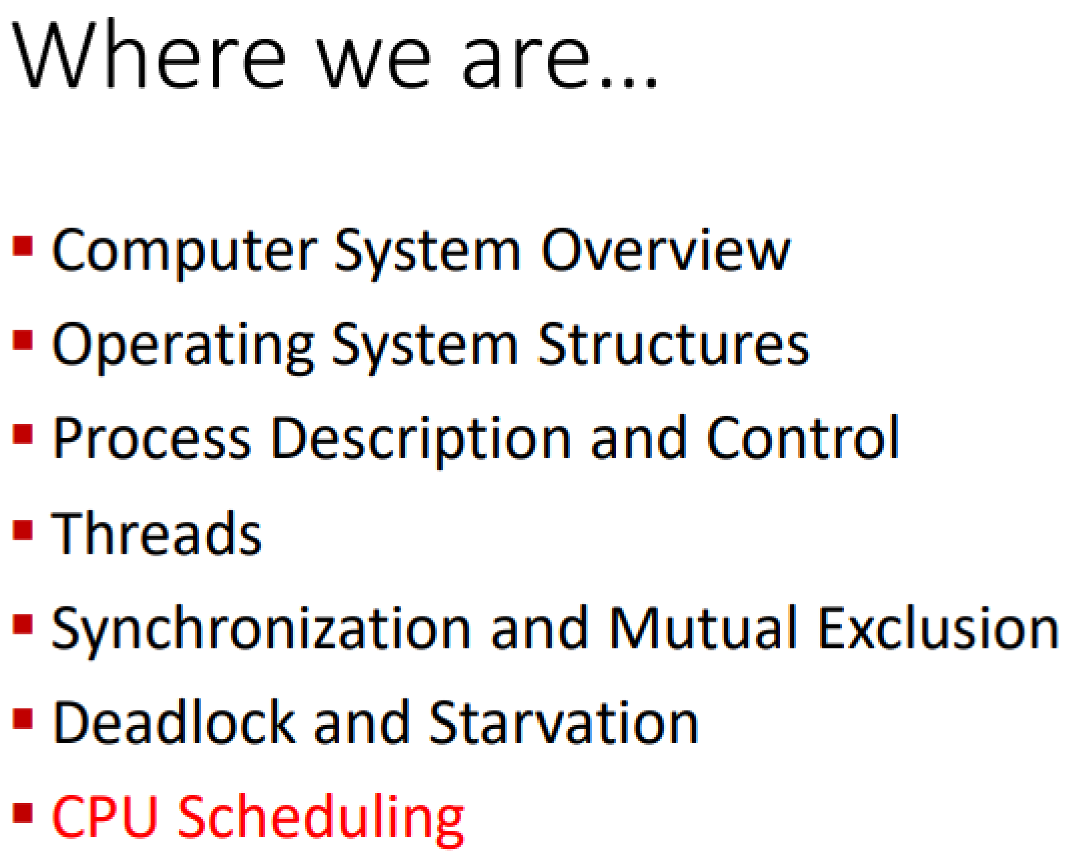
toc:

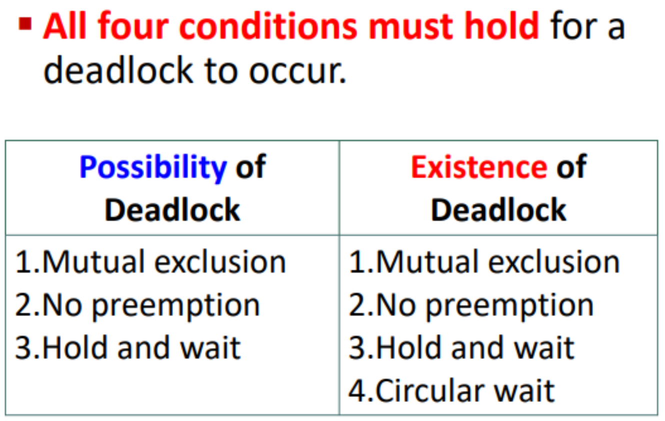
- Recap
- Basics
- Scheduling Criteria
- Scheduling Algos
- Multi-Processor Scheduling
    - Multicore
    - Multithreaded Cores
    - Load Balancing
    - Processor Affinity
    - NUMA Systems
    - Hetereogeneous Multiprocessing
- Real-Time CPU Scheduling
    - Priority Based Scheduling
- Algo Eval
    - Deterministic
    - Queueing Models
    - Simulations
    - Real Implementation



Where we are...

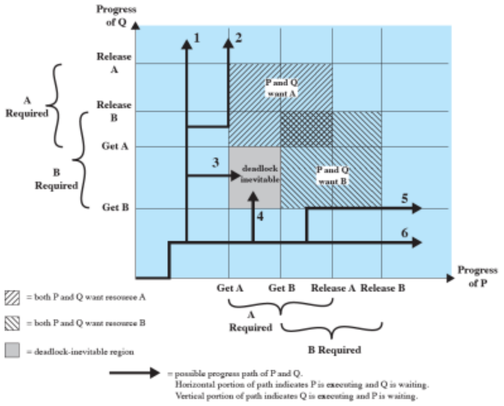
- Computer System Overview
- Operating System Structures
- Process Description and Control
- Threads
- Synchronization and Mutual Exclusion
- Deadlock and Starvation
- CPU Scheduling

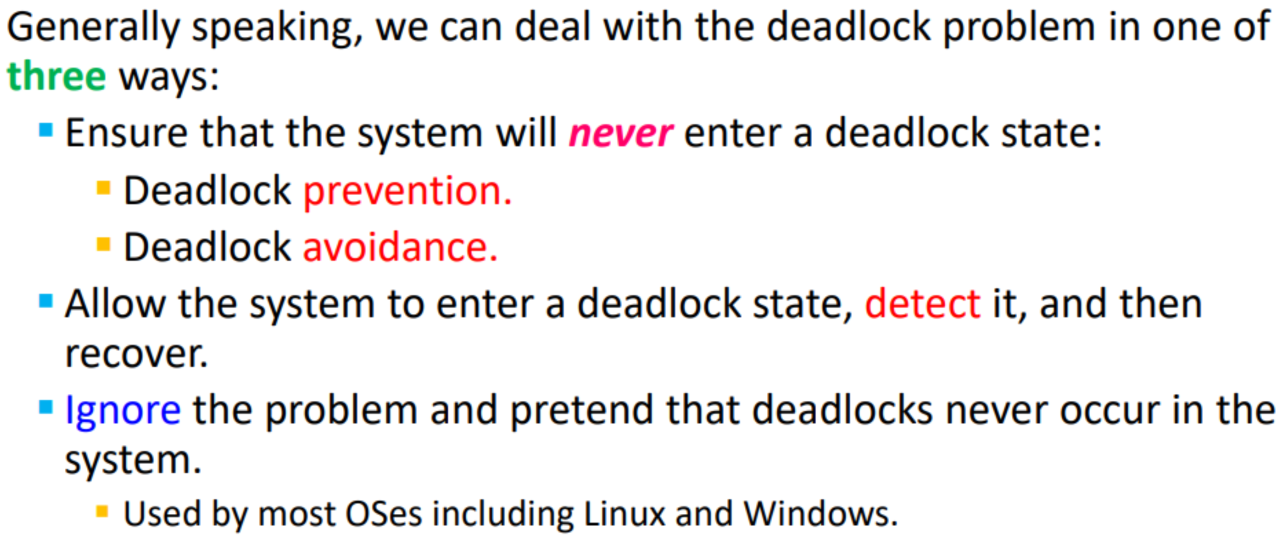
Recap



- **All four conditions must hold** for a deadlock to occur.

Possibility of **Deadlock**	**Existence** of **Deadlock**
1.Mutual exclusion 2.No preemption 3.Hold and wait	1.Mutual exclusion 2.No preemption 3.Hold and wait 4.Circular wait


Progress of Q
Release A
Release B
Get A
Get B
A Required
B Required
1
2
3
4
5
6
P and Q want A
P and Q want B
deadlock inevitable
Progress of P
Get A
Get B
Release A
Release B
A Required
B Required
▨ = both P and Q want resource A
▧ = both P and Q want resource B
■ = deadlock-inevitable region
= possible progress path of P and Q.
Horizontal portion of path indicates P is executing and Q is waiting.
Vertical portion of path indicates Q is executing and P is waiting.



Generally speaking, we can deal with the deadlock problem in one of **three** ways:

- Ensure that the system will ***never*** enter a deadlock state:
  - Deadlock prevention.
  - Deadlock avoidance.
- Allow the system to enter a deadlock state, detect it, and then recover.
- Ignore the problem and pretend that deadlocks never occur in the system.
  - Used by most OSes including Linux and Windows.

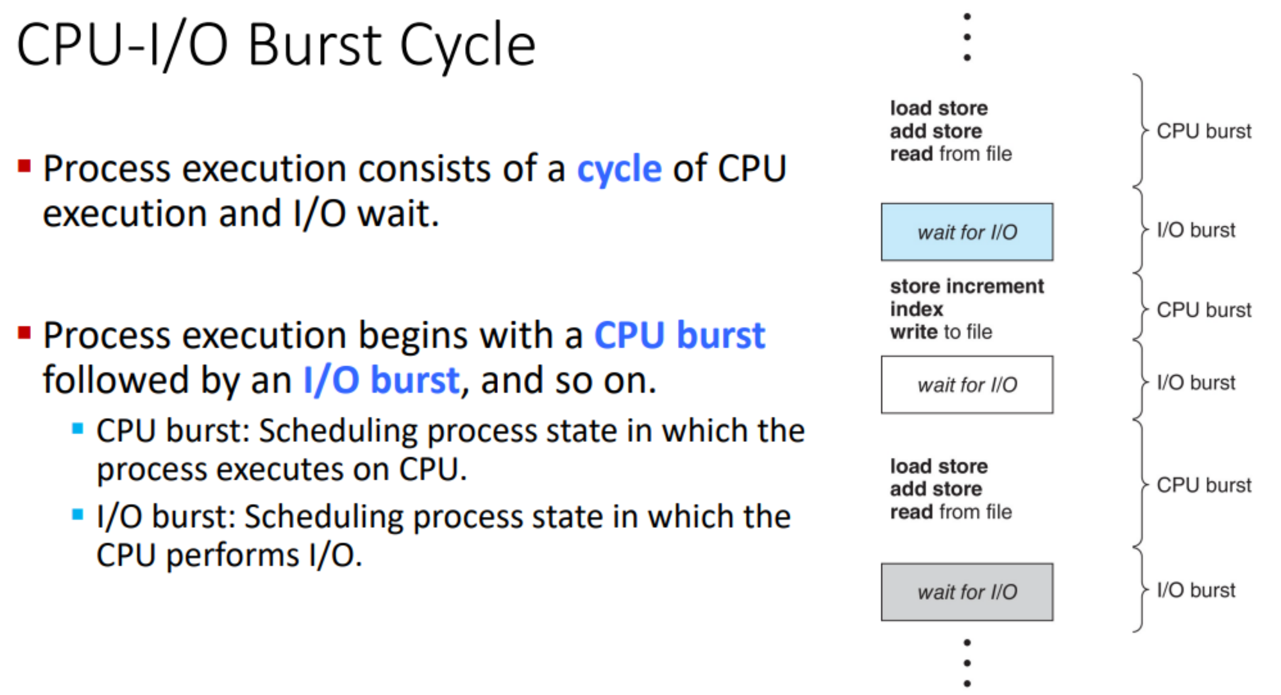
Basics

Kernel threads are ht eones being scheduled.

