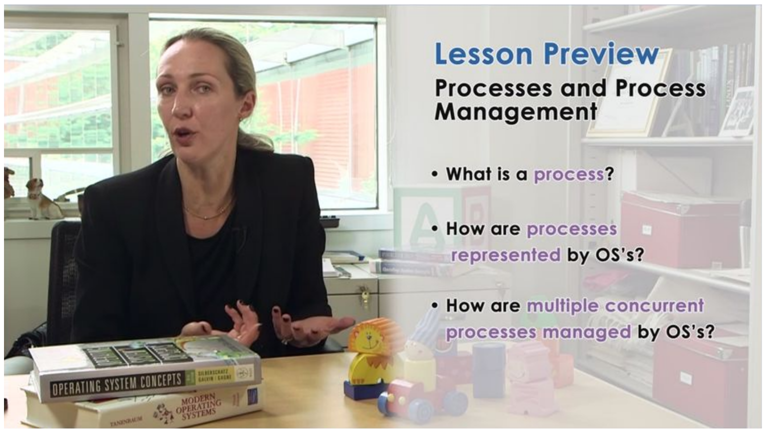
P2L1 - Process and Process Management

01 - Lesson Preview

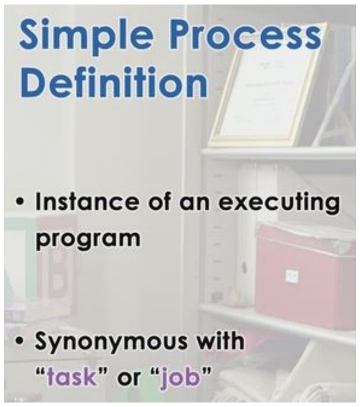


Lesson Preview

Processes and Process Management

- What is a process?
- How are processes represented by OS's?
- How are multiple concurrent processes managed by OS's?

One of the key abstractions that Operating Systems support is that of a process. In this Lecture, I will explain what is a process, how an operating system represents a process, and also what an operating system must do in order to manage one or more processes, particularly when multiple processes share a single physical platform.



Simple Process Definition

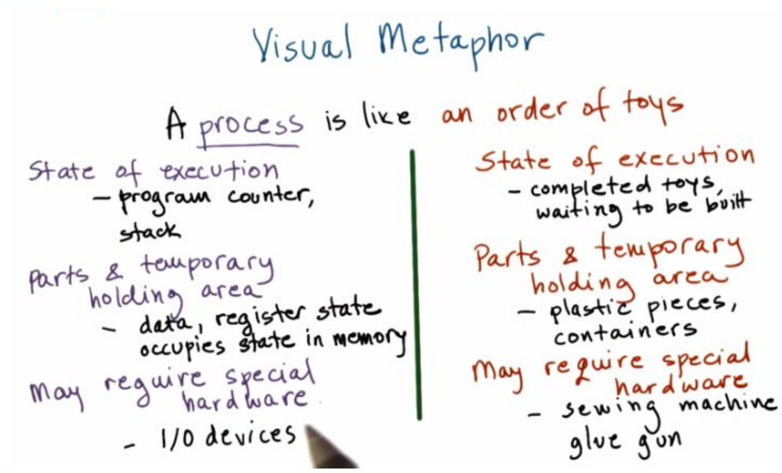
- Instance of an executing program
- Synonymous with "task" or "job"

Before we begin lets define what a process is. In the simplest terms, a process is an instance of an executing program. Sometimes it makes sense to use the terms "task" or "job" interchangeably with a "process".

02 - Visual Metaphor

We will use again a visual metaphor to describe what a process is. Continuing with a toy shop as an example, you can think of a process as an order of toys. An order of toys has its state of execution, it requires some parts, and a temporary holding area, and even may require some special hardware.

For instance, it's state of execution, may include the completed toys, the toys that are waiting to be built, that are part of that order, and other things. Building the toys may require various parts, like plastic pieces, wooden pieces, and these come in different containers, or we may require some other temporary holding area for the pieces.



Visual Metaphor

A process is like an order of toys

State of execution
- program counter, stack

Parts & temporary holding area
- data, register state occupies state in memory

May require special hardware
- I/O devices

State of execution
- completed toys, waiting to be built

Parts & temporary holding area
- plastic pieces, containers

May require special hardware
- sewing machine glue gun

And, finally, to actually build a toy, we may need some special hardware. We may need sewing machines, glue guns, or other types of tools. So, how does all of this then compare to a process in an operating system?

Well, a process also has a state of execution described with the program counter, the stack pointer. All this information is used by the operating system to decide how to schedule the process, how to swap between multiple processes, and for other management tasks.

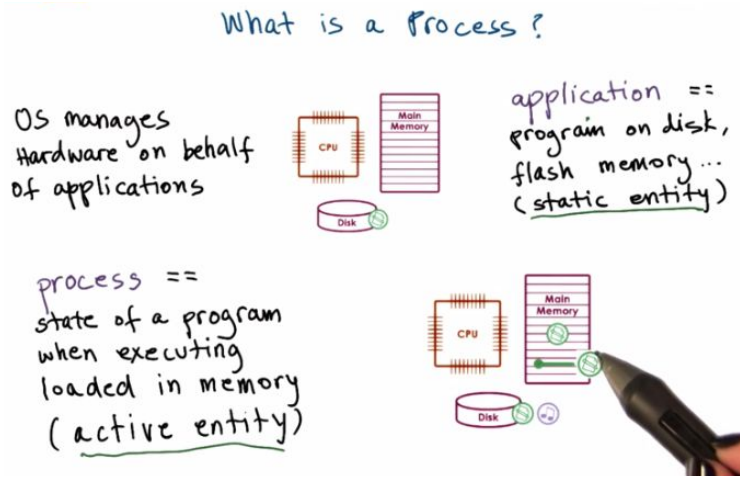
In order to execute, the process needs some data. There's some state in registers. And, it also has some temporary holding area. For instance, it occupies state in memory.

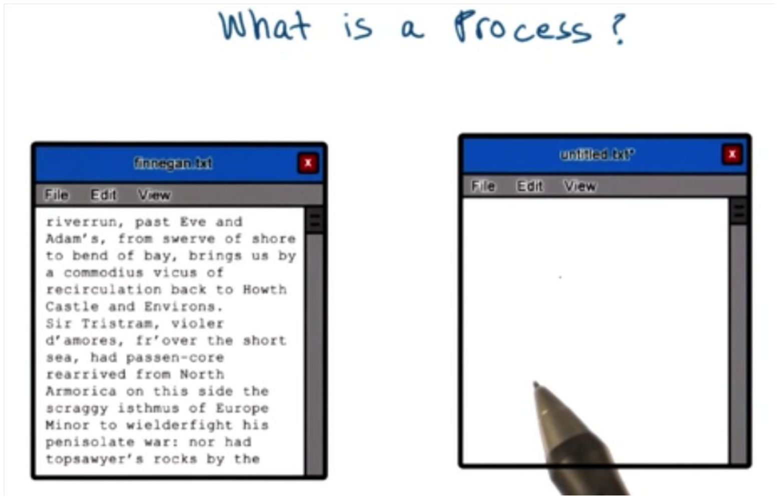
Finally, executing a process may require some special hardware like I/O devices like discs, or network devices. The operating system has to manage these devices and control which of the processes that are perhaps executing concurrently at the same time gets access to which hardware components.

This is similar to what would happen in a toy shop where the toy shop manager has to control how the special hardware, like the sewing machine or the glue gun, are used. Which particular order of toys will get to be assigned the usage of these more designated hardware components.

03 - What is a Process

Let's talk now, more specifically, about processes, and, we'll start first by understanding, what is a process? To do this, recall that one of the roles of the operating system is to manage the hardware on behalf of applications. An application is a program that's on disk, in flash memory, even in the cloud. But it's not executing, it's a static entity. Here, for instance, in this picture, we have some application that's stored on disk.


What is a Process?
OS manages hardware on behalf of applications
CPU
Main Memory
Disk
application == program on disk, flash memory... (static entity)
process == state of a program when executing loaded in memory (active entity)
CPU
Main Memory
Disk



riverrun, past Eve and Adam's, from swerve of shore to bend of bay, brings us by a commodius vicus of recirculation back to Howth Castle and Environs. Sir Tristram, violer d'amores, fr'over the short sea, had passen-core rearrived from North Armorica on this side the scraggy isthmus of Europe Minor to wielderfight his penisolate war: nor had topsawyer's rocks by the

Once an application is launched, it's loaded in memory here, and it starts executing. Then it becomes a process. So a process is an active entity. If the same program is launched more than once, then multiple

processes will be created. These processes will executing the same program, but potentially will have very different state. In fact, very likely they will have very different state.

For instance, a process can be one instance of the word editor program. Here, you're displaying some notes from a previous lecture. And perhaps you're just reviewing it, you're not really modifying this. And then you can have a second process, another instance of the exact same word editor program to take notes from this lecture.

