CPSC 457 Notes

Brian Pho

June 28, 2017

Table of Contents

Chapter 1

Slide Set 1: Administrative and Introduction

1.1 Main Topics

- Processes
- Threads
- Concurrency
- Scheduling
- Deadlock
- Memory
- File Systems
- Input/Output

1.2 What is an Operating System?

OS is a piece of software between hardware and applications. Deals with

- Performance
- Resource allocation
- Security

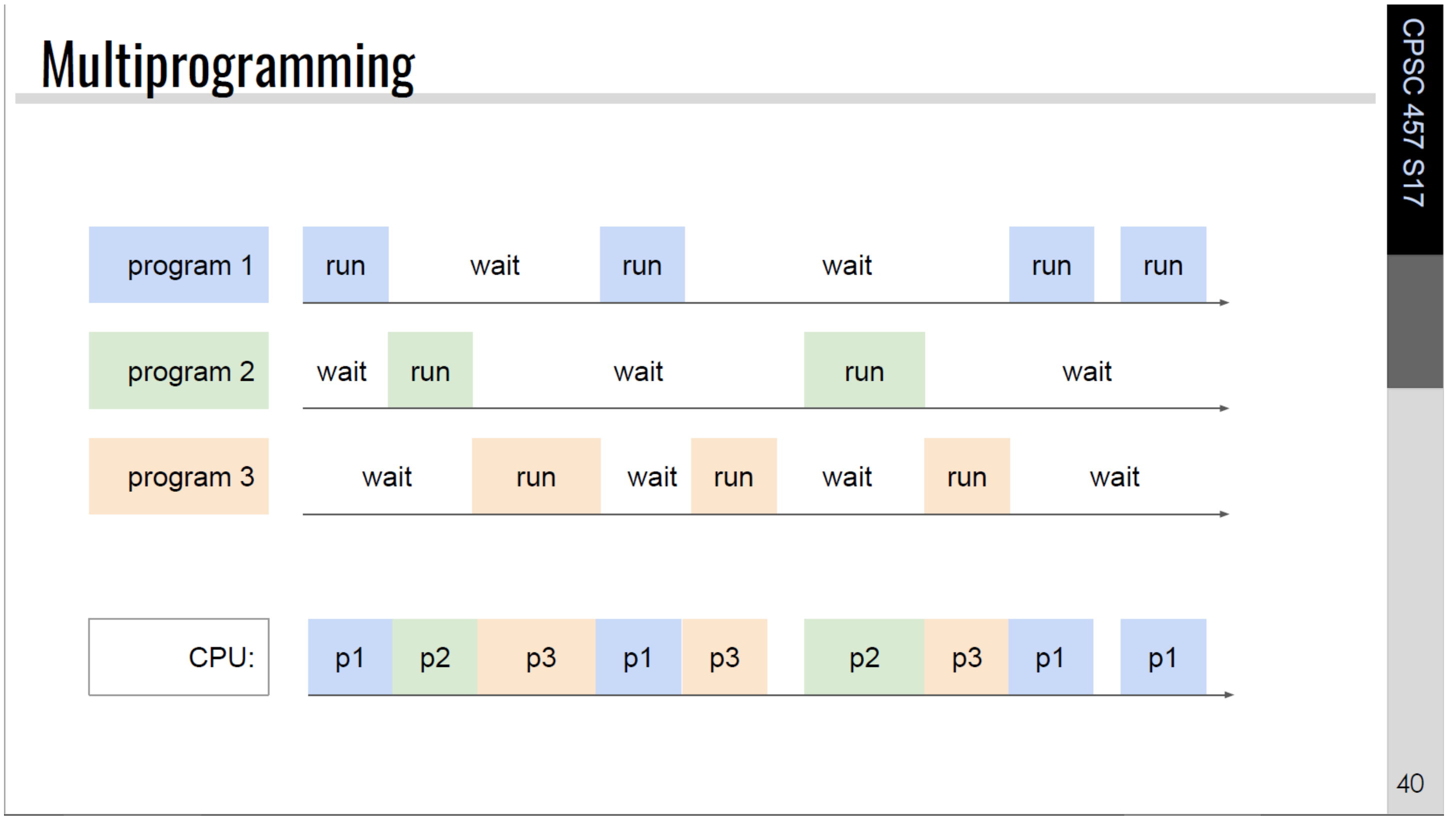
It manages all of the resources. Files are an abstraction by the OS in place of manual disk writing/reading. It is also the software that runs all of the time (in kernel mode). An OS may be views as an extended machine (abstraction) or as a resource manager. A batch system is when similar operations are groups together. The GUI is not part of the OS (kernel) but rather the shell

1.3 Important Concepts

- **Multiprogramming**: the illusion of parallel processing by quickly switching programs (hold multiple programs in memory) (like pipelining)
- **Spooling**: a buffer for IO devices
- **Timesharing**: multiple users using the same machine or an interactive service for multiple users (extension of multiprogramming)
- **Multiplexing**: the sharing of resources in time and space

Buffering vs Spooling: Buffer is actively asking, Spooler is actively receiving.

1.4 Multiprogramming



Multiprogramming

Chapter 2

Slide Set 2: Hardware, Booting, Interrupts, and VMs

2.1 CPU

The unit in a computer that performs computations. 4 steps in a CPU cycle:

1. Fetch
2. Decode
3. Execute
4. Repeat steps 1-3

Special purpose registers

- Program counter (PC): holds address of next instruction
- Stack pointer: points to the top of the current stack in memory
- Status register: interrupt flag, privilege mode, zero flag, carry flag

CPU pipelining: perform multiple steps per cycle. A superscalar CPU uses a buffer to allow multiple FDE units to run. The speed of a pipeline is the speed of the slowest unit in the pipeline.

2.2 Caches

Memory that is quickly accessible by the CPU.
A *cache hit* is when the data/instruction needed by the CPU is found in the cache.
A *cache miss* is when the data/instruction needed by the CPU is not found in the cache.
There is L1 and L2 cache. L1 cache is inside each core while L2 cache is shared by all of the cores. Important memory is kept *very close* to the CPU.

2.3 Memoization

Optimization technique similar to caching. Trade memory for speed. Method is what makes dynamic programming work. Stores the results of an expensive computations.

2.4 Hardware Review - I/O

Device Controller: chip(s) that control a device
Device: connected to the computer through the controller
Device Driver: software that talks to a controller
Buses: wires that transfer data between computer components

2.5 Booting

Steps when a computer is turned on

1. BIOS started
2. Check RAM, keyboard, and other devices
3. Record interrupt levels and I/O addresses of the devices
4. Determine the boot device
5. Read from boot device into memory
6. Read secondary boot loader into memory
7. Loader reads in the OS
8. OS queries BIOS to get configuration info and initializes all device drivers
9. OS creates device table and all necessary background processes

2.6 Kernel

Basic unit of the OS and is always running. "Running at all times" mostly means listening and responding to events from hardware. Not all parts of the OS run in kernel mode, some parts run in user mode. User mode applications cannot directly talk to hardware, it must go through the OS.

2.6.1 Kernel Mode

The kernel runs in kernel mode (unrestricted mode, god mode). All instructions, I/O operations, and memory are allowed and accessible. There are two modes, kernel and user mode. User mode is limited and has only a subset of the operations that kernel mode has. Some of the OS operates in user mode.

2.6.2 User Mode

For a user to access I/O operations, the user invokes a trap (aka makes a system call). A trap is a special instruction that switches from user mode to kernel mode and pauses the application.

2.7 I/O

The kernel can do I/O in many ways

- CPU constantly asks I/O if it is done (busy wait)
- CPU asks if I/O is done, if not then CPU sleeps for a set amount of time and retries (busy wait with sleep)
- CPU sleeps and is woken by an interrupt (interrupt)

2.8 Interrupts

A literal interruption of the CPU. The CPU suspends what it is doing to handle the interrupt and then resumes it's previous operation.

2.9 Direct Memory Access (DMA)

A piece of hardware used for bulk data movement between main memory and I/O. This is to bypass the CPU and let the CPU perform more operations not for I/O. DMA acts as a buffer for the CPU to prevent it from being overwhelmed by interrupt requests.

2.10 Interrupts vs. Traps

The source of interrupts are external while traps are internal. Interrupts are asynchronous while traps are synchronous. Both put the CPU in kernel mode and both save the current state of the CPU.

2.11 Kernel Designs

- Monolithic: entire OS runs as a single program in kernel mode (faster but buggier)
- Micro: only essential components in kernel mode (slower but less buggy)
- Modular (hybrid): some components are in kernel mode, some aren't
- Layered: components are organized into a hierarchy of layers

2.12 Virtual Machine

Abstract hardware of a single computer into several different execution environments. A hypervisor is the software that runs VMs. Different versions of hypervisors are:

- Bare-metal: runs directly on hardware
- Hosted: runs on top of another OS
- Hybrid: runs on both top and directly of hardware

Chapter 3

Slide Set 3: System Calls and Processes

3.1 System Calls

Steps when a system call is called:

- Application issues system call
- OS switches from user mode to kernel mode
- OS saves application state
- OS does the requested operation
- OS switches back to user mode and restores application state
- Application resumes

Think of system calls as an API provided by the OS. System calls are minimalist and not very easy to use. OSs often provide libraries for applications to use to access system calls. A library provides a set of functions that are available to a programmer.

3.2 Processes

A process is a program in execution. It holds all of the info needed by the OS to run the program. It has the following:

- Address space
- Set of resources
- PC, stack pointer, registers
- Process ID

Processes are allowed to create new processes. An orphaned process is a process that doesn't have a parent process. Orphaned processes are adopted by init. Two processes can communicate with each other via a pipe. Pipes are accessed using file I/O APIs. Processes allow multitasking instead of idling while doing I/O. It gives the illusion of parallelism. Each process gets:

- Address space
- Text section

- Data section
- Heap
- Stack
- PC

3.3 Program vs Processes

A program is a passive entity while a process is an active entity. A program becomes a process when it is loaded into memory for execution. A program can have more than one process.

- fork(): create and run child process, a duplicate of parent.
- exec(): replace current process
- wait(NULL): wait for the child process to change states

fork() returns 0 in the child process and the pid of the child process in the parent. fork() creates and starts a new processes.

3.4 Process Control Block (PCB)

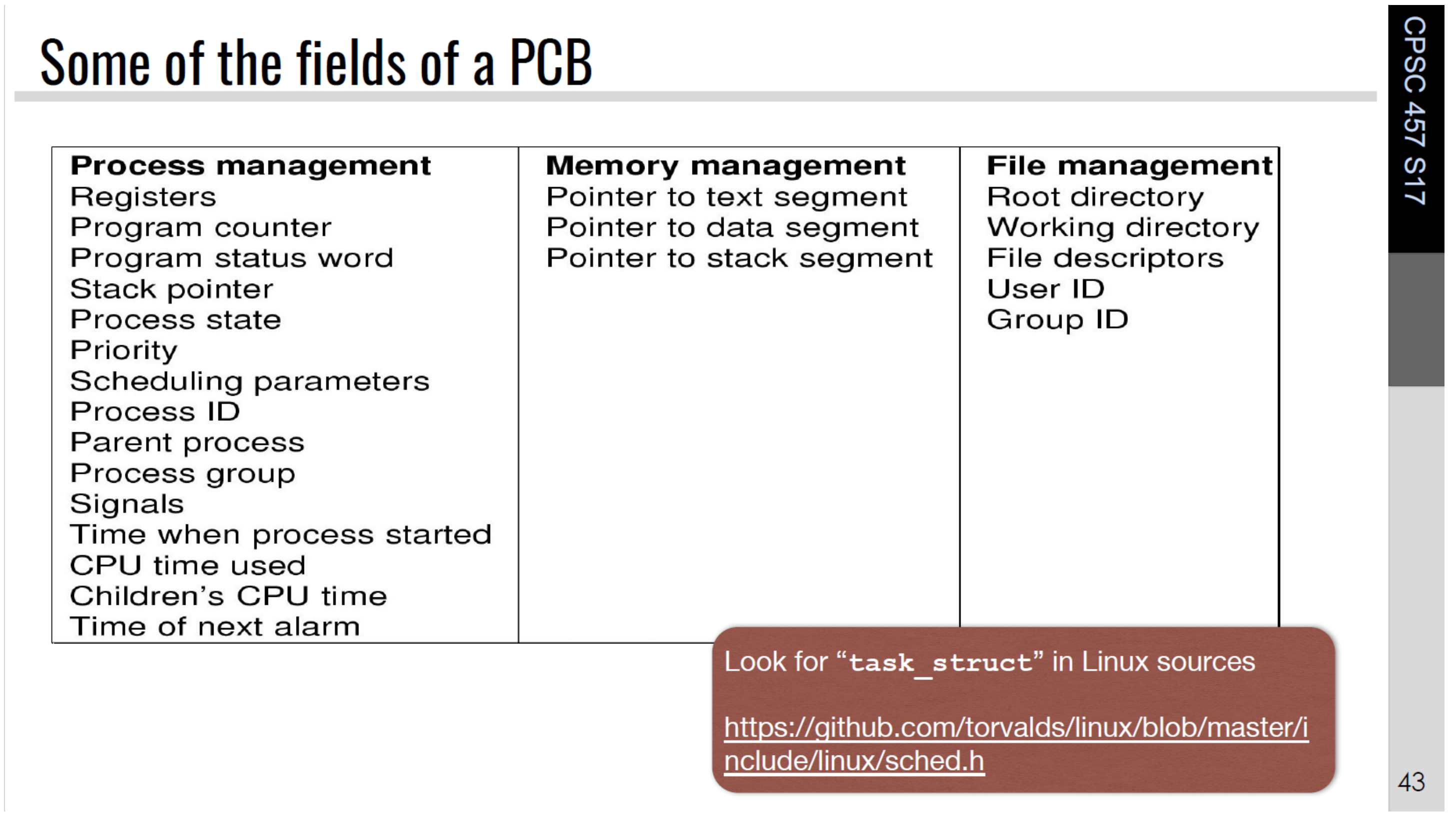
Each process is represented in the OS by a PCB that includes:

- Process state
- Program counter
- CPU registers
- CPU scheduling
- Memory management info
- Accounting info
- I/O status info

The process table is a collection of all PCBs. In Linux, the PCB is called task_struct. Some fields of the PCB:

- Process Management
  - Registers
  - PC
  - Stack pointer
  - Process state
- Memory Management
  - Pointer to (text/data/stack) segment

- File Management
    - Root directory
    - Working directory
    - User/Group ID



Some of the fields of a PCB

Process management	Memory management	File management
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Look for "`task_struct`" in Linux sources

https://github.com/torvalds/linux/blob/master/include/linux/sched.h

CPSC 457 S17

43

3.5 Operations on Processes

Each process gets a *pid* which is a unique process identifier.

- fork(): process creation
- exit(); process termination

3.6 init

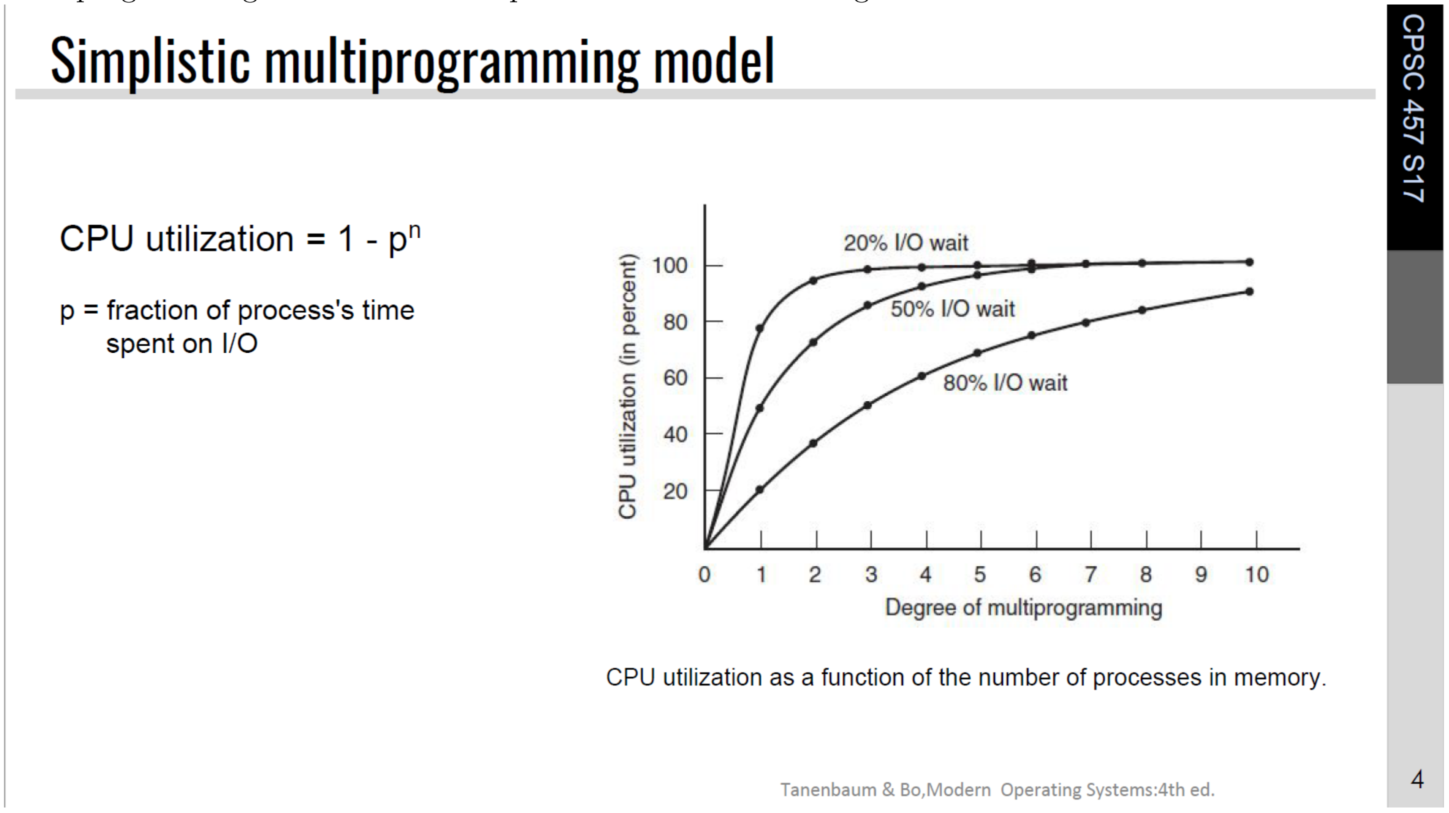
init is the first process started during the booting of the computer and is the ancestor of all user processes. Has a pid of 1. System processes are not descendants of init. Orphaned processes are adopted by init.