“Process scheduling” and “thread scheduling” are often used interchangeably.

We say CPU to mean CPU core as a computation unit

Maximum CPU utilization obtained with multiprogramming

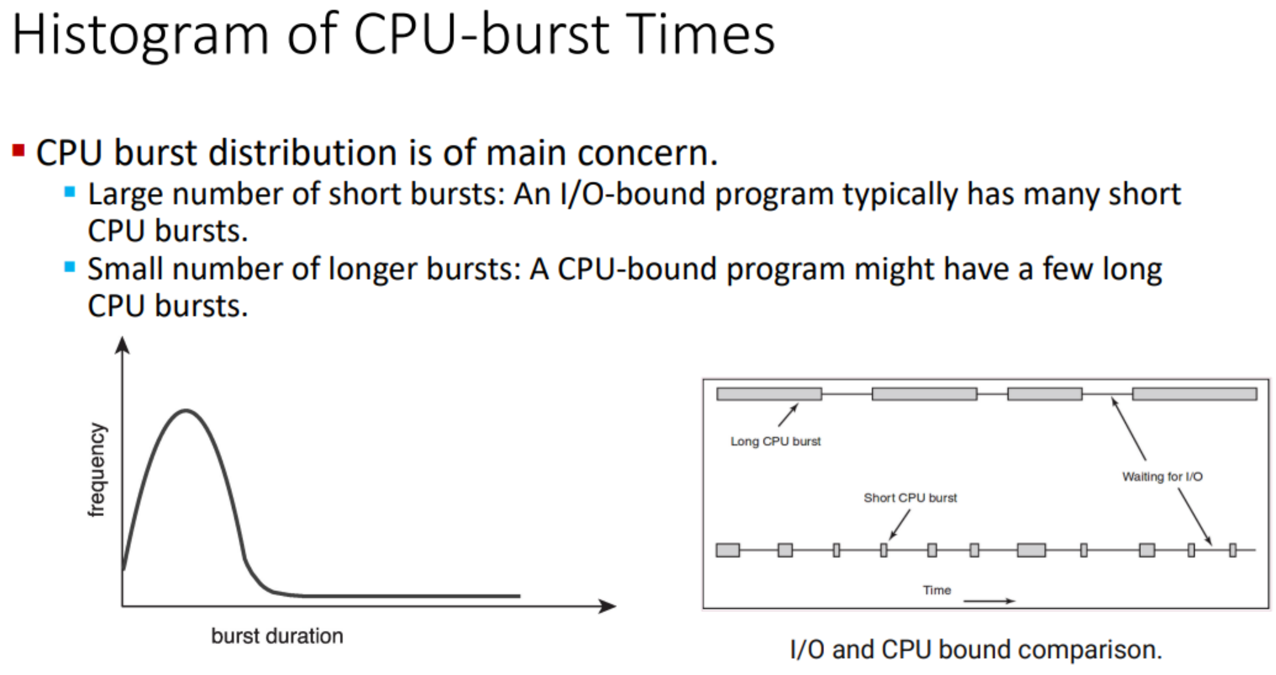


CPU-I/O Burst Cycle

- Process execution consists of a **cycle** of CPU execution and I/O wait.
- Process execution begins with a **CPU burst** followed by an **I/O burst**, and so on.
  - CPU burst: Scheduling process state in which the process executes on CPU.
  - I/O burst: Scheduling process state in which the CPU performs I/O.

We alternate working (cpu) and wait (i/o) bursts.

We end on an I/O burst as the process is ending



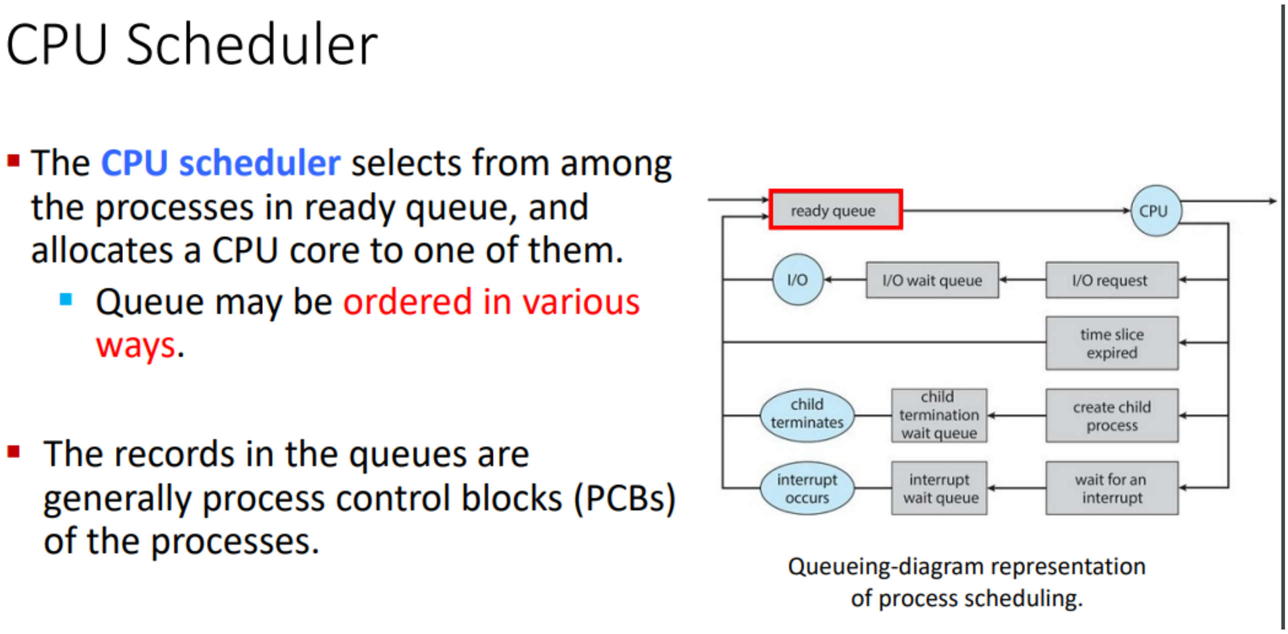
Histogram of CPU-burst Times

- CPU burst distribution is of main concern.
  - Large number of short bursts: An I/O-bound program typically has many short CPU bursts.
  - Small number of longer bursts: A CPU-bound program might have a few long CPU bursts.

I/O and CPU bound comparison.

On the histogram:

- above is cpu-bound
- below is i/o bound

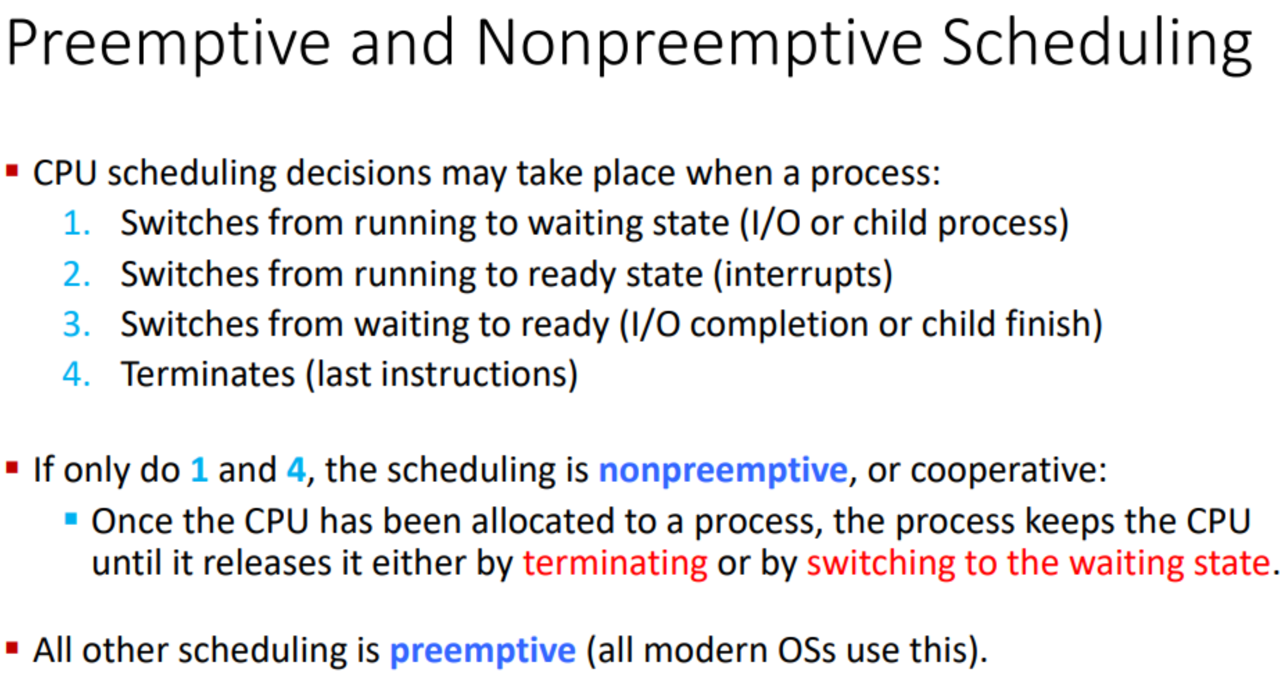


CPU Scheduler

- The **CPU scheduler** selects from among the processes in ready queue, and allocates a CPU core to one of them.
  - Queue may be ordered in various ways.
- The records in the queues are generally process control blocks (PCBs) of the processes.

Queueing-diagram representation of process scheduling.