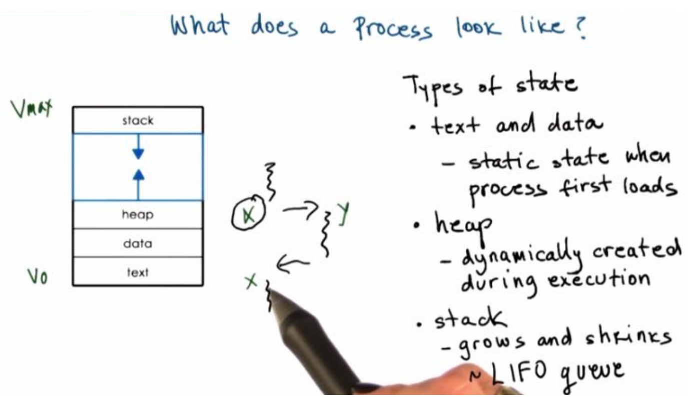
Given that we just started, this probably doesn't have many notes, so it has relatively small state, and it may have some unsaved edits. So, process therefore represents the execution state of an active application. It doesn't mean necessarily that it's running. It may be waiting on input like user input to type in certain notes. Or it may be waiting for another process that's currently running on the CPU, in case there's just one CPU in the system.

04 - What does a Process look like?

So what does a process look like? **A process encapsulates all of the state of a running application. This includes the code, the data, all the variables that that application needs to allocate. Every single element of the process state has to be uniquely identified by its address. So an OS abstraction used to encapsulate all of the process state is an *address space (amit - basically think in terms of stack, heap, static data etc and all that needs to be maintained and that collectively is our address space)*.** This is what we have here.

The address space is defined by a range of addresses from v_0 to some v_{max} and different types of process state will appear in different regions in this address space. What are the different types of state in a process? First we have the code (the text), and the data that's available when the process is first initialized. So all of this is static state that's available when the process first loads. ***Then during execution, the process dynamically create some state, allocates memory, stores the temporary results, reads data from files. This part of the address space we call a heap (amit - is heap limited? Or it is per process? Or it can keep on growing as long as the system can support it)***.

In this picture here, the heap is shown as contiguous portion of the address space starting immediately after the data, but in reality there may be holes in this space so it may not be contiguous. There may be portions of it that don't have any meaning for that particular process and in fact the process isn't even allowed to access them. I will talk in a little bit, how the operating system knows what's okay for the process to access vs what isn't.

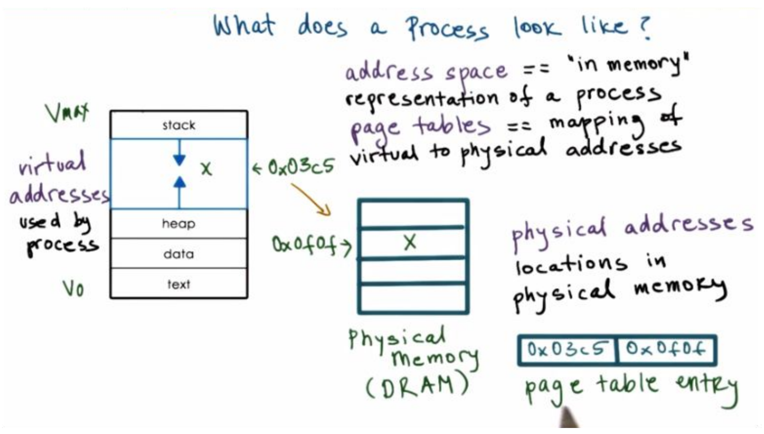

What does a Process look like?
Vmax
stack
heap
data
text
V0
x
y
x
Types of state
• text and data
– static state when process first loads
• heap
– dynamically created during execution
• stack
– grows and shrinks
~ LIFO queue

Another very important part of the address space is what we call a stack. It's a dynamic part of the address space state in that it grows and shrinks during execution, but it does so in a LIFO (Last In First Out) order. Whatever you put on the stake will be the very first item to be returned when your trying to read from the stack.

Consider for instance we are executing a particular portion of the process and now we need to call some procedure to jump to a different part of the address space. We want to make sure that the state, that we were in at this point (pointing at x) of the execution before we called (drawing arrow to y) this other procedure, is saved and then that it will be restored once we come (draw arrow back from y) back from the execution. We can place stat on the stack and jump to execute this portion (pointing at y) of the code, so the procedure y. When we're finished with y, the state x will be popped from the stack and we can continue the execution in the same state that we were in before the call to y was made. So there are lots of points during a process execution where this LIFO behavior is very useful. So the stack is a very useful data structure.

05 - Process Address Space

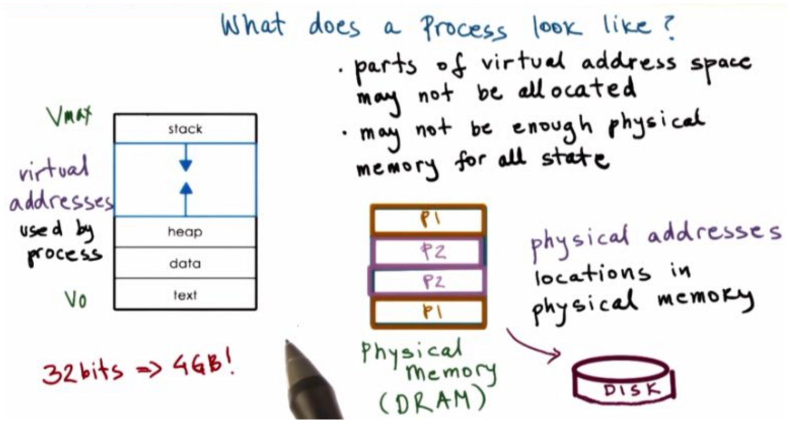
As a whole we refer to this process representation as an ***address space***. We said earlier that the potential range of address from v_0 to v_{max} represents the maximum size of the process address space and we call these addresses ***virtual addresses*** so v's between v_0 and v_{max} are the addresses that are used by the process to reference some of it state. ***We call these addresses "virtual" because they don't have to correspond to actual locations in the physical memory. Instead the memory management hardware and operating system components responsible for memory management, like page tables, maintain a mapping between the virtual addresses and the physical addresses*****s. By using this kind of mapping, we decouple the layout of the data in the virtual address space, which may be complex and it may depend on the specifics of the application or the tools that we use like how to compiler chose to lay that data out. That's completely decoupled with how that data is laid out in physical memory**. And that will allow us to maintain physical memory management simple and not in any way dictated by the data layout of processes that are executing.


What does a Process look like?
address space == "in memory" representation of a process
page tables == mapping of virtual to physical addresses
VMAX
stack
x
virtual addresses used by process
heap
data
text
V0
0x03c5
0x0f0f
X
physical addresses locations in physical memory
Physical Memory (DRAM)
0x03c5
0x0f0f
page table entry

For instance, the variable x may be at a location 0x03c5 in the virtual address space and this may correspond to a completely different address 0x0f0f in physical memory. The way this happens is when the process requests some memory to be allocated to it at a particular virtual address, the address of the physical memory that the operating system actual allocates may be completely different and instead of notifying the process about the details of where exactly in memory that variable really is, the operating system will create a mapping between this virtual address 0x03c5 and the physical address 0x0f0f where x actually is. So then whenever the process tries to access x this mapping is referenced and reality the exact physical location where x is will be access. As long as the mapping between 0x03c5 and 0x0f0f is present in this mapping table, this is a page table and this is a page table entry, any access of the process to x will in fact access the correct physical location where x is stored.

06 - Address Space and Memory Management

We said already not all processes require the entire address space from v_0 to v_{max}, there may be portions of this address space that are not allocated (amit - IMP - I THOUGHT OTHERWISE). Also we may simply not have enough physical memory to store all this state even if we do need it. **For instance, if we have a virtual addresses that are 32 bits long, this address space can have up to 4 GB, and if we have several such processes running at the same time, even on a pretty expensive machines we will quickly run out of physical memory. To deal with this the operating system dynamically decides which portion of which address space will be present where in physical memory.**


What does a Process look like?
· parts of virtual address space may not be allocated
· may not be enough physical memory for all state
Vmax
stack
heap
data
text
V0
virtual addresses used by process
P1
P2
P2
P1
physical addresses locations in physical memory
32bits => 4GB!
Physical memory (DRAM)
DISK

For instance in some system with processes P1 and P2, they may share the physical memory in this manner (referring to pictorial in center of figure to the right), so the regions marked with orange belong to P1, and the regions marked with purple belong to process P2. **Both P1 and P2 may have some portions of their address space not present in memory but rather swapped temporarily on disk (AMIT - TAKEAWAY - SWAPPING HAPPENS WITH THE DISK)**. And this portion of the address space will be brought in whenever it's needed and perhaps that will cause some other parts of either P1's or P2's address space to be swapped to disk to make room. So the operating system must maintain information where these virtual addresses actually are in memory or on disk since it maintains the mapping between the virtual addresses and the physical location of every part of the process address space.

