Operating Systems Lecture Notes
Prof. Ghassan Shobaki

May 9, 2025

Contents

Operating Systems Lecture 4: OS Introduction (Part 4): Multiprogramming vs Time Sharing

Intro: High Level view at Multiprogramming

This is an important OS concept. In both cases, the OS has multiple processes to handle. In a multiprogramming system, the OS is going to give the CPU to one process. And it will wait until that process requests IO. Or has to go into the waiting state for other reasons (we will be covering these other reasons later on, but for now the only reason that we have covered is requesting IO). So if a process goes into the waiting state, the OS is going to give the CPU to another process, that is MULTIPROGRAMMING.

Intro: High Level view at Timesharing

Time sharing means that the Kernel is not going to wait for a process to request I/O or go into the waiting state. The Kernel will keep switching processes, even if a process doesn't need to wait, the OS is GOING TO TAKE THE CPU FROM IT and it is going to give it to another process.

Detailed info on multiprogramming (Batch system)

It is needed for efficiency.

- Single user cannot keep CPU and I/O devices busy at all times
- Multiprogramming organizes jobs (code and data) so CPU always has one to execute.
- A subset of total jobs in system is kept in memory.
- One job selected and run via job scheduling
- When it has to wait (for I/O for example) OS switches to another job.

Detailed info on TimeSharing (multitasking)

- It is a logical extension in which CPU switches jobs so frequently that users can interact with each job while it is runnijng, creating interactive computing.
- Response timem should be $<$ 1 second
- Each user has at least one program executing in memory $=>$ process
- If severeal jobs ready to run at the same time $=>$ CPU scheduling
- If processes don't fit in memory, swapping moves them in and out to run
- Virtual memory allows execution of processes not completely in memory.

Timing diagram for Multiprogramming and Time sharing

Multiprogramming:

```
CPU [[Kernel][P1     ][K][P2.....]]
```

TimeSharing:

```
CPU[[Kernel][p1][Kernel][P2][Kerne][P1][Kernel][P2]]
```

So time sharing is switching between processes. We may have many processes and we don't know which process gets the CPU next. Which process gets the CPU next is a scheduling decision.

Why would a CPU do Time-sharing? Or why would it keep switching between processes?

It is about "user experience". So the user here will feel that multiple processes are making progress at the same time. So if we just assume that all of these processes belong to the same computer. Like you have your MS-Word open, and you have your browser active and at the same time you have your email active and your email receives messages and you get notified when you are typing. So there are multiple processes. So when this is happening, the user is getting the feel that multiple processes are making progress. Even though, strictly speaking, only one of them will be running (or be active on the CPU) at any given time but the user will get the feel or the illusion that multiple processes are running at the same time. So this is MORE RESPONSIVE. This system is going to be more responsive to the user.

But saying that TimeSharing is "faster" is NOT true

This is not faster in fact. So when you say that which one is faster? So you have to define "faster" here. Faster is not well defined here. So I can argue that TimeSharing is faster from a user point of view. But I can argue that multiprogramming is faster. How? What is the argument? If you ask what is the total time needed to execute a given set of processes? So the total time needed to finish them, assuming that you can always overlap CPU with I/O. Assuming that I/O doesn't take a long time. Asumming that when P1 goes to I/O and P2 gets the CPU and when P2 requests I/O again, the I/O for P1 is done. Then assuming that the I/O deson't take a long time here all processes will finish faster. Why? Because we have spent less time on the Kernel. Or in other words, Kernel has spent less time on the CPU. You can think of Kernel time as wasted time. So assuming that the I/O is fast enough then we will need less time i.e the total time to execute both processes is going to be less.

Thinking of "fast" in terms of "Throughput"

What's the total number of processes that you can complete executing within a given period of time. If I give you one minute, what is the total number of processes that you can finish in that time. Obivously, in multiprogramming you will be able to complete more processes. Again, assuming that the I/O is fast enough of the I/O can always be overlapped with CPU.

Conclusively

From throughput point of view, Multiprogramming is better. From user responsiveness point of view, Time sharing is better. That is why the word "faster" is not welll defined.

User interaction on multiprogramming

This is definitely less interactive and if this process does not request the I/O for a long time then the user will feel that the SYSTEM IS SLOW.

Big Question: How does the OS implement Time Sharing? Timed Interrupt

The OS needs a mechanism for setting a limit on the TIME that process#1 gets. So initially, Kernel has the CPU. Now when it gives the CPU to process#1, it needs a mechanism to get the CPU back after a specific or a certain time period that is determined by the Kernel. So the Kernel wants to say that, "OK, I am going to give the CPU to process#1 for 10 milliseconds and after 10 milliseconds, I, the Kernel, want the CPU back". So how does the Kernel implement this?

