Operating Systems Lecture Notes
Prof. Ghassan Shobaki

May 9, 2025

Contents

Operating Systems Lecture 2: OS Introduction (Part 2): CPU, I/O and Interrupts

Lecture Intro

Most of today's lecture is going to be about the interrupts and how the interrupts are used to drive and operate the system. So last time, we started talking about the I/O devices. Each I/O device, like a keyboard has a controller that controls that device and each device has a "**LOCAL BUFFER**" for storing the data that that device will use or that device will consume.

Difference b/w a device controller and a device driver

The driver is a software and the controller is a hardware. So the device controller is the electronic part of the device. It is an electronic circuit that controls the device. It can be viewed as an "**INTERFACE**" between the device (which could be a mechanical device like a mouse, or a keyboard) but it has an electronic piece that interfaces with the computer. So that electronic piece is the device controller. And the device driver is a piece of software that belongs to the OS and it talks to that device controller. So it is between the operating system and the device controller.

Example of a keyboard

Now suppose the device is a keyboard. It is going to write the data into some local buffer. Now the data is written into the local buffer by the I/O device itself. But at some point, we need to move that data from the local buffer to the main memory. And in particular we need to move it from the local buffer to the memory location that belongs to the process that requested that I/O service. So then this data will be transferred from the local buffer to the main memory (the address space of a certain process).

Who does this transfer?

This transfer is done by the CPU and not by the I/O device, and it is something that the OS does.

How does the OS (or Kernel) know that an I/O device has completed a certain request?

The answer is that an I/O device will generate an interrupt to notify the OS that it has completed a certain request.

Timing Diagram and talk on CPU Scheduler

When the system starts, the Kernel (after the bootstrap program) is running. Then the Kernel will select a process to get the CPU. So how does it select a process? It invokes what we call a "**CPU scheduler**" (a very important part of OS). For now we only need to understand what the CPU scheduler does. And what it does is select one process to get the CPU next. How does it select that? It selects it based on a certain scheduling algorithm. And we will not study scheduling algorithms until we get to chapter 6. So in chapter 6, we will be studying different scheduling algorithms and policies. So we will understand the "how" later on. For now, we need to understand the "what". What does the CPU Scheduler does? It selects a process to get the CPU. So according to certain policies or algorithms to get the CPU. Suppose, it selects Process#1 then our timing diagram looks like this:

```
CPU [[Kernel] [P1]]
```

So process#1 is now running on the CPU. Typically a process will not be doing CPU only. It will need to do I/O at some point, because it needs to get a user response from the keyboard or the mouse, or it needs to write something to the screen or it needs to access a file. So it is hard to imagine a process that doesn't require I/O. So it will request I/O from the Kernel. And that request is made via a **SYSTEM CALL**. So that process that is running on the CPU is going to make a system call. Obviously, a system call means that you are calling the OS (or the Kernel) to do something or to perform a service for you. And one of the most important services is the I/O. Accessing the I/O or making an I/O request. The SYSTEM CALL is implemented via an **INTERRUPT**. So basically this process will make a system call via an interrupt. Now the Kernel is in charge again. Now, something to keep in mind is that whenever an interrupt is generated, the control is given to the Kernel.

Interrupt service routine

When an interrupt is generated, what will be invoked is an "**Interrupt Service Routine**". For each interrupt, there is an interrupt service routine that services that interrupt.

Interrupt Vector

It is basically a look up table. That is indexed by the **INTERRUPT NUMBER**. And for each interrupt number, there is a corresponding ISR (Interrupt Service Routine) that services that interrupt.

Interrupts and Kernel

And these interrupts belong to the Kernel. Whenever an ISR is invoked, you are basically giving control to the Kernel.

What will the Kernel do when an ISR is invoked?

It will satisfy that I/O request by sending the request to the corresponding I/O device. Now to send a request to the corresponding I/O device. See the timing diagram now at this point:

```
CPU [[Kernel][P1][Kernel]]
I/O               []
```

What is happening above? This is the I/O device, at some point the request will be sent to the I/O device. Now the I/O device may or may not be available immediately to process that request because that I/O device typically has a Queue of requests. So there may be other requests that are currently in the queue or that are waiting to get serviced. So the I/O device has a queue. The queue may have other requests in it or it may be empty. So assuming that the queue is empty (as depicted in the above timing diagram) then that means that there are no other requests and thus this request will get processed by the I/O device at that point.

```
CPU [[Kernel][P1][Kernel]]
I/O                [P1]
```

So now the I/O device is servicing P1, then now the Kernel is done (it has sent that request to the I/O device). Now the I/O device is going to take some time to process that request, and we expect I/O device to be SLOW (or to take some time). So during this time, the Kernel is NOT JUST GOING TO WAIT. It is not going to waste system resources waiting for this I/O request to get completed. So what do you think the Kernel will do at this point? It cannot give the CPU to P1 again because it WILL NOT WORK. Why will it not work?

Is a certain process P1 is being serviced by the I/O, can the Kernel give the CPU to P1 during this time?

It cannot do that because it will not work. What is an I/O? An I/O is like reading from the keyboard. Your process#1 is reading from the keyboard.

```
read x;
y = x+5;
---
---
```

Listing 1: Illustrative code snippet for I/O

I cannot go and execute things that are after this read operation because I am reading x. And then I am using x. So until I read "x", I cannot execute `y = x+5;` And in-fact, Operating Systems never re-order the instructions in a given process. The OS is not in the business of reordering instructions within any given process. The OS just executes a program in-order.

So what does the Kernel do during this waiting time?