I will talk about memory management in a later lesson, but at the very least you must understand that for each process the operating system must maintain some information regarding the process address space. **We mentioned the page tables for instance. And**

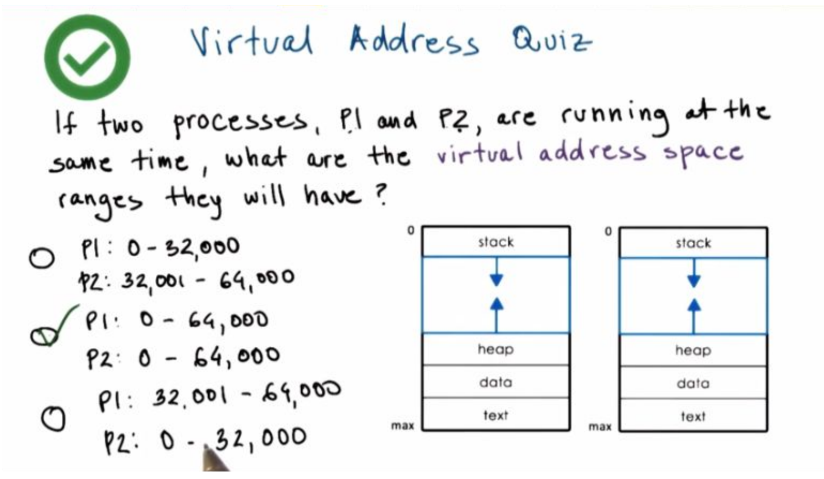
in the operating system uses this information to both *maintain mappings between the virtual address and the physical location where the state is actually stored and also to check the validity of accesses to memory to make sure that a process is actually allowed to perform a memory access (AMIT - TAKE AWAY - OS MAKES SURE THAT EVERY MEMORY ACCESS BY A PROCESS IS WITHIN THE MEMORY LIMIT SPECIFIED FOR IT IN ITS ADDRESS SPACE)*.

07 - Virtual Addresses Quiz

To review this, let's take a quiz. If two processes, P1 and P2, are running at the same time, what are the ranges of their virtual address space that they will have?

The first choice is P1 has address ranges from 0 to 32,000, and P2 from 32,001 until 64,000. The second choice is that both P1 and P2 have address ranges from 0 to 64,000. And the last choice, P1 has an address space range from 32,001 to 64,000, and P2 has address ranges from 0 to 32,000. So the reverse from the first one. So go ahead and mark all the ones that you think are correct answers.

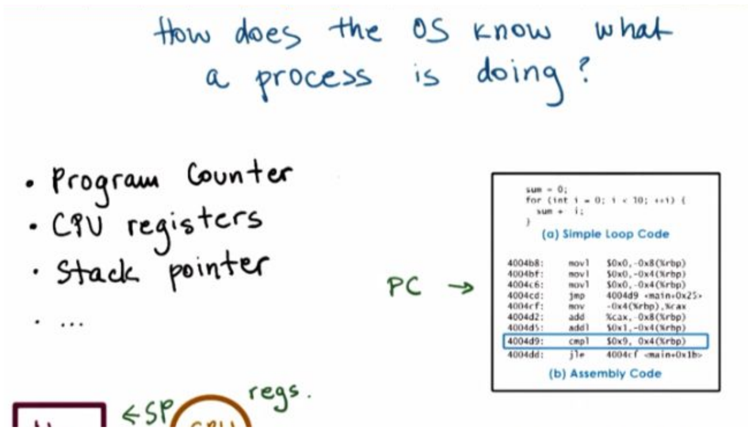
08 - Virtual Addresses Quiz



The correct answer is the second one. Both P1 and P2 can have the exact same virtual address space range from 0 to 64,000 in this case. The operating system underneath will map P1's virtual addresses to some physical locations, and P2's virtual addresses to other physical locations.

The fact that we have decoupled the virtual addresses that are used by the processes from the physical addresses where data actually is makes it possible for different processes to have the exact same address space range and to use the exact same addresses. The OS will make sure that they point to distinct physical memory locations if that's what's required.

09 - Process Execution State



```
sum = 0;
for (int i = 0; i < 10; ++i) {
  sum += i;
}
```

(a) Simple Loop Code

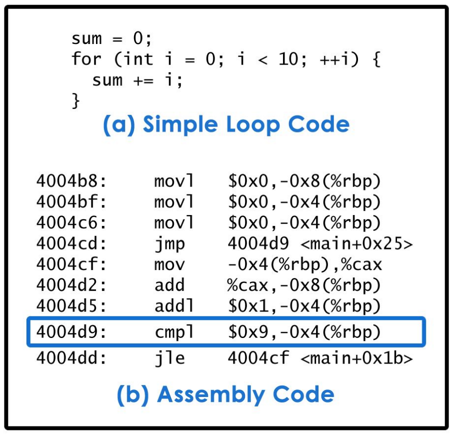
```
4004b8:    movl    $0x0,-0x8(%rbp)
4004bf:    movl    $0x0,-0x4(%rbp)
4004c6:    movl    $0x0,-0x4(%rbp)
4004cd:    jmp     4004d9 <main+0x25>
4004cf:    mov     -0x4(%rbp),%eax
4004d2:    add     %eax,-0x8(%rbp)
4004d5:    addl    $0x1,-0x4(%rbp)
4004d9:    cmpl    $0x9,0x4(%rbp)
4004dd:    jle     4004cf <main+0x1b>
```

(b) Assembly Code

For an operating system to manage processes, it has to have some kind of idea of what they are doing. If the operating system stops a process, it must know what it was doing when it was stopped so that it can restart it from the exact same point. So how does an operating system know what a process is doing? Let's think about the underlying hardware, the CPU, and think how it executes applications.

Applications, before they can execute, their source code must be compiled, and a binary is produced. The binary is a sequence of instructions, and they're not necessarily executed sequentially. There may be some jumps, loops, or even there may be interrupts that will interrupt the execution of the binary. **At any given point of time, the CPU needs to know where in this instruction sequence the process currently is.** ***So we call this the program counter, PC***.

The program counter is actually maintained on the CPU while the process is executing in a register and there are other registers that are maintained on the CPU (amit - takeaway - PC and registers). These hold values necessary during the execution. **They may have information like addresses for data, or they may have some status information that somehow affects the execution of the sequence. So these are also part of the state of the process**.



```
sum = 0;
for (int i = 0; i < 10; ++i) {
  sum += i;
}
```

(a) Simple Loop Code

```
4004b8:    movl    $0x0,-0x8(%rbp)
4004bf:    movl    $0x0,-0x4(%rbp)
4004c6:    movl    $0x0,-0x4(%rbp)
4004cd:    jmp     4004d9 <main+0x25>
4004cf:    mov     -0x4(%rbp),%cax
4004d2:    add     %cax,-0x8(%rbp)
4004d5:    addl    $0x1,-0x4(%rbp)
4004d9:    cmpl    $0x9,-0x4(%rbp)
4004dd:    jle     4004cf <main+0x1b>
```

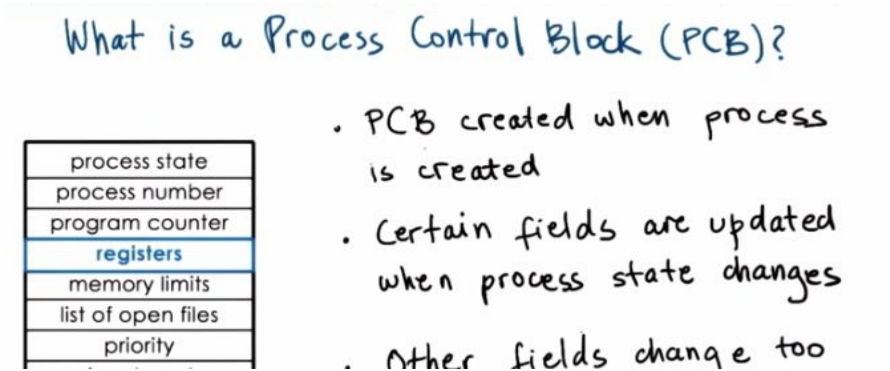
(b) Assembly Code

Another piece of state that defines what a process is doing is the process stack. And the top of the stack is defined by the stack pointer (amit - Process Stack and Stack Pointer). We need to know the top of the stack because we said the stack exhibits this LIFO behavior, so whatever item was the last one to come on top of the stack needs to be the very first item that we can retrieve from the stack. But the stack pointer maintains this information.

And similarly, there are other bits and pieces of information that help the operating system know what a process is actually doing at a particular point of time. **To maintain all of this useful information for every single process, the operating system maintains what we call a process control block, or a PCB (amit - takeaway - PCB).**

10 - Process Control Block

Let's see now what is a Process Control Block (PCB). A PCB is a data



What is a Process Control Block (PCB)?