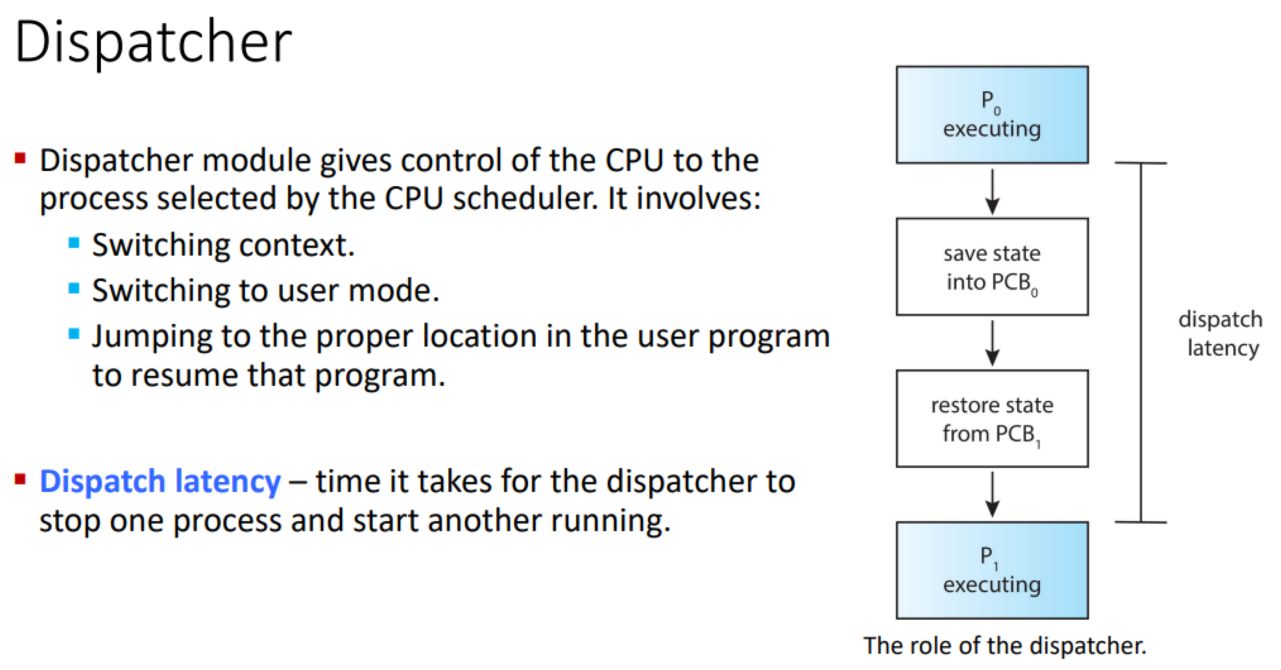
Reminder: the whole process itself isn't in the queue, just the pcb (which has all the info and points to where the process actually is)



Preemptive and Nonpreemptive Scheduling

- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state (I/O or child process)
  2. Switches from running to ready state (interrupts)
  3. Switches from waiting to ready (I/O completion or child finish)
  4. Terminates (last instructions)
- If only do 1 and 4, the scheduling is **nonpreemptive**, or cooperative:
  - Once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state.
- All other scheduling is **preemptive** (all modern OSs use this).

Preemptive scheduling makes decisions at all these times.



Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the CPU scheduler. It involves:
  - Switching context.
  - Switching to user mode.
  - Jumping to the proper location in the user program to resume that program.
- Dispatch latency – time it takes for the dispatcher to stop one process and start another running.

The role of the dispatcher.

Knowledge Check

- which of the following is true of cooperative scheduling
  - b. A process keeps the CPU until the releases the CPU either by terminating or by switching to the waiting state.
- In preemptive scheduling, the sections of code affected by interrupts must be guarded from simultaneous use
  - True
  - we want to ensure mutual exclusion
  - we want to make sure no other process is accessing at the same time

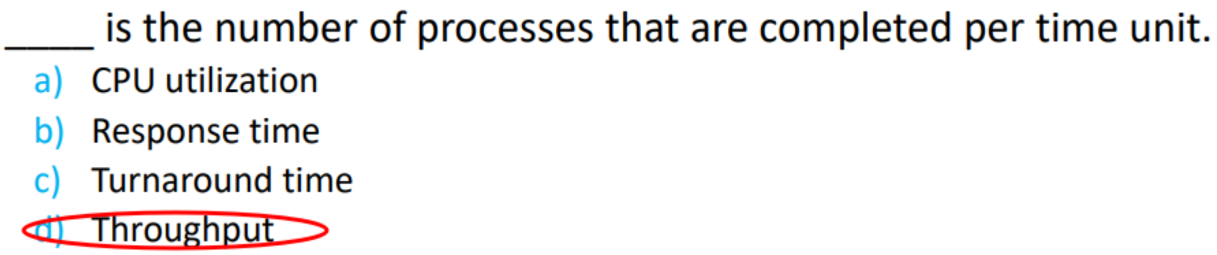
Scheduling Criteria

- cpu util
  - keep as busy as possible
  - maximize this
- throughput
  - num of processes that complete their execution per time unit
  - maximize this
- turnaround time
  - amount of time to execute a particular process
  - minimize this
- waiting time
  - aggregate amount of time a process has been waiting int eh ready queue
  - will be affected by scheduling algo
  - minimize this
- response time
  - time between request submission and first response for interactive systems

- minimize this

All numbers in milliseconds (ms) unless stated otherwise.

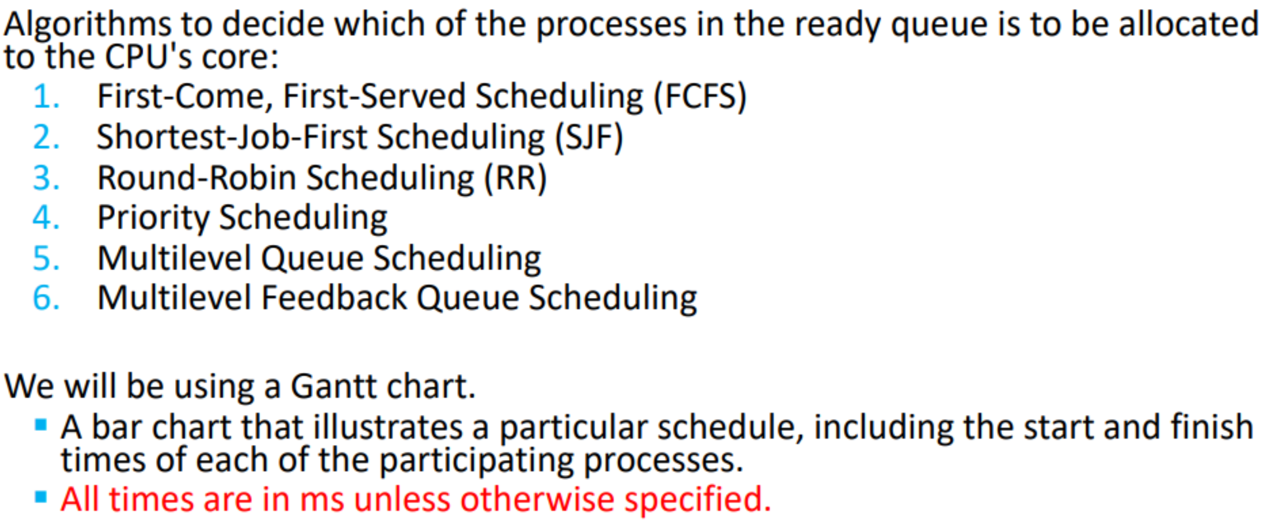
For now we assume we have 1 cpu.



_____ is the number of processes that are completed per time unit.

a) CPU utilization
b) Response time
c) Turnaround time
[✓ d) Throughput]

Scheduling Algos

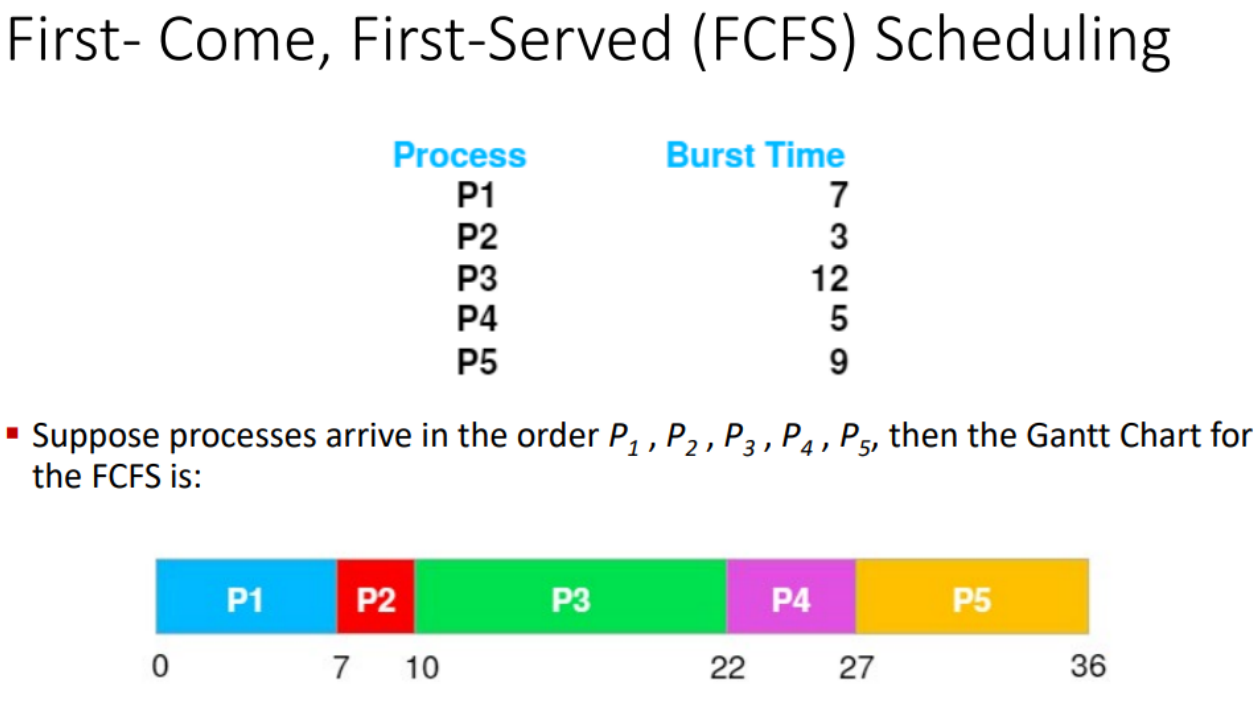


Algorithms to decide which of the processes in the ready queue is to be allocated to the CPU's core:

1. First-Come, First-Served Scheduling (FCFS)
2. Shortest-Job-First Scheduling (SJF)
3. Round-Robin Scheduling (RR)
4. Priority Scheduling
5. Multilevel Queue Scheduling
6. Multilevel Feedback Queue Scheduling

We will be using a Gantt chart.

- A bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes.
- All times are in ms unless otherwise specified.



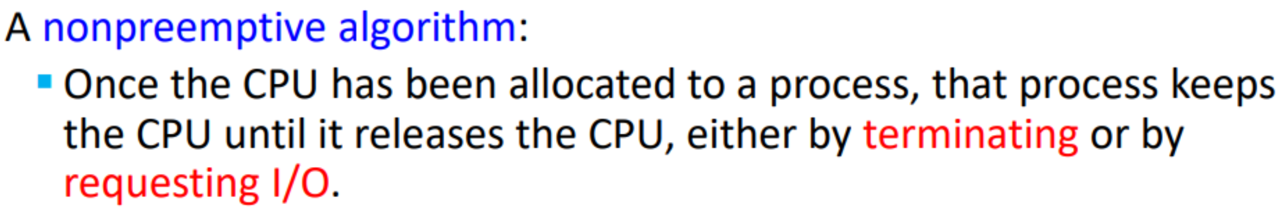
First- Come, First-Served (FCFS) Scheduling

Process	Burst Time
P1	7
P2	3
P3	12
P4	5
P5	9

- Suppose processes arrive in the order P_1 , P_2 , P_3 , P_4 , P_5, then the Gantt Chart for the FCFS is:

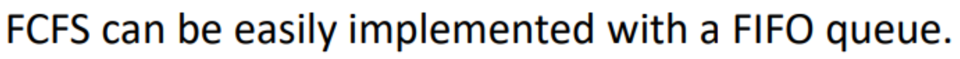
P1
P2
P3
P4
P5
0
7
10
22
27
36

we're assuming these all arrived at (more or less) the same time since there is no `arrival time` column in the chart. We assume in order of number just because it's convenient here.