The kernel will have to find another process to give the CPU to. Usually there are multiple processes. So it will find a process like P2 and it will give it the CPU.

```
CPU[[Kernel][P1][Kernel][P2]]
I/O                [P1]
```

Now what is interesting here is that if we take a snapshot at this point, we will see that multiple processes are making progress. We are using two resources. P2 is making progress on the CPU. P1 is making progress on the I/O device. Now what is the state of each process? With the operating system terminology, P2 in this case is in the running state. So running means that you are on the CPU. So P2 is running. And P1 is waiting.

```
Process   State
P1        waiting
P2        running
```

Now even though P1 is waiting, it is actually making progress but according to the terminology that we will be using in this course, this process is waiting because it is not on the CPU and it is basically waiting for the I/O device to complete processing this request. But just the point here is that if we say that a process is waiting it doesn't mean that it is wasting time. It could be that it is getting serviced by the I/O device.

```
CPU[[Kernel][P1][Kernel][P2]]
I/O             [P1]
```

So basically this above timing diagram is one form of concurrency between the CPU and the I/O device. Now what will happen next? Eventually the I/O device will complete processing this request.

What will happen when the I/O device completes processing a request?

It must notify the kernel that it has completed processing a request. And it must do that by generating an **INTERRUPT**. And whenever an interrupt gets generated who gets control? The **KERNEL**. So for simplicity's sake, we are assuming that P2 is the process that has kept CPU for this long, but this is not always true. The CPU may have been given to some other process from P2 during this time. But anyways, for simplicity's sake, we are assuming that P2 has kept the CPU for this long.

```
CPU[[Kernel][P1][Kernel][P2][Kernel]]
I/O             [P1      ]
```

Anyways, now that the Kernel gets the CPU after the interrupt is generated by the I/O, what is the logical thing that the Kernel should do in this case? It will do scheduling BUT BEFORE THAT it will have TO COPY THE DATA FROM THE I/O device to the main memory. It is basically the I/O device saying: "OK I am done with this request. I am ready. I have the data in my local buffer, so come and pick it up". So you have to pick the data up from I/O device's local buffer and put it in main memory and put it in the location that belongs to the process that is requesting that I/O. That is the job that the OS will have to do. It will have to make the data available to the process that requested the I/O. Then after doing this, it will mark PROCESS#1 as **READY**. Ready means "**READY TO TAKE THE CPU**".

What happens after when the data is put into the location of process requesting I/O inside main memory?

That certain process (in our case Process#1 or P1) will be marked as "**READY**" by the Kernel. Ready means "**READY TO TAKE THE CPU**". Who decides whether it will get it or not? The scheduler. So being ready means that it will be one of the candidates that the scheduler will consider for giving the CPU next to. Also, the CPU is not necessarily given back to P1 after all this because who gets the CPU is based on the scheduling policy. And based on the scheduling policy P1 could be a lower priority process. P1 may get the CPU or may not get the CPU. It all depends on the scheduling algorithm that is being used.

Now that the I/O has completed...what now?

Now that P1 is labeled "**READY**". Now based on certain scheduling algorithm's criteria that we will be studying later in chapter 6, it will decide what to do next.

What if there are multiple processes requesting the same resource?

The I/O device has a queue, which means that multiple requests can be sent to the I/O device. It will queue them. And it will service them one at a time, in order. When it completes one of them, it will generate an **INTERRUPT**. So there is an interrupt for each request. So when the I/O device notifies the Kernel that it has completed a request (it is just one specific request that belongs to one specific process) then there may be other requests in the queue and they will be serviced one request at a time.

What happens according to the diagram if P2 decides to access the I/O?

Then it will make a system call. So now there will be a different diagram. At some point, there will be another I/O request. So the Kernel will be in-charge and it will send a request to the I/O device. Now if it sends the request to the same I/O device as P1 or a different one can make a difference. If it sends it to the same I/O device, what do you think will happen? It will be put on the Queue, because that I/O device is already busy servicing P1 and it is at least not ready now. But suppose that it sends it to another I/O device and if that I/O device happens to be available (meaning its queue is empty) then maybe that request for P2 may get serviced immediately.

```
CPU[[Kernel][P1][Kernel][P2][Kernel]]
I/O             [P1             ]
I/O#2                   [P2...]
```

And then whichever request completes first, it is going to generate an interrupt. So if P1 started first then it is going to notify the Kernel that is done.

If both P1 and P2 are getting serviced, what will the Kernel do?

It will find another process to give the CPU to. So it may give the CPU to process#3.

Is CPU ever in-active?

When the Kernel doesn't find a process that is ready then the Kernel will keep the CPU. This is known as the "**Idle state**". This is not something that we want. In general, we would like to minimize Kernel intervention. The CPU is to service the processes and the Kernel is the manager. And that manager should minimize the time that it spends on the CPU. It is not useful time. The Kernel time, from an application's point of view, is not useful time. So the Kernel should manage the system with minimal CPU time.

Two Types of Interrupts

1. Interrupts that are used by the I/O device to notify the system that it has completed processing an I/O request

2. Interrupts that implement System Calls

Other types of Interrupts: Exceptions

In addition to the above two, there are other kind of interrupts such as the "**Exceptions**". So exceptions are generated for different reasons. Exceptions may be generated if a process does something wrong or we do an invalid operation such as:

1. Divide by zero. So this will generate an exception.
2. If a process runs into STACK OVERFLOW.
3. A floating point overflow.
4. It executes an invalid operation.
5. Out of bound memory access (we won't fully understand how these type of exceptions are caught until we get to the memory management chapter – which is chapter 8 and chapter 9).

When an exception is generated, Kernel gets the control and the Kernel will then terminate that process in order to protect the system.