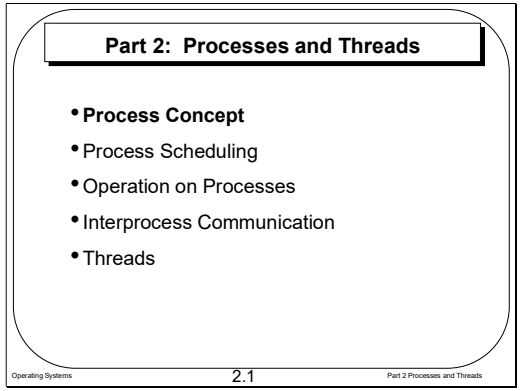
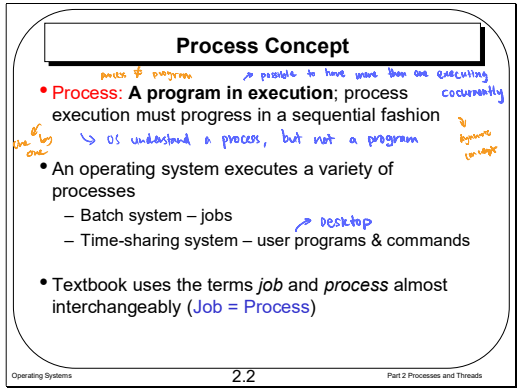
Slide 1



Slide 2

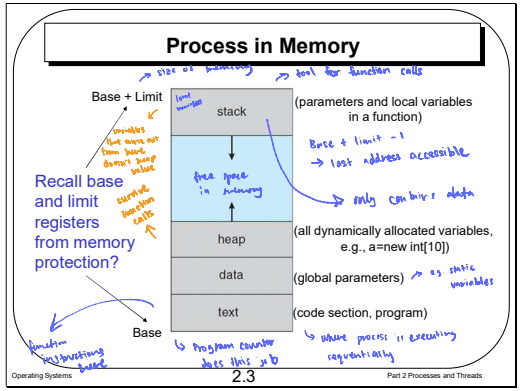


So what is a process? There are many similar terms in operating system. An operating system executes many kinds of programs: a batch system executes jobs. A time sharing system executes user programs and commands.

The textbook uses job and process interchangeably. Here, Job=process. Formally speaking, process is a formal term for operating system; and a job is a larger concept.

A process is defined to be a program in execution. It is a dynamic concept. Process is different from program. The program is an executable file (static concept). If you click the executable (that is the program) multiple times, you will create multiple processes. Those processes have their own main memory and other resources. The program is like a template for creating multiple processes. In the reminder of this chapter, one important part is to learn how the process changes during its lifetime. The process execution must be in a sequential order, one by one instruction. Note, on modern CPUs, out of order execution is supported; but still the instructions are committed in a sequential order.

Slide 3

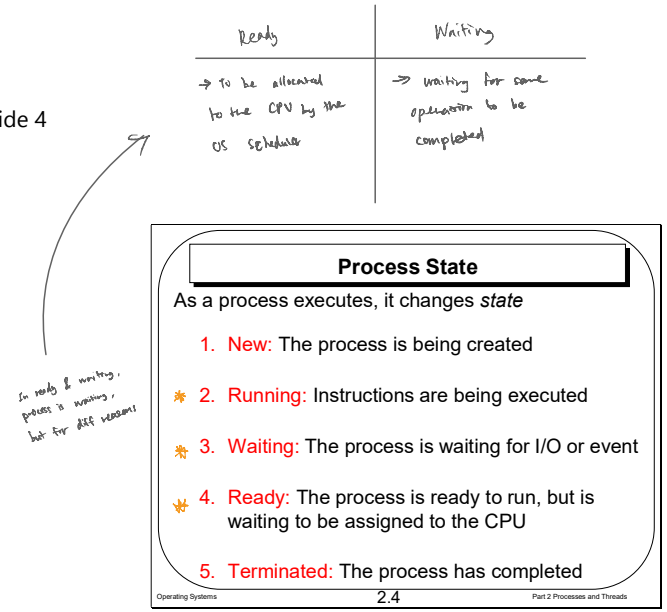


PCB store Base & Limit value

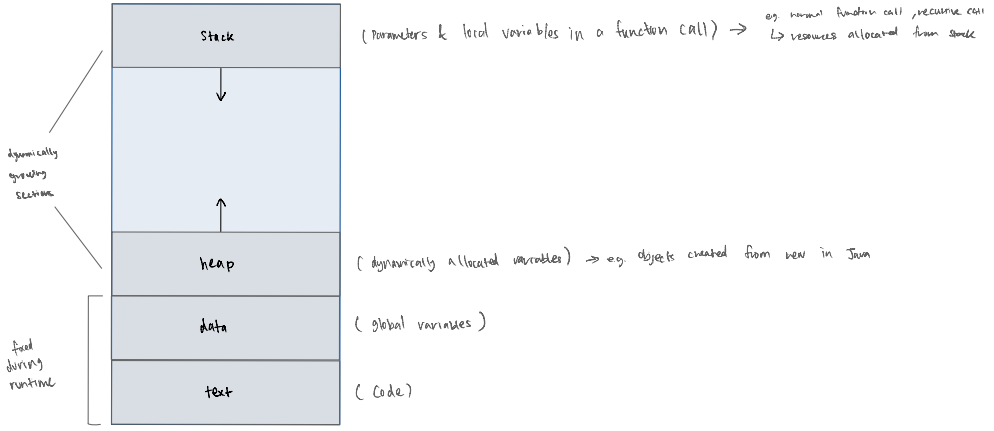
This figure shows the memory layout of a process. It has four sections: text, data, heap and stack. The memory region is defined from Base to Base+limit. Remember those two registers from memory protection slides?