- PCB created when process is created
- Certain fields are updated when process state changes
- Other fields change too

structure that the **operating system maintains for every one of the processes that it manages. From what we've seen so far the PCB must contain processes state like the program counter, the stack pointer, really all of the CPU registers (their values as they relate to the particular process), various memory mappings that are necessary for the virtual to physical address translation for the process, and other things (amit - just think in terms of what all will be required for OS to run this process).**

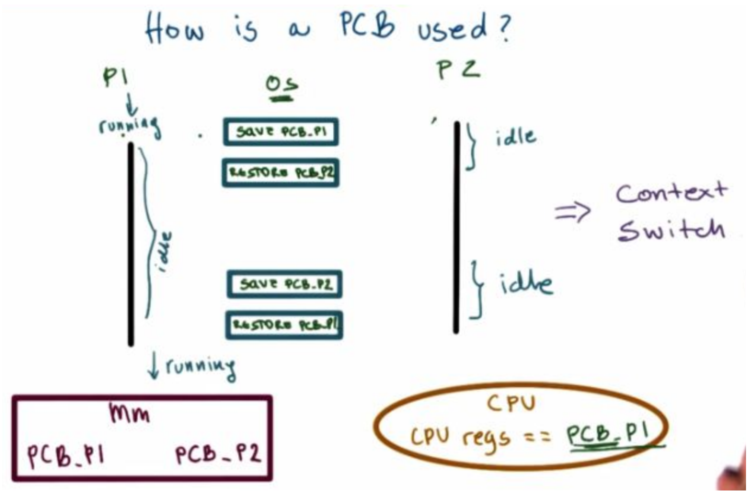
Some of the other useful information includes a list of open files for instance, information that's useful for scheduling like how much time this particular process has executed on the CPU, how much time it should be allocated in the future (this can depend on the process priority), etc.

The PCB data structure is created when the process is initially created itself and it's also initialized at that time. For instance, the program counter will be set to point to the very first instruction in that process.

Certain fields of the PCB are updated whenever the process state changes. For instance, when a process requests more memory the OS will allocate more memory and will establish new valid virtual to physical memory mappings for this process. This will reflect the memory limits information as well as the information regarding the valid virtual address regions for this process. And this perhaps doesn't happen too often.

Other fields of this PCB structure change pretty frequently. For instance, during the execution of program the program counter changes on every single instruction *(amit - question can be - where is PC maintained? In PCB or in CPU? And why?)*. We certainly don't want the OS for every instruction that the process executes to have to spend time to write this new PCB value for the program counter. The way this is handled is that the CPU has a dedicated register which it uses to track the current program counter (PC) for the currently executing process (amit - reason is performance and takeaway is that PC has its own register). This PC register will get automatically updated by the CPU on every new instruction. It is the OS's job however to make sure to collect and save all the information that the CPU maintains for a process and to store it in the PCB structure whenever that particular process is no longer running on the CPU.

11 - How is a PCB Used?



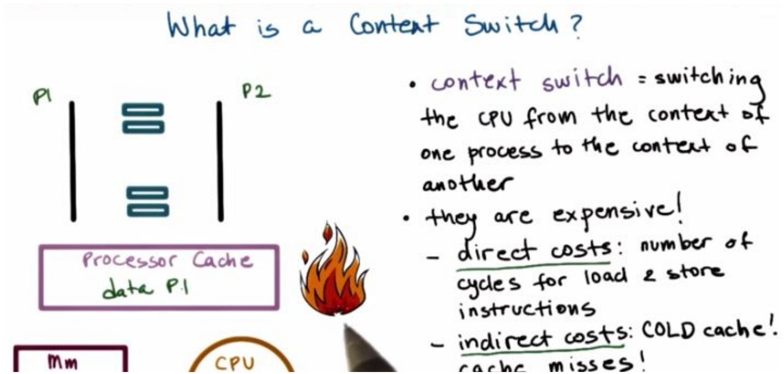
Let's see what we mean by this. Let's assume the OS manages two processes P1 and P2. It has already created them and their PCBs and these PCBs are stored somewhere in memory. Let's say P1 is currently running on the CPU and P2 is idle. ***What this means that P1 is running is that the CPU registers currently hold a values that correspond to the state of P1 so they will ultimately need to be stored in PCB of P1***.

Then at some point the OS decided to interrupt P1, so P1 will be come idle. Now what the OS has to do, is it has to save all the state information regarding P1 including the CPU registers into the PCB for P1. Next the OS must restore the state about P2 so that P2 can execute. What that means is that it has to update the CPU registers with values that correspond to those of the PCB for P2. If at some point during it's execute P2 needs more physical memory it will make a request via the malloc call for instance and the OS will allocate that memory and establish new virtual to physical address mappings for P2 and update as appropriate the PCB data structure for P2.

When P2 is done executing or when the OS decides to interrupt P2, it will save all the information regarding P2's state in the PCB for P2 and then it will restore the PCB for P1. P1 will now be running and the CPU registers will reflect the state of P1. Given that the value of the PCB for P1 corresponds exactly to the values it had when we interrupted P1 earlier. That means that P1 will resume its execution at the exact same point where it was interrupted earlier by the OS.

Each time the swapping between processes is performed the OS performs what we call ***context switch***.

12 - Context Switch

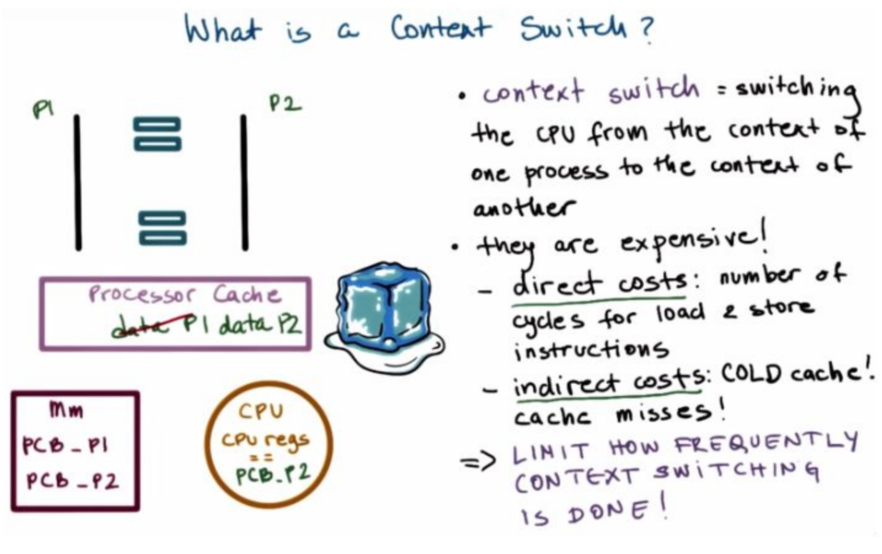

What is a Context Switch?
P1
P2
Processor Cache
data P1
mm
CPU
• context switch = switching the CPU from the context of one process to the context of another
• they are expensive!
– direct costs: number of cycles for load & store instructions
– indirect costs: COLD cache! cache misses!

Recall our illustration that shows how the OS swaps between P1 and P2 for them to share the CPU. In this illustration, the PCBs for P1 and P2 reside in memory and the values of the CPU will change depending on which process is currently executing.

Now we can more formally state that a context switch is the mechanism used by the operating system to switch the execution from the context of one process to the context of another process. In our diagram, this is happening both when the operating system switches from the execution of P1 to the execution of P2. And then again a second time when the OS switches from the execution of P2 back to the execution of P1.

This operation can be expensive, and that's for two reasons. First, there are ***direct costs***, and this is basically the number of cycles that have to be executed to simply load and store all the values of the process control blocks to and from memory. There are also ***indirect costs (amit - takeaways - direct cost and indirect cost and hot and cold cache)***. When process 1 is running on the CPU, a lot of its data is going to be stored into the processor cache. As long as P1 is executing, a lot of its data is likely going to be present somewhere in the processor cache hierarchy.

In the picture, we show a single processor cache, but in practice, modern CPUs have a hierarchy of caches from L1 to L2, down to the last level cache. And each one is larger, but potentially slower than the previous one. More importantly, however, accessing this cache is much, much faster than accessing the memory. For instance, the accesses along the processor cache hierarchy will be on the order of cycles, whereas the accesses to memory will be on the order of hundreds of cycles, for instance. **When the data we need is present in the cache, in this case, that's P1's data, we call this that the cache is hot**. But when we context switch to P2, some, or even all, **of the data belonging to P1 in the cache will be replaced to make room for the data needed by P2. So, the next time P1 is scheduled to execute, its data will not be in the**


What is a Context Switch?
P1
P2
Processor Cache
~~data~~ P1 data P2
MM
PCB_P1
PCB_P2
CPU
CPU regs
==
PCB_P2
• context switch = switching the CPU from the context of one process to the context of another
• they are expensive!
– direct costs: number of cycles for load & store instructions
– indirect costs: COLD cache! cache misses!
=> LIMIT HOW FREQUENTLY CONTEXT SWITCHING IS DONE!

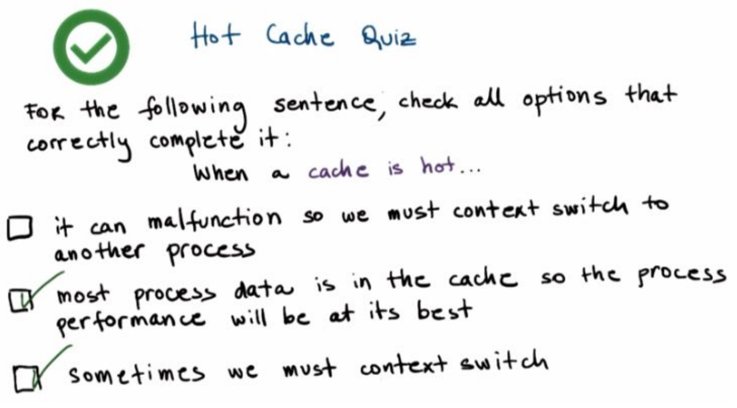
cache. It will have to spend much more time to read data from memory, so it will incur cache misses. We call this the cold cache. Running with a cold cache is clearly bad because every single data access requires much longer latency to memory and it slows down the execution of the process. As a result, we clearly want to limit the frequency with which content switching is done.

