language and the operating system. Language support is necessary to allow programs to express the desired behavior. Examples of this support include the exception facility of Ada and the Java try-catch mechanism.

Language implementations alone can deal with program-generated exceptions, but a more general exception-handling mechanism requires OS support. A comprehensive example is provided by the Unix signal mechanism. This mechanism allows various predefined signals, each represented by a small integer, to be sent to a process by itself, another process, or the system. Processes may define handlers for each signal; if no handler is defined, default actions are determined by the OS when the signal occurs.

6.8 SUMMARY

Interrupts are extremely important in the implementation of operating systems. Whether they are process complete interrupts, service request interrupts, or error interrupts, the operating system must handle them efficiently and completely. Interrupts cause a transfer to special locations determined by the hardware, where different types must be distinguished and the interrupt processed. Two common hardware techniques for distinguishing interrupts are the fully vectored approach and the use of first and second level interrupt handlers. Selected interrupts may be blocked or disabled using an interrupt mask register or a system of priority levels.

FOR FURTHER READING

A brief general discussion of interrupts is contained in Silberschatz & Gavin [1994]. A more extended treatment is provided by Krakowiak [1988]. Interrupt-driven I/O is examined by Milenkoviç [1987] and by Shay[1992].

-

Because interrupts are very specific to a computer's architecture, and the handling of interrupts is specific to particular operating systems, various manuals accompanying each OS are the best source for understanding interrupts and interrupt handlers in a particular environment. Discussions about the IBM-compatible microcomputers' interrupts can be found in Norton [1985], while examples of interrupt handlers can be found in various IBM Technical Reference Manuals (Personal Computer XT, IBM, 1984 and Personal Computer AT, IBM, 1984), which describe the PC-DOS operating system's basic input/output system (BIOS). The IBM MVS Supervisor Logic Manual provides an excellent discussion of interrupt handling as well as timer queue management in the MVS environment. VMS interrupt handling is described by Sewell[1992] and by Kenah and Bate[1984].

REVIEW QUESTIONS

1. What is the purpose of an interrupt? What mechanisms exist to transfer control to interrupt handlers?

2. List three broad categories of interrupts and explain the types of interrupts in each category.

3. Explain the role of each of the following in interrupt processing in a computer system: program counter; interrupt vector; processor status (word); second-level interrupt handler; first-level interrupt handler.

ASSIGNMENTS

1. From the programmer's point of view, what are the advantages and disadvantages of a computer system containing fully vectored interrupts?

2. From the programmer's point of view, what are the advantages and disadvantages of a computer system that does not have fully vectored interrupts?

3. What is the advantage of having hardware stack support in servicing interrupts? What problems are there in a hardware environment that does not have a hardware stack?

4. Why would a programmer wish to process program fault interrupts? In what circumstances would one choose not to process program fault interrupts and let the operating system perform default servicing on such interrupts?

5. What would happen if an interrupt handler masked all interrupts and never unmasked them prior to returning from servicing the interrupt?