The text and data sections are usually at the bottom, because they are fixed during runtime. Text is for the code and data is for global variables. In C/C++ and java, you can declare global variables. They are allocated in the data section.
Stack and heap are two dynamically growing sections. Stack grows from top down, for parameters and local variables in a function call. Consider a normal function call or even recursive calls. The resources are allocated from stack.
Heap grows from bottom up, for dynamically allocated variables. For example, objects created from new in java.

Slide 4

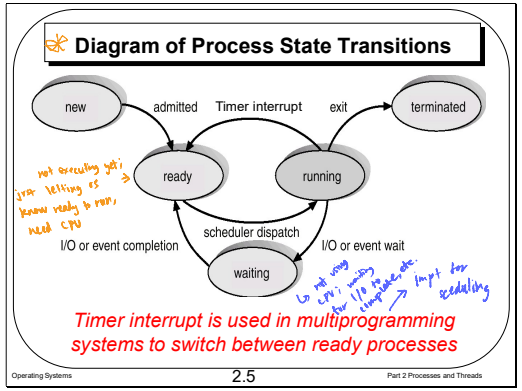


As a process executes, it changes state. A process can have five states: new, running, waiting, ready and terminated. Let's use a figure to illustrate the process state transitions. Note, in both ready and waiting, the process is waiting, although for different reasons. It is waiting in the ready state to be allocated to the CPU by the OS scheduler, whereas it is waiting in the waiting state for some operation to complete.



At any time there is only 1 process in running state
↳ if multicore, yes, able to run in parallel
↳ if not, no

Slide 5



At any time there is only 1 process in running state
Is it possible to go from ~~waiting~~ waiting to ~~waiting~~ running state? Why or why not?

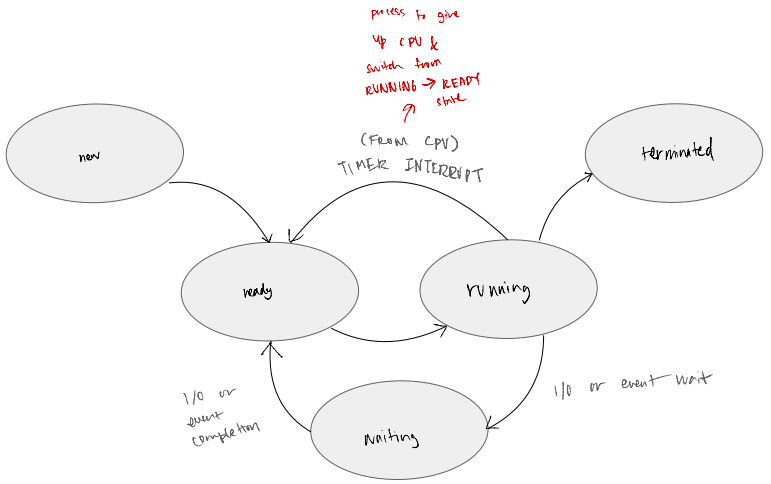
This figure is very important. When a process is being created, its state is new. Then, it is admitted into the system, when the initialization is done (for example, the memory region and other resources are allocated). And it becomes ready for scheduling. The state is ready. In practice, there can be multiple processes that are ready to execute on the CPU. Then, OS chooses a ready process to execute, that is scheduler dispatch. The selected process changes its state from ready to running. We will talk about the details on process scheduling, that is, which process to select next week.

The transitions from running to other states are complex. If it finishes, then the process becomes terminated. The main memory and other resources can be deallocated, and reused by other processes.

There is a timer inside the CPU. If it receives a timer interrupt, the process needs to give up CPU and is switched from running to ready state. The timer interrupt is generated from the CPU every time slice (say, 100 ms) in multiprogramming system (remember in part 1, when we say, the CPU is multiplexed among processes). Recall that in a multiprogramming system, the CPU is multiplexed among processes. When we say multiplexing, we mean: every 100ms, a timer interrupt is generated and then the current process transits from running to ready, and OS picks another ready process to execute on the CPU.

If the process issues an I/O request or will wait for other events, its state will be changed to waiting. We will see the example of event wait in the later slides. When the I/O or the event finishes, it is switched to ready.

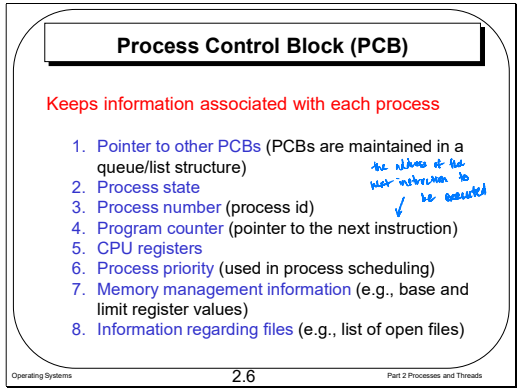
You should be able to understand all the transitions and whether this transition is legal or not. For example, can we directly switch from waiting to running state? No. because at this moment, the other process is running on the CPU, and it must be put into the ready state for further scheduling.



State	Steps
NEW	· newly created process
READY	· Process admitted into system & initialized (e.g. resources allocated) Ready for scheduling.
RUNNING RUNNING → TERMINATED	· OS selects a ready process to execute; scheduler dispatch · Process finished → main memory & other resource deallocated & reused by other processes

WAITING TO RUNNING NOT ALLOWED.
→ other process is running on CPU, it must be put into the ready state for further scheduling

Slide 6

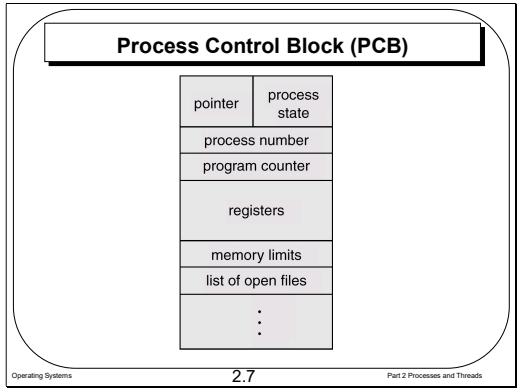


The processes take turn to use the CPU, according to the timer interrupt or I/O/event wait. How does OS support this switch?