13 - Hot Cache Quiz

Here's a quick quiz about the processor cache. For the following sentence, check all options that correctly complete it. The sentence start says, when a cache is hot, and here are the choices.

- ❏ When a cache is hot, it can malfunction, so we must context switch to another process.
- ❏ When a cache is hot most process data is in the cache, so the process performance will be at its best.
- ❏ Or, the last choice, when a cache is hot sometimes we must context switch.

14 - Hot Cache Quiz



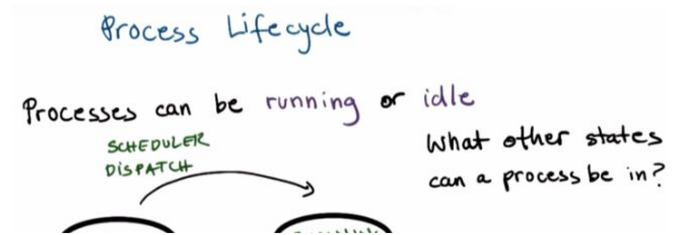
Hot Cache Quiz

For the following sentence, check all options that correctly complete it:

When a cache is hot...

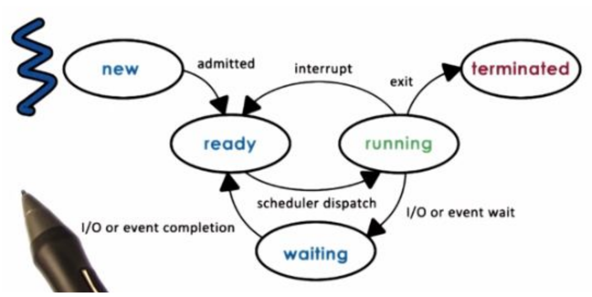
- ☐ it can malfunction so we must context switch to another process
- [✓] most process data is in the cache so the process performance will be at its best
- [✓] sometimes we must context switch

The first option implies that the hot cache means that the cache is physically getting hot, then it will malfunction. However, the term hot cache has nothing to do with the actual temperature of the cache. It merely refers that many of the cache accesses will actually resolve in a cache hit; the data will be found and cached. So in this context, the more cache hits means that the cache is hot. Now coincidentally, this also will lead to a rise in temperature. However, the effects of that aren't going to be that the operating system will context switch to another process. Modern systems and platforms do have a lot of mechanisms to deal with temperature rises, but that's beyond the scope of this lecture. Let's look at the second option. The second option is actually the most correct one. If data is present in the cache, it will be accessed much faster than if data is accessed from memory. So, executing with a hot cache actually corresponds to the state when the process performance is at its best. And unfortunately, three is correct as well. Although hot cache means best performance, sometimes we must context switch although the process cache is hot. And that's because there is another process that maybe has higher priority that needs to execute. Or maybe we have a policy where we have to timeshare the CPU between two processes and P1's time has expired, so we have to context switch and give the CPU to P2.

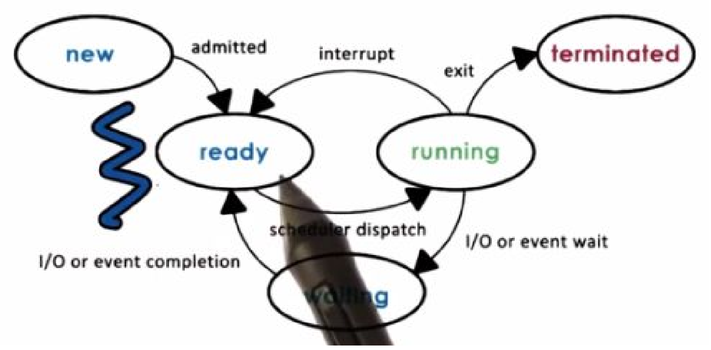

Process Lifecycle
Processes can be running or idle
What other states can a process be in?
SCHEDULER DISPATCH

15 - Process Life Cycle States

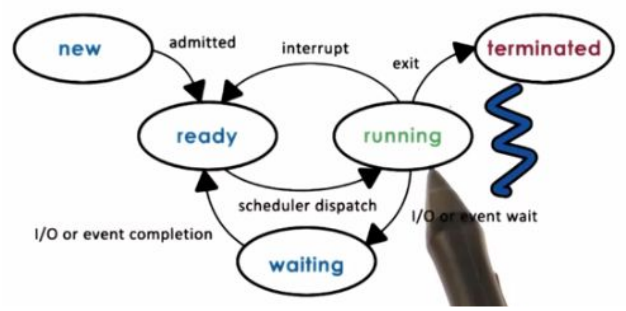
During the context switch discussion, we said that P1 and P2 were going back and forth between *running and idling*. So they were in two states; they were either running or idling. **When a process is running, it can be interrupted and context-switched. At this point, *the process is idle*, but it's in *what we call a ready state*. It is ready to execute, except it is not the current process that is running from the CPU. At some later point, the scheduler would schedule that process again, and it will start executing on the CPU, so it will move into the running state**. What other states can a process be in? And how is that determined?



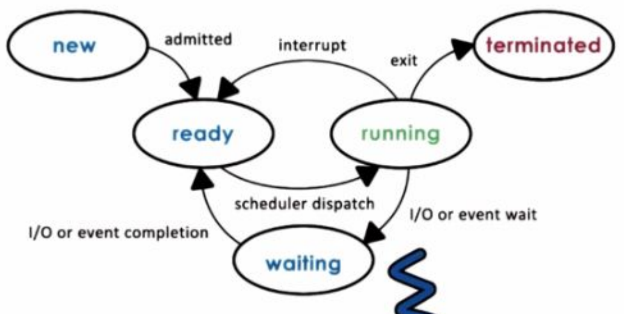
To answer that question, let's look at a general illustration of the states that a process is going through throughout its life cycle. Initially, when a process is created, it enters the new state. This is when the OS will perform admission control, and if it's determined that it's okay, the operating system will allocate and initiate a process control block and some initial resources for this process. Provided that there are some minimum available resources, the process is admitted, and at that point, it is ready to start executing.



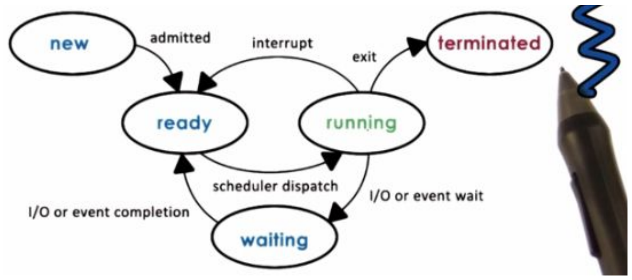
It is ready to start executing, but it isn't actually executing on the CPU. It will have to **wait in this ready state (ready means ready to execute but havn't got the CPU yet)** until the scheduler is ready to move it into a running state when it schedules it on the CPU.



So, once the scheduler gives the CPU to a ready process, that ready process is in the running state. And from here, a number of things can happen. First, the running process can be interrupted so that a context switch is performed. This would move the running process back into the ready state.



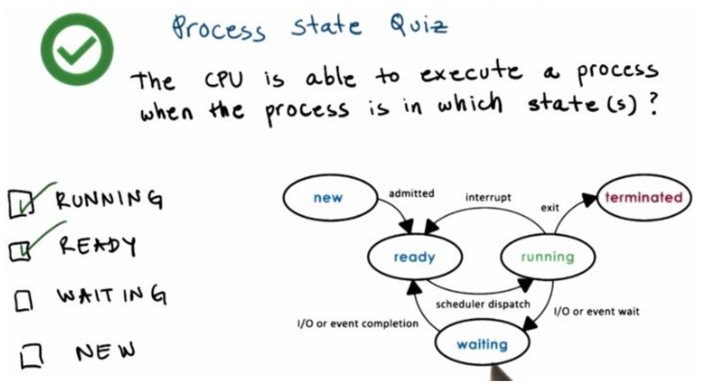
Another possibility is that a running process may need to initiate some longer operation, like reading data from disk or to wait on some event like a timer or input from a keyboard. At that point, the process enters a waiting state. When the event occurs or the I/O operation completes, the process will become ready again.



Finally, when a running process finishes all operations in the program or when it encounters some kind of error, it will exit. It will return the appropriate exit code, either success or error, and at that point, the process is terminated.

16 - Process State Quiz

Let's take a quiz now. Using the process life cycle diagram, let's answer the following question. The CPU is able to execute a process when the process is in which of the following states? You'll need to check all that apply and here are the choices. Running, ready, waiting, or new.

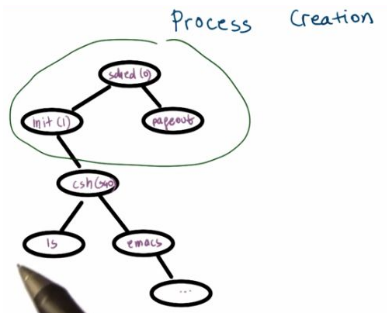


17 - Process State Quiz

A running process is already executing, so it should be marked as a correct answer (amit - wordings of the quiz were confusing, so we need to think carefully). Any of the processes that are in ready state, the CPU is able to execute them. They're just waiting for the operating system's scheduler to schedule them on the CPU.