Chapter 4

Slide Set 4: Processes and Threads

4.1 CPU Utilization

CPU Utilization $= 1 - p^n$ where p = fraction of processes' time spent on I/O and n is the degree of multiprogramming or the number of processes that are running at a time.



Simplistic multiprogramming model

CPU utilization = 1 - p^n

p = fraction of process's time spent on I/O

20% I/O wait
50% I/O wait
80% I/O wait
CPU utilization (in percent)
100
80
60
40
20
0
1
2
3
4
5
6
7
8
9
10
Degree of multiprogramming

CPU utilization as a function of the number of processes in memory.

Tanenbaum & Bo,Modern Operating Systems:4th ed.

CPSC 457 S17

4

4.2 Process Creation

- init is created at boot time
- only existing processes can create new processes via fork()
- all processes are descendants of init

Each process has its own address space.

- Stack

- Heap
- Data
- Text
- PCB

4.3 Resource Allocation

- child obtains resource directly from OS
- child obtains portion of resources allocated to the parent
- child shares some/all resources with the parent

4.4 Process Termination

What causes a process to terminate?

- self termination
- exceptions
- errors
- killed by users
- normal exit (voluntary)
- error exit (voluntary)
- fatal error (involuntary)
- killed (involuntary)

When terminating a process, what happens in memory?

- free memory
- assign child process a new parent
- delete PCB

What about other resources?

- free all allocated resources
- remove processes from process table

Default behavior on Linux is to reparent the child process to the init process.

4.5 Process Scheduling

The objective is to maximize CPU utilization by having a process executing on the CPU at all times. The process scheduler creates a scheduling queue:

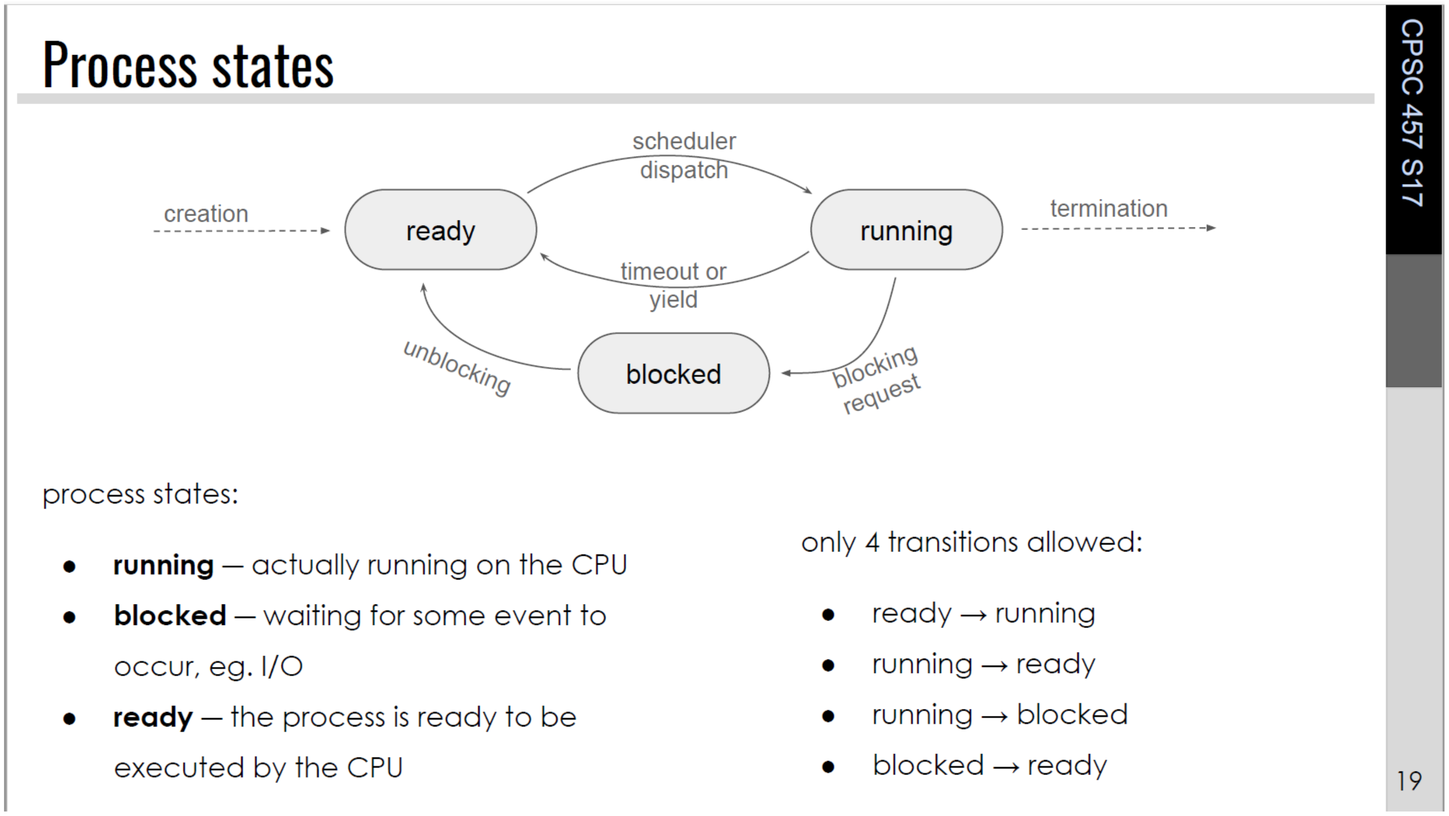
- Job queue: all processes in the system
- Ready queue: processes in memory that are ready to be executed
- Device queue: processes waiting for a particular I/O device

4.6 Process States

- running (actually using the CPU at that instant)
- ready (runnable; temporarily stopped to let another process run)
- blocked (unable to run until some external event happens)

Logically, the first two states are similar. In both cases the process is willing to run, only in the second one, there is temporarily no CPU available for it. The third state is fundamentally different from the first two in that the process cannot run, even if the CPU is idle and has nothing else to do. Only 4 transition allowed:

- Ready − > Running (scheduler)
- Running − > Ready (timeout)
- Running − > Blocked (blocking request)
- Blocked − > Ready (unblocking)



Process states

process states:

- **running** — actually running on the CPU
- **blocked** — waiting for some event to occur, eg. I/O
- **ready** — the process is ready to be executed by the CPU

only 4 transitions allowed:

- ready → running
- running → ready
- running → blocked
- blocked → ready

19

4.7 Context Switching

The switching of one process to another. Occurs in kernel mode. Time slice is the allocated amount of time a process is given. When the OS switches between processes:

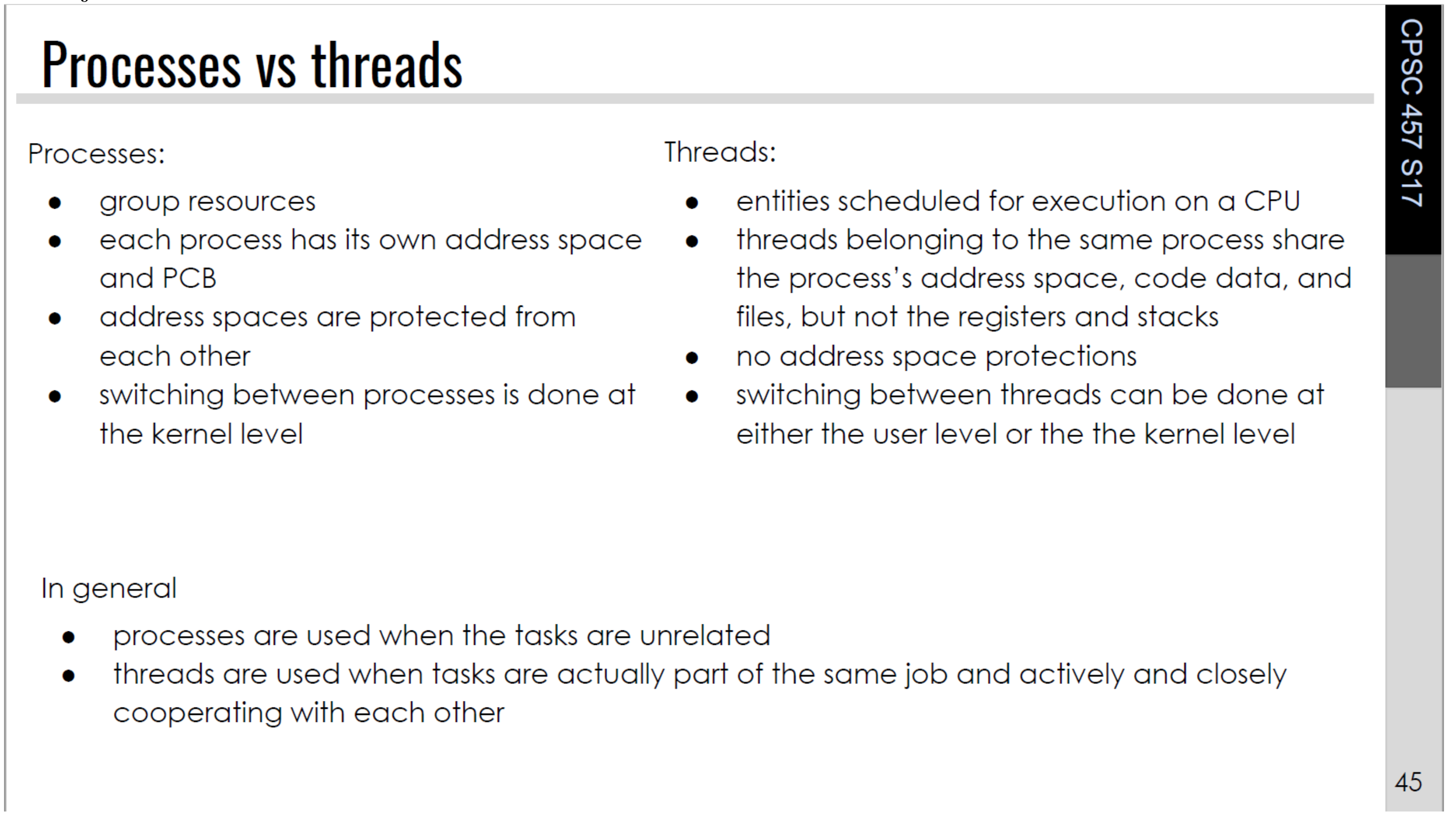
- OS saves state of the old process in PCB
- OS loads saved state of the next process from PCB

4.8 Thread

A thread is a process within a process. It is also a unit of execution of a process. A process is a program in execution and a thread is a unit of execution of a process. Threads have individual registers, stack, PC, and state.

4.9 Process vs Thread

Threads are mostly about improving performance. Processes are expensive while threads are cheap. Processes are typically independent while threads exist as "subsets" of a process. Threads belong to the same process and share many resources such as address space and open files. Processes only interact through OS mechanisms while threads are more flexible. Processes have protected address spaces while threads do not. Processes are used when the tasks are unrelated while threads are used when tasks are part of the same job.



Processes vs threads

Processes:

- group resources
- each process has its own address space and PCB
- address spaces are protected from each other
- switching between processes is done at the kernel level

Threads:

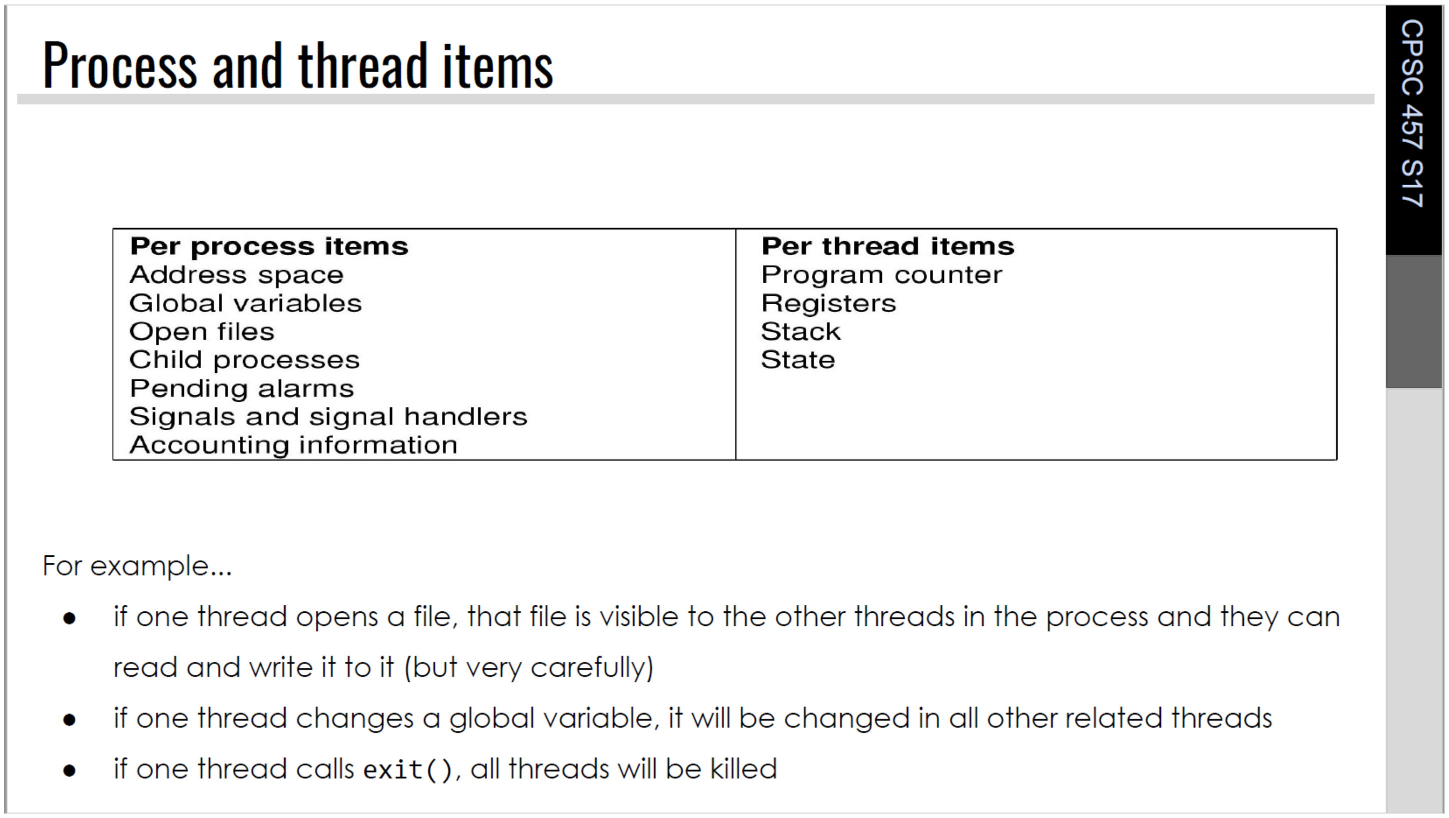
- entities scheduled for execution on a CPU
- threads belonging to the same process share the process's address space, code data, and files, but not the registers and stacks
- no address space protections
- switching between threads can be done at either the user level or the the kernel level

In general

- processes are used when the tasks are unrelated
- threads are used when tasks are actually part of the same job and actively and closely cooperating with each other

CPSC 457 S17

45



Process and thread items

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

For example...

- if one thread opens a file, that file is visible to the other threads in the process and they can read and write it to it (but very carefully)
- if one thread changes a global variable, it will be changed in all other related threads
- if one thread calls `exit()`, all threads will be killed

CPSC 457 S17

4.10 Why Threads

- Multiple activities within an application
- Lighter weight than processes
- Useful for parallel computing

4.11 Benefits of Threads

- Responsiveness: multiple actions at once
- Resource Sharing: share data
- Economy: more economical to create and switch
- Scalability: more tasks can be scheduled
- Ease of Use: alternatives are much more complicated

4.12 Why Threads are Lightweight than Processes

Process creation is ”expensive”, because it has to set up a complete new virtual memory space for the process with it’s own address space. ”expensive” means takes a lot of CPU time. Threads don’t need to do this, just change a few pointers around, so it’s much ”cheaper” than creating a process. The reason threads don’t need this is because they run in the address space, and virtual memory of the parent process.

Every process must have at least one thread. So if you think about it, creating a process means creating the process AND creating a thread. Obviously, creating only a thread will take less time and work by the computer.