We need to use some data structure to maintain the state of the process. The data structure is called PCB, process control block. PCB includes
Pointer to other PCB, because in OS, PCBs of all processes are maintained in a queue structure.
Process state
Process id,
Programmer counter: the address of the next instruction to be executed.
A series of CPU registers.
CPU scheduling information like priority.

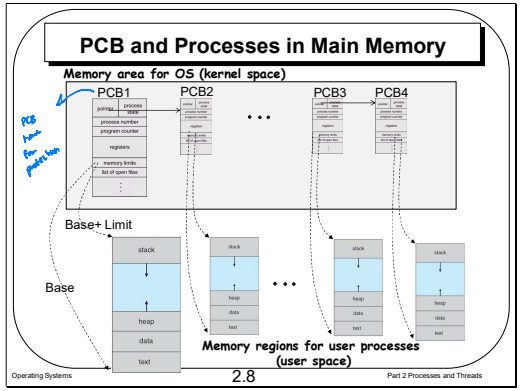
Where is PCB stored? Each process has its PCB stored in the main memory, and hardware protection is enabled for PCBs.

Slide 7



In the main memory, PCB is laid out in this manner. If we need to change the process state, we need to change the corresponding field.

Slide 8



process memory
& base & limit define memory region for a process

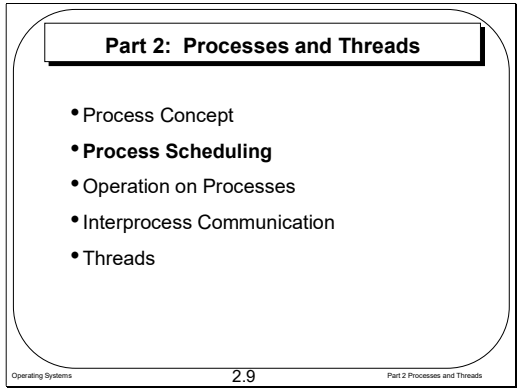
Let's put the concepts together. Let's first look at PCB and memory layout. The entire main memory is divided into two parts: memory area for OS (kernel space) and memory regions for user processes (user space). PCBs are stored in kernel space for protection. The base and limit defines the memory region for a process. The process memory is stored in user space.

Changes to PCB must happen in kernel mode.

Here let me give you one question: suppose PCB1 belongs to Process 1. What happens if Process 1 executes the code "a=new int[10]", that is to allocate one new array? The heap of Process 1 grows.

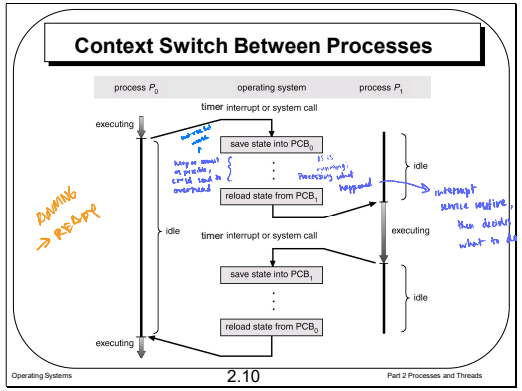
Suppose PCB1 belongs to Process 1. What happens if Process 1 executes the code "a = new int [10]", that is to allocate one new array?

Slide 9



memory can always be access, the difference is who can access

Slide 10



First, let's talk about CPU switch, also called context switch. In a time sharing system, every 100ms, there is a timer interrupt generated, and the running process gives up the CPU. Then, another process is picked to run on the CPU.

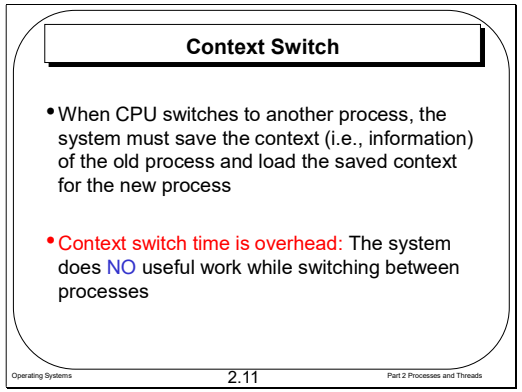
This is one example of context switch. We have two processes P0 and P1. The operations of OS is shown in the middle. At the beginning, P0 is executing. Then, an interrupt occurs (for example, the timer interrupt occurs). P0 will transit from running to ready, and OS should schedule another process for execution. The operating system takes control at this point. It saves the PCB of P0, and loads the state from PCB1. After loading PCB1, we get the program counter etc., and resume the execution accordingly.

What is the mode for those PCB modifications? User mode or kernel mode? Kernel mode.

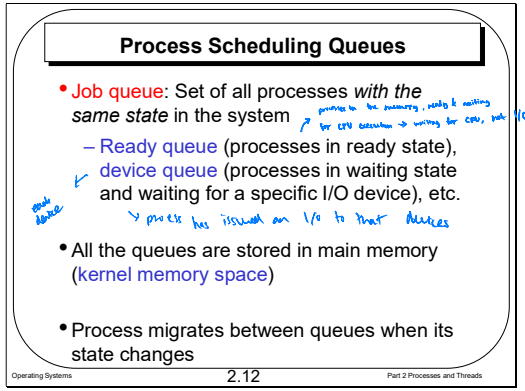
After some time, the timer interrupt occurs, and PCB1 is saved, and PCB0 is reloaded. Then, P0 resumes the execution.

We can see, when the CPU is switched from one process to another, the operating system must save the context of the old process and then load the saved context of the new process. However, context switch does not do any useful work. It is considered as overhead to the system. In OS research, hardware and software approaches have been developed to reduce the context switch overhead.

