Computer System Overview: Part 2

3 Interrupts

Interrupt is a very important concept for not only understanding computer hardware, but also using facilities provided by high-level programming languages. You may not be familiar with hardware interrupt, but you probably have known some well-known terms, like *event-driven* used in Windows GUI programming, or *event subscription and listening* in Java. For example, the following are two implementation skeletons of a text editor, respectively in Pascal and in Visual C. The former is classic procedural programming, while the latter is based on event driven mechanism. The event driven method is similar to interrupt. When there is no event occurring, from the viewpoint of application programmers, no CPU resource is consumed regarding the program (although Windows itself may run an event dispatcher thread or process checking all the time if there is an event available to be dispatched, which is similar to `"while (!has_char()) ;"` in the first case).

- Pascal-like

```
...
init();
...
while (true) {
  while (!has_char()) ;

  ch = get_char();

  if (ch == ^X) {
    savefile();
    exit (0);
  } else
  ...
  } else
  if ('z' >= ch && 'a' <= ch) {
    insertChar(ch);
  }
}
```

- Windows-VC-like

```
...
CreateWindow();
EnableEvent(WM_CLOSE);
...
void eventOccurred(Event e) {
  switch (e.code) {
  case WM_CLOSE:
    savefile();
    exit(0);
  case 'a'-'z':
    insertChar(e.code);
    break;
  default:
    break;
  }
}
```

3.1 What is interrupt?

A rough definition of *interrupt* is that: interrupt is a mechanism by which computer components, like memory or I/O modules, may interrupt the normal processing of the processor and request the processor to perform a specific action.

According to the source where they are generated, interrupts may be categorized into four classes:

- program: generated due to the appearance of some condition as result of an instruction execution, such as arithmetic overflow, ?/0, attempt to execute an illegal machine instruction, etc. Some program interrupts may otherwise be desirable, like `INT 21H` invocation.
- I/O: generated by an I/O controller, to signal normal completion of an operation or to signal a variety of error conditions.
- timer: generated by a timer within the processor. This enables the operating system to perform some action periodically.
- hardware failure: generated due to a hardware failure, e.g. memory parity error.

When we address a subject, we usually take the three-step approach by answering *what*, *why*, and *how*. We have discussed *what*. Then *why* the computer system needs interrupt?

3.2 I/O processing with interrupts

In brief, interrupts are provided primarily as a way to improve processing efficient.

Suppose we have a program that needs to output something to a printer, then the program may be abstracted in the way as Figure 1.5 in the textbook gives. When WRITE is called, the flow of control will proceed to the corresponding I/O program, which may consists of three sections: section 1, prepares for the I/O operation, e.g. preparing the parameters for accessing the I/O module; section 2, issues actual I/O command to I/O module; section 3, completes the I/O operation, e.g. setting a flag indicating the success or failure of the operation. Based on this abstract model, we may come up with the CPU time consumed by running this program:

$$t_1 + t_2 + t_3 + 2 * (t_4 + t_{IO} + t_5)$$

As we know, most I/O devices are much slower than the processor, since they may perform mechanical operations, which is much slower than the signal transmission of an electronic circuit,

$$t_{IO} >> t_i$$

thus at section 2, CPU has to wait there or periodically check the status of I/O device until a success or failure signal is obtained. In this case, considerable amount of time is wasted simply waiting or *polling*. The solution is interrupt.

The idea is that while the I/O operation is in progress, the processor, instead of idling, may switch to work on other programs. When the I/O operation is finished, the I/O module sends an interrupt request signal to the processor. The processer responds by suspending the current operation and branching off to a program to service that particular I/O device, known as *interrupt handler*, which is similar to section 3 mentioned above. After the interrupt is processed, then the user program of concern may proceed.