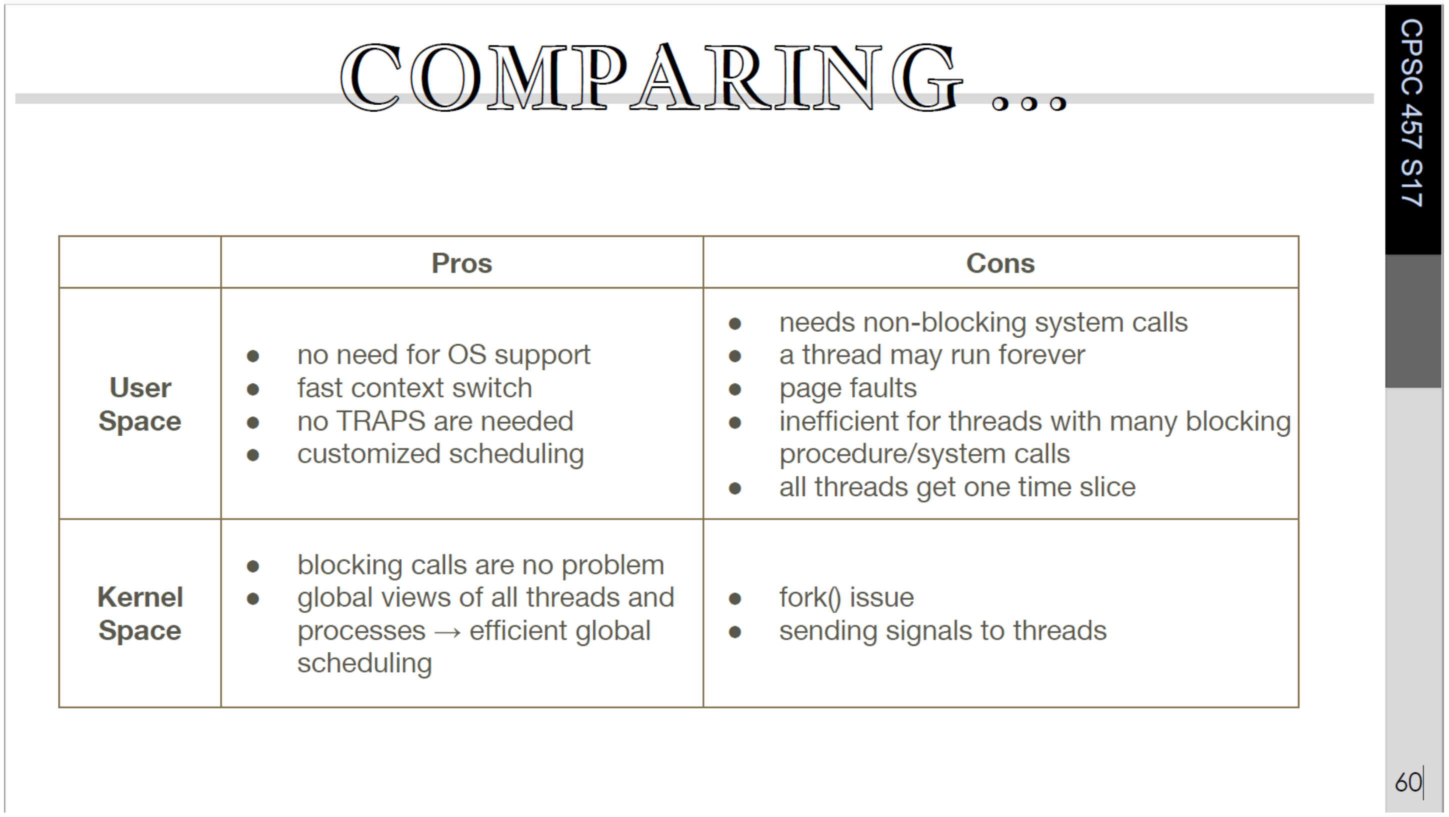
In addition, threads are ”lightweight” because threads can interact without the need of inter-process communication. Switching between threads is ”cheaper” than switching between processes (again, just moving some pointers around). And inter-process communication requires more expensive communication than threads.

4.13 User-Level Threads

Thread implementation is entirely in user space and requires no support from the OS. Each process has its own thread table and scheduler. No need to trap into kernel when switching treads so efficient.

4.14 Kernel-Level Threads

One master thread table at kernel level. Better for processes doing a lot of blocking I/O. Less efficient since thread operations need to trap into kernel.



COMPARING ...

CPSC 457 S17

	Pros	Cons
User Space	● no need for OS support ● fast context switch ● no TRAPS are needed ● customized scheduling	● needs non-blocking system calls ● a thread may run forever ● page faults ● inefficient for threads with many blocking procedure/system calls ● all threads get one time slice
Kernel Space	● blocking calls are no problem ● global views of all threads and processes → efficient global scheduling	● fork() issue ● sending signals to threads

60

4.15 Thread Models

- N:1 (many to one or user-level threads) many user-level threads to one kernel thread
- 1:1 (one to one or kernel-level threads) one user-level thread to one kernel thread
- M:N (many to many or hybrid threads) combination of the two above

Chapter 5

Slide Set 5: Concurrent Programming (Thread Pools, Critical Sections, Mutexes, Dining Philosophers)

5.1 Thread Cancellation

Asynchronous Cancellation

- one thread immediately terminates the target thread
- killed thread has no chance to clean up which may leave data in an undefined state

Deferred/Synchronous Cancellation

- the target thread periodically checks whether it should terminate
- check only at cancellation points at which it can be canceled safely
- less performance and may run for a while after cancellation request

5.2 Unix Signals

- a form of IPC (inter-process communication)
- asynchronous
- a signal is used to notify a process/thread that a particular event has occurred
- a signal must be generated/sent, delivered, and then handled via signal handler

5.3 Reentrant Functions

Reentrant functions are functions

- that can be interrupted in the middle of an operation
- and then called again (re-entered)
- and finally the original function call can finish executing

Used in interrupt handlers, signal handlers, multi-threaded applications. Don't use global variables and don't call other non-reentrant functions. Similar to a leaf function in Assembly.

5.4 Thread Pools

The program creates and maintains a set/pool of worker threads. When a program needs a thread, it takes one out of the pool, when it is done, the thread is back to the pool (thread recycling). Thread queues are usually combined with a task queue. Instead of asking for a thread, a task is inserted into a task queue and available threads take up tasks from the task queue.

5.5 Race Conditions

A race conditions is a behavior where the output is dependent on the sequence or timing of other uncontrollable events. We want to avoid race conditions:

- Prevent more than one process/thread from accessing a shared resource at any given time
- Identify critical sections in the code
- Enforce mutual exclusion

5.6 Critical Sections and Mutual Exclusion

A critical section is the part of the program that accesses the shared resource in a way that could lead to races. Mutual exclusion is when no two processes/threads will ever be in their critical sections at the same time. Requirements to avoid race conditions:

- No two processes may be inside their critical regions at the same time
- No assumptions about speed or the number of CPUs
- No process running outside it's CS may block other processes
- No process should have to wait forever to enter its CS

Definitions

- **Critical Section**: part of the program where a shared resource is accessed
- **Mutual Exclusion**: ensuring only one process can access a resource at a time
- **Mutex/Lock**: a mechanism to achieve mutual exclusion
- **Deadlock**: a state where each process/thread is waiting on another process/thread to release a lock
- **Livelock**: states of the processes change but none are progressing
- **Starvation**: one process does not get to run at all
- **Fairness**: stronger requirement than starvation-free, all processes get equal opportunity to progress
- **Random Timeout**: a mechanism for preventing deadlocks by randomly timing out

5.7 Mutex (aka Lock)

A mutex is a synchronization mechanism for ensuring exclusive access to a resource in concurrent programs. It has two states: locked and unlocked. Only the process/thread that locks the mutex can unlock it.

Chapter 6

Slide Set 6: Concurrent Programming (More Synchronization Mechanisms)

6.1 Condition Variables

Another synchronization primitive that is useful for implementing critical sections containing loops waiting for a condition. It is a variable that waits on a condition to be satisfied before unlocking/locking the thread. It is useful for implementing CS containing loops waiting for a condition.

6.2 Semaphore

Another synchronization primitive that is a special integer variable used for signaling among processes. The value indicates number of available units of some resource. Supports three operations:

- initialization
- increment
- decrement

6.2.1 Binary Semaphore

A special type of semaphore with value of either 0 or 1. When the binary semaphore is locked by a thread, it can be unlocked by any thread as opposed to mutex, where locking/unlocking must be done by the same thread. Each semaphore maintains a queue of processes blocked on the semaphore.

6.2.2 Counting Semaphore

A general semaphore where the value is not limited to 0 and 1 like the binary semaphore. If:

- S > 0: value of S is the number of processes/threads that can issue a wait and immediately continue to execute
- S $= 0$: all resources are busy, the calling process/thread must wait
- S < 0: represents the number of processes that are waiting to be unblocked

A common problem with semaphores is that if a thread/process requests more resources than available, the program can get into a deadlock, race condition, or other forms of unpredictable and irreproducible behaviour.

6.3 Monitors

A monitor is a programming language construct that controls access to shared data. A monitor is also a module that encapsulates

- Shared data structures
- Procedures that operate on the shared data structure
- Synchronization between concurrent procedure invocations

Data in monitors can only be accessed via the published procedures. A monitor is a higher-level construct compared to mutexes and semaphores. Only one thread can execute any monitor procedure at any time. Think of all bodies of all procedures being critical sections, protected by one mutex. Monitors with condition variables are only accessible from within the module.

6.4 Monitors vs Semaphores vs Mutex

Once a monitor is correctly programmed, access to the protected resource is correct for accessing from all processes. With semaphores or mutexes, resource access is correct only if all of the processes that access the resource are programmed correctly.

6.5 Spinlocks

Another synchronization mechanism. Locks are implemented using busy waiting loops, a lightweight alternative to mutex. Often implemented in Assembly using atomic operations.
Atomic Operation: an operation that appears to execute instantaneously

6.6 Compare-and-Swap(CAS)

Another synchronization mechanism. The general algorithm

- compare contents of memory to value 1
- if they are the same, change the memory to value 2
- return the old contents of memory

6.7 More Synchronization Mechanisms

Event Flags: a memory word where different event may be associated with each bit in a flag
Message Passing: processes send each other messages that can contain arbitrary data

6.8 Priority Inversion

The situation where a processes may have higher priority but has to wait for lower priority processes to finish, effectively making the priority inverted. The solution is to use priority-inheritance where if a higher priority item wants to use a resource, it gets to immediately.

Chapter 7

Slide Set 7: Other Synchronization Mechanisms & CPU Scheduling

7.1 Good Race-Free Solution

Recall that a good solution satisfies the following four requirements:

- **Mutual Exclusion**: No two processes/threads may be simultaneously inside their critical sections (CS).
- **Progress**: No process/threads running outside its CS may block other processes/threads.
- **Bounded Waiting**: No process/thread should have to wait forever to enter its CS.
- **Speed**: No assumptions may be made about the speed or the number of CPUs.

7.2 Achieving Mutual Exclusion

7.2.1 Disabling Interrupts

The idea is for each process to disable all interrupts just before entering its CS and re-enabling them just before leaving the CS. Once a process has disabled interrupts, it can examine and update the shared memory without interventions from other processes.

The problem is that a process may never re-enable the interrupt or that on multi-CPU system, disabling interrupts only affects one CPU.

7.2.2 Lock Variables

The idea is a single, shared (lock) variable, initialized to 0. If lock == 0 then no process is in CS. If lock == 1 then a process is in CS. A process can only enter its CS if lock == 0. Otherwise, it must wait.

The problem is that there is no mutual exclusion. The solution is to make entering the CS an atomic operation.

7.2.3 Strict Alternation

The idea is that two processes alternate between entering their CS using a global variable turn.

The problem is that:

- Busy Waiting (process is busy checking the condition while waiting for the condition to change)
- Only works for 2 processes
- No progress (one process is blocked by another process not in its CS)

7.2.4 Peterson's Algorithm

The idea is that there is a shared integer turn and shared array flag[2] indicating whose turn it is and who is interested in entering CS. It satisfies mutual exclusion, progress, and bounded waiting.

The problem is that it may fail on some CPUs with out-of-order execution or memory reordering.

7.2.5 Synchronization Hardware

Race conditions are prevented by ensuring that critical sections are protected by locks.

- A process must acquire a lock before entering a CS.
- A process releases the lock when it exits the CS.

Many modern computer systems provide special hardware instructions that implement useful atomic operations that can be used to create atomic locking/unlocking mechanisms.
Advantages

- Avoids system calls
- Can be more efficient if expected wait time is short
- Only makes sense on multi-CPU/core systems

Disadvantages

- Busy-waiting (spinlocks)
- Extra coding

7.2.6 Compare-and-Swap (CAS)

General Algorithm

- Compare contents of memory to value1
- If they are the same, change the memory to value2
- Return the old contents of memory

Pseudo-code:

```c
int case(int* mem, int val1, int val2)
{
    int old = *mem;
    if (old == val1)
    {
        *mem = val2;
    }
    return old;
}
```

7.2.7 Test-and-Set

General Algorithm

- Remember contents of memory (atomic)
- Set memory to true (atomic)
- Return the old contents of memory

Pseudo-code:

```c
int testAndSet(int* mem)
{
    int old = *mem;
    *mem = TRUE;
    return old;
}
```

7.2.8 Swap

General Algorithm

- Atomically swap contents of two memory locations

Pseudo-code:

```c
void swap(int* a, int* b)
{
    int tmp = *a;
    *a = *b;
    *b = temp;
}
```

7.2.9 Bounded Waiting with Synchronization Hardware

- When used correctly, the atomic operations, such as compare-and-swap, test-and-set, swap can be used to achieve mutual exclusion, progress, and speed
- But too low level to achieve bounded waiting
- Bounded waiting can be 'added' via two shared variables such as a lock

7.3 CPU Scheduling

Recall multi-programming, the main objective is:

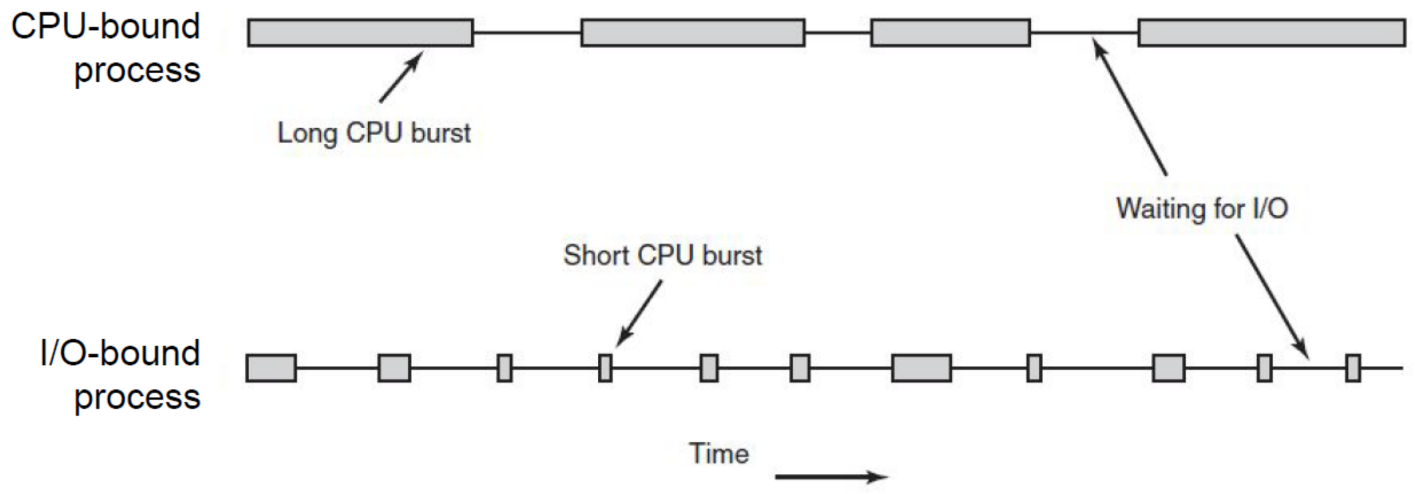
- Maximize CPU utilization by having a process running at all times
- Several processes kept in memory
- A process runs until it must wait

The software the decides which process runs next is called a *scheduler* that is usually part of a kernel.

7.4 Process Behavior

Most processes alternate between bursts of CPU activity and bursts of I/O activity. As CPUs get faster, processes tend to get more I/O-bound. It takes quite a few I/O-bound processes to keep the CPU fully occupied.

- CPU-Bound: long CPU bursts and infrequent I/O waits
- I/O-Bound: short CPU bursts and frequent I/O waits



7.5 When to Schedule

Variety of situations when scheduling is needed:

- Process creation
- Process termination
- Blocking system call (I/O or mutex)
- I/O interrupt
- Periodic clock interrupt (time slice)

Scheduling Algorithms

- **Non-preemptive**: process runs until it does I/O, exits or voluntarily yields CPU

- **Preemptive**: processes are swapped at regular intervals (time slice) or kicked out when a condition is satisfied

Note: On systems without clock interrupt, only non-preemptive scheduling is possible.