Slide 11



Slide 12

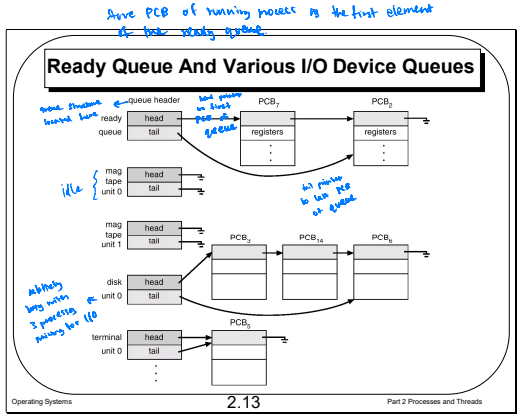


Now, let's talk about process scheduling. But before that, let's see how OS organizes processes.

The operating system uses the queue structure to maintain all the processes with the same state in the operating system. These queues are called job queues. Therefore, a single job queue contains all the processes with the same state in the system.
The ready queue is a set of all processes in the ready state. That is, the processes in the memory, ready and waiting for CPU execution. Note: it is not waiting state. It is just waiting for CPU, not I/O or other events.
There is a device queue for each device. A device queue includes a set of processes waiting for the I/O on that device. This means, the process has issued an I/O to that device.

As the process state changes, process is migrated among different queues. Let's see one example in the next slide.

Slide 13



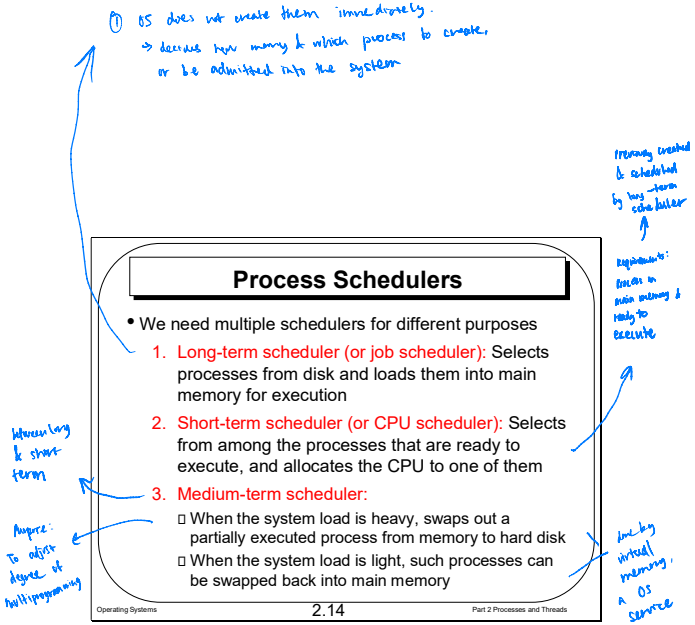
This is an illustration about the ready queue and device queues.

Each queue structure is located at the queue header. In the queue header, there is a head pointer to the first PCB in the queue, and a tail pointer to the last PCB in the queue.

You can also see that there is a device queue for each device. We can judge whether a device is busy or not from the length of the device queue. For example, mag unit 0 is idle, and disk unit 0 is relatively busy, with three processes waiting for I/O.

PCB7 is running. We usually store the PCB of the running process as the first element of the ready queue. Say, PCB7 issues an I/O to the disk. Its PCB is put in the queue of the hard disk. Then after the I/O finishes, it is moved from the device queue of the hard disk to the ready queue.

Slide 14



Summary of last lecture: A process is a program in execution. You should be able to tell the difference between program and process. Then, we spoke about process state transition. You should be able to understand each state transition in the transition diagram. Then, we have spoke about process control block or PCB. Then, we studied how PCB and process state changes during a context switch.

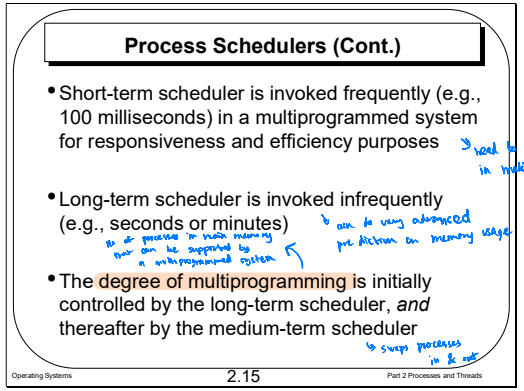
Consider a user clicking an executable many times. Ideally, one click will create one process. However, each process needs some main memory and other resources. Thus, a machine can only support a limited number of processes in the main memory. Once the processes have been created in the main memory, we need to choose from among the ready processes to execute on the CPU. The operating system needs to carefully schedule which process to use the CPU, and to choose the set of processes to remain in the main memory. These decisions are somehow independent of each other. Therefore, we can have multiple levels of schedulers.

The first one is job scheduler. The job scheduler selects the processes from the disk and loads them into memory for execution. This is also called the long-term scheduler. Even if you run many executables, OS will not create them immediately. Instead, the long-term scheduler decides how many and which processes to create, or to be admitted into the system.

The second one is the CPU scheduler, which selects a process from the ready queue for CPU execution. The CPU scheduler is also called the short-term scheduler. The consideration of the short-term scheduler is the processes that are in the main memory and are ready to execute. They have been previously created and scheduled by the long-term scheduler.

The third one is called the mid-term scheduler, a scheduler in between long-term and short-tem scheduler. It is responsible for adjusting the degree of multiprogramming. When the system load is heavy, we need to swap out a partially executed process from memory to disk. When the system load becomes light, we swap in some processes from disk to memory for execution. The OS service that enables this swapping is called virtual memory (2nd half of the course).

Slide 15



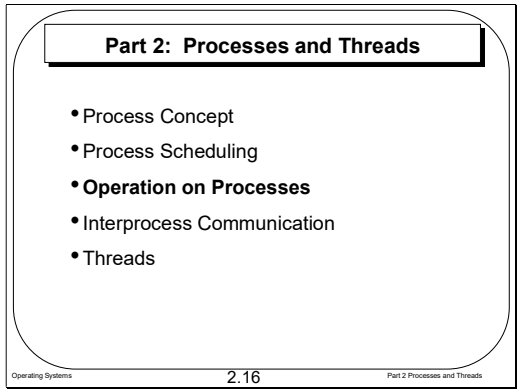
As the name suggests, short-term scheduler is invoked very frequently. It must be fast in making decision.