To analyze the performance improvement while using interrupt, suppose we have another program which performs totally calculation work and it needs t_0 CPU time to finish. Thus, in the first case without interrupt, to finish the two programs, CPU needs to run:

$$T_1 = t_1 + t_2 + t_3 + 2 * (t_4 + t_{IO} + t_5) + t_0$$

while in the second case with interrupt, the time needed is:

$$T_2 = t_1 + t_2 + t_3 + 2 * (t_4 + t_5) + t_0$$

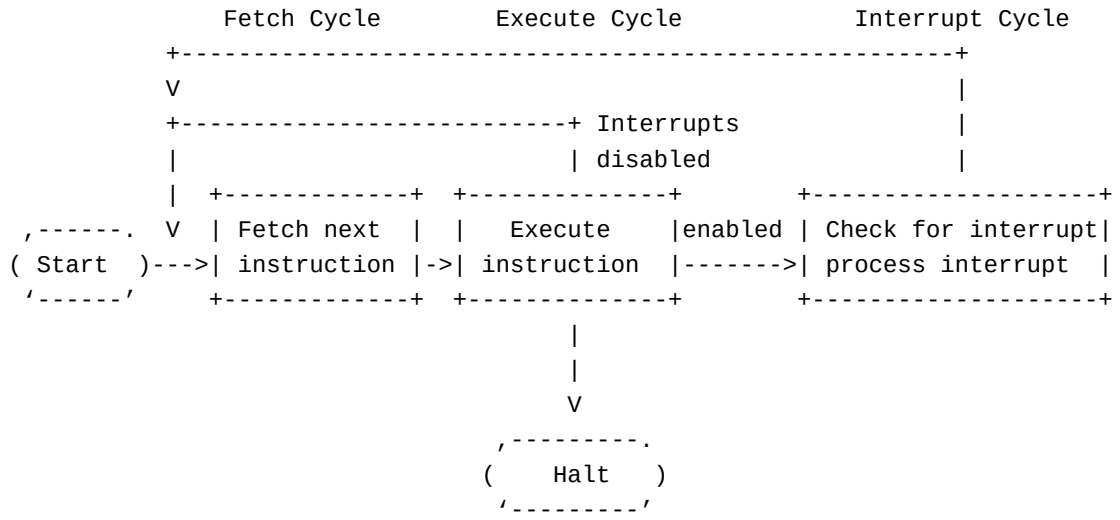
If the I/O operations involve a big fraction of the program, then $n * t_{IO}$ will be very large, at least much larger than $t_1 + t_2 + t_3 + n * (t_4 + t_5)$. If t_0 is large enough, or comparable to t_{IO}, then T_2 will be only a fraction of T_1.

(The textbook addresses this case wrongly, or at least misleading, though it later does mention multiple programming. The same user program cannot simply continue to run before the preceding I/O operation is finished. A vivid counter example is that after you finished editing a file and pressed Ctrl-S to save it to, say a floppy disk, then usually you will see a mouse pointer with the shape of sandglass, which means you are not allowed to do anything before it finishes. Although this case is at a very high level, it does tell the nature of a sequential program.)

It is worth mentioning that there may be some strategies determining what to do next when an interrupt happens, either responding to the interrupt request and resuming the process that was suspended due to the I/O operation, or simply continuing the current operation. This issue will discussed later in the chapter of process scheduling.

3.3 Interrupt cycle

To accommodate interrupts, An interrupt cycle is added into instruction cycle for checking the availability of interrupts as follows:



3.4 Interrupt processing in detail

The following gives the detailed interrupt processing procedure:

1. I/O device issues the interrupt.
2. The processer finishes the execution of an instruction.
3. The processor checks for an interrupt. If there is one, it then sends an acknowledgement signal to the I/O device that issued the interrupt. This signal allows the device to remove its interrupt signal.
4. To switch to run interrupt handler, information about the current program is stored, so that its execution may be resumed later, including PSW and PC.
5. The processor loads the program counter with the entry location of the interrupt handler. A typical case is there are a set of routines, each for one type of interrupt, or each for one device.
6. The interrupt handler may continue to save other information that is considered as part of process state.
7. The handler performs the interrupt processing.
8. When the handler finishes, the saved register values are restored into the registers that originally hold them when the interrupt handler returns.
9. Finally, PSW and PC values of the interrupted program are restored, thus the program may continue to execute.