7.6 Categories of Scheduling Algorithms

Batch

- Usually on mainframes
- HPC systems
- No interactivity needed
- No preemption needed

Interactive

- General systems: running many tasks, many of them must remain interactive

Real Time

- Applications are guaranteed CPU cycles
- Often tied closely to some hardware (planes, cars, gaming)

7.7 Scheduling Metrics

Individual Statistics

- **Arrival Time**: the time a process arrives
- **Start Time**: the time a process first gets to run on the CPU
- **Finish Time**: when the process is done
- **Response Time**: how long before you get first feedback (often response = start - arrival)
- **Turnaround Time**: time taken to fulfill a request (turnaround = finish - arrival)
- **CPU Time**: how much time the process spent on CPU
- **Waiting Time**: total time spent in waiting queue (waiting = turnaround - CPU - I/O)

Overall Statistics

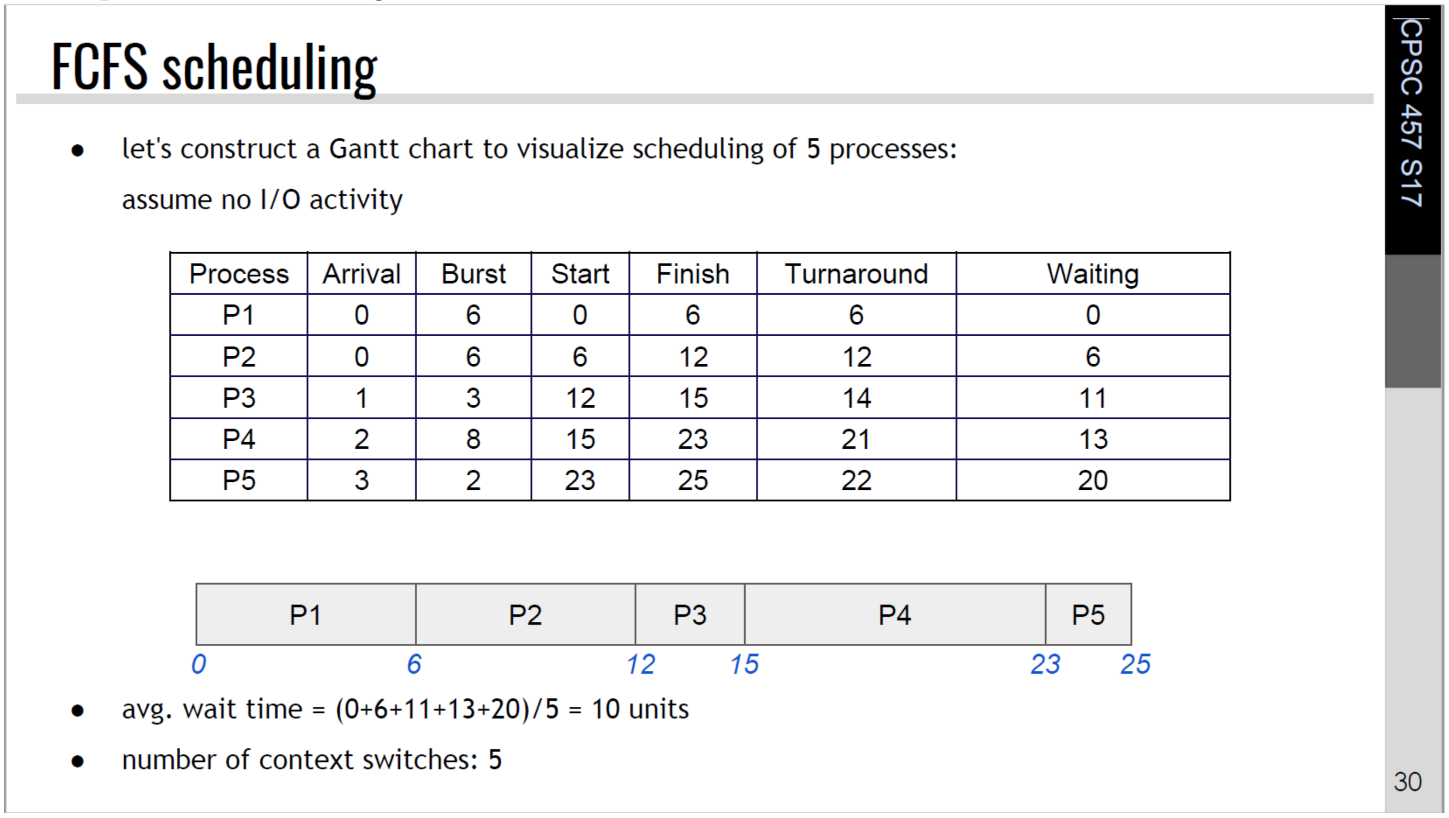
- **Average turnaround time**: average of the turnaround time
- **Average wait time**: average of the wait time
- **Throughput**: number of jobs finished per unit of time

7.8 Scheduling Algorithms

7.8.1 First Come First Serve (FCFS)

- Non-preemptive
- CPU is assigned in the order the processes request it using a FIFO ready queue
- New jobs are appended to the ready queue
- A running job keeps the CPU until it is either finished or it blocks
- When a running process blocks, next process from the ready queue starts to execute
- When a process is unblocked, it is appended at the end of the ready queue
- The minimum number of context switches is only N switches for N processes

Disadvantage of FCFS is the *convoy effect* where a CPU-bound process will tie up the CPU, making I/O-bound processes wait a long time to execute.



FCFS scheduling

- let's construct a Gantt chart to visualize scheduling of 5 processes: assume no I/O activity

Process	Arrival	Burst	Start	Finish	Turnaround	Waiting
P1	0	6	0	6	6	0
P2	0	6	6	12	12	6
P3	1	3	12	15	14	11
P4	2	8	15	23	21	13
P5	3	2	23	25	22	20

- avg. wait time = (0+6+11+13+20)/5 = 10 units
- number of context switches: 5

CPSC 457 S17

30

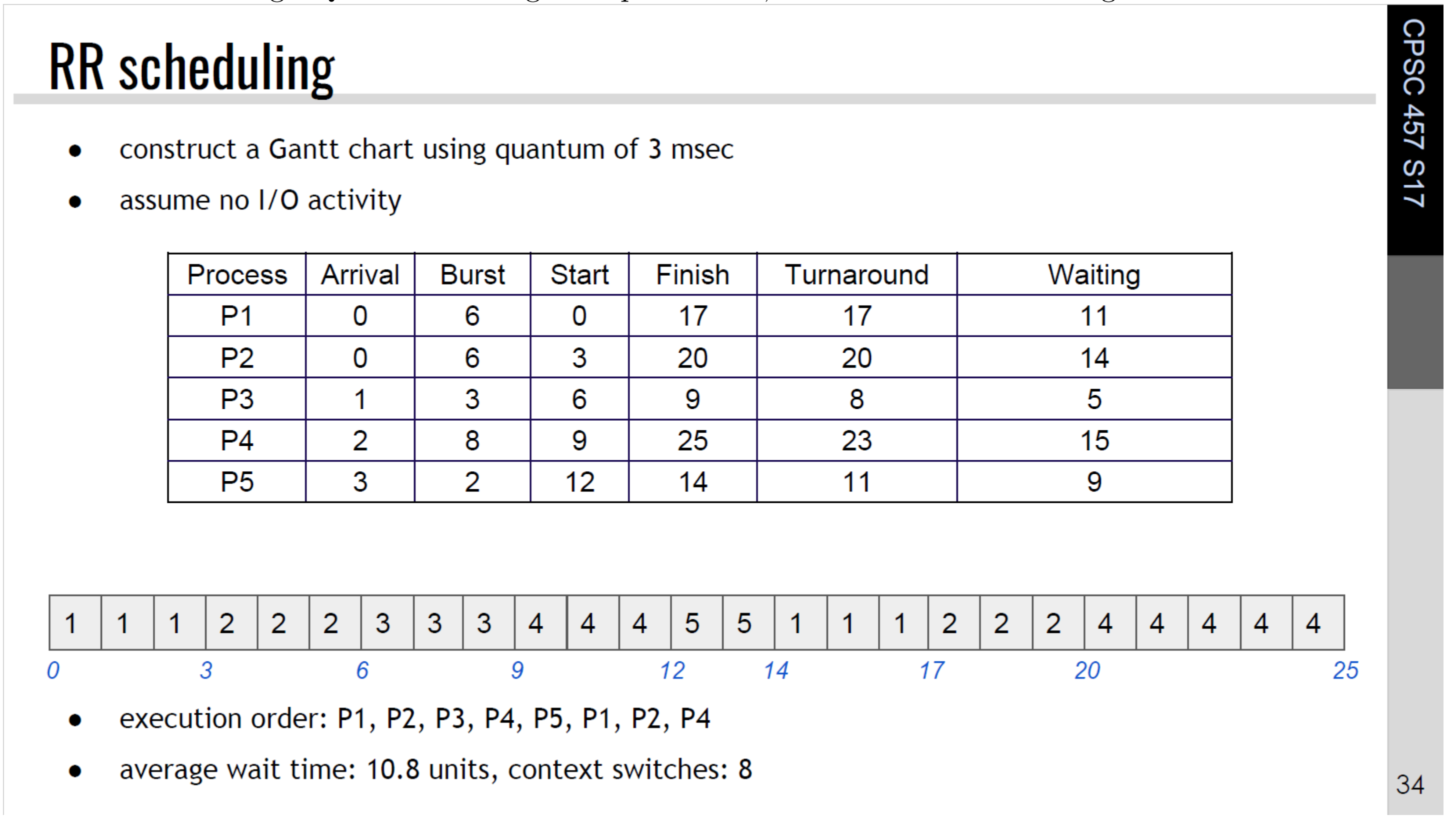
7.8.2 Round Robin (RR)

- Preemptive version of FCFS
- Each process is assigned a time interval called a time slice/quantum
- If the process exceeds the quantum, the process is preempted (context switched), and the CPU is given to the next process in ready queue
- Preempted process goes at the back of the ready queue

The performance of RR depends significantly on the size of the time quantum (Q) and the time required for a context switch (S).

- Very small Q implies heavy overhead but a highly responsive system.
- Very large Q implies minimum overhead but a non-responsive system.

Conclusion: Although Q should be large compared to S, it should not be too large.



RR scheduling

- construct a Gantt chart using quantum of 3 msec
- assume no I/O activity

Process	Arrival	Burst	Start	Finish	Turnaround	Waiting
P1	0	6	0	17	17	11
P2	0	6	3	20	20	14
P3	1	3	6	9	8	5
P4	2	8	9	25	23	15
P5	3	2	12	14	11	9

1	1	1	2	2	2	3	3	3	4	4	4	5	5	1	1	1	2	2	2	4	4	4	4	4

0 3 6 9 12 14 17 20 25

- execution order: P1, P2, P3, P4, P5, P1, P2, P4
- average wait time: 10.8 units, context switches: 8

CPSC 457 S17

34

7.8.3 Shortest Job First (SJF)

- Non-preemptive
- Applicable to batch systems where job length is known in advance
- When the CPU is available it is assigned to the shortest job
- Ties are resolved using FCFS
- SJF is similar to FCFS but the ready queue is sorted with the shortest job
- Sorting can be either basic (execution time can be static) or advance (dynamically compute based on the history of CPU bursts)

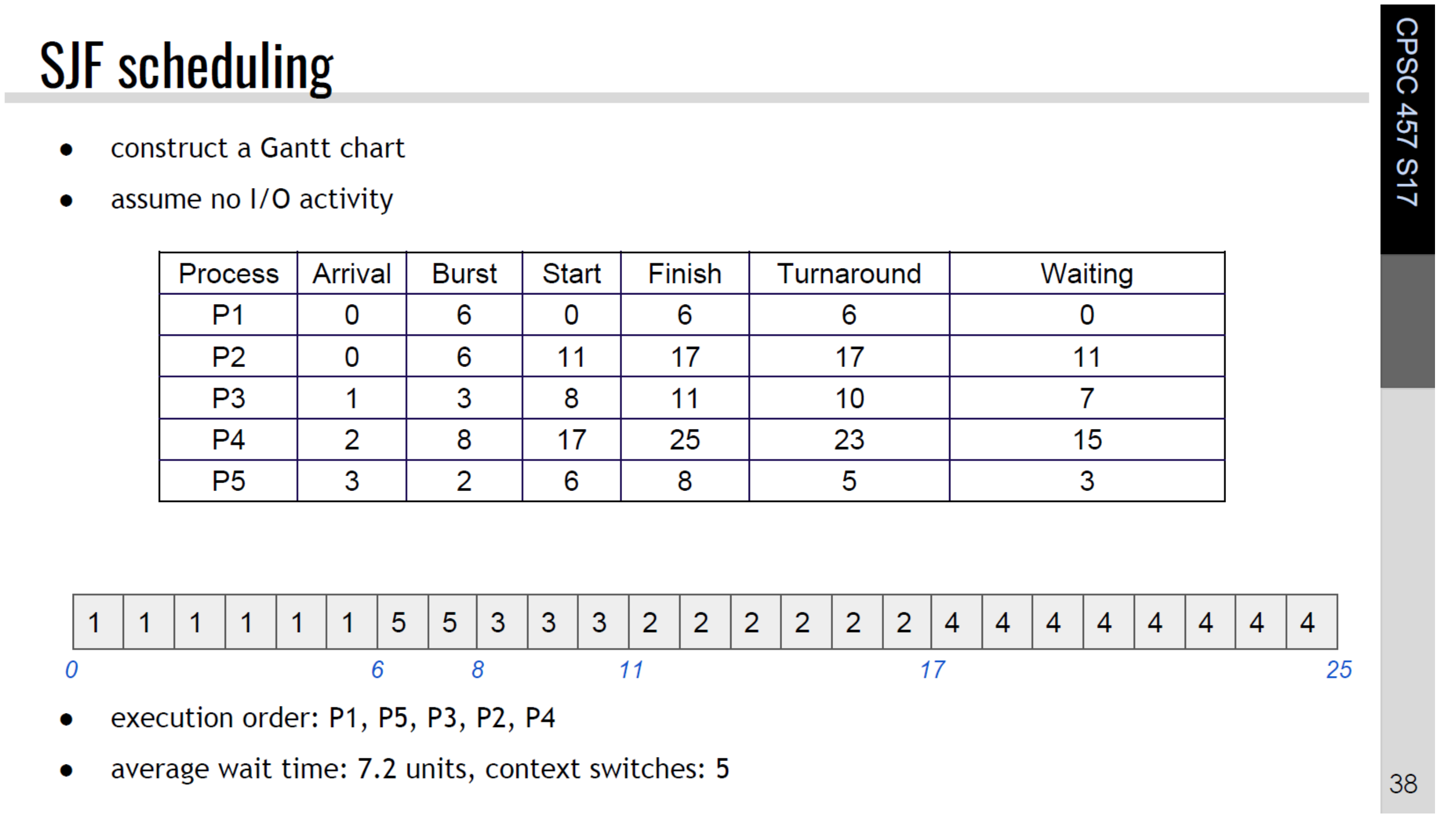
Advantages

- Minimum number of context switches
- Optimal turnaround time if all jobs arrive simultaneously

Disadvantages

- Requires advance knowledge of how long a job will execute

- Has potential for job starvation (solved by aging)
- Aging: priority = total wait time / estimated run time



SJF scheduling

- construct a Gantt chart
- assume no I/O activity

Process	Arrival	Burst	Start	Finish	Turnaround	Waiting
P1	0	6	0	6	6	0
P2	0	6	11	17	17	11
P3	1	3	8	11	10	7
P4	2	8	17	25	23	15
P5	3	2	6	8	5	3

1	1	1	1	1	1	5	5	3	3	3	2	2	2	2	2	2	4	4	4	4	4	4	4	4

0 *6* *8* *11* *17* *25*

- execution order: P1, P5, P3, P2, P4
- average wait time: 7.2 units, context switches: 5

CPSC 457 S17

38

7.8.4 Shortest Remaining Time Next (SRTN)

- Preemptive version of SJF
- The next job is picked based on the remaining time
- Remaining time = total time - time already spent on CPU
- SRTN is similar to RR
- The ready queue is sorted by the remaining time
- Context switches can happen as a result of adding a job

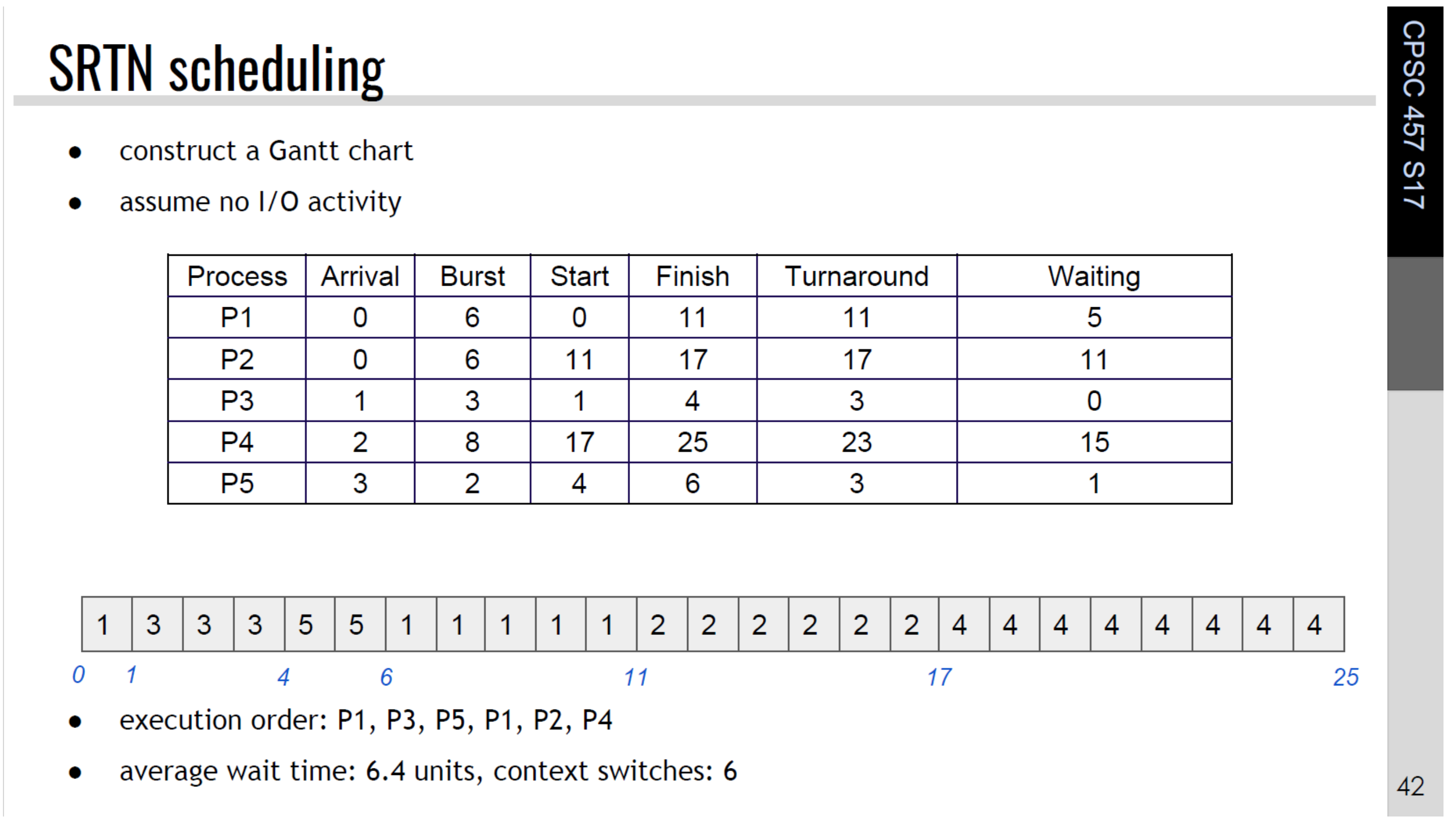
Advantages

- Similar to SJF
- Optimal turnaround time even if jobs don't arrive at the same time
- Slightly more context switches