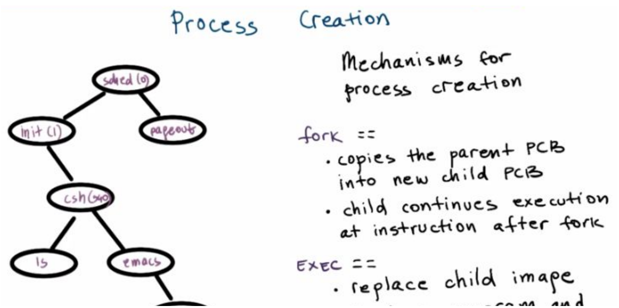
You should remember that as soon as a ready process is scheduled on the CPU, it will continue its execution from the very first instruction that's pointed by the process program counter. It is possible that this is the very first instruction in the process, if the process entered the ready queue for the first time after being newly created. And the other option is that it's some other random instruction in the process binary, depending on when the process was interrupted last time. Either when it was interrupted by the scheduler or because it had to stop executing since it had to wait on an I/O or some kind of external event.

18 - Process Life Cycle Creation



You may be asking yourself how are processes created; what came first? In OS a process can create child processes. In this diagram here, you see that all processes will come from a single root and they will have some relationship to one another where the creating processes is a parent and the created process is the child of that parent. Some of these will be privileged processes; that will be root processes. In fact, this is how most OSes work. **Once the initial boot process is done and the OS is loaded on the machine it will create some number of initial processes. When a user logs into a system a user shell process is created and then when the user types in command like ls, or emacs, then new processes get spawned from that shell parent process. So the final relationship looks like this tree (amit - takeaway - process creation is a tree hierarchy and there is a root to it).**

Most OSes support **two basic mechanisms for process creation, fork and exec. A process can create a child via either one these mechanisms. With the fork mechanism the OS will created a new PCB for the child and then it will copy the exact same values from the parent PCB into the child PCB. At that point,**


Process Creation
sched (0)
init (1)
pageout
csh
ls
emacs
Mechanisms for process creation
fork ==
• copies the parent PCB into new child PCB
• child continues execution at instruction after fork
Exec ==
• replace child image

both the parent and the child will continue their execution at the instruction that is immediately after the fork. And this is because both the parent and the child have the exact same values in their PCB and this includes the value of the program counter. So after the OS completes the fork, both of these processes will start their execution at the exact same point.

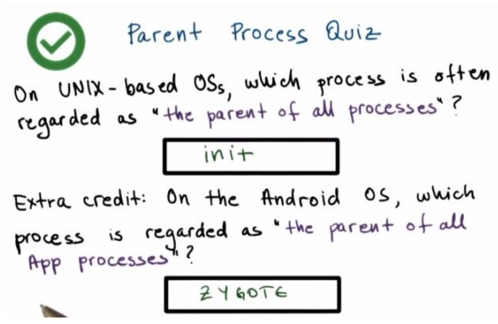
Exec behaves differently; it will take a PCB structure created via fork, but it will not leave its values to match the parent's values like with fork. Instead the OS replaces child's image it will load a new program and the child's PCB will now point to values or describe values that describe this new program and in particular **the program counter for the child will now point to first instruction of the new program (amit - remember that it is the first instruction, which is expected)**.

So the behavior of actually creating a new program is like you call fork, where fork creates the initial process and then you call exec which replaces the child's image, the image that was created in the fork, with the image of this new program.

19 - Parent Process Quiz

Since we have been talking about process creation, let's take a quiz about some special parent processes. The first question is, on UNIX-based operating systems, which process is often regarded as the parent of all processes? And the second question, which is not required but it's extra credit, on the Android OS, which process is regarded as the parent of all App processes? Feel free to use the Internet to find the answer for these questions.

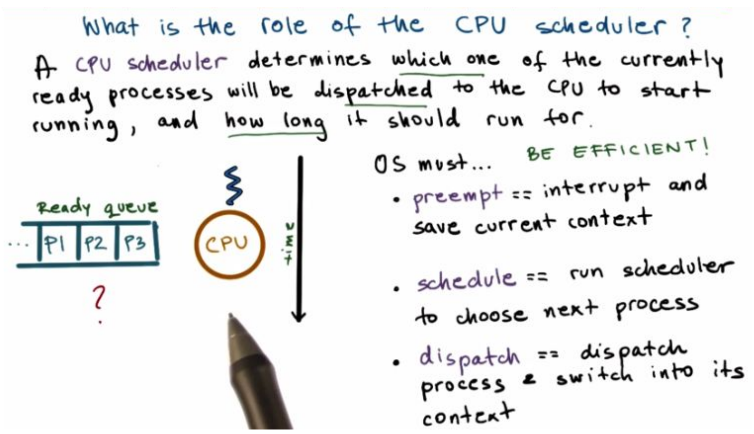
20 - Parent Process Quiz


Parent Process Quiz
On UNIX-based OSs, which process is often regarded as "the parent of all processes"?
init
Extra credit: On the Android OS, which process is regarded as "the parent of all App processes"?
ZYGOTE

On UNIX-based systems, init is the first process that starts after the system boots. And because all other processes can ultimately be traced to init, it's referred to as the parent of all processes. On the Android OS, Zygote is a daemon process which has the single purpose of launching app processes. The OS accomplishes this by forking the Zygote process every time a new app needs to be created, so the Zygote process is the parent of all of the apps.

21 - Role of the CPU Scheduler

Let's talk about process scheduling next. For the CPU to start executing a process the process must be ready first. The problem is, however, there will be multiple ready processes waiting in **the ready queue (amit - takeaway - we had a ready queue so we have a ready state)**. How do we pick what is the right process that should be given the CPU next, that should scheduled on the CPU?

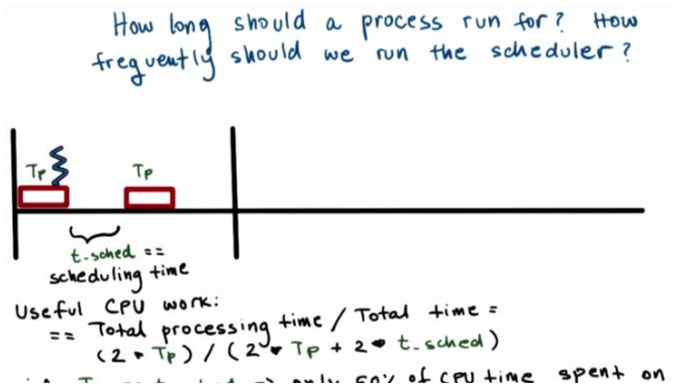


What is the role of the CPU scheduler?
A CPU scheduler determines which one of the currently ready processes will be dispatched to the CPU to start running, and how long it should run for.
Ready queue
... P1 P2 P3
?
CPU
time
OS must... BE EFFICIENT!
• preempt == interrupt and save current context
• schedule == run scheduler to choose next process
• dispatch == dispatch process & switch into its context

This is a simple diagram where we have a ready queue with several processes waiting in it and the CPU which has currently one process scheduled on it. So the question is which process do we run next? This is determined by a component call the ***CPU scheduler***. The CPU scheduler is an OS component that manages how processes use the CPU resources. It decides <u>which one</u> of the currently ready processes will be <u>dispatched</u> to the CPU so that it can start running, start using the CPU, and it also determines <u>how long</u> this process should be allowed to run for. Over time this means that in order for to manage the CPU the OS must be able to preempt, to interrupt the executing process and save its current context. This operation is called ***preemption***. Then the OS must run the scheduling algorithm in order to choose one of the ready processes that should be run next. And finally once the processes is chosen, the OS must dispatch this process onto the CPU and switch into its context so that that process can finally start executing.

Given that the CPU resources are precious, the OS needs to make sure that CPU time is spent running processes and not executing scheduling algorithms and other OS operations. So it should minimize the amount of time that it takes to perform these tasks (preemption, scheduling, and dispatching); the OS must be <u>efficient</u> in that respect. What that means is that it is important to both have efficient designs as well as efficient implementations of the various algorithms that are used, for instance in scheduling, as well as efficient data structures that are used to represent things like the waiting processes or any information that's relevant to make scheduling decisions. **This includes information about the priority of the processes, about their history like how long did they ran in the past, other information may also be useful.**

22 - Length of Process



How long should a process run for? How frequently should we run the scheduler?
Tp
Tp
t_sched == scheduling time
Useful CPU work:
== Total processing time / Total time =
(2 • Tp) / (2 • Tp + 2 • t_sched)
only 50% of CPU time spent on

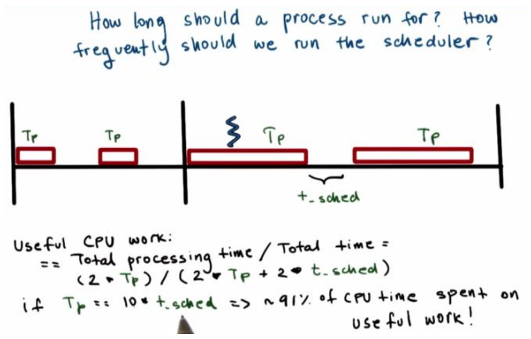
Another issue to consider is: **How often do we run the scheduler? The more frequently we run it the**

more CPU time is wasted on running the scheduler vs running application processes (amit - simple efficiency thing). So another way to ask the same question is: How long should a process run? The longer we run a process the less frequently we are invoking the scheduler to execute.