This is implemented using a TIMED INTERRUPT. Remember that whenever there is an interrupt, the Kernel gets control. So, the Kernel is going to set up a timed interrupt, an interrupt that gets triggered after a certain period of time that is determined by the Kernel. So the Kernel is going to set a certain counter. Of-course, it is going to be in binary. So the Kernel is going to put 10 milliseconds (in binary) inside the counter and then it will get decremented with the SYSTEM CLOCK. So whenever the clock ticks that amount of time, that period is going to get decremented from what is in the counter and then eventually that counter will reach 0.

What happens when the counter reaches zero?

When the counter reaches zero then an interrupt gets generated. And when that interrupt gets generated, the CPU is in control again. So this is extremely important. Otherwise, the Kernel will not have control. So this is a very important mechansim. This will allow the Kernel to determine how much time each process spends on the CPU. And obviously, this ability to set up this timed interrupt (or setting the value of this counter), obviously this previllege should be given only to the Kernel. A user process should not be allowed to touch that timed interrupt. And that is why, this requires "Hardware Support" by supporting "DUAL-MODE OF OPERATION". So the hardware must have a way of distinguishing between a Kernel and a user process. And that requires hardware support and this is implemented using the dual mode hardware feature.

Dual Mode Hardware Feature

This dual mode hardware feature is implemented by a mod bit. So there is a certain bit that indicates whether we are Kernel mode or in user mode. If we are in Kernel mode then the Kernel is allowed to execute some PREVILLEGED INSTRUCTIONS. Among these instructions includes the instuction that set up this timed interrupt. If you are in user mode then you cannot execute the previleged instructions.

Interrupts and timers

Timer prevents infinite loop / process hogging resources

- Timer is set to interrupt the computer after some time period
- Keep a counter that is decremented by the physical clock
- Operating system sets the counter (previleged instruction)
- When counter reaches zero, generate an interrupt
- Set up before scheduling process to regain control or terminate program that exceeds alloted time.

All kinds of interrupts that we have studied so far

1. Exceptions
2. I/O completion
3. Timed interrupt
4. System call (they can be made for many different reasons. Including the normal termination of a process. For example: exit() is a system call).

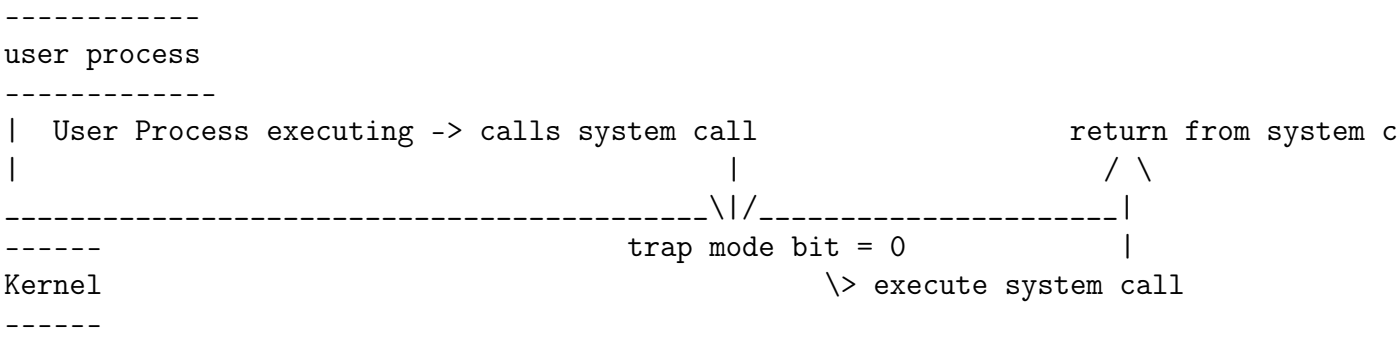
Also "process completition" is a special case of a more general kind of interrupt.

More on dual mode

Dual mode operation allows OS to protect itself and other system components

- user mode and kernel mode
- Mode bit provided by hardware
  - Provides ability to distinguish between user code and kernel code
  - Some instructions designated as previleged, only executable in kernel mode
  - System call changes mode to kernel, return resets it to user
- Lack of Hardware-supported dual mode can cause serious problems. Example: DOS on Intel 8088
- Some modern machiens support multiple modes (e.g third mode for virtual machine manager).

Transition from user to kernel mode



WHat is happening above?

We have a user process that is executing in user mode. When it makes a system call then the mod bit is changed to 0 and now we are in Kernel mode. The kernel is executing. When the Kernel is done, it is going to set the mode bit back to 1. So basically whenever the Kernel gives the CPU to a process, it must set the mode bit to 1 to indicate that it is the user that is running on the system, so the hardware does not allow a user process to execute previleged instructions (THIS IS A KEY IDEA).