In contrast, the long-term scheduler is very infrequent. We can do very advanced prediction on the memory usage. See the second half of the course.

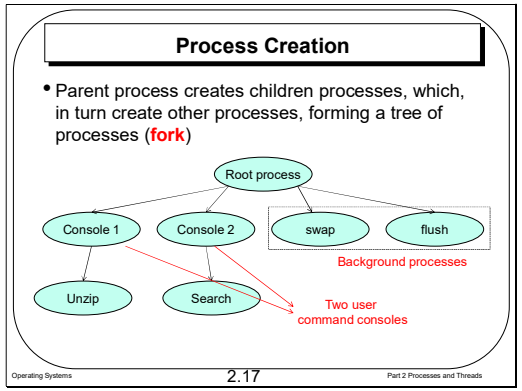
The long-term scheduler controls the *degree of multiprogramming* initially. Then medium-term scheduler does a similar job after that, by swapping in/out processes.

Degree of multiprogramming is the number of processes in the main memory that can be supported by a multiprogrammed system.

Slide 16



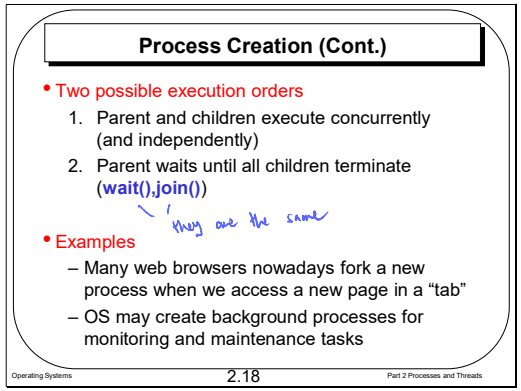
Slide 17



Now, we talk about process creation and process termination.

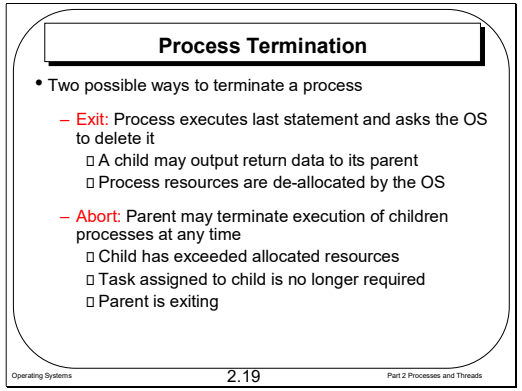
Operating system provides system calls to create a child process. So, the parent process can create a child process, and the child process can create its own child process. The relationship forms a tree. Thus, all the processes are the children of a root process. Then, users may open different consoles and create more processes for their command. There may also be some background processes that are automatically created by the root when the system starts, such as swap (used in the mid-term scheduler) and flush (for writing dirty pages to I/O devices).

Slide 18



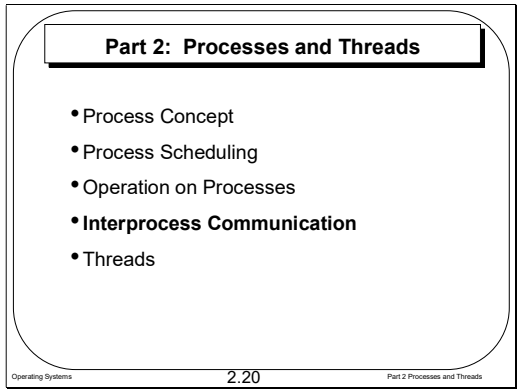
There are two execution orders for parent and child processes. One is to execute concurrently (independently). The other is that parent waits until children terminate. Remember in the process state transition, we have one event wait, causing a process transiting from running to waiting. The parent calls a system call wait, and its process state becomes waiting. Its state will change to ready when the child process finishes execution. In Nachos, the API is join ☺. You can check out the lab briefing slides and video.

Slide 19

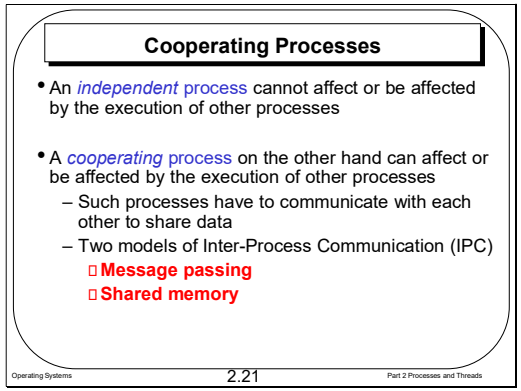


Usually, a child process is not allowed to kill the parent process. OS typically kills a child process if its parent process terminates. This is called cascaded termination.

Slide 20

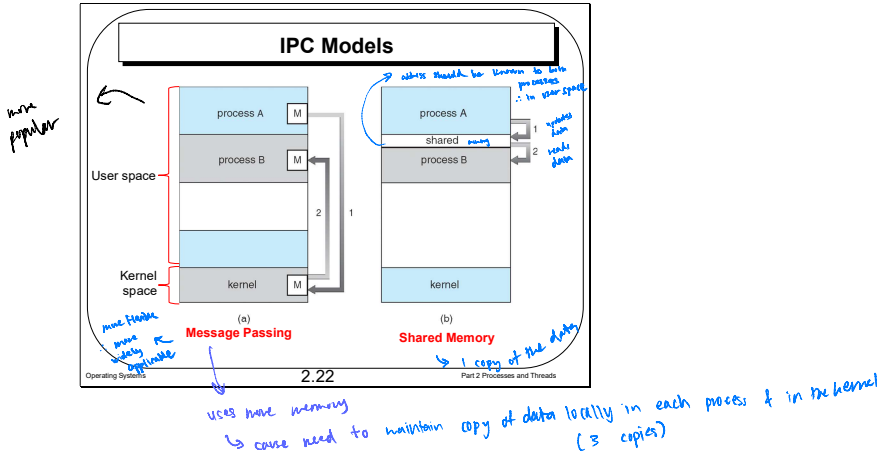


Slide 21



So far, we have talked about the operations for a single processes. Now, we talk about two or more processes working together. Two processes can be independent or cooperating. Two cooperating processes usually have shared data, and they communicate with each other.