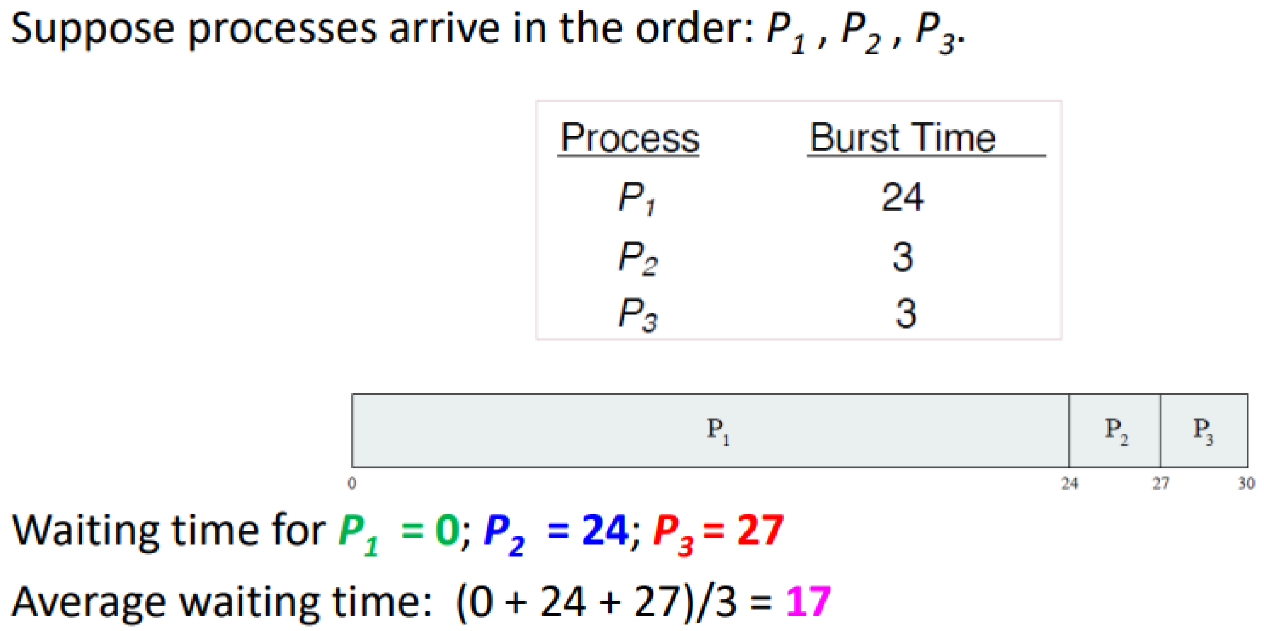
A nonpreemptive algorithm:

- Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O.



FCFS can be easily implemented with a FIFO queue.

different orders will result in different wait times but FCFS can't account for that

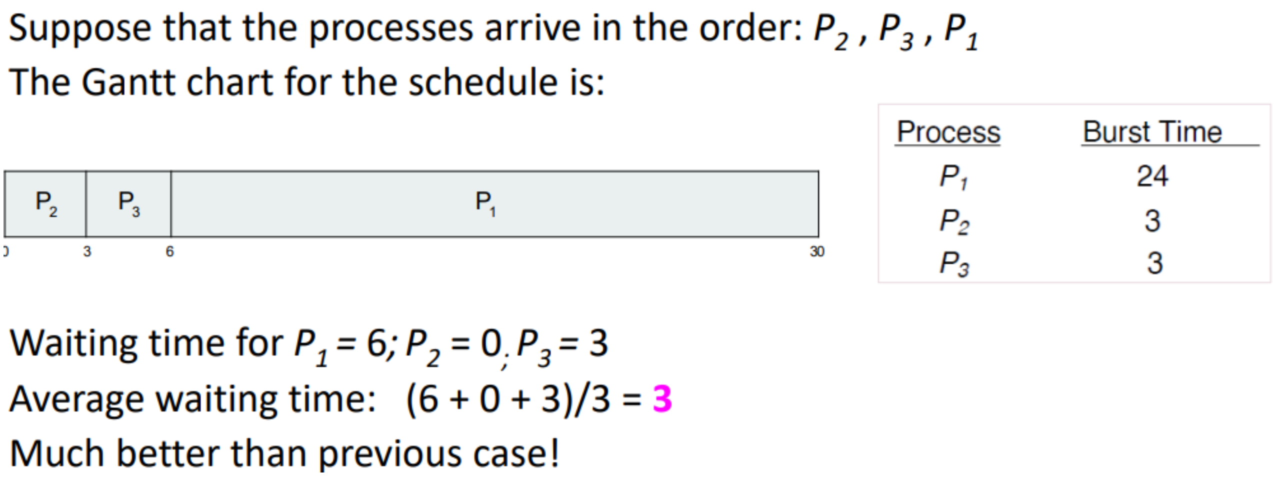


Suppose processes arrive in the order: P_1 , P_2 , P_3.

Process	Burst Time
P_1	24
P_2	3
P_3	3

Waiting time for P_1 = 0; P_2 = 24; P_3 = 27

Average waiting time: (0 + 24 + 27)/3 = 17



Suppose that the processes arrive in the order: P_2 , P_3 , P_1

The Gantt chart for the schedule is:

Process	Burst Time
P_1	24
P_2	3
P_3	3

Waiting time for P_1 = 6; P_2 = 0, P_3 = 3

Average waiting time: (6 + 0 + 3)/3 = 3

Much better than previous case!

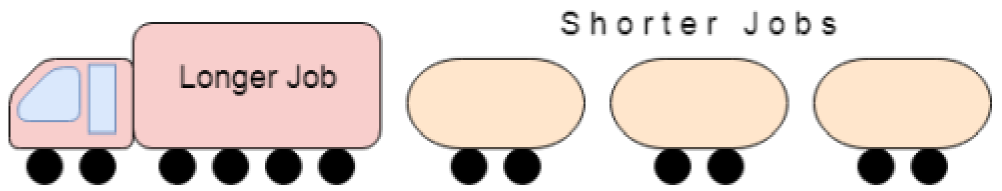
Convoy effect:

- short process behind long process
- consider one cpu-bound and many i/o-bound processes

FCFS is not good for interactive systems

- important that each process get a share of the cpu at regular intervals

The Convoy Effect, Visualized Starvation

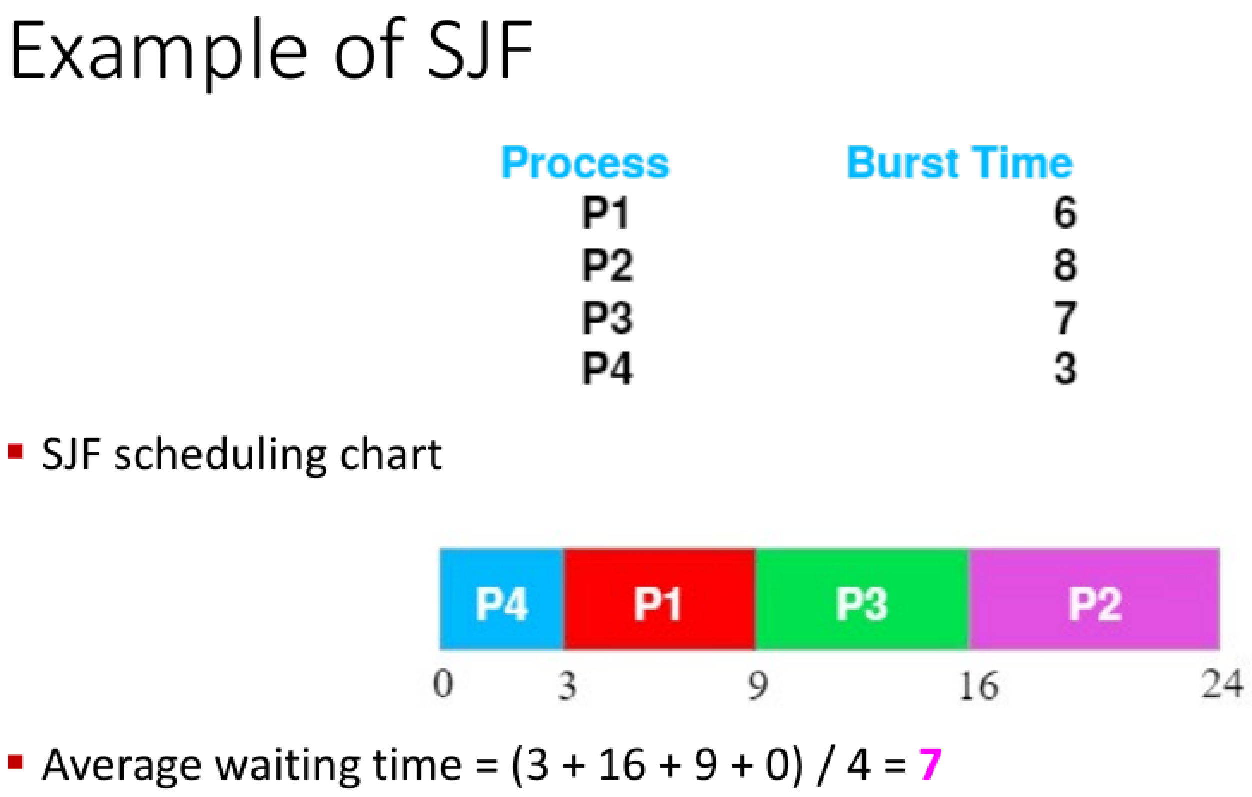


Shortest-Job-First (SJF) Scheduling

- for each process, associate the length of its next CPU burst
  - use these lengths to schedule the process with the shortest time
  - shortest-next-cpu-burst is a more accurate description
- SJF is optimal
  - gives minimum average waiting time for a given set of processes

If the next bursts of 2 processes are the same, FCFS scheduling is used to break the tie.

But how do we know how long the next cpu burst is? We can ask the user but there's more to it.



Example of SJF

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

- SJF scheduling chart

- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Remember that waiting time is the time that the process waits to ***start*** not to finish.