3.5 Multiple interrupts

The above only discussed the case in which a single interrupt happens. Actually, in a computer system, there are multiple interrupt signal sources, so more than one interrupt requests may happen at the same time or during a same period. The typical two approaches are: **sequential interrupt processing** - by disabling interrupt request while an interrupt is being processed, all interrupts will be processed sequentially (usually PSW contains a bit for this purpose); **nested interrupt processing** - all the interrupts may be assigned different priorities, so that whenever an interrupt occurs while an interrupt handler is running, their priorities will be compared first, and the further action will be determined according to the result. These two approaches are illustrated by the following figures:

It also comes out how to design the interrupt subsystem to accommodate multiple interrupts from different devices, i.e. how to recognize where an interrupt comes from. One solution is providing multiple interrupt signal lines, each for one I/O module. Alternatively, there can be a single interrupt line, but additional lines specifying which device generated the interrupt signal.

4 I/O communication techniques

We further discuss the typical techniques used for I/O communication. The first technique is called *Programmed I/O*, which is actually the case we discussed above before introducing interrupt.

4.1 Programmed I/O

With this method, the processer, executing the instructions in programs, prepares data in memory for output, and then makes request to access I/O device by communicating with I/O module. The I/O module performs the requested action and then sets the appropriate bits in the I/O status register. It is the responsibility of the processer to check the status periodically until it finds that the operation is complete.

To make things look real, assembly language program instead of diagrams is used here to show more details. Suppose the I/O module we will use have 4 ports for CPU to access: 60H for control, 61H for status, and 62H for data. To request READ operation, the processer may send 00H to port 60H, and 01H for WRITE operation. The following example shows how data are read from the I/O device.

```
    ...
  start:
```

```
  MOV AL, 00H
  MOV DX, 60H
  OUT DX, AL  ; issue read command

checking:
  MOV DX, 61H
  IN AL, DX  ; read status in AL
  CMP AL, 01H
  JNE checking  ; continue checking is not ready

  MOV DX, 62H
  IN [DS : CX], DX  ; read a byte from port 62H
  INC CX
  CMP CX, max
  JLE start
  ...
```

4.2 Interrupt-driven I/O

To avoid the waste of time for CPU to wait for the finish of I/O device, we may alternatively use interrupt-driven method, which has been stated above.

4.3 Direct memory access (DMA)

The interrupt mechanism is not perfect yet, and also has its problem, i.e. the processor has to be involved all the way through the I/O processing. When large volumes of data are to transferred between memory and I/O devices, the processor will be interrupted hundreds or even more times to process interrupts. To solve this problem, a more efficient technique called, *direct memory access*, is used. To achieve the goal, a new module, DMA controller, is introduced to take CPU's role in I/O processing. It may be a separate module attached to system bus, or embedded into a I/O module. Although it works independently, it doesn't need to be complex as a CPU, since the data transmission between memory and I/O modules is kind of simple. Only the following information is needed to complete the operation:

- what: whether a read or write is requested
- where$_1$: the address of the I/O device involved
- where$_2$: the starting location in memory to read from or write to
- how much: the number of words to be written or read

An issue worth being mentioned is that system bus may be occupied by DMA operation while CPU is trying to use for accessing memory, but the delay caused here is merely a bus cycle, and the CPU needn't to save the current context. So it is more efficient than the previous approach.

5 The memory hierarchy

Three key characteristics of memory - *cost*, *capacity*, and *access time* - cause a dilemma facing system designers. Three relationships hold between every two of them:

- the faster, the greater cost per bit
- the greater capacity, the smaller cost per bit
- the greater capacity, the slower

Thus it is not possible to meet the need of all the users who would like to have memory of large capacity, short access time, and low price, since the first requirement naturally means more money, and lower speed.

The answer to the dilemma is to rely on a *memory hierarchy*, instead of a single memory component or technology. As Figure 1.14 shows, from top to bottom, the speed decreases while the capacity increases and the prices become much lower. Thus, it is possible to accommodate multiple types of storages gaining a balance.