Slide 22



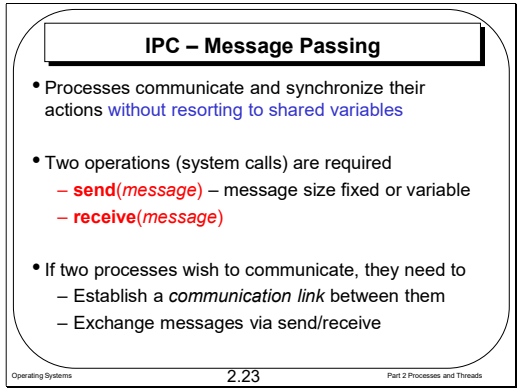
Let's use one figure to illustrate the communication models. Figure (a) is message passing and (b) is shared memory.

In figure (b), Process A updates the data in the memory buffer, and process B reads the data from this shared buffer (shared variables). The communication is done by using memory read/write operations. The address of the shared memory should be known to both the processes. So, it is in user space.

In figure (a), Process A puts a message in a storage named "mailbox" in the kernel space (that is, copy a message from A to kernel space). Process B can take the message from the kernel space (copying the message from kernel space to B).

You can see that shared memory is simple; just memory read/write operations. Message passing is complicated, because we need to have some mailbox as a proxy. However, message passing is more flexible and thus more widely applicable.

Slide 23

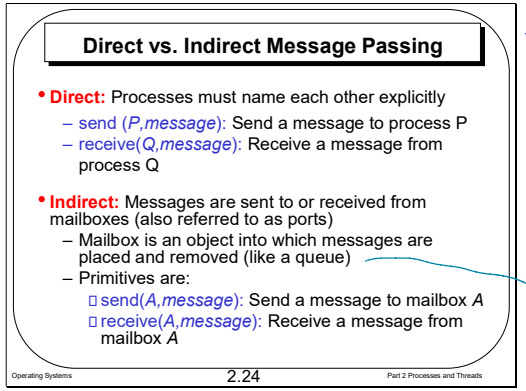


Message passing is a mechanism for processes to communicate and to synchronize their actions without shared variables. Message passing is more flexible. Shared variables are usually for a single machine. Message passing is more general, and can be used for distributed systems.

If two processes P and Q wish to communicate, they need to (a) establish a communication link between them, and (b) send and receive messages with the send/receive APIs.

mail box full → have to wait /
or it might get overriden

Slide 24

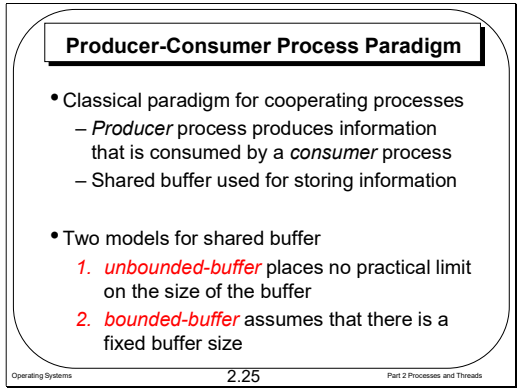


There's a mail box in the kernel that stores the data

There are many implementation details for message passing. I have condensed those details and focus on two key important methods: direct and indirect communication for message passing.

Direct communication means to send and receive with the explicit name of processes, and indirect communication does not have this requirement. In indirect communication, a mailbox is used. The mailbox is an object where messages are placed and removed. The mailbox can be implemented as a queue structure. Each mailbox has a unique id so that senders and receivers can identify the mailbox. Processes can communicate only if they share a mailbox.

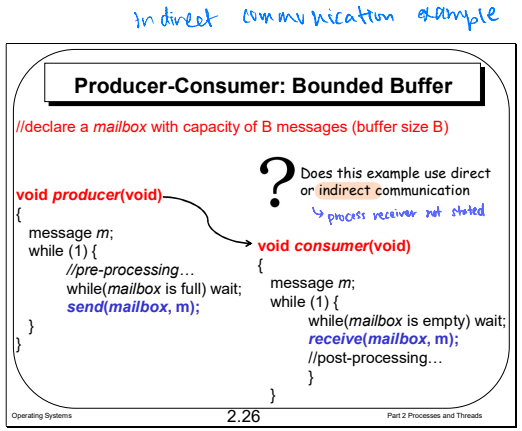
Slide 25



Let's see one example of the classical producer-consumer problem. It is a basic paradigm for cooperating processes. There are many producer-consumer examples in our daily life. For example, the manufacturers produce products and consumers buy the products. Thus, the manufacturers and consumers are one kind of producers and consumers. Also, usually, there can be a warehouse for manufacturers to temporarily store the products, it is like a buffer in this case.

Producer generates messages that is consumed by the consumer. There is a buffer shared by the producer and the consumer. The buffer can be unbounded or bounded. An unbounded buffer places no practical limit on the size of the buffer, basically infinite capacity. But a bounded buffer assumes there is a fixed buffer size.

Slide 26



We first declare a mailbox with capacity of B messages. This is a bounded buffer.

Let's first look at the producer process. It is a while loop. That means, it keeps doing the same thing. It first performs some pre-processing, and then checks whether the mailbox is full or not. if it is full, the producer has to wait. Then, it sends a message to the consumer.
In consumer, it first checks whether the mailbox is empty or not. if so, it has to wait. Then the consumer receives a message from the mailbox and performs some post-processing on the message.