Disadvantages

- Requires advance knowledge of how long a job will execute
- Has potential for job starvation (solved by aging)

- Needs to consider the cost of a context switch



SRTN scheduling

- construct a Gantt chart
- assume no I/O activity

Process	Arrival	Burst	Start	Finish	Turnaround	Waiting
P1	0	6	0	11	11	5
P2	0	6	11	17	17	11
P3	1	3	1	4	3	0
P4	2	8	17	25	23	15
P5	3	2	4	6	3	1

1	3	3	3	5	5	1	1	1	1	1	2	2	2	2	2	2	4	4	4	4	4	4	4	4

0 1 4 6 11 17 25

- execution order: P1, P3, P5, P1, P2, P4
- average wait time: 6.4 units, context switches: 6

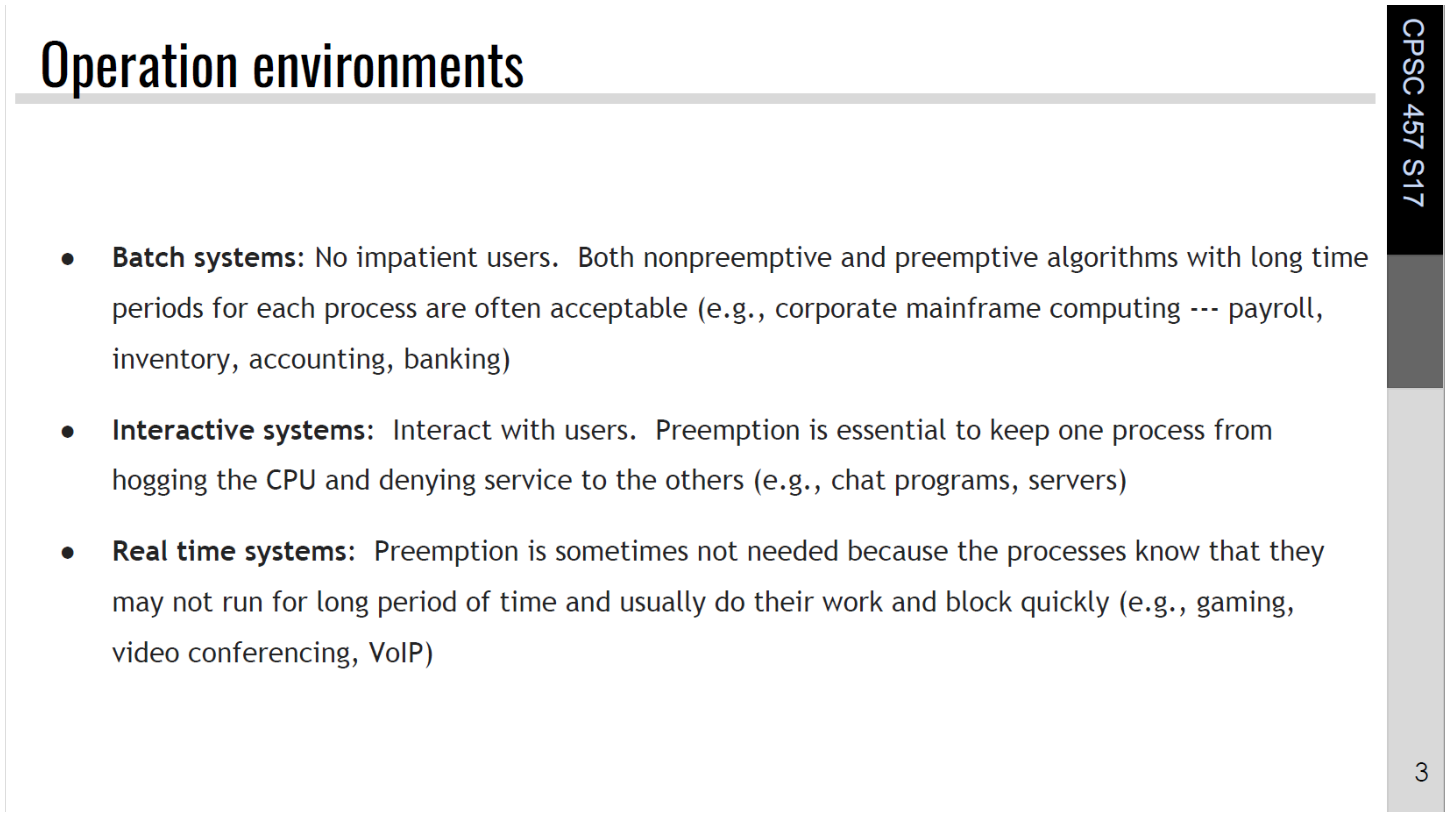
CPSC 457 S17

42

Chapter 8

Slide Set 8: CPU Scheduling

8.1 Operation Environments



Operation environments

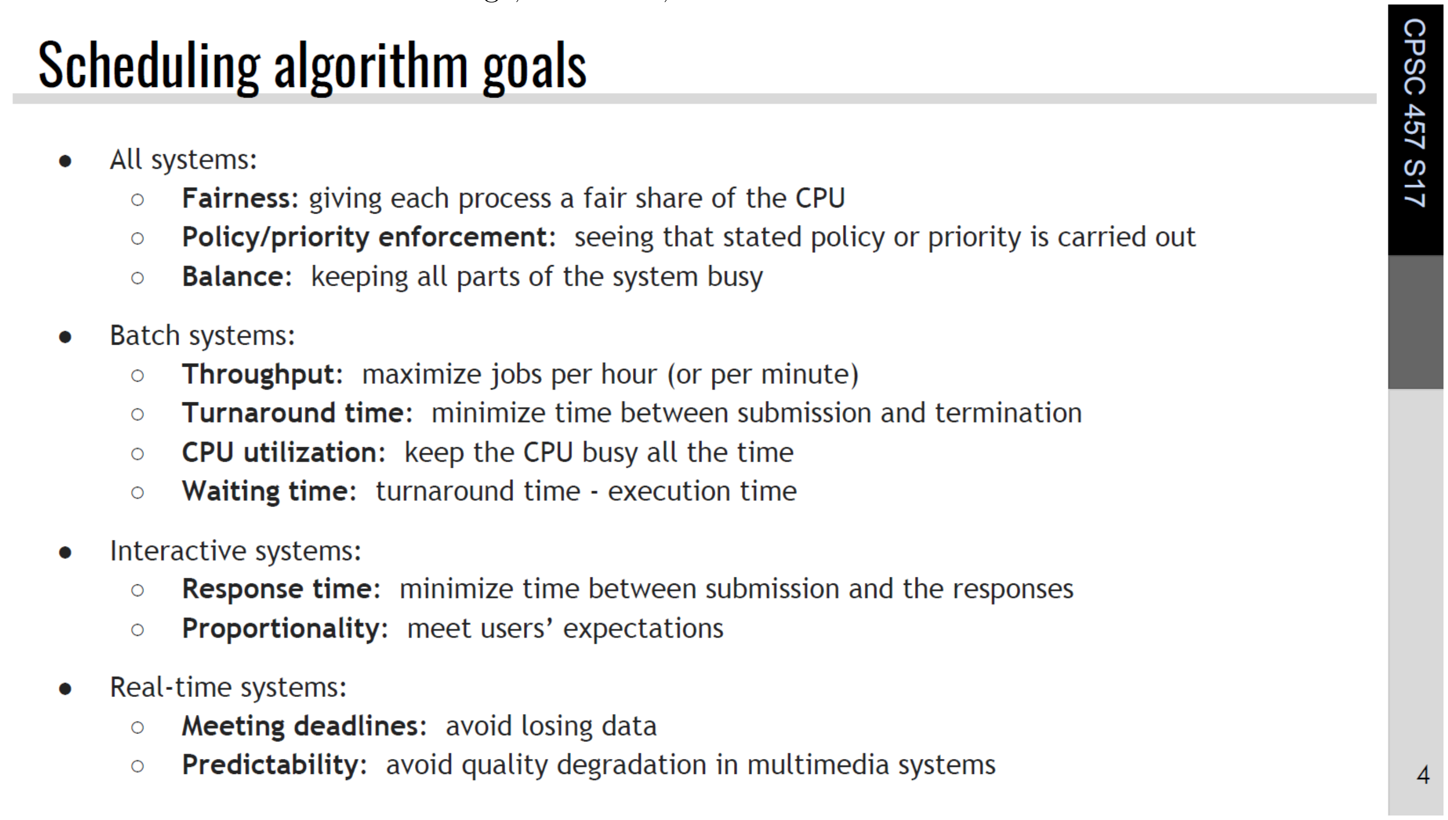
- **Batch systems:** No impatient users. Both nonpreemptive and preemptive algorithms with long time periods for each process are often acceptable (e.g., corporate mainframe computing --- payroll, inventory, accounting, banking)
- **Interactive systems:** Interact with users. Preemption is essential to keep one process from hogging the CPU and denying service to the others (e.g., chat programs, servers)
- **Real time systems:** Preemption is sometimes not needed because the processes know that they may not run for long period of time and usually do their work and block quickly (e.g., gaming, video conferencing, VoIP)

CPSC 457 S17

3

8.2 Scheduling Algorithm Goals

These statistics can reference average, minimum, or maximum.



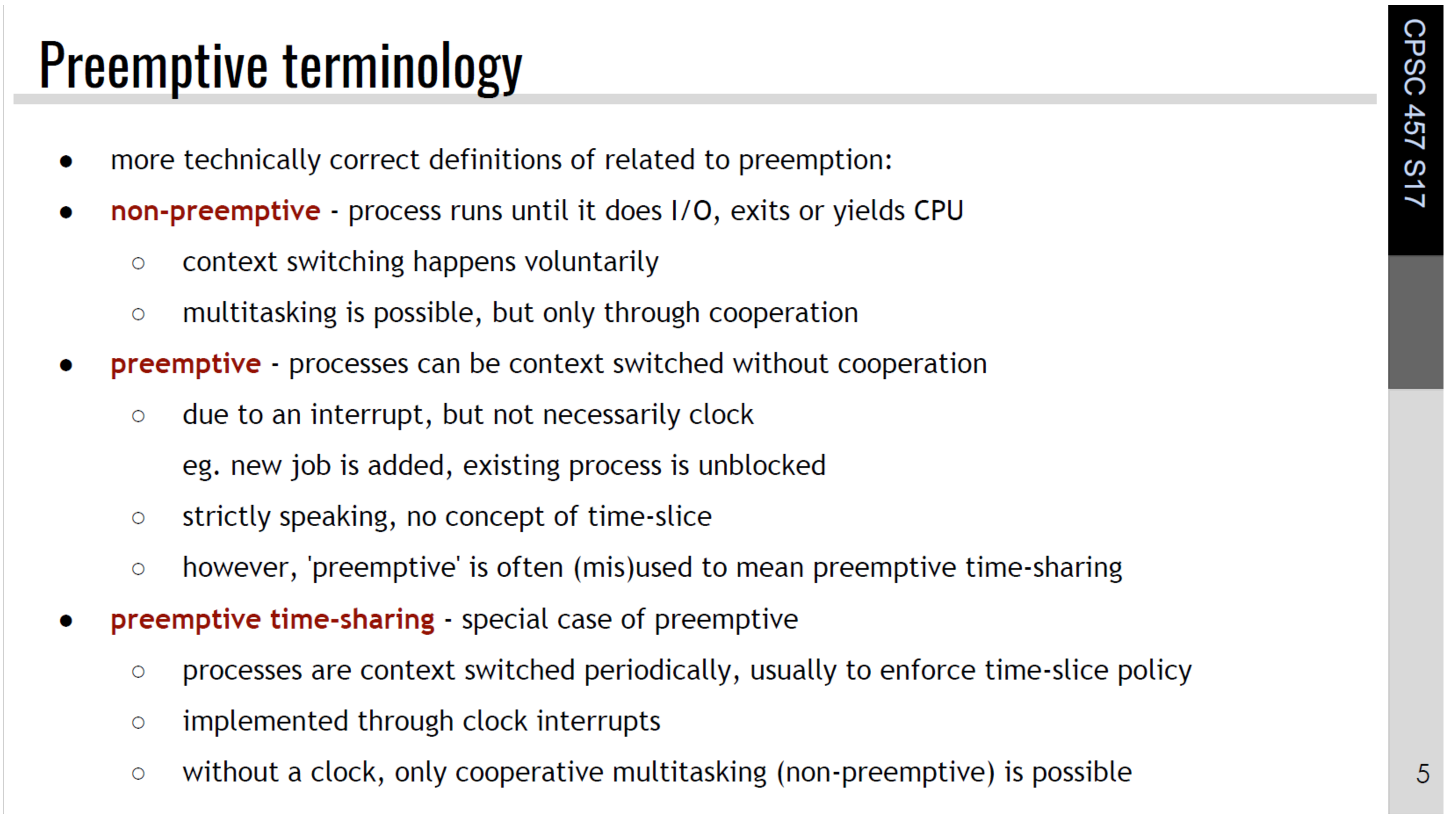
Scheduling algorithm goals

- All systems:
  - **Fairness:** giving each process a fair share of the CPU
  - **Policy/priority enforcement:** seeing that stated policy or priority is carried out
  - **Balance:** keeping all parts of the system busy
- Batch systems:
  - **Throughput:** maximize jobs per hour (or per minute)
  - **Turnaround time:** minimize time between submission and termination
  - **CPU utilization:** keep the CPU busy all the time
  - **Waiting time:** turnaround time - execution time
- Interactive systems:
  - **Response time:** minimize time between submission and the responses
  - **Proportionality:** meet users' expectations
- Real-time systems:
  - **Meeting deadlines:** avoid losing data
  - **Predictability:** avoid quality degradation in multimedia systems

CPSC 457 S17

4

8.3 Preemptive Terminology



Preemptive terminology

- more technically correct definitions of related to preemption:
- **non-preemptive** - process runs until it does I/O, exits or yields CPU
  - context switching happens voluntarily
  - multitasking is possible, but only through cooperation
- **preemptive** - processes can be context switched without cooperation
  - due to an interrupt, but not necessarily clock

    eg. new job is added, existing process is unblocked
  - strictly speaking, no concept of time-slice
  - however, 'preemptive' is often (mis)used to mean preemptive time-sharing
- **preemptive time-sharing** - special case of preemptive
  - processes are context switched periodically, usually to enforce time-slice policy
  - implemented through clock interrupts
  - without a clock, only cooperative multitasking (non-preemptive) is possible

CPSC 457 S17

5

8.4 CPU Scheduling on Batch Systems

Already covered in previous slide set. Many of these ideas can be adapted to interactive and even real-time systems.

- FCFS
- RR
- SJF
- SRTN

8.5 CPU Scheduling on Interactive Systems

- RR
- Shortest-process-next
- Fair-share
- Priority
  - Lottery
  - Multilevel-queues
  - Multilevel feedback queues

8.5.1 Shortest-Process-Next

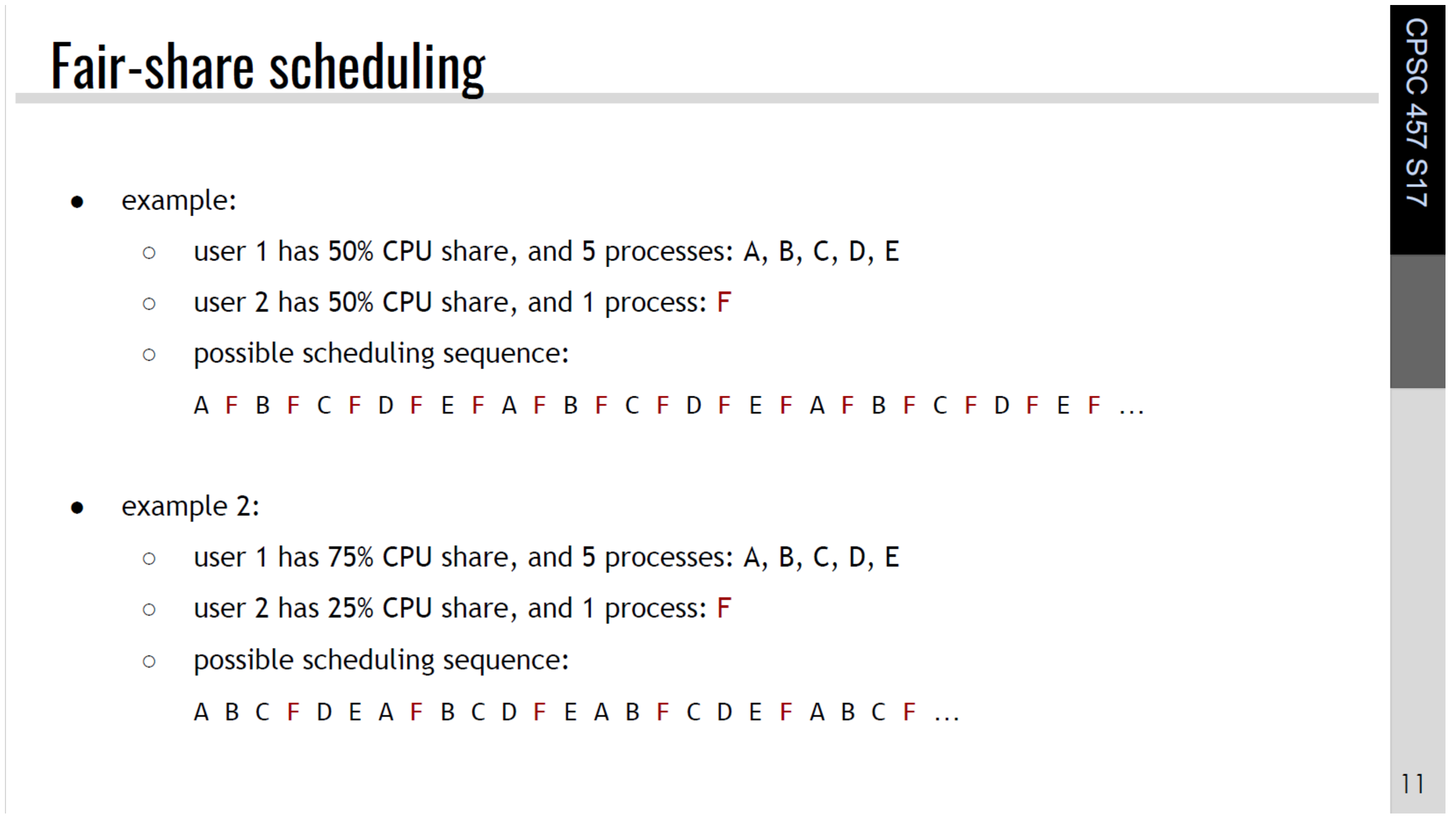
Very similar to SJF scheduling where a ready queue is sorted by a predicted next CPU burst and uses time-sharing preemption. The prediction is done using *exponential averaging.*

$$P = a * B + (1 - a) * P'$$

- P = new prediction of how much a process' burst will be
- P' = previous prediction
- a = smoothing factor (commonly set to 0.5)
- B = burst time

8.5.2 Fair-Share

A scheduling algorithm that takes into account the owners of the processes. It's used to ensure that all users of a system get a fair share of the CPU by allocating the CPU among users/groups instead of processes. All processes belonging to an owner have to share the owner's CPU share.