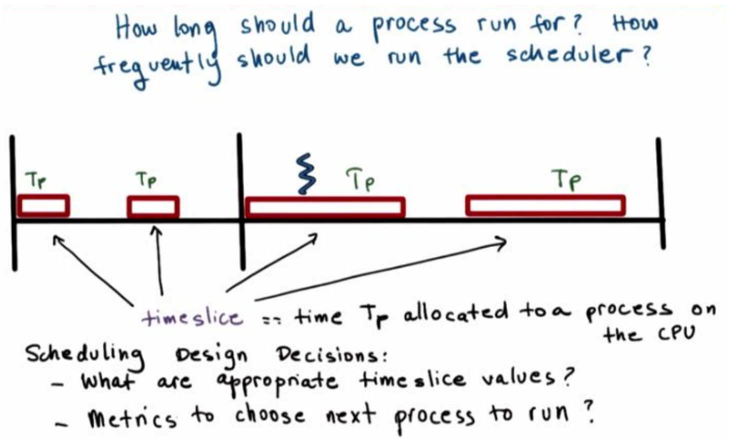
Consider this scenario in which we are running processes for the amount of time T_p and the scheduler takes some amount of time T_{sched} to execute. If we want to understand how well the CPU was utilized, we have to divide the total processing time that was performed during an interval (amit - takeaway here is that - for CPU efficiency think in terms of good time meaning time for process and total time meaning including scheduling as that needs context switching also). So during this interval that was ($2 * T_p$), and then divide that by the total duration of the interval, so the total duration of the interval is ($2 * T_p + 2 * T_{sched}$).

If the processing time and the scheduling time are equal as in this picture, that means that only 50% of the CPU time is spent on useful work! Half of the time, during the time in this interval, the CPU was basically doing systems processing work, scheduling, and that time should be considered overhead.

Let's now look at the second interval where the processing time T_p is much larger than the scheduling time T_{sched}. And let's assume that its actually 10 times the scheduling time, not to scale. So if we work out the math here, we will find out that almost 91% of the CPU time was spent on actually doing useful work! So we're doing much better in this interval in terms of the efficiency of the CPU, how much of it is used for useful application processing vs in this previous example.

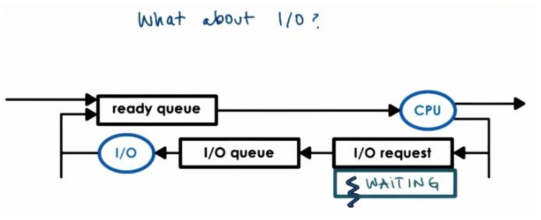

How long should a process run for? How frequently should we run the scheduler?
Tp
Tp
Tp
Tp
t_sched
Useful CPU work:
== Total processing time / Total time =
(2 * Tp) / (2 * Tp + 2 * t_sched)
if Tp == 10 * t_sched => ~91% of CPU time spent on useful work!

In these examples, T_p refers to the time that allocated to a process that has been scheduled to run and so the time that that process can consume on the CPU. We refer to this time as the ***timeslice***.

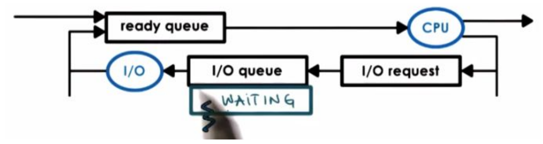

How long should a process run for? How frequently should we run the scheduler?
Tp
Tp
Tp
Tp
timeslice == time Tp allocated to a process on the CPU
Scheduling Design Decisions:
- What are appropriate timeslice values?
- Metrics to choose next process to run?

As you can see there are a lot of design decisions and tradeoffs that we must make when we're considering how to design a scheduler. Some of these include deciding what are appropriate timeslice values, for instance, or deciding what would be good metrics that are useful when the scheduler is choosing what's the next process it should run. I will discuss these design issues in a later lesson, but for now you need to be aware that some decisions need to be made.

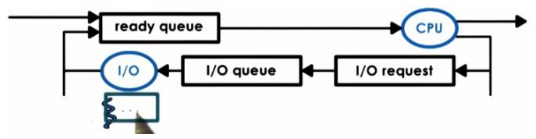
23 - What about I/O?



Before we move forward, we need to consider how I/O operations affect scheduling. So far we know the OS manages how processes access resource on the

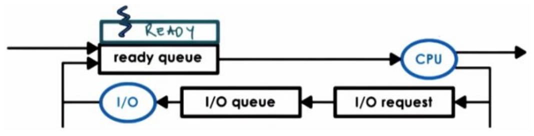


hardware platform and this in addition to the CPU and memory will include I/O devices, peripherals like keyboards, network cards, disks,



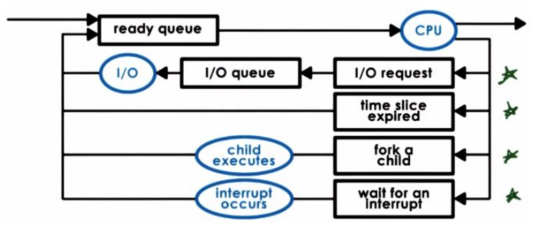
etc. So in this diagram, **imagine a process had made an I/O request, the OS delivered that request, for instance, it was a read request to disk. And then placed the process on the I/O queue that's associated with that particular disk device. So the process is now waiting in the I/O queue (amit - so we had ready queue, now we have I/O queue)**.

The process will remain waiting in the queue until the device completes the operations, so the I/O event is complete, and responds to that particular request. ***So once the I/O request***



is met, the process is ready to run again and depending on the current load in the system it may be placed in the ready queue or it may be actually scheduled on the CPU if there is nothing else waiting in the ready queue before it (amit - between ready queue and it can be direct on cpu).

So to summarize, a process can make its way into the ready queue in a number of ways (amit - imp - question can be how do a process go to ready queue). A process which was waiting on an I/O event ultimately found its way into the ready queue. A process which was



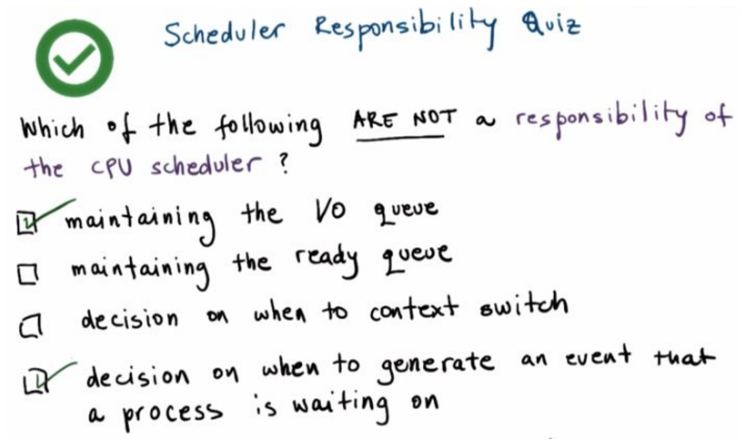
running on the CPU but its timeslice expired goes back on the ready queue. When a new process is created via the fork call it ultimately ends it way on the ready queue. Or a process which was waiting for interrupt, once the interrupt occurs it will also be placed on the ready queue.

24 - Scheduler Responsibility Quiz

To make sure you understand the responsibilities of a CPU scheduler, let's take a quiz. The question is, which of the following are not a responsibility of the CPU scheduler? The options are:

- ❏ maintaining the I/O queue
- ❏ maintaining the ready queue
- ❏ deciding when to context switch
- ❏ or deciding when to generate an event that a process is waiting on.

You should pick all that apply.



Scheduler Responsibility Quiz

Which of the following ARE NOT a responsibility of the CPU scheduler?