So the above calculation w/ start time s_i for process p_i would look like this:

average waiting time $= (w_1 + w_2 + w_3 + w_4)/4$

For FCFS we'd have

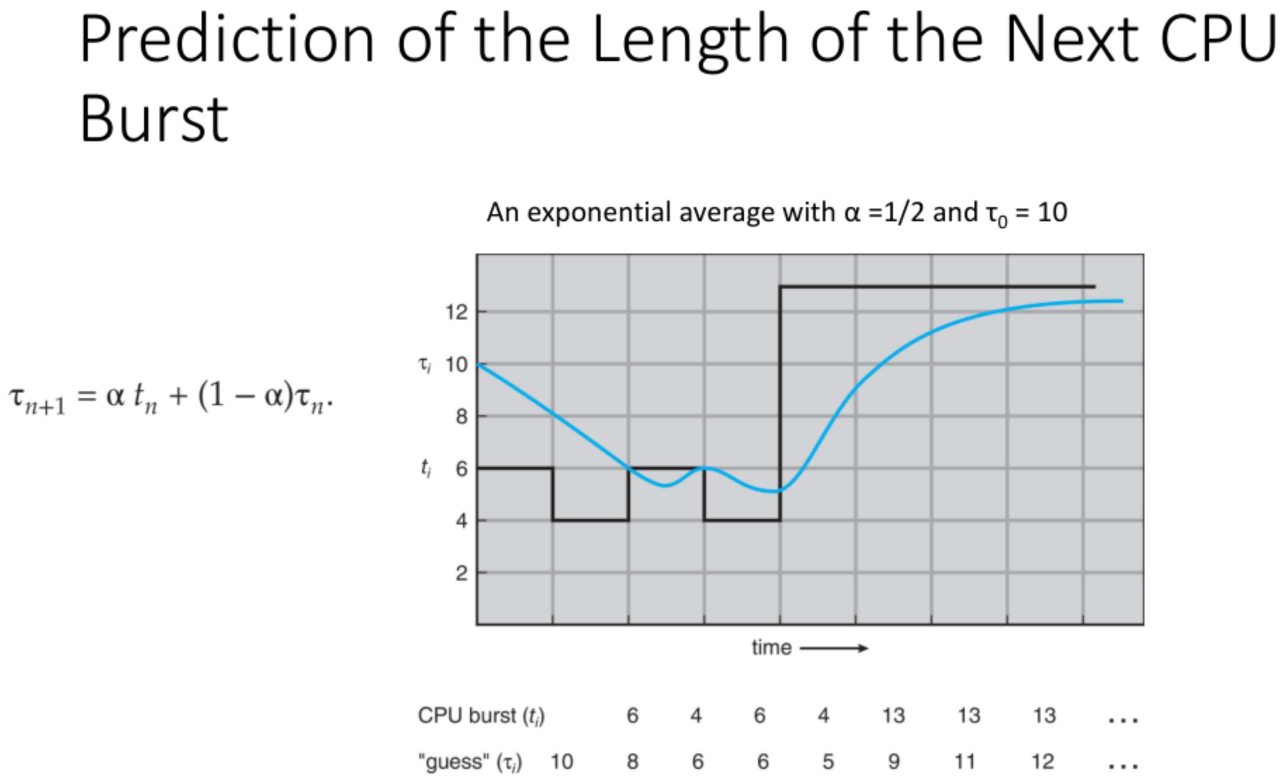
average waiting time $= (0+6+14+21)/4 = 10.75$

We can only estimate the length, typically assumiing it to be similar to the previous one. Then pick the process with the shortest predicted next CPU burst.

We use an exponential averaging of the length of the previous cpu bursts

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$$

- t_n = actual length of n^{th} CPU burst
- τ_{n+1} = predicted value of the next CPU burst
- $\alpha \in [0,1]$, typically $\frac{1}{2}$ for our applications



Prediction of the Length of the Next CPU Burst

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n.$$

An exponential average with α =1/2 and τ_0 = 10

CPU burst (t_i)		6	4	6	4	13	13	13	…
"guess" (τ_i)	10	8	6	6	5	9	11	12	…

We start by predicting 10 ms for the first burst.

It actually uses 6 ms.

We use these values to predict the next burst.

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n = \frac{1}{2}6 + (1-\frac{1}{2})10 = 8$$

Then we check against the next actual burst length.

So on and so forth.

The predictions aren't the tightest at times but they're not too too far off.

Note that we're using the actual previous burst time as well as the prebious prediction in our calculations to predict the next burst time.

More on exponential averaging:

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = \alpha t_n$
  - only the actual last CPU burst counts

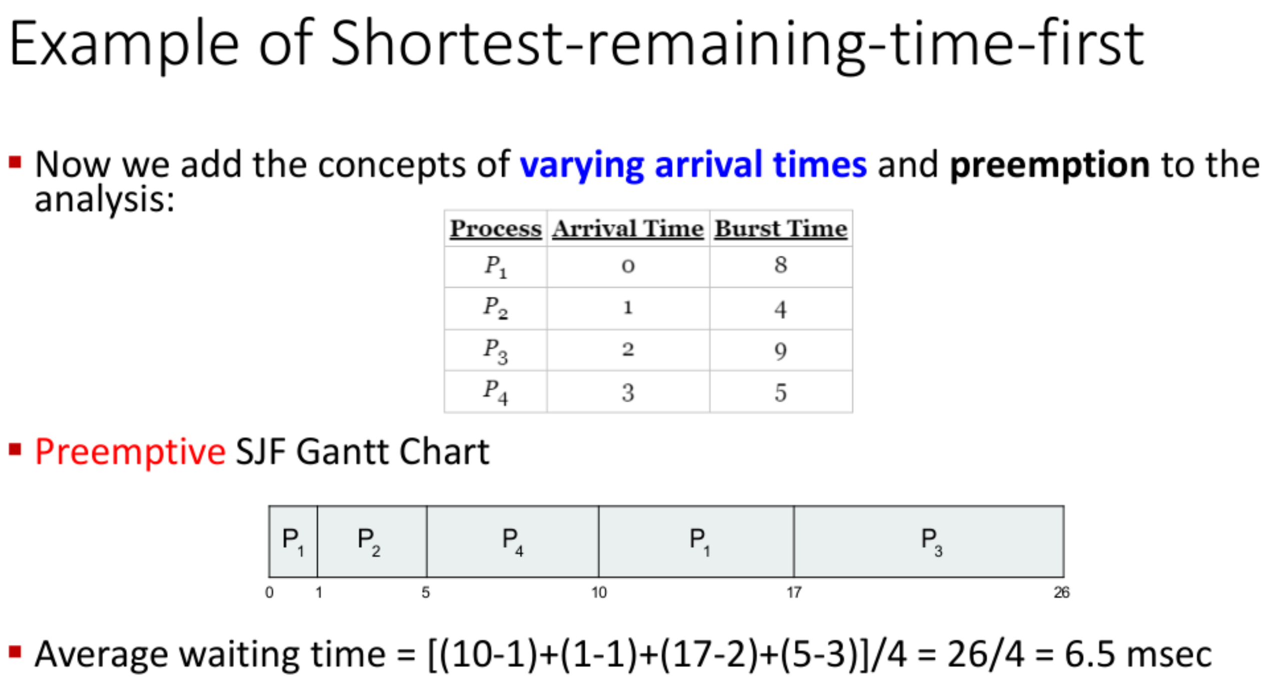
Exponential averaging is a series that we use for prediction in genearl, not just for processors.

Tweaking the α value will allow us to put more weight on near or far.

Preemptive vs Non-Preemptive

- SJF can be either
- the choice arises when anew process arrives at the ready queue while a previous process is still executing
- the next burst of the newer process may be shorter than what is left of the currently executing process
- in this scenario
  - if preemptive
    - preempt the currently executing process
    - sometimes known as `shortest-remaining-time-first-scheduling`
  - if non-preemptive
    - allow the currently running process to finish its CPU burst

The context switches take about 10 microseconds so preempting here is fine for us as we're typically just working in milliseconds.



Example of Shortest-remaining-time-first

- Now we add the concepts of **varying arrival times** and **preemption** to the analysis:

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- Preemptive SJF Gantt Chart

P_1
P_2
P_4
P_1
P_3
0
1
5
10
17
26

- Average waiting time = [(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5 msec

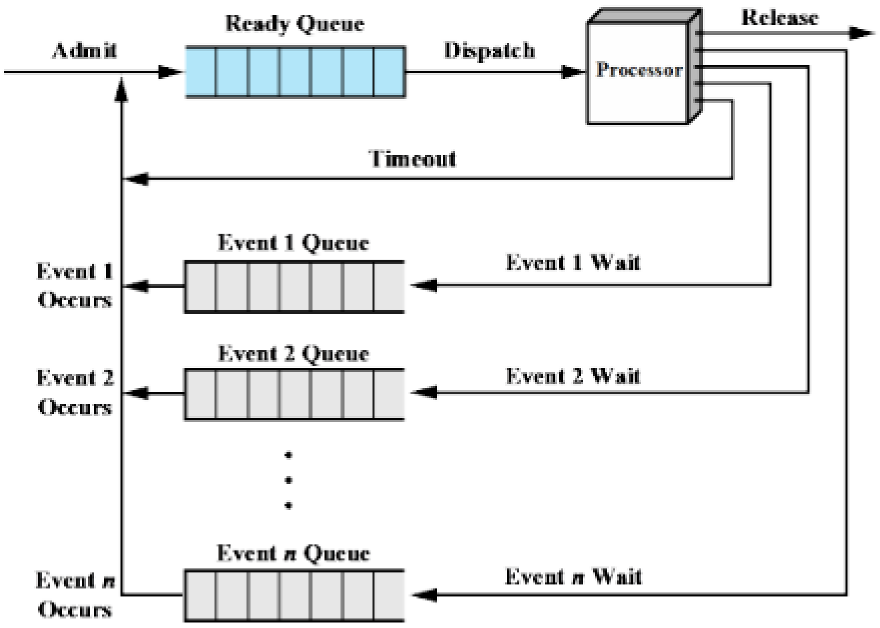
- P_1 arrives and works for 1 ms
- P_2 arrives and has 4 ms compared to P_1's 7 ms
- P_1 is preempted and P_2 begins execution
- P_3 arrives and has 9 ms compared to P_2's 3 ms
- P_4 arrives and has 5 ms compared to P_2's 2 ms
- $@t = 5, P_2$ finishes
- We choose P_4 as it has the least remaining time to finish
- P_4 finishes
- P_1 is chosen to execute as it has 7 ms remaining
- P_3 is chosen after P_1 finishes as it has 9 ms reamining

Average waiting time $= [P_1 + P_2 + P_3 + P_4]/4$

Remember that waiting time is the time waiting in the queue. So we have to add in the 9 ms that P_1 was sent back to the queue for.

Round Robin (RR) Scheduling

- each process gets a time quantum (q)
  - a small unit of cpu time
  - usually 10-100 ms
- after the quantum has passed
  - process is preempted
  - process added to end of ready queue



○ (b) Multiple blocked queues

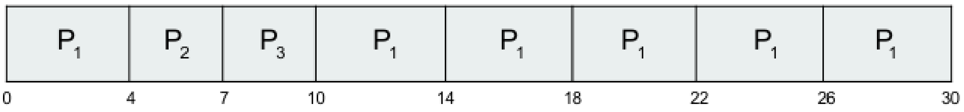
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once.
- No process waits more than (*n*-1)x*q* time units.
- Timer interrupts every quantum to schedule next process.
- Performance:
  - *q* large ⇒ FCFS
  - *q* small ⇒ *q* must be large with respect to context switch, otherwise overhead is too high.

- The average waiting time under the RR policy is often long.