Fair-share scheduling

- example:
  - user 1 has 50% CPU share, and 5 processes: A, B, C, D, E
  - user 2 has 50% CPU share, and 1 process: F
  - possible scheduling sequence:

    A F B F C F D F E F A F B F C F D F E F A F B F C F D F E F …
- example 2:
  - user 1 has 75% CPU share, and 5 processes: A, B, C, D, E
  - user 2 has 25% CPU share, and 1 process: F
  - possible scheduling sequence:

    A B C F D E A F B C D F E A B F C D E F A B C F …

CPSC 457 S17

11

8.5.3 Priority

A scheduling algorithm that can schedule processes with different priorities. Important processes should get more CPU time than less important processes. Priorities can be either static or dynamic.

Implementing Priority Scheduling

With non-preemptive scheduling

- Consider SJF-like scheduling where ready queue is sorted by priority
- All jobs would eventually finish
- Although adding a new job with a very high priority might have to wait

With preemptive scheduling (more complicated)

- Consider RR-like scheduling where ready queue is sorted by priority
- Problem is starvation where a high priority CPU-bound process would ensure that no other process gets to run

8.5.4 Multilevel Queues

A preemptive time sharing scheduling algorithm. The ready queue is partitioned into separate queues to separate processes based on their priority (foreground vs background) and each queue can have a different scheduling algorithm. A process is permanently assigned to a queue.

Option 1: Static Priority Scheduling

- Each queue has a different but fixed priority
- The scheduler moves from the highest priority queue down
- Once a queue is empty, the scheduler moved to the next queue
- Starvation is a problem

Option 2: Each Queue Gets a Fixed CPU Share

- E.g. Higher priority queue gets 80% CPU time while the lower queues get 20%
- Not a very dynamic solution

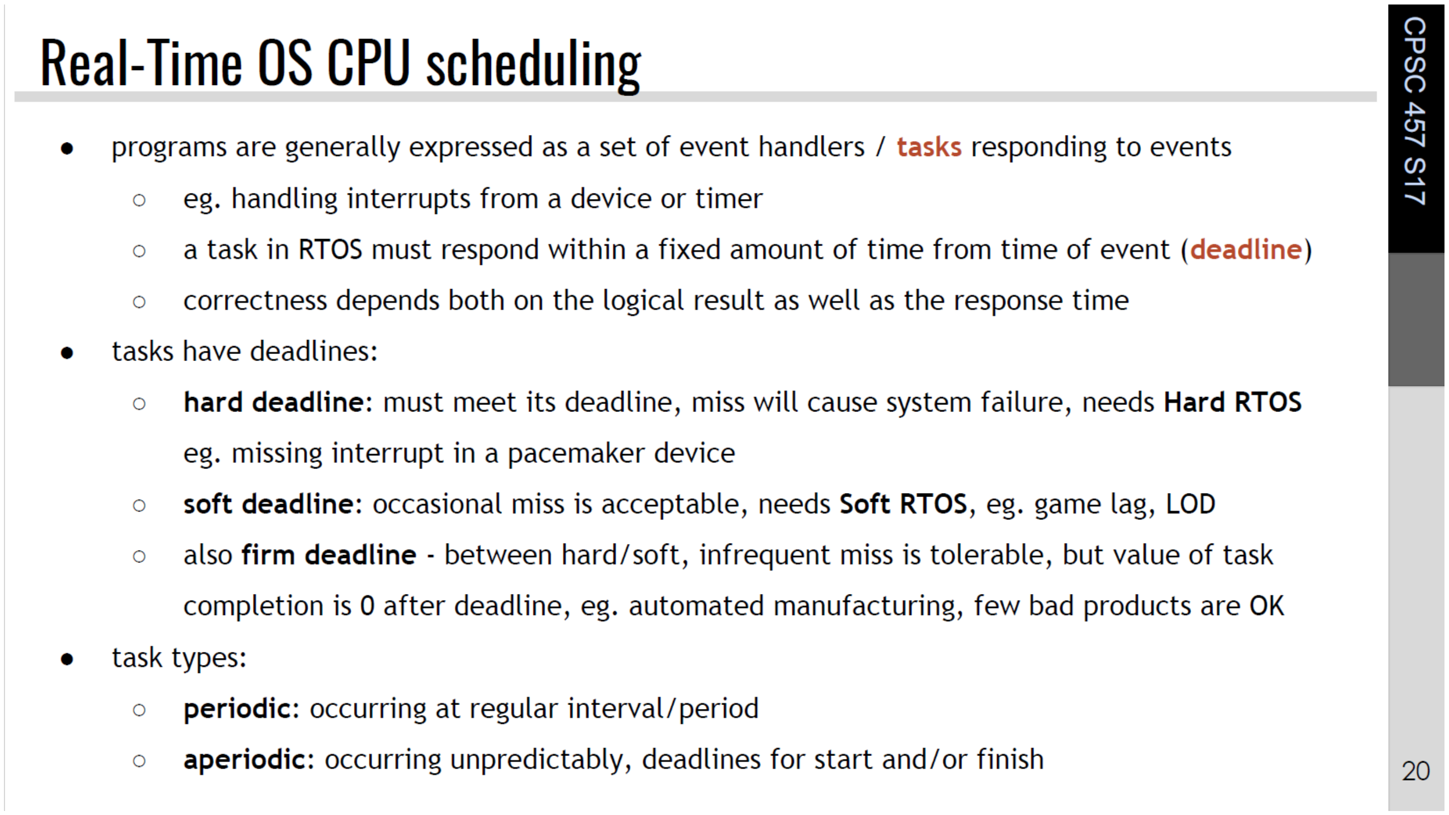
8.5.5 Multilevel Feedback Queues

Similar to multilevel queues but processes can move between queues (move up by aging, down by expiring time slice). This scheduling algorithm solves the starvation problem and dynamic allocation of resources problem. Quantum is the time allocated to each process in a queue. If a process uses up the time slice, then it is moved down to a lower queue. If a process has been waiting for a long time to run in a lower queue, then it is moved up to a higher queue.

8.5.6 Lottery (Probabilistic)

Another preemptive time sharing algorithm. Each process gets some random number of ”lottery tickets” where the number of tickets determine the process’ priority (higher priority $\rightarrow$ more tickets). The scheduler picks a random number and the process with that number wins time slice of CPU (With more tickets, higher probability of being picked). In the long run, a job’s priority will determine the job’s total CPU share. Starvation is not a problem because all processes have a ticket and it is possible to pick it. However, there are some technical problems of implementing this scheduling algorithm for a large number of processes, tickets, jobs. Also, there is overhead and time spent on random selection of a process to run.

8.6 CPU Scheduling on Real-Time Systems



Real-Time OS CPU scheduling

- programs are generally expressed as a set of event handlers / **tasks** responding to events
    - eg. handling interrupts from a device or timer
    - a task in RTOS must respond within a fixed amount of time from time of event (**deadline**)
    - correctness depends both on the logical result as well as the response time
- tasks have deadlines:
    - **hard deadline:** must meet its deadline, miss will cause system failure, needs **Hard RTOS** eg. missing interrupt in a pacemaker device
    - **soft deadline:** occasional miss is acceptable, needs **Soft RTOS**, eg. game lag, LOD
    - also **firm deadline** - between hard/soft, infrequent miss is tolerable, but value of task completion is 0 after deadline, eg. automated manufacturing, few bad products are OK
- task types:
    - **periodic:** occurring at regular interval/period
    - **aperiodic:** occurring unpredictably, deadlines for start and/or finish

CPSC 457 S17

20

8.6.1 Rate-Monotonic (RM)

An algorithm that is suited for periodic tasks. The priority is calculated by inverting the period of the task (shorter period = higher priority, longer period = lower priority). Usually preemptive (higher priority task will preempt lower priority task). A simple formula can be used to determine whether a set of tasks is *schedulable*. The disadvantage of this method is when tasks are aperiodic.

8.6.2 Earliest-Deadline-First (EDF)

Similar to RM except the priority is calculated differently. The priority is calculated by using the deadline instead of the time the task takes. Usually a preemptive scheduler. A simple formula can determine whether tasks can be scheduled without missing deadlines.

8.7 Thread Scheduling

8.7.1 User-Level Threads

- Kernel assigns a quantum to process
- Threads within the process share the quantum
- Each process has its own thread scheduler
- No clock to interrupt a thread that runs too long
- Very fast context switch
- Not possible to allow other threads from other processes to run during a process' thread execution

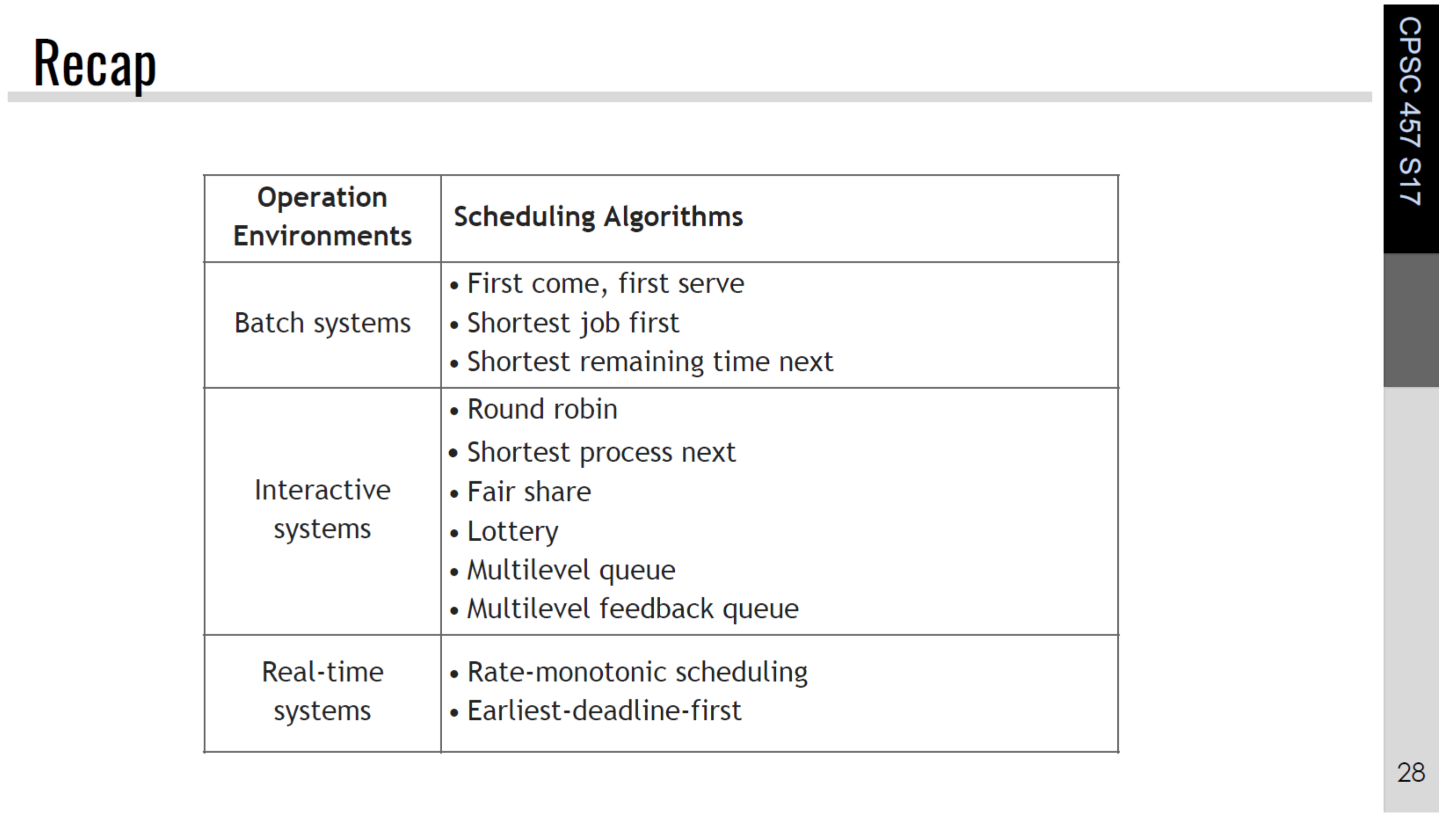
8.7.2 Kernel-Level Threads

- Kernel assigns a quantum to threads
- A full context switch is required
- A thread blocking on I/O will not suspend the entire process
- Threads from other processes may run between current process' threads

8.8 Other Schedulers

- Long-term: Decides which programs to run and which ones to delay
- Medium-term: Decides which programs to swap in/out when running low on resources or when a process is idle for too long
- Short-term: Responsible for allocating CPU to processes in memory
- I/O: Ordering in the I/O queue to increase throughput

8.9 Recap



Recap

Operation Environments	Scheduling Algorithms
Batch systems	• First come, first serve • Shortest job first • Shortest remaining time next
Interactive systems	• Round robin • Shortest process next • Fair share • Lottery • Multilevel queue • Multilevel feedback queue
Real-time systems	• Rate-monotonic scheduling • Earliest-deadline-first

CPSC 457 S17

28

Chapter 9

Slide Set 9: Deadlocks

9.1 Definition

A set of processes is deadlocked if:

- Each process in the set is waiting for an event (resource becoming available, mutex, message arriving)
- That event can be caused only be another process in the set

9.2 System Model

A system consists of processes and resources. There are n processes $(P_1, P_2, ..., P_n)$ and m resource types $(R_1, R_2, ..., R_m)$. The resources could be CPU, memory space, I/O devices. Each resource type, R_i, has W_i instances such as 1 CPU, 5 disks, and 3 printers.
Each process utilizes a resource follows in the same manner:

- **Request** a resource - may block
- **Use** a resource - for a finite amount of time
- **Release** a resource - potentially unblock related process

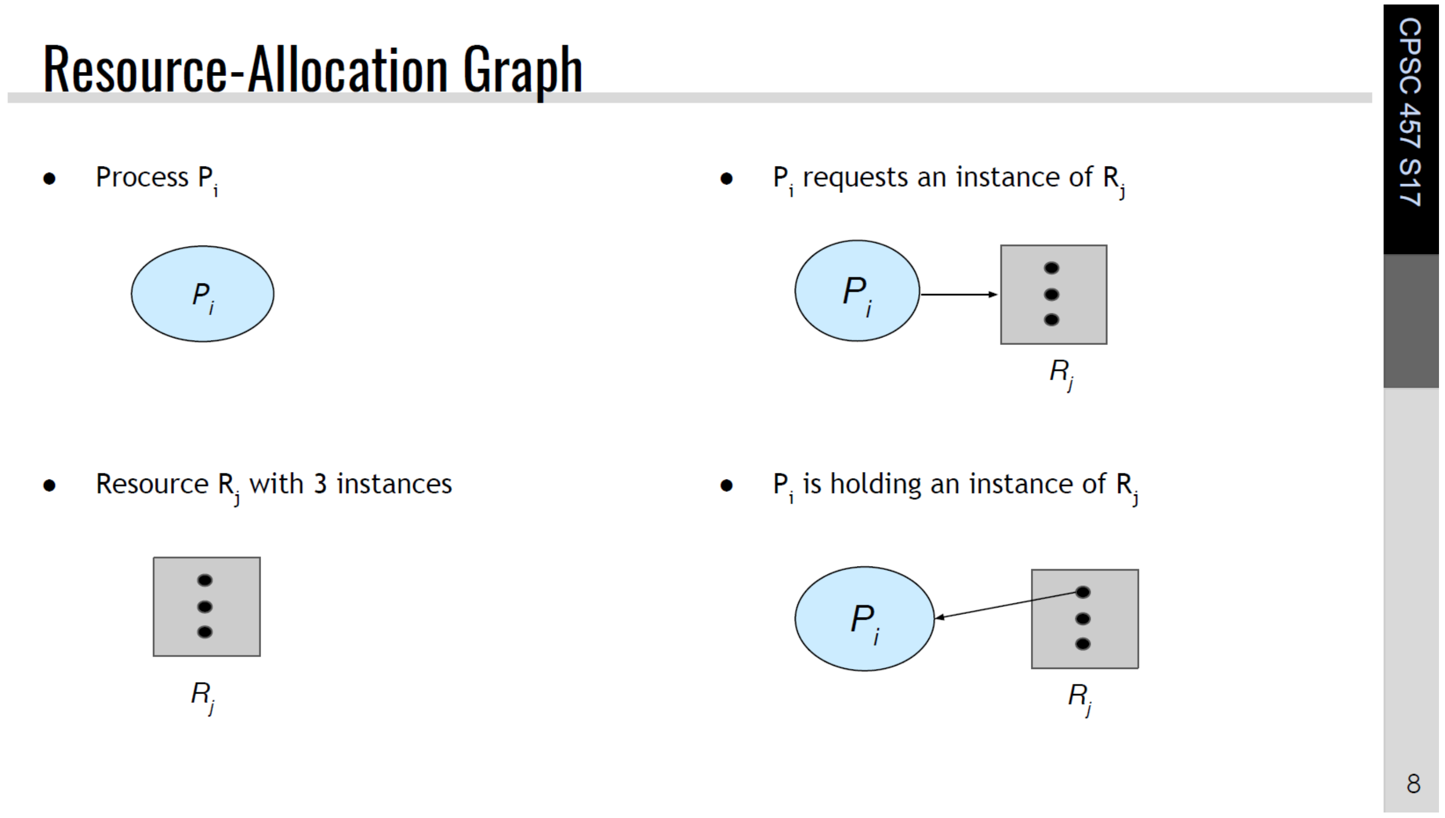
9.3 Necessary Deadlock Conditions

The following four conditions must hold **simultaneously**.