Does this example use direct or indirect communication? Indirect communication.

Slide 27

Part 2: Processes and Threads

- Process Concept
- Process Scheduling
- Operation on Processes
- Interprocess Communication
- **Threads**

Slide 28

Threads

- This part is for **self-learning**

within a process; can share memory

don't share memory

- **About Labs**
  - Nachos uses term “thread”, but means “process”
  - All concepts and mechanisms that we learnt about processes are also applicable to Nachos threads
    - Thread control block = Process control block
    - Thread state = Process state
    - System calls: fork, exit, etc.

Note: In our labs, Nachos uses the term “thread”. However, all concepts and mechanisms that we learnt about processes can be applicable to Nachos threads.
For example, in threads, there is a data structure called thread control block, which is equivalent to PCB for processes.
Thread state is equivalent to process state, although some terms can be different. For example, in Nachos “Just created”, is the same as “New” in our lecture.
Moreover, they have similar system call definitions: fork to create a process/thread, exit means termination.

Slide 29

Threads

- Overview
- Single vs Multithreading Process
- Benefits of threads
- Types of threads
- Multithreading models

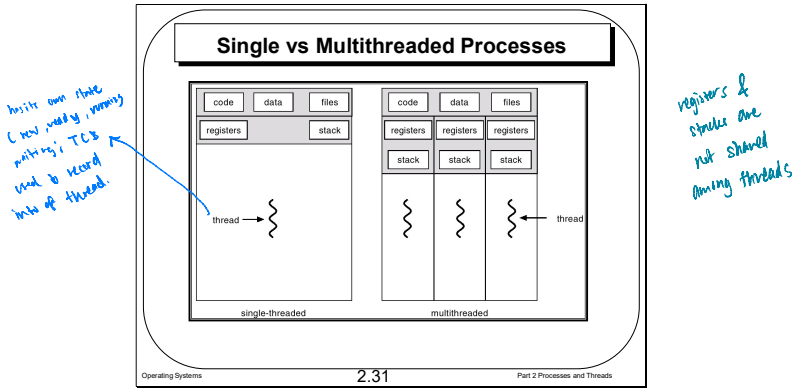
Note: Only in this part, we refer to the concepts of threads. All other parts in this course, we refer to process (single-threaded process).

Slide 30

Overview

- A thread (or lightweight process) is a basic unit of CPU utilization; it consists of:
  - Thread id
  - Program counter
  - Register set
  - Stack space
- A thread shares with its peer threads in the same process:
  - Code and data sections
  - Operating system resources (open files, etc.)
- A traditional or heavyweight process is an executing program with a single thread of control

Slide 31



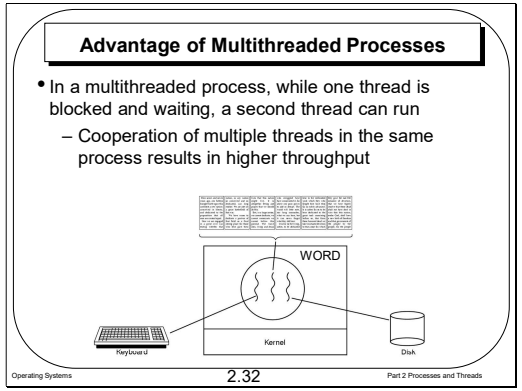
This figure illustrates the difference between a single-threaded process and a multi-threaded process.

On the left we have the single-threaded process, and on the right we have the multi-threaded process. The single-threaded process is just the same as a conventional process that we introduced earlier. Also, a thread has its state (new, ready, running, waiting). A data structure called thread control block (TCB) is also used to record the information of a thread. Thus, a thread has very similar state transitions as a process.

In the multi-threaded process, we can see that all the threads share the code, the data, heap and files. The data section includes all the global variables. Each thread has its own registers and stack.

Why are threads also called lightweight processes? Because the resources for code, data and files are shared among multiple threads. When we create one more thread, those resources are not required to be allocated again. Thus, it is lightweight.

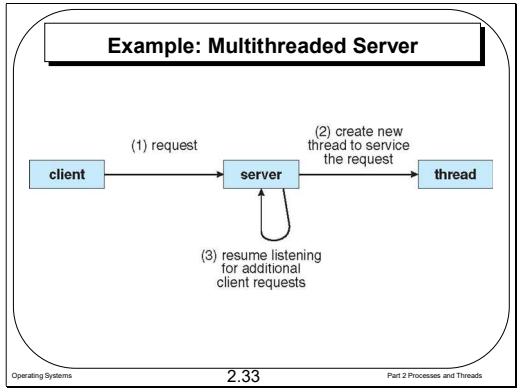
Slide 32



Most operating systems support multithreaded processes. In a multithread process, when one thread is blocked or waiting for an event, the other thread can run. Therefore, cooperation of multiple threads in the same process can deliver better performance.

For example, a word processor is only one process in the windows system. It has multiple threads: one thread for displaying the inputs, the other thread for accepting the keystrokes from users, one thread for auto-saving the document into the hard disk. Even if a thread (say the one accepting the keyboard input) is blocked, other threads can still run for other functionalities of WORD.

Slide 33



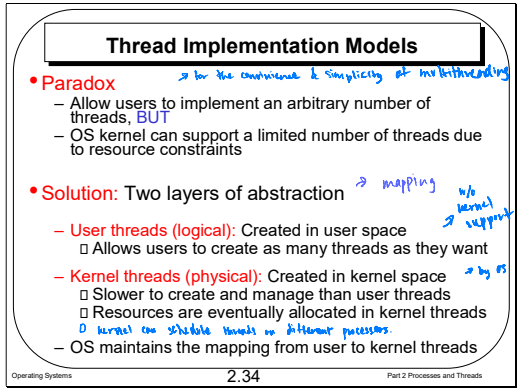
This example is a multithreaded server. This architecture is very common in web servers, FTP servers, etc. There can be multiple clients sharing the same server. A client is a process in the client machine, and the server is a process in the server machine. Let's see how the server handles a request from the client. First, a client submits a request to the server. Next, the server create a new thread to serve the request. And then, the server resume listening for additional client requests. The thread will serve the request independently. When finished, the thread is deallocated. Since creating a new thread requires a small amount of resource, this architecture is more efficient compared with creating a process for each request. There is a more advanced programming paradigm called thread pool. You can google it. here I will not go into details.