Example of RR with Time Quantum = 4

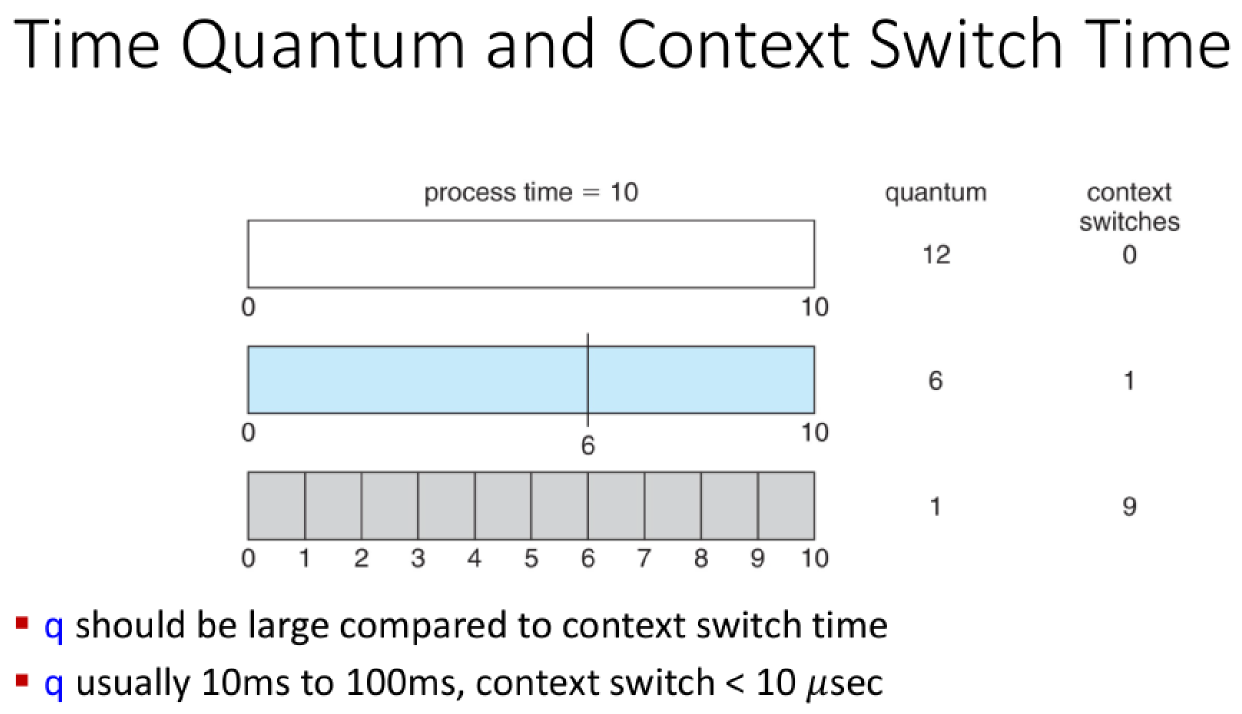
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



- P_1 starts executing
- 4 ms pass and the cpu checks the queue
- P_1 is preempted and P_2 is put in its place for execution
- P_2 finishes execution
- P_3 executes and finishes
- P_1 begins to execute once more and runs the course of the 4 ms quantum

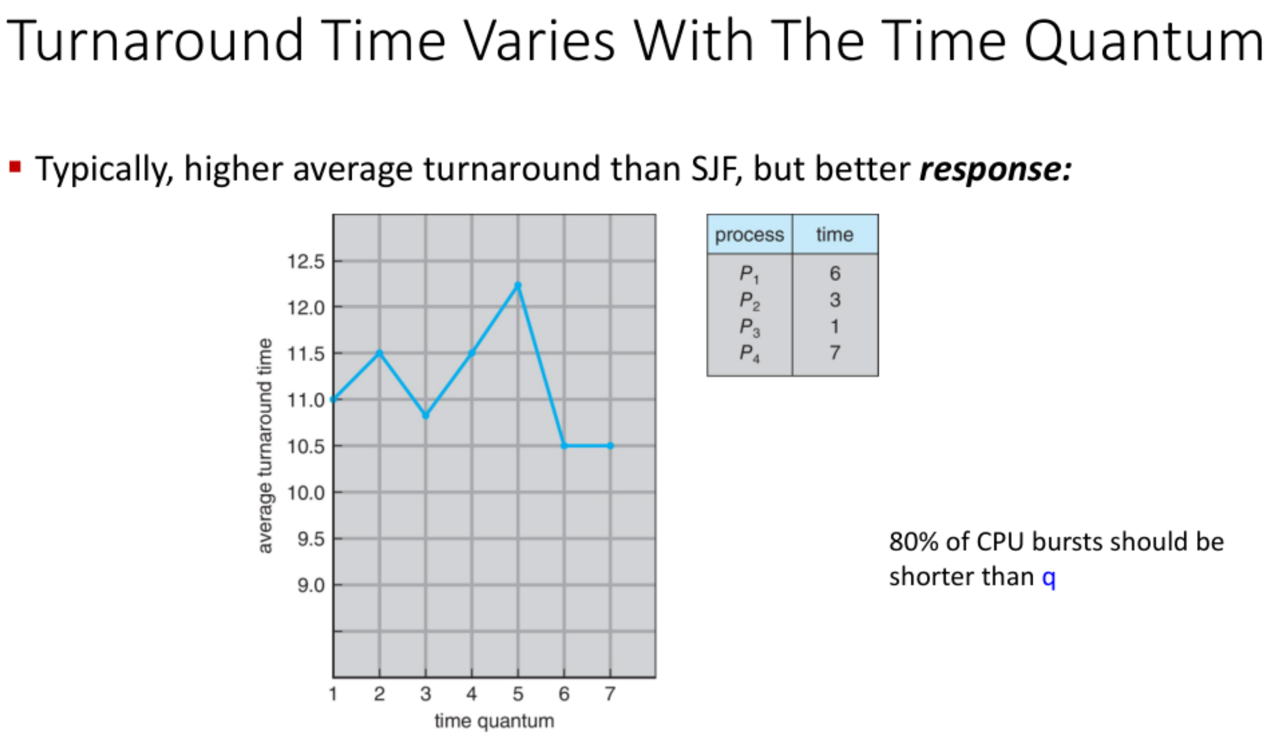
averaging waiting time $= \frac{6+4+7}{3} = 5.666...$



Time Quantum and Context Switch Time

quantum	context switches
12	0
6	1
1	9

- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 μsec



Turnaround Time Varies With The Time Quantum

- Typically, higher average turnaround than SJF, but better ***response:***

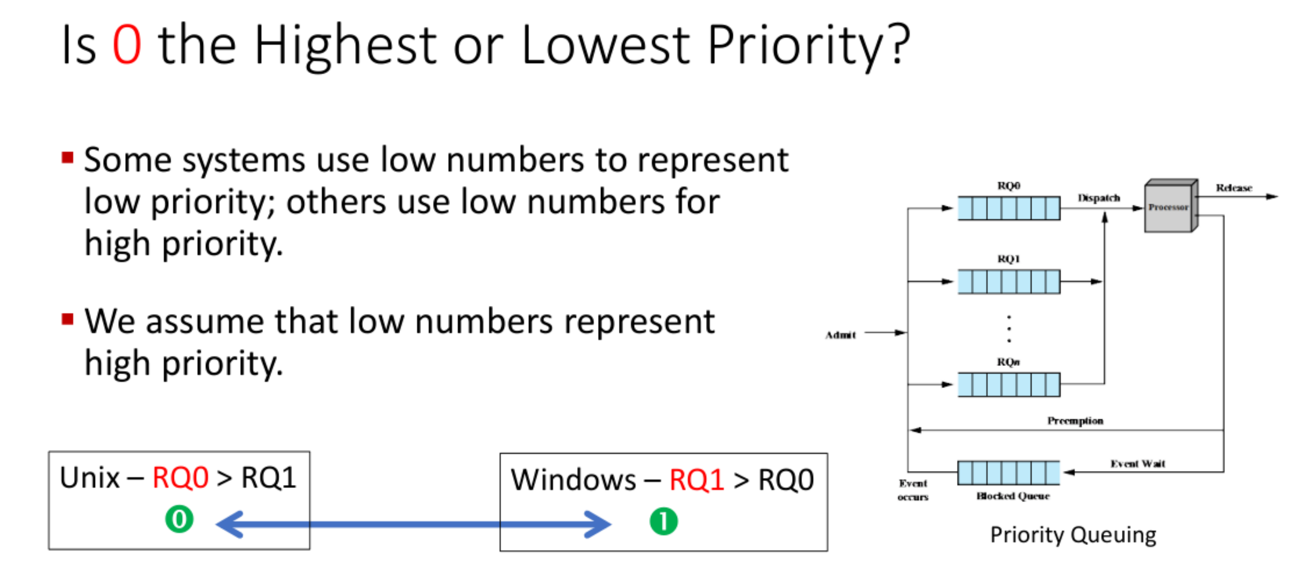
process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts should be shorter than q

Priority Scheduling

- priority number (integer) associated w/ each process

- CPU is allocated to the process with the highest priority
  - can be done preemptively or nonpreemptively
- SJF is priority scheudling where priority is the inverse of predicted next CPU burst time
  - larger burst = lower priority
  - smaller burst = higher priority



Is 0 the Highest or Lowest Priority?

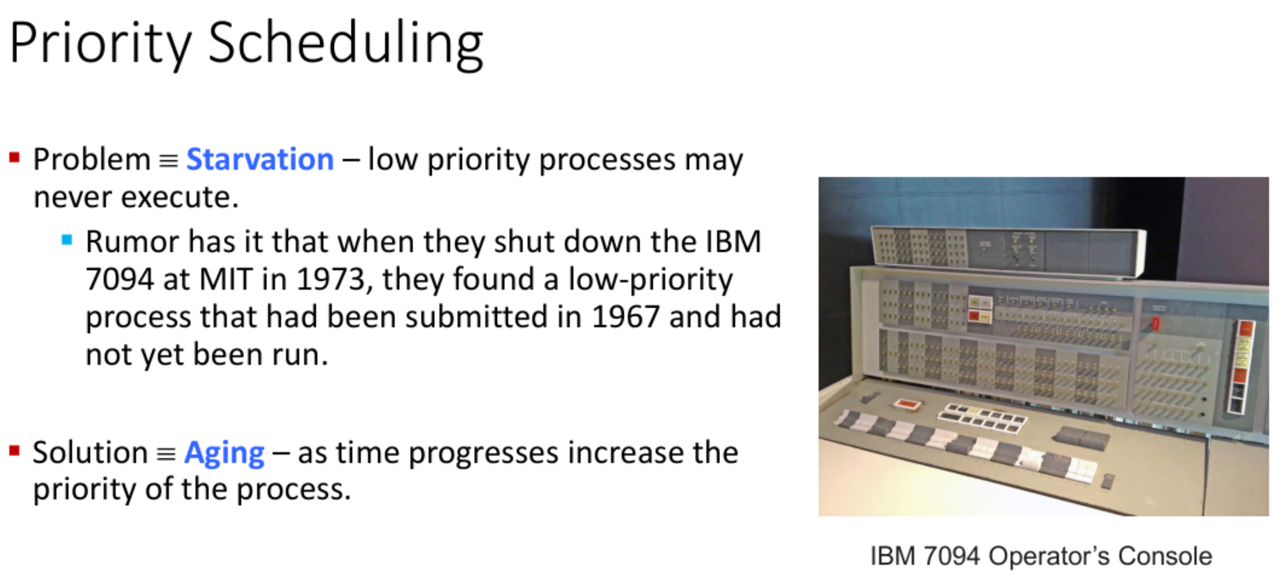
- Some systems use low numbers to represent low priority; others use low numbers for high priority.
- We assume that low numbers represent high priority.