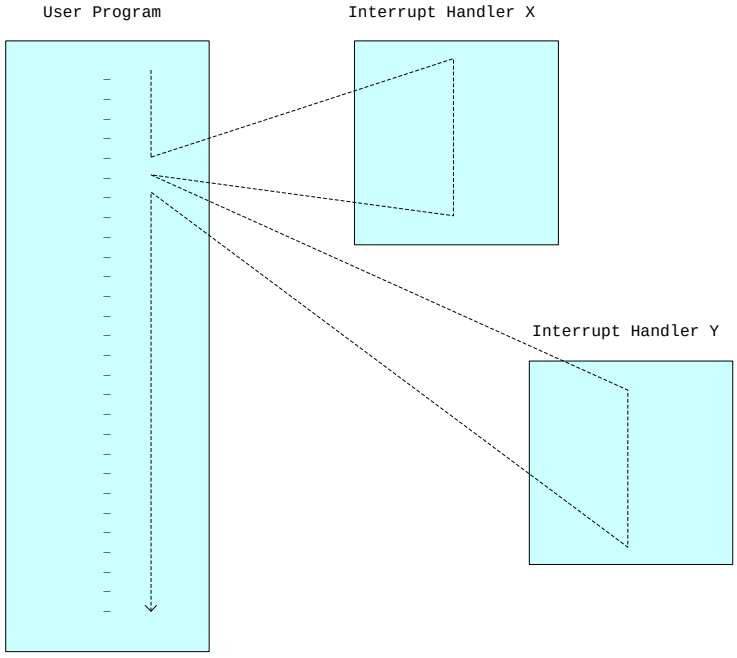
But how could such a combination work? Suppose the system has two levels of memory. L1 is cache inside the processer, and L2 is main memory. The initial state is that user programs and data have been loaded into memory, and cache is empty. And whenever the processer tries to access a location of memory, it first checks with cache and determines if cache already has a copy of that location's value. If it has, we called it a *hit*; otherwise *miss*. If the desired data is found in cache, then the processer needn't bother to visit memory; if the data is not there, then the processer will have to access memory. To speed up the following multiple visit to the same memory location, that value will be maintained in cache until other data from memory need to be stored in cache, and based on some strategy, the former data item is one that should be transferred back to memory. A popularly used value to measure how well a multiple level memory system works is *hit ratio*, which is defined as the fraction of all memory accesses that are found in the cache. Suppose hit ratio $H = 95\%$, the access time to L1 is $T_1 = 0.1\mu s$, and the access time to L2 is $T_2 = 1mus$. Then the average time to access a data item can be expressed as:

$$H * T_1 + (1 - H) * T_2 = 0.15\mu s$$

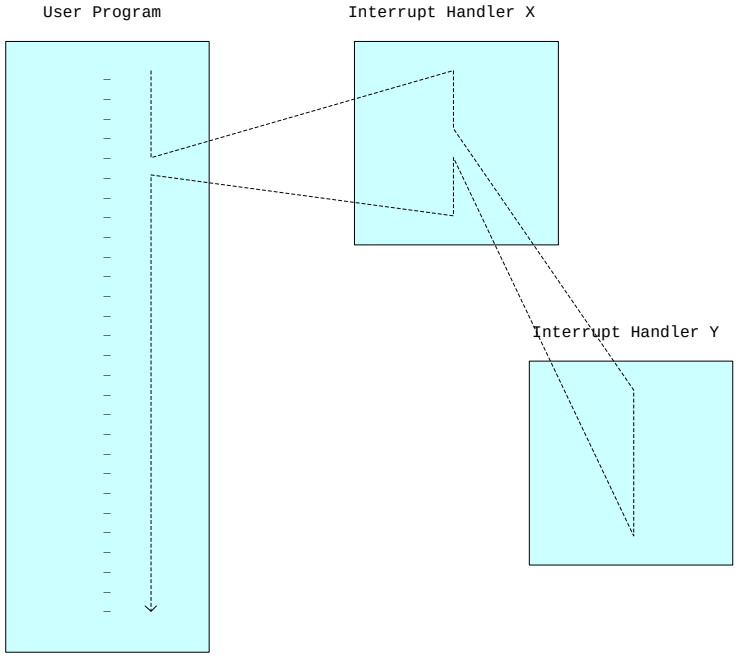
A question that may be raised is why we assign 95% to H, and why not 5%. This assumption is based on a principle called *locality of reference*, which states that a program usually visits a same data block frequently over a limited time, e.g.

```
for (int i=0; i<100; i++) {
  sum += i;
}
```

While the loop goes on, the location containing i's value will be accessed frequently. Thus once it is transferred to the cache, no visit to main memory is necessary, so the consecutive visits will be very fast.



(a) Sequential interrupt processing



(b) Nested interrupt processing