- **Mutual Exclusion**: The involved resources must be non-shareable
- **Hold and Wait**: A process holding at least one resource is waiting to acquire additional resources
- **No Preemption**: A resource can be released only by the process holding it (voluntary)
- **Circular Wait**: There is an ordering of processes such that there is a cycle (P1 waits for P2, P2 waits for P3, P3 waits for P1)

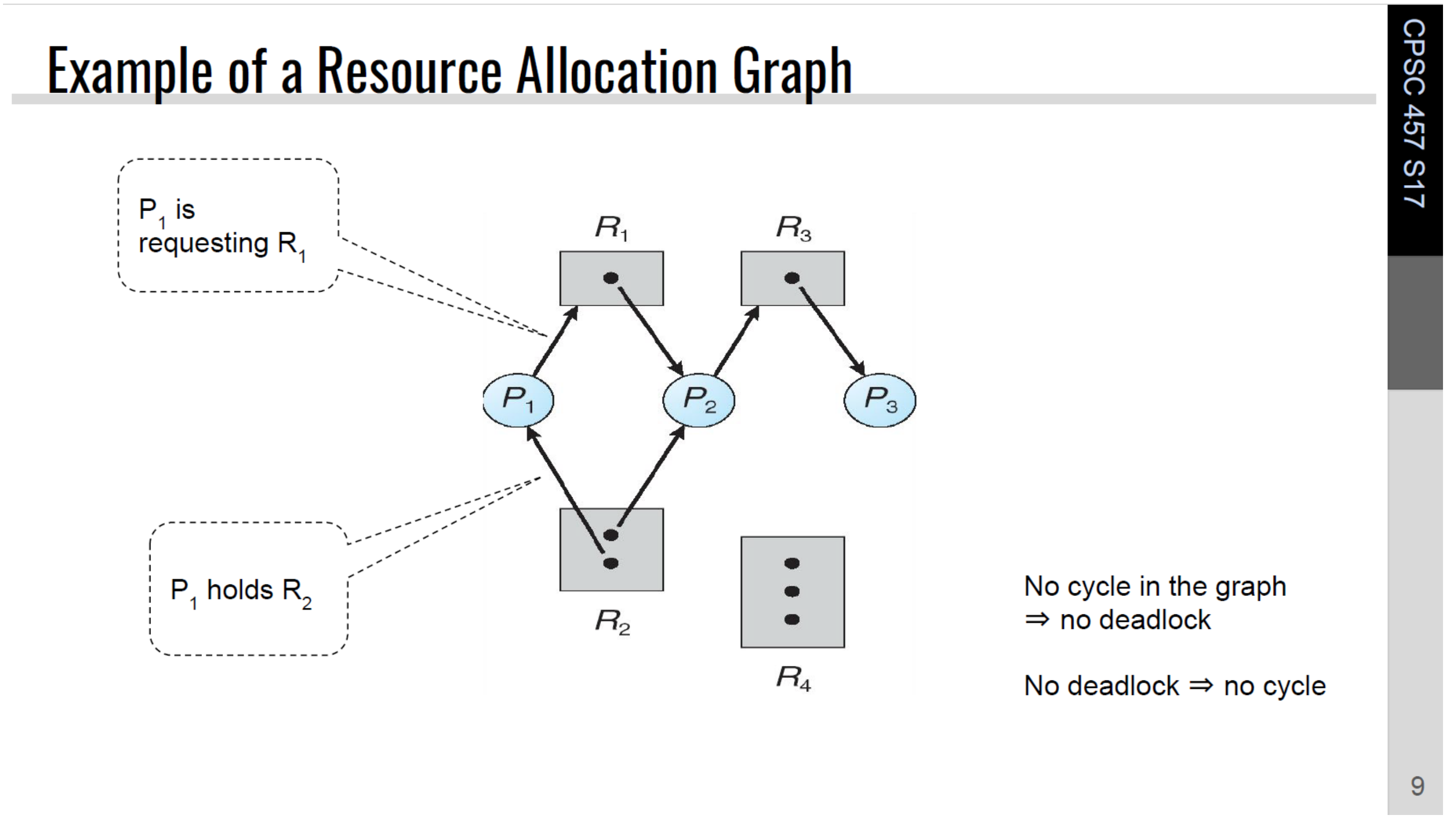
9.4 Resource Allocation Graph

- A set of vertices V and a set of edges E.
- Vertices are partitioned into 2 types:
  - $P = P_1, P_2, ..., P_n$, the set consisting of all the processes in the system
  - $R = R_1, R_2, ..., R_n$, the set consisting of all resource types in the system
- Request edge - directed edge from process to resource $(P_i \rightarrow R_j)$
- Assignment edge - directed edge from resource to process $(R_j \rightarrow P_i)$



Resource-Allocation Graph

- Process P_i
- P_i requests an instance of R_j
- Resource R_j with 3 instances
- P_i is holding an instance of R_j



Example of a Resource Allocation Graph

No cycle in the graph
⇒ no deadlock

No deadlock ⇒ no cycle

- No cycle in the graph → No deadlock
- No deadlock → No cycle
- Deadlock → Cycle
- Cycle $\not\to$ Deadlock

The operation order *does* affect if a system will go into deadlock.

9.5 Basic Facts

- If a graph contains no cycles → no deadlock
- If a graph contains a cycle →
  - If only one instance per resource type then *guaranteed* deadlock
  - If multiple instances per resource type then *possible* deadlock

9.6 Methods for Dealing with Deadlocks

Ignore

- Pretend that deadlocks never occur in the system
- Up to applications to address their deadlock issues

Prevention and Avoidance

- Ensure that a system will never enter a deadlock state

- They are different

Recover

- Allows system to enter deadlock and then recover
- Requires deadlock detection and recovery

9.7 Deadlock Prevention

A set of algorithms that removes one of the 4 necessary conditions for deadlock.

9.7.1 Avoid Mutual Exclusion

- Not required for shareable resources
- Required for non-shareable resources

9.7.2 Avoid Hold and Wait

- Must guarantee that whenever a process requests a resource, it does not hold any other resource

9.7.3 Avoid No Preemption

- Allow preemption
- If a process (holding some resource) requests another resource, have the process release all of it's currently held resources

9.7.4 Avoid Circular Wait

- Impose an ordering of all resource types and require that each process requests resources in an increasing order of enumeration

9.8 Deadlock Avoidance

- Can lead to low resource utilization
- Can increase resource utilization if some a priori (known before) information is available
- Have each process declare the maximum number of resources of each type that it may need. Then use an algorithm to ensure that there can never be a circular wait condition

9.8.1 Safe State

A system is in safe state if there exists a sequence of all running processes in the system where they can all finish without deadlock. If a system is in a safe state then no deadlocks are possible, if a system is not in a safe state then there is a possibility of deadlock.

9.8.2 Deadlock Avoidance Algorithms

Single instance per resource type

- Use resource-allocation graph algorithm

Multiple instance per resource type

- Use banker's algorithm

Single Instance: Resource-Allocation Graph Algorithm

Suppose that a process requests a resource, then the request can be granted only if allowing the request will not violate safe state. E.g. Converting the request edge to an assignment edge does not result in the formation of a cycle.

Multiple Instances: Banker's Algorithm

A more general avoidance algorithm than the resource-allocation graph that works with multiple instances per resource type.

- **Available**: the number of instances of resource type are available
- **Max**: the maximum instances of a resource type that the process can request
- **Allocation**: the current resource instances held by the process
- **Need**: the number of resource type instances that a process currently requires to execute (max = allocation)

The following is the steps for the algorithm.

- Check if request > need (Hey! You're asking more than you said you needed!)
- Check if request > available (Not enough resource for the request)
- Suppose the request was granted
- Then check if state is still safe
- Continue allocating resource until all processes are either finished or some of the processes cannot run
- If all finished, then grant request, else do not grant request

9.9 Deadlock Detection

9.9.1 Detection Algorithm

Detection algorithms are often expensive since you have to spend too many CPU cycles on useless work.

- Single instance per resource type
- Multiple instances per resource type

Single Instance

- Maintain a **wait-for** graph
- Nodes are processes
- $P_i \rightarrow P_j$ If P_i is waiting for P_j
- Uses a collapsed resource-allocation graph that does not show the resources but rather which process is waiting on who
- If there is a cycle, there exists a deadlock

Multiple Instances

Pretty much the same as the Banker's algorithm.

9.10 Deadlock Recovery

Works regardless of the number of instances per resource type.

9.10.1 Process Termination

Options

- Abort all deadlocked process
- Abort one process at a time until the deadlock cycle is eliminated (how to decide which one to abort?)

9.10.2 Process Rollback

Similar to termination but

- Programs can cooperate
- Use checkpoints to save current state or rollback
- Useful for long computations/simulations

9.10.3 Resource Preemption

Steps

- Pick a victim process
- Suspend victim process
- Save state of victim's process
- Give victim's resources to other deadlocked processes
- When the other processes release the resources, restore their state
- Return resources to the victim
- Resume the victim

Chapter 10

Slide Set 10: Memory

10.1 Addresses

Use to find processes in memory.

10.1.1 Physical Addresses

Not a good idea to work with physical addresses. Protection is a problem for jump/branch instructions.

10.1.2 Logical Addresses

Logical addresses are not real addresses but virtual addresses. Used so programs can see one contiguous space of memory but is really split across physical memory.

10.2 Binding of Instructions

When programs are written, the physical address space of the process is not known. Thus programs could be expressed in a way that allows them to be relocated or when needed, have the address bind to the actual memory location. The binding of instructions and data can happen at three different stages:

- **Compile Time**
- **Load Time**
- **Execution Time**

10.3 Memory-Management Unit

The hardware device that maps logical/virtual addresses to physical addresses.

10.4 Swapping

Processes can be swapped temporarily out of memory to a backing store and then brought back into memory. The backing store is a fast disk large enough to hold processes. Swapping allows the OS to run more processes than the available physical memory (RAM).

10.5 Memory Allocation

Deals with the issue of how much memory to give each process. The approaches can lead to fragmentation where there are lots of tiny, free chunks of memory but none of them can satisfy any request. The solution comes in two approaches.

10.5.1 Fixed Partitioning

Memory is divided into equal sized partitions that do not change. The problem is that this can lead to **internal fragmentation** where memory internal to a partition becomes fragmented.

10.5.2 Dynamic Partitioning

Memory is divided into partitions that fit the request perfectly. The problem is that this can lead to **external fragmentation** where memory external to all partition becomes fragmented.

10.6 Implementation

How do we keep track of free and allocated memory? Use the following data structures.

10.6.1 Bitmaps & Fixed Partitions

Memory is fixed partitioned. OS maintains a bitmap where 0 = free and 1 = used.

10.6.2 Linked List

Memory is dynamically partitioned. OS maintains a list of allocated and free memory using a linked list.

10.7 Memory Allocation Algorithms

- **First Fit**: find the first hole that is big enough, leftover space becomes new hole
- **Best Fit**: find the smallest hole that is big enough, leftover space becomes new hole (but very likely useless)
- **Next Fit**: same as first fit, but start searching at the location of last placement
- **Worst Fit**: find the largest hole, leftover space is likely to be usable
- **Quick Fit**: maintain separate lists for common request sizes

10.8 Virtual Memory

Virtual memory is a memory management technique that allows the OS to present a process with logical address space that appears contiguous. However, the physical address space can be discontiguous.

10.9 Paging

Virtual address space is divided into pages, physical address space is divided into frames. Pages and frames are the same size. Pages map to frames via a lookup table called a page table. This method avoid external fragmentation since there are no holes. Each process has its own page table.
If a program tries to access a page that does not map to a physical memory address, it is called a **page fault**.

10.10 Demand Paging Performance

Use this formula to calculate the effective access time (EAT).

$$EAT = (1 - p) * ma + p * pfst$$

where

- EAT = effective access time
- p = probability of page fault
- ma = memory access time
- pfst = page fault service time

10.11 Address Translation

- **Page Number**: Used as an index into a page table which contains base address of corresponding frame in physical memory
- **Page Offset**: Combined with base address to define the physical memory address that is sent to the memory unit

10.12 Memory Protection

Use a protection bit. Use valid/invalid bit (dirty bit).