Unix – RQ0 > RQ1
⓿

Windows – RQ1 > RQ0
❶

Priority Queuing

Linux and Windows use the same numbering system



Priority Scheduling

- Problem ≡ **Starvation** – low priority processes may never execute.
  - Rumor has it that when they shut down the IBM 7094 at MIT in 1973, they found a low-priority process that had been submitted in 1967 and had not yet been run.
- Solution ≡ **Aging** – as time progresses increase the priority of the process.

IBM 7094 Operator's Console

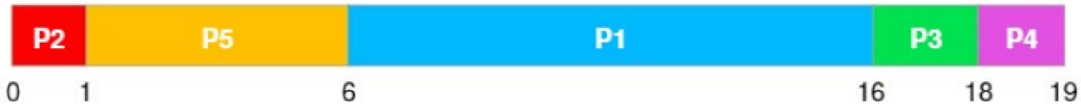
Starvation is a problem for both preemptive and nonpreemptive.

We up priority for old processes

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart



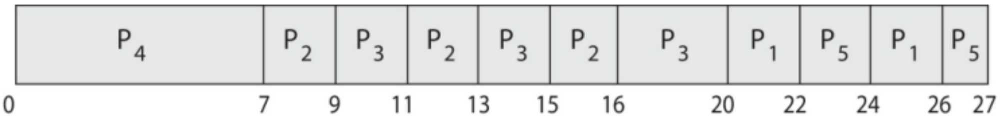
- Average waiting time = 8.2 msec (exercise).

Priority Scheduling w/ Round-Robin

- Run the process with the highest priority. Processes with the same priority run round-robin.

Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

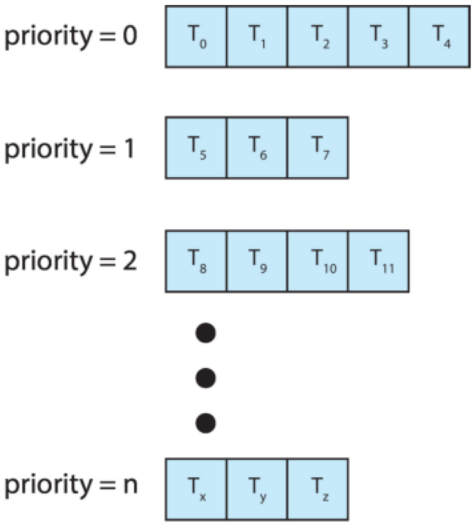
- Gantt Chart with 2 ms time quantum



A process can't be preempted for a process with a lower priority. If a process is the only process within its priority tier and there are no other higher priority processes then the process gets to run with impunity.

Multilevel Queue Scheduling

- With priority scheduling, have separate queues for each priority.
- Schedule the process in the highest-priority queue.



priority = 0 T_0 T_1 T_2 T_3 T_4

priority = 1 T_5 T_6 T_7

priority = 2 T_8 T_9 T_{10} T_{11}

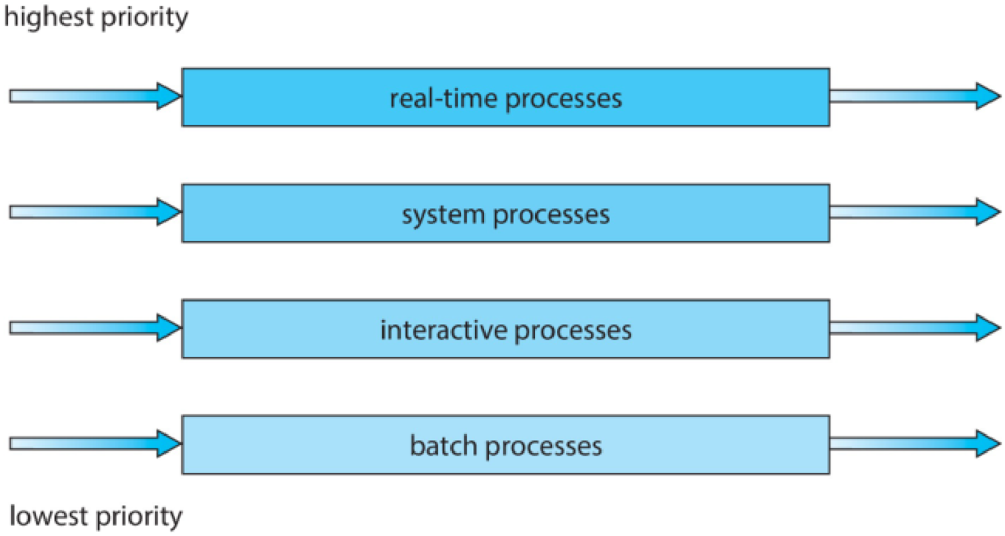
●
●
●

priority = n T_x T_y T_z

Separate queues for each priority.

Multilevel Queue Scheduling

- Prioritization based upon process type



highest priority

real-time processes

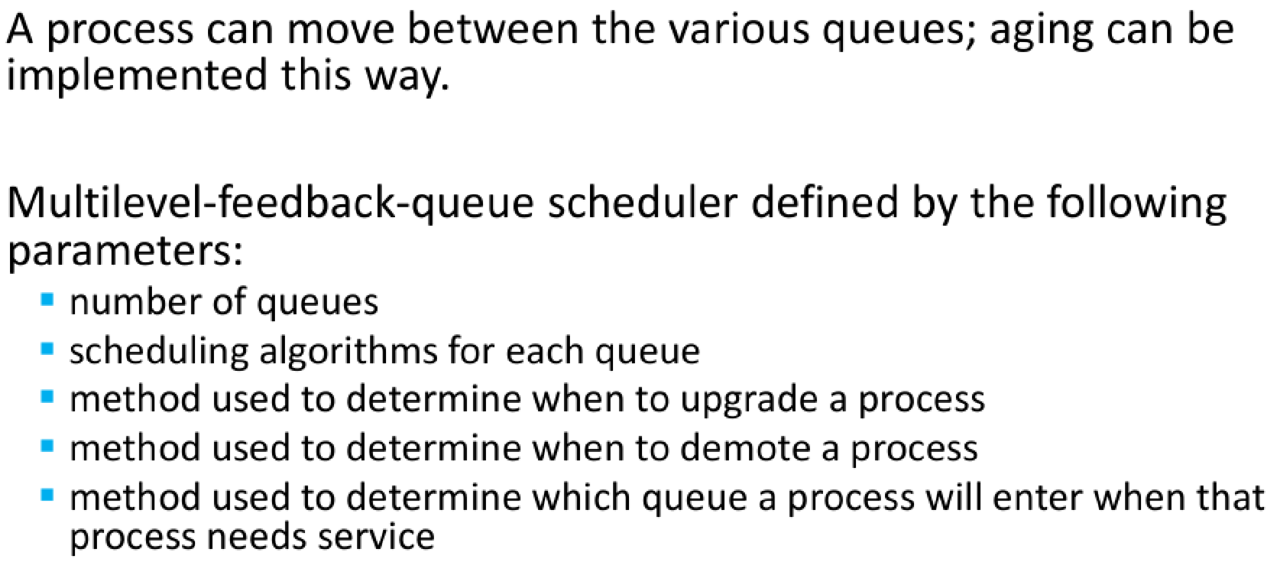
system processes

interactive processes

batch processes

lowest priority

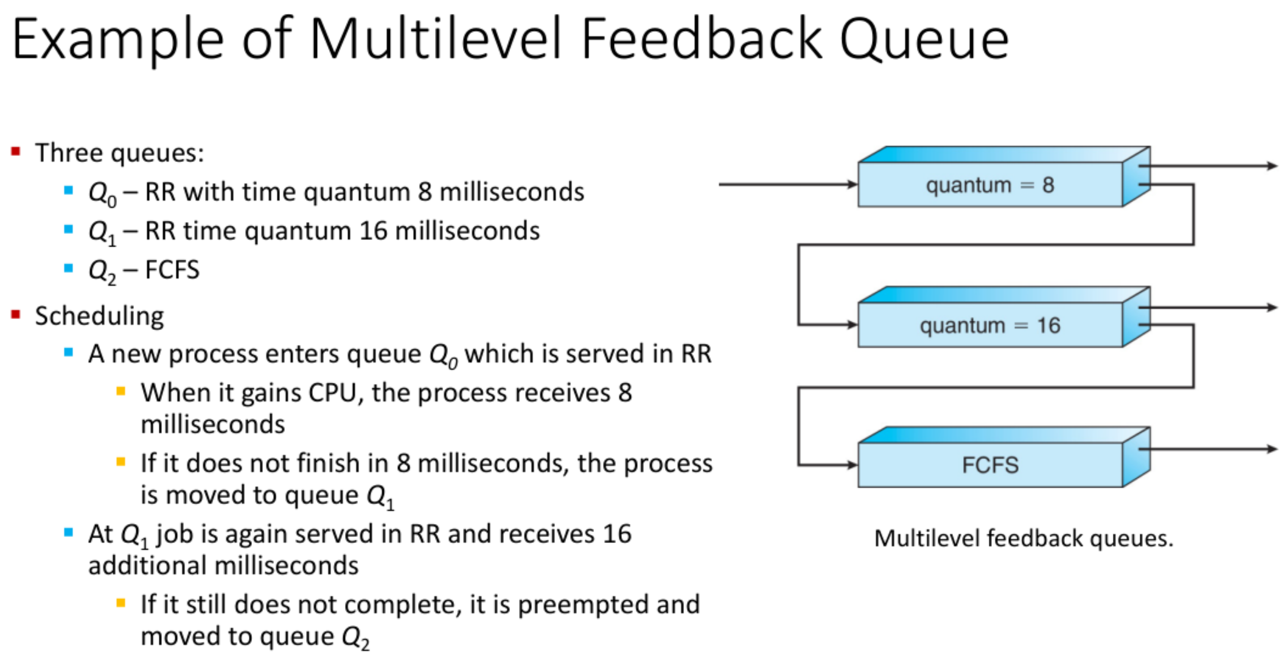
Multilevel queue scheduling.



A process can move between the various queues; aging can be implemented this way.

Multilevel-feedback-queue scheduler defined by the following parameters:

- number of queues
- scheduling algorithms for each queue
- method used to determine when to upgrade a process
- method used to determine when to demote a process
- method used to determine which queue a process will enter when that process needs service



Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new process enters queue Q_0 which is served in RR
    - When it gains CPU, the process receives 8 milliseconds
    - If it does not finish in 8 milliseconds, the process is moved to queue Q_1
  - At Q_1 job is again served in RR and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2

Multilevel feedback queues.

Aging is done via moving them do a different queue.