Extra Knowledge:
if the server is implemented by a singlethreaded process, the response time is bad when there are many clients. If the server forks a process to serve the I/O request, the overhead of creating a process significantly increases the response time. Even worse, resources such as code, global data, and files cannot be shared among the processes. Therefore, the server is implemented as a multithreaded process.

pthreads — open-source library

Thread stack is stored in process memory space

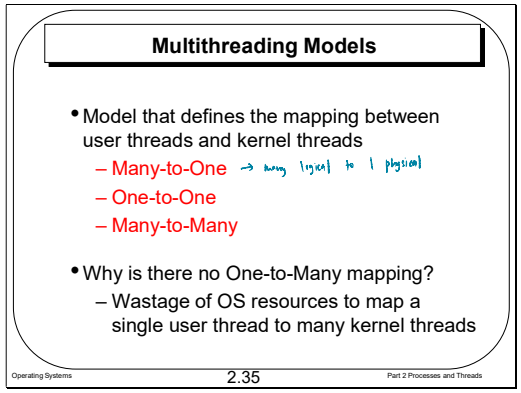
Slide 34



On one hand, we want to allow users to implement an arbitrary number of threads. This is for the convenience and simplicity of multithreading programming. On the other hand, the OS kernel can only support a limited number of threads due to resource constraints.
The solution is to have two layers of abstraction. There are two types of threads, user threads and kernel threads. User threads are created and managed by users/developers, and kernel threads are created and managed by OS. Then, OS maintains the mapping from user threads to kernel threads.

Let's look at more details about user threads and kernel threads. User threads are supported at the user space, without the kernel support. This is to allow users to create as many threads as they want. The kernel threads are supported at the kernel level. They are slower to create and manage than user thread. Resources are allocated in kernel threads. The kernel can schedule threads on different processors.

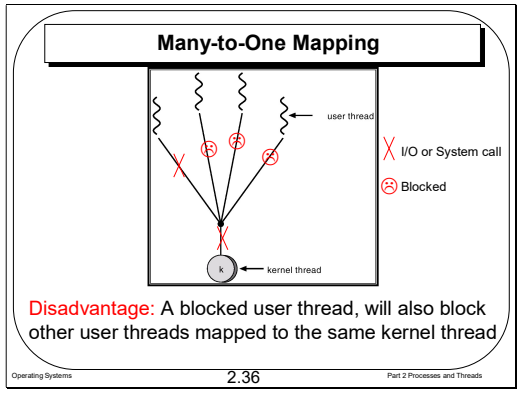
Slide 35



From user space to kernel space, have a mapping from the user threads to the kernel threads. 3 multithreading models: **many-to-one, one-to-one, and many-to-many**.

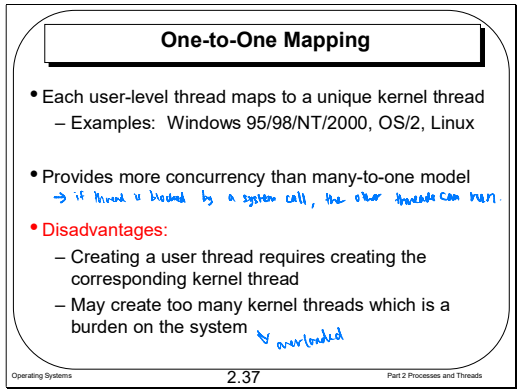
No **one to many** -> it does not make sense to have one user thread mapping to multiple kernel threads. It is a waste of OS resources.

Slide 36



In the **many-to-one** model, many user-levels threads are mapped to a single kernel thread. *For those threads mapped to the same kernel thread*, can't run in parallel due to the single kernel thread. That is, if a thread is stuck by a system call or I/O, the other user threads mapped to the same kernel thread are also blocked.

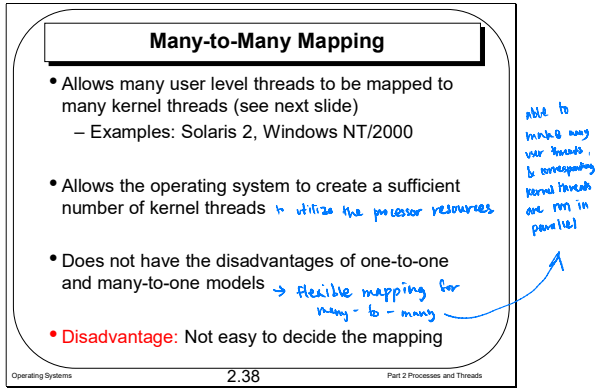
Slide 37



The second model is one-to-one: each user level thread maps to a kernel thread.
Both Windows and Linux support one-to-one model.
This model has the advantage of providing more concurrency than the many-to-one model. If a thread is blocked by a system call, the other threads can run.
The disadvantage is mainly on the thread creation overhead.

- Creating a user thread requires the creation of a kernel thread.
- If too many kernel threads are created, the system can be overloaded.

Slide 38



The last model is many-to-many. This model allows a more flexibile mapping: multiple user level threads can be mapped to multiple kernel threads.
It allows the operating system to create a sufficient number of kernel threads to utilize the processor resources. For example, the operating system can create more kernel threads on a multi-core machine.
Finally, due to the flexible mapping, the many-to-many model does not have the disadvantages of one-to-one and many-to-one models. The developer can create as many user threads as they wish, and the corresponding kernel threads are run in parallel.

Is there any disadvantage? Mapping complexity.

Slide 39