Chapter 11

Slide Set 11: Paging

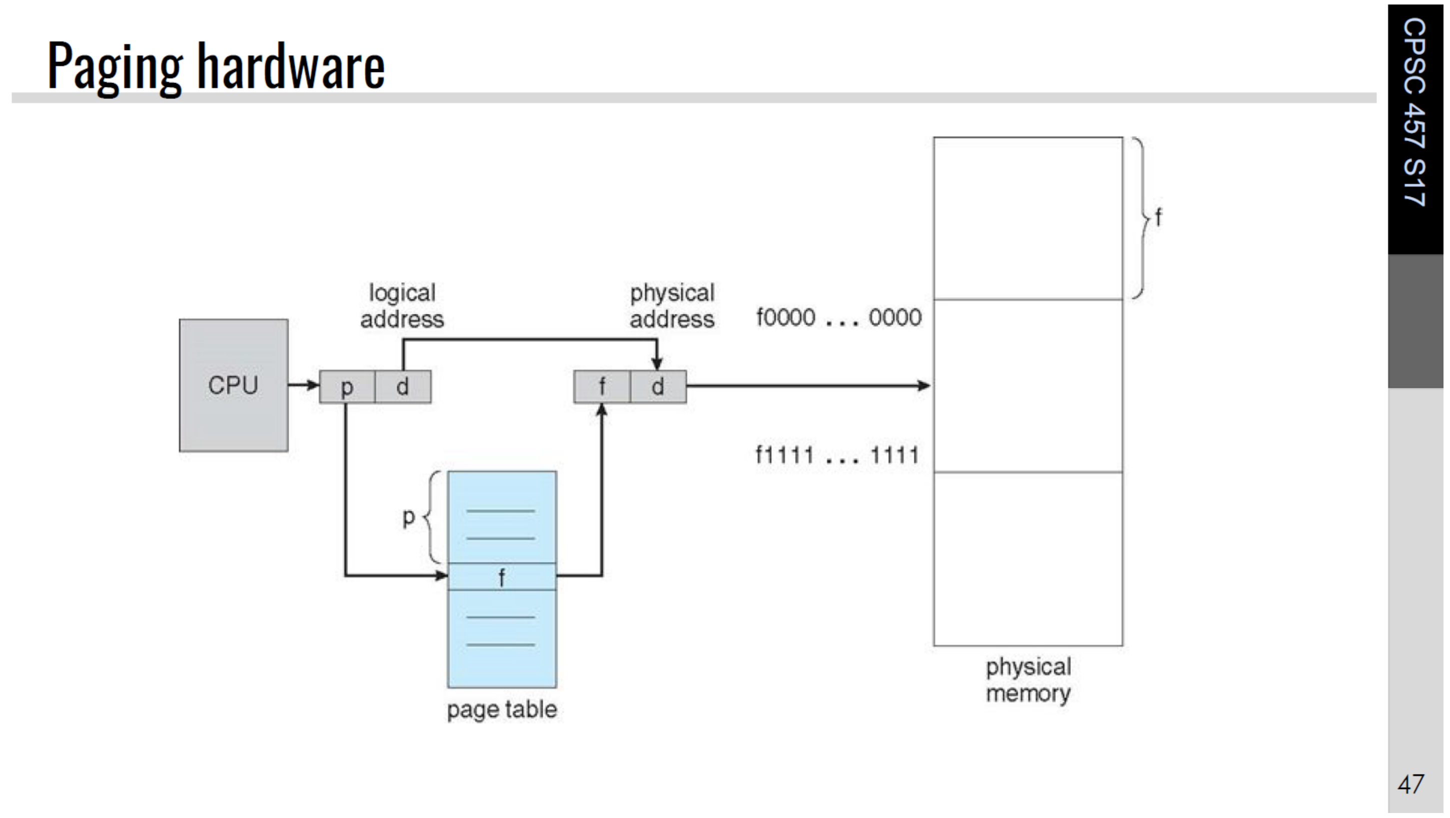
11.1 Shared Pages

11.2 Page Fault Handling

11.2.1 Belady's Anomaly

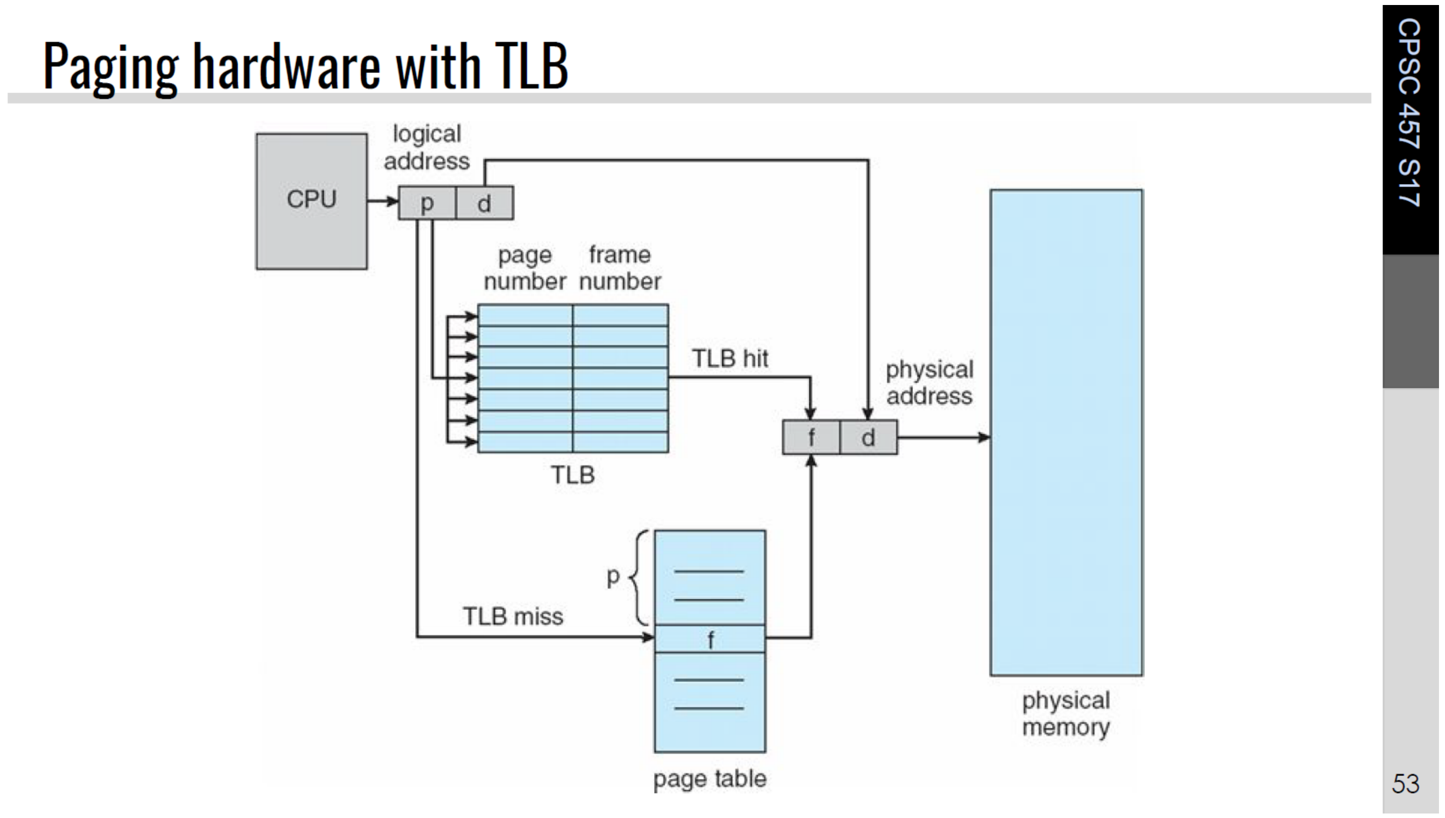
11.3 Page Table Implementations

11.4 Hardware



Paging hardware

11.4.1 Translation Look-Aside Buffers (TLBs)



-

11.4.2 Simple

11.4.3 Hierarchical

11.4.4 Inverted

11.4.5 Hashed IPT

11.5 Page Replacement Algorithms

11.5.1 First-In-First-Out

11.5.2 Optimal

11.5.3 Least Recently Used

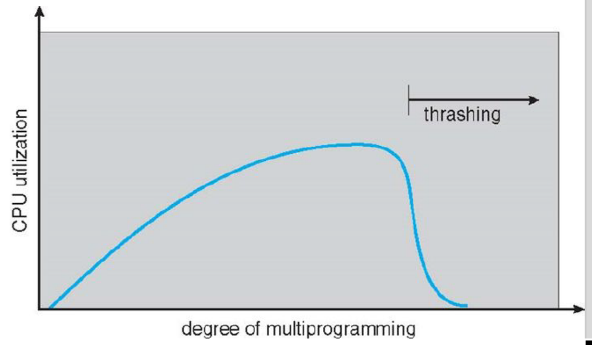
11.5.4 CLOCK Replacement

11.6 Thrashing

If a process does not have access to a sufficient number of memory pages, a futile, repetitive swapping condition known as "thrashing" often arises, and the page fault rate typically becomes high. This frequently leads to high, runaway CPU utilization that can grind the system to a halt.

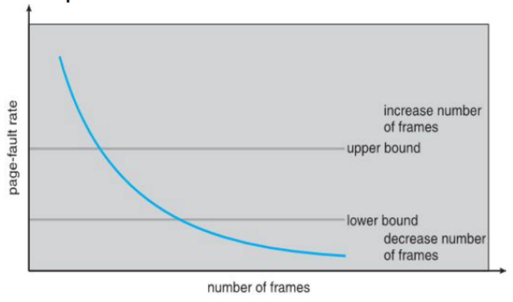
Thrashing

- if a process does not have “enough” pages, the page-fault rate is very high
  - page fault to get page
  - replace existing frame
  - but quickly need replaced frame back
- **thrashing process** = process is progressing slowly due to frequent page swaps
- this can lead to an entire system thrashing:
  - many processes thrashing → low CPU utilization
  - OS thinks that it needs to increase the degree of multiprogramming
  - OS adds another process to the system making things even worse

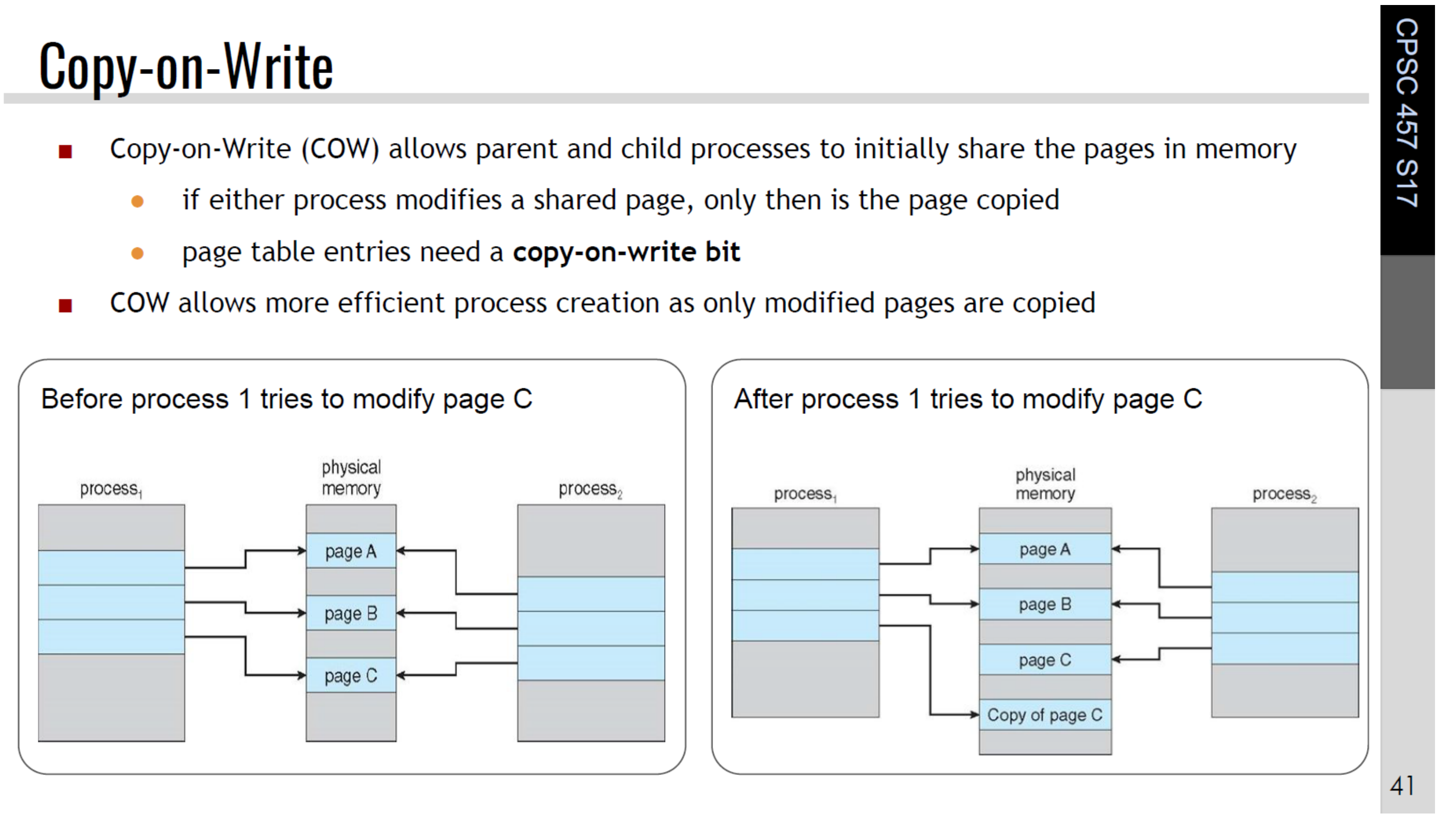


Dealing with thrashing

- local page replacement
  - when a process is thrashing, OS prevents it from stealing frames from other processes
  - at least the thrashing process cannot cause the entire system to thrash
- working set model
  - OS keeps track of pages that are actively used by a process (working set)
  - working set of processes changes over time
  - OS periodically updates the working set for each process, using a moving time window
  - before resuming a process, OS loads the working set of the process
- page fault frequency
  - establish acceptable bounds on page fault rate
  - if actual page fault rate of a process too high
    → process gains a frame
  - if actual page fault rate of a process too low
    → process loses a frame



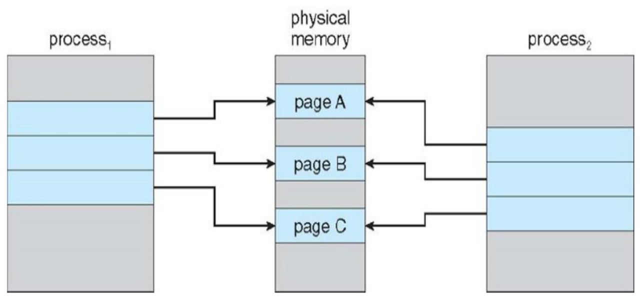
11.7 Copy on Write



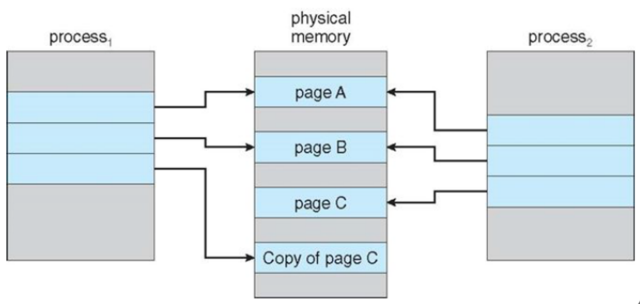
Copy-on-Write

- Copy-on-Write (COW) allows parent and child processes to initially share the pages in memory
  - if either process modifies a shared page, only then is the page copied
  - page table entries need a **copy-on-write bit**
- COW allows more efficient process creation as only modified pages are copied

Before process 1 tries to modify page C



After process 1 tries to modify page C



CPSC 457 S17

41

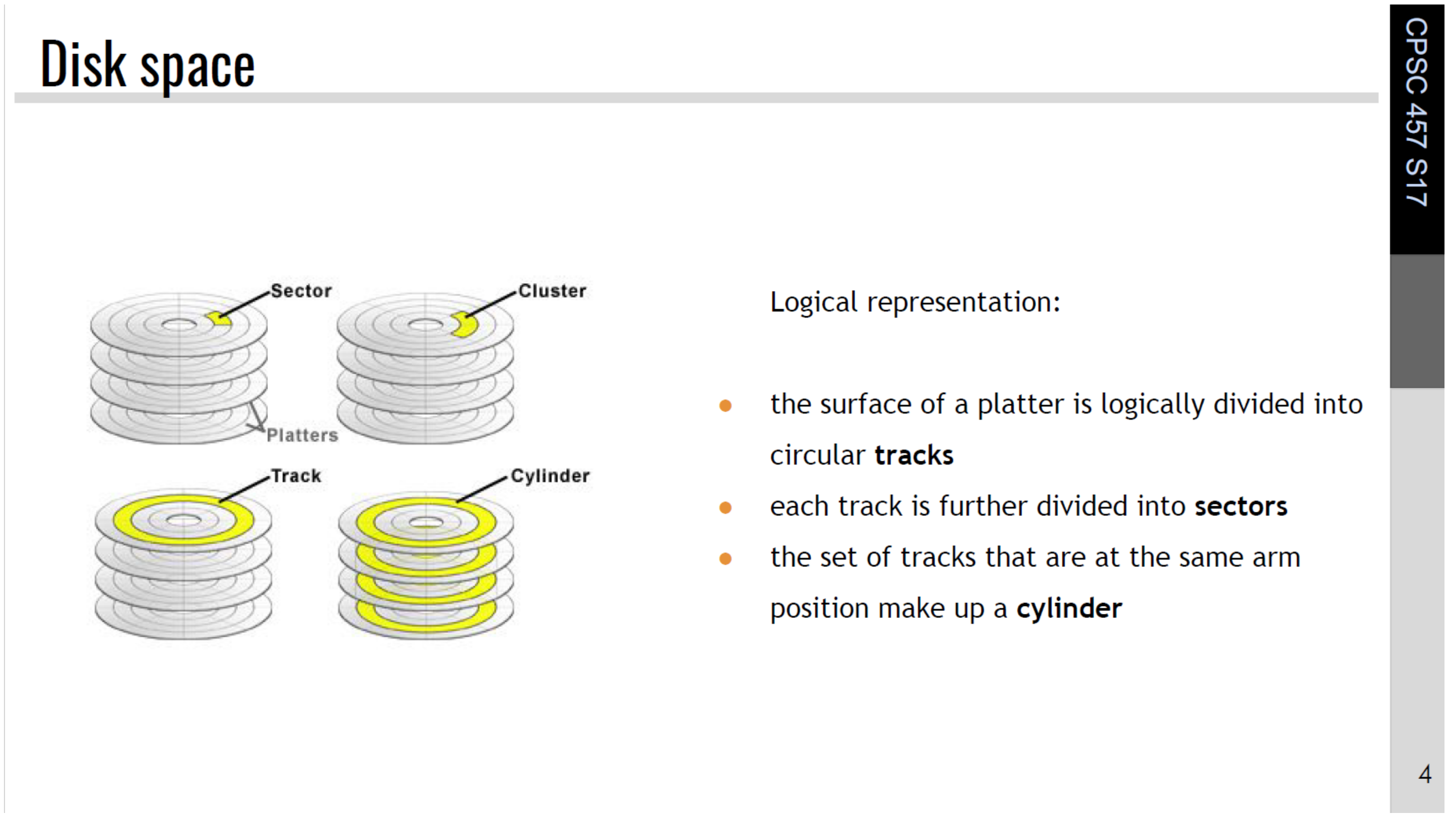
Chapter 12

Slide Set 11: Input/Output

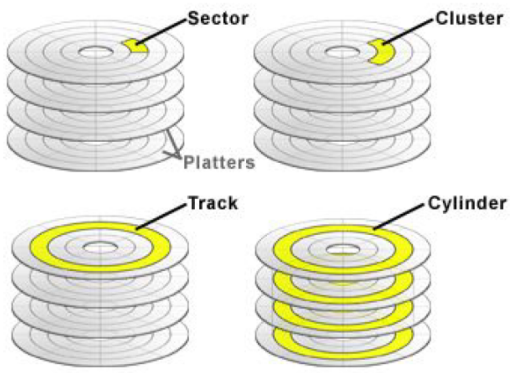
12.1 Disk Structure

A **head crash** is when the read/write head makes contact with the disk surface, causing permanent damage to the disk. In decreasing size:

- Hard Drive
- Platter
- Cylinder
- Track
- Sector



Disk space



Logical representation:

- the surface of a platter is logically divided into circular **tracks**
- each track is further divided into **sectors**
- the set of tracks that are at the same arm position make up a **cylinder**

12.1.1 Mapping

A **logical block** is the smallest unit of transfer between the disk and memory. **Mapping** is the converting of logical block numbers into physical disk address that consists of a cylinder number, a head number, and a sector number.

12.2 Disk Scheduling

The time required for reading/writing a disk block is determined by 3 factors:

- Seek Time: the time to move the arm to the correct cylinder
- Rotational Delay: the time for the correct sector to rotate under the head
- Disk Bandwidth: the time to actually transfer data

12.2.1 FCFS

The first request to enter is accessed.

12.2.2 SSTF

The request with the least seek time from the current head position is accessed. This scheduling could lead to starvation among requests.

12.2.3 SCAN

The head continuously scans back and forth across the disk and serves the requests as it reaches each cylinder

12.2.4 C-SCAN

Same as SCAN in one direction, but when reaching last cylinder, head returns to 1st cylinder

12.2.5 C-LOOK

Small optimization of C-SCAN, head only goes as far as needed by the next request and not to the end of the cylinder.

12.3 RAID

RAID stands for "redundant array of inexpensive disks."

- **RAID 0**: Split data across all disks
- **RAID 1**: Copy data across all disks
- **RAID 5**: Data is distributed with 1 parity strip per disk
- **RAID 6**: Data is distributed with 2 parity strips per disk
- **RAID 10**: A combination of RAID 0 and RAID 1

12.4 I/O Hardware

Block Devices

- Store information in fixed-size blocks
- Hard disk
- USB
- DVD

Character Devices

- Delivers/accepts a stream of characters without regard to block structure
- Printer
- Keyboard
- Mouse

Other Devices

- Clock/Timers

Chapter 13

Slide Set 12: File Systems

-