☑ maintaining the I/O queue

☐ maintaining the ready queue

☐ decision on when to context switch

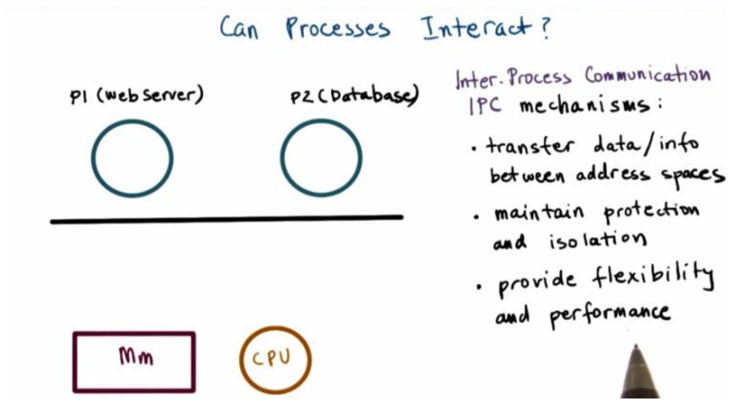
☑ decision on when to generate an event that a process is waiting on

25 - Scheduler Responsibility Quiz

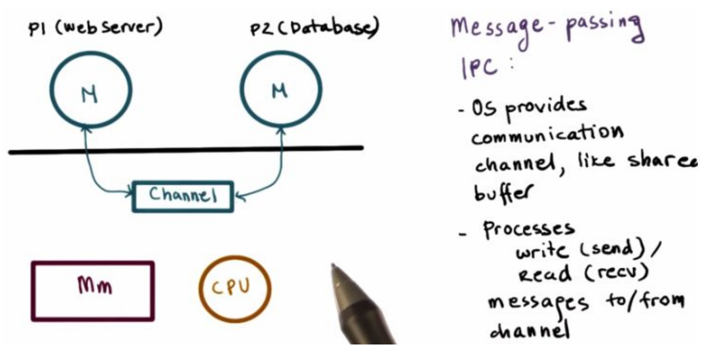
Let's see what the correct answers are. So which of the following are not a responsibility of the scheduler? ***First, the scheduler has no control over when I/O operations occur*****. So clearly the first choice should be marked**. One exception are the timer interrupts. Depending on the scheduling algorithm, the scheduler chooses when a process will be interrupted, so when it will context switch, so clearly it has some influence over when events based on the timer interrupt will be generated. This also answers the third question. It is the scheduler, based on the scheduling algorithm, that decides when a process should be context switched, so this clearly is responsibility of the scheduler. Similarly, it is the scheduler that's in charge of maintaining the ready queue. It is the one that decides which one of the processes in the ready queue will be scheduled to execute next. And finally, the scheduler really has no control over when external events can be generated, other than the timer interrupt as we discussed. So it has no control over events that a process may be waiting on. So this choice should be marked as well.

26 - Inter Process Communications

Another natural question can be, can processes interact? And the simple answer to this is yes. An operating system must provide mechanisms to allow processes to interact with one another. And today in fact, more and more of the applications we see are actually structured as multiple processes. So these multiple processes have to be able to interact to contribute to a common goal of a more complex multi-process application.


Can Processes Interact?
P1 (web server)
P2 (Database)
Mm
CPU
Inter-Process Communication
IPC mechanisms:
· transfer data/info between address spaces
· maintain protection and isolation
· provide flexibility and performance

For instance, here's an example of a web application consisting of two processes on the same machine. The first one is the web server, the front-end, that accepts the customer request. And the second one is the backend, the database that stores customer profiles and other information. This is a very common case in many enterprise and web applications. So, how may these processes interact? Now, before we answer that, remember that the operating systems go through a great deal to protect and isolate processes from one another. Each of them is a separate address space. They control the amount of CPU each process gets, which memory is allocated, and accessible to each process. So these communication mechanisms that we will talk about somehow have to be built around those protection mechanisms. These kinds of mechanisms are called inter-process communication mechanisms, or we refer to them as IPC. The IPC mechanisms help transfer data and information from one address space to another, while continuing to maintain the protection and isolation that operating systems are trying to enforce. Also, different types of interactions between processes may exhibit different properties. Periodic data exchanges, continuous stream of data flowing between the processes, or coordinated applet, to some shared single piece of information. Because of all these differences, IPC mechanisms need to provide flexibility as well as clearly performance.


P1 (web server)
P2 (Database)
M
M
Channel
Mm
CPU
Message-passing IPC:
- OS provides communication channel, like shared buffer
- Processes write (send)/ read (recv) messages to/from channel

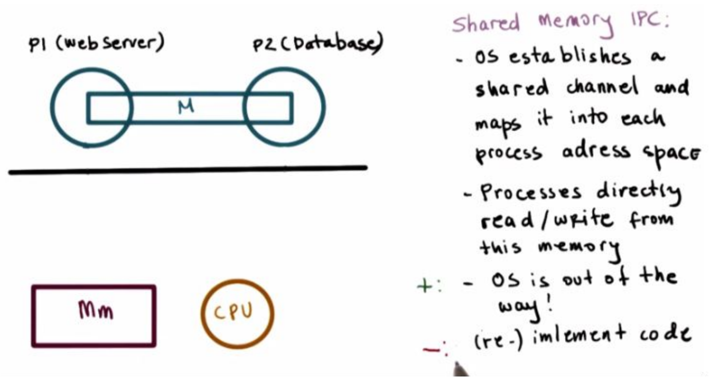
One mechanism that operating systems support is message passing IPC. The operating system establishes a ***communication channel***, like a shared buffer for instance, and the processes interact with it by writing or sending a message into that buffer. Or, reading or receiving a message from that shared communication channel. So, it's message passing because every process has to put the information that it wants to send to the other process, explicitly in a message and then to send it to this dedicated communication channel.

+: OS manages
-: overheads

The benefits of this approach is that it's really the operating system who will manage this channel, and it's the operating system that provides the exact same APIs, the exact same system

calls for writing or sending data, and the reading or receiving data from this communication channel.

The downside is the overhead. Every single piece of information that we want to pass between these two processes we have to copy from the user space of the first process into this channel that's sitting in the OS, in the kernel memory. And then back into the address space of the second process.

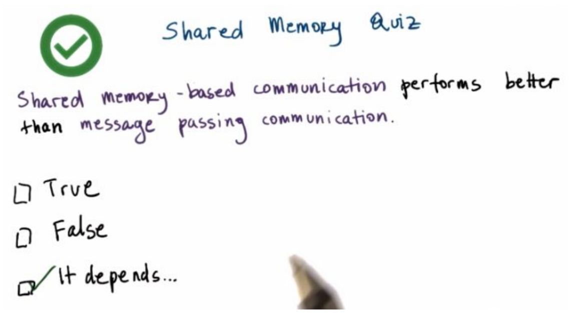

P1 (web Server)
P2 (Database)
M
Mm
CPU
Shared Memory IPC:
- OS establishes a shared channel and maps it into each process adress space
- Processes directly read/write from this memory
+: - OS is out of the way!
-: (re-) imlement code

The other type of IPC mechanism is what we call ***shared memory*** IPC. ***The way this works is the operating system establishes the shared memory channel, and then it maps it into the address space of both processes.*** The processes are then allowed to directly read and write from this memory, as if they would to any memory location that's part of their virtual address space. ***So the operating system is completely out of the way in this case. That in fact is the main advantage of this type of IPC (amit - takeaway - in shared memory OS does not participate much)***. That the operating system is not in the fast path of the communication. So the processes, while they're communicating are not going to incur any kind of overheads from the operating system. ***The disadvantage of this approach is because the operating system is out of the way it no longer supports fixed and well defined APIs how this particular shared memory region is used. For that reason, its usage sometimes becomes more error prone, or developers simply have to re-implement code to use this shared memory region in a correct way***.

27 - Shared Memory Quiz

Let's provide a little bit of more information through this shared memory quiz. Let's look at this statement. Shared memory-based communication performs better than message passing communication. So, you think this statement is true? It is false? Or, whether it depends on something.

27 - Shared Memory Quiz



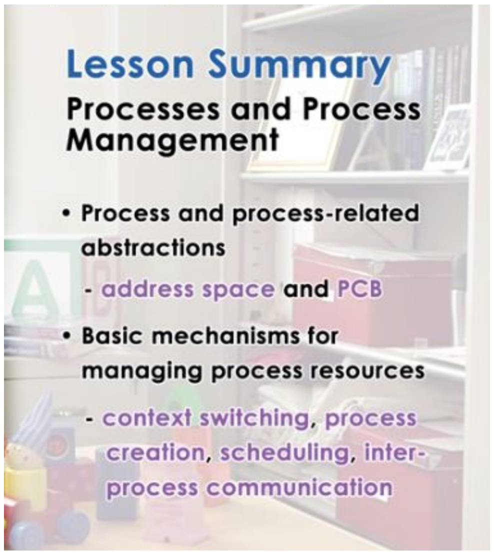
Shared Memory Quiz

Shared memory-based communication performs better than message passing communication.

- ☐ True
- ☐ False
- ☑ It depends...

The correct answer to this is, it depends. With shared memory based communication, the individual data exchange may be cheap, because they don't require that the data is copied in and out of the kernel. **However, the actual operation of mapping memory between two processes, that operation itself is expensive. So, it only makes sense to do shared memory-based communication if that cost, the setup cost, can be amortized across a sufficiently large number of messages. That's why the real answer is, it depends (amit - takeaway - when we see "it depends" think twice for reasons..like in shared memory we need to syn, whats the cost etc etc)**.

29 - Lesson Summary



Lesson Summary

Processes and Process Management

- Process and process-related abstractions
  - address space and PCB
- Basic mechanisms for managing process resources
  - context switching, process creation, scheduling, inter-process communication

In this lesson we learned how a process is represented by OSes. We learned how process is laid out in memory, how OSes use the PCB structure to maintain information about a process during its lifetime. We looked at some of the key mechanisms that OSes support to manage processes like process creation and process scheduling. And then finally, we reviewed some aspects of memory management that are necessary for your understanding of some of the decisions and overheads that are associated with process management.