Knowledge Check

- ___ scheduling is approximated by predictin the next CPU burst with an exponential average of the measued lengths of previous CPU bursts
  - d. SJF
- The ___ scheduling algo is designed especially for time-sharing systems
  - c. RR
- which of the following is ture of multilevel queue scheduling?
  - b. each queue has its own scheduling algorithm
  - the most general CPU-scheduling algo is multilevel feedback
  - processes can only move between queues using the feedback variant

Multi-Processor Scheduling

CPU scheduling is more complex when multiple CPUs available.

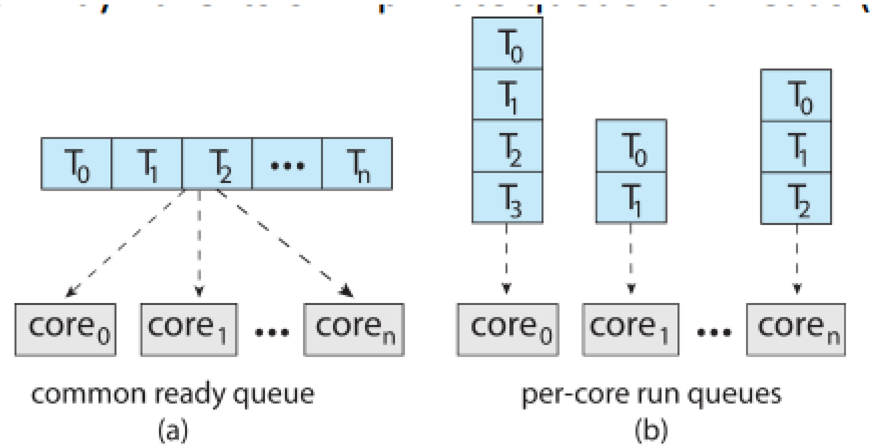
multiprocessor may be any one of the following architectures:

- multicore CPUs
- multithreaded cores
- NUMA systems
- heterogeneous multiprocessing

Multicore

Symmetric Multi Processing

- each processor is self scheduling
- all threads may be in a common ready queue
- each processor may have its own private queue of threads


T_0 T_1 T_2 ••• T_n
$core_0$ $core_1$ ••• $core_n$
T_0 T_1 T_2 T_3
T_0 T_1
T_0 T_1 T_2
$core_0$ $core_1$ ••• $core_n$

common ready queue
(a)

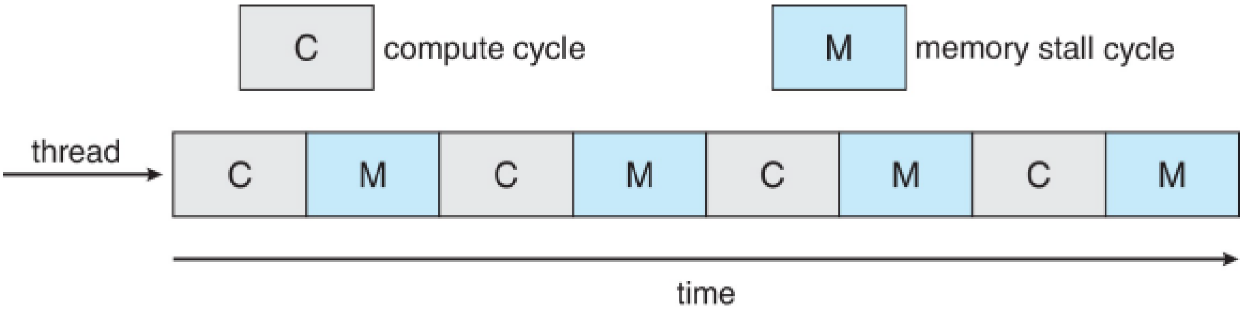
per-core run queues
(b)

Organization of ready queues.

Lately we've been making multicore chips that are faster and less power hungry.

Multithreaded cores are also a recent advancement we've made.

This takes advantage of memory stall to make progress on another thread while memory retrieve happens.



Memory stall.

What is Memory Stall?

When a processor acceses memory it waits around for the data to become available.

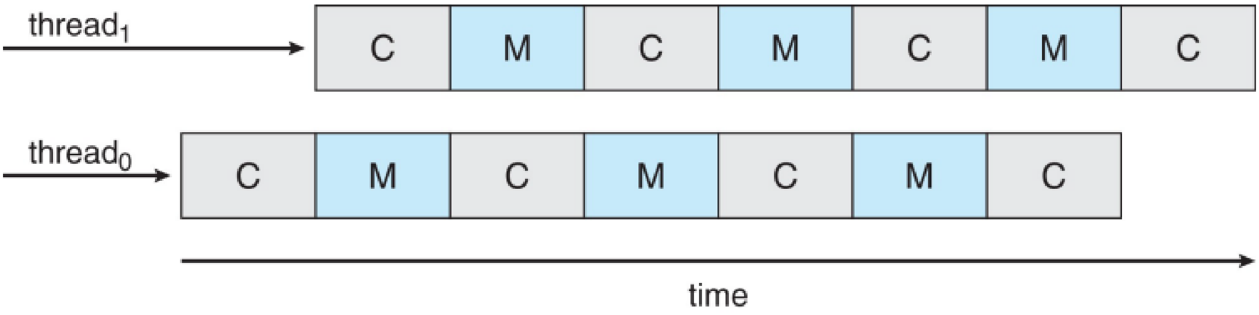
THe processor is much faster than the memory so it has to wait on the memory.

THis can also occur during due to a cache miss

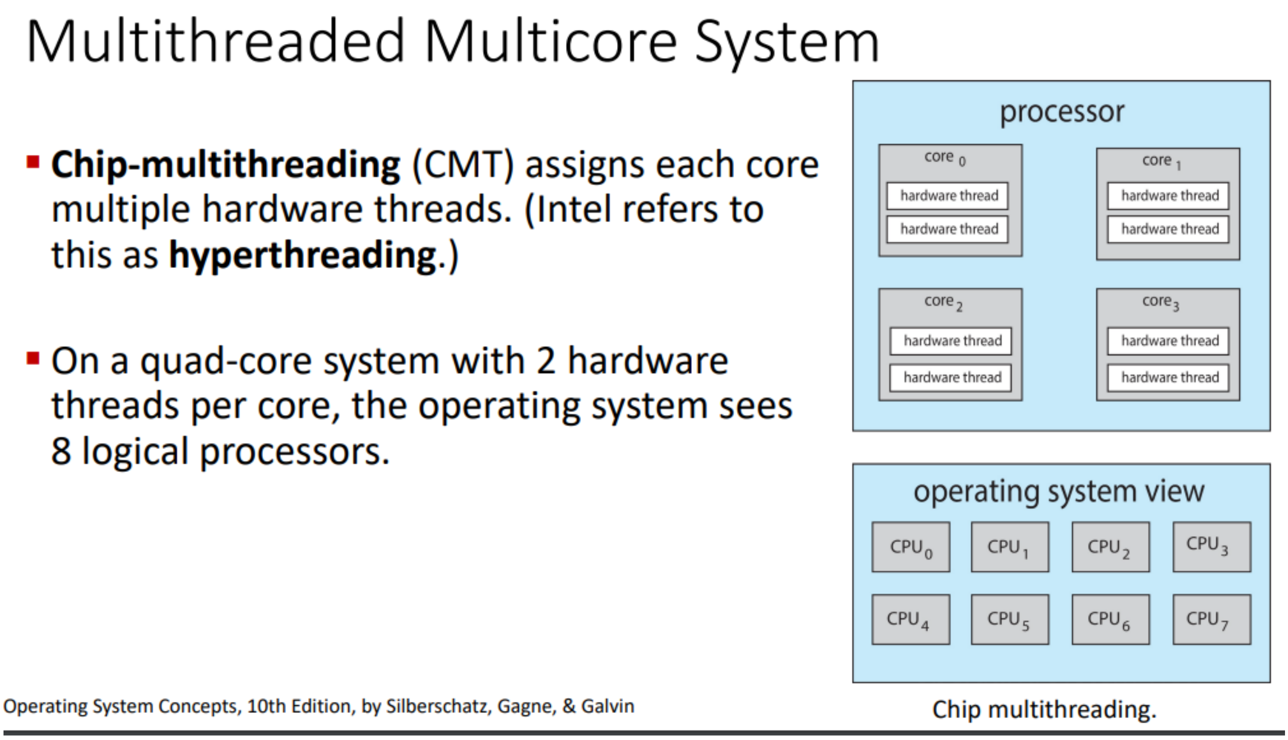
Multithreaded Cores

Each core as > 1 hardware thread

if one thread has amemory stall then we switch to another thread for 0 downtime.



Multithreaded multicore system.



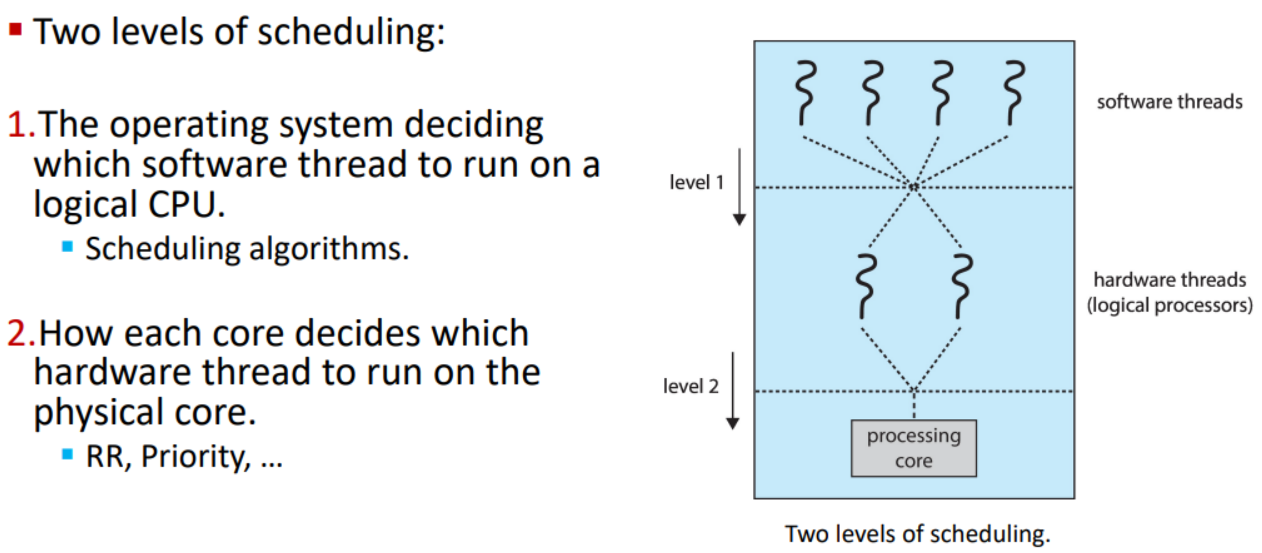
Multithreaded Multicore System

- **Chip-multithreading** (CMT) assigns each core multiple hardware threads. (Intel refers to this as **hyperthreading**.)
- On a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors.

Operating System Concepts, 10th Edition, by Silberschatz, Gagne, & Galvin

Chip multithreading.

each thread is basically treated as their own cpu from a logical perspective



- Two levels of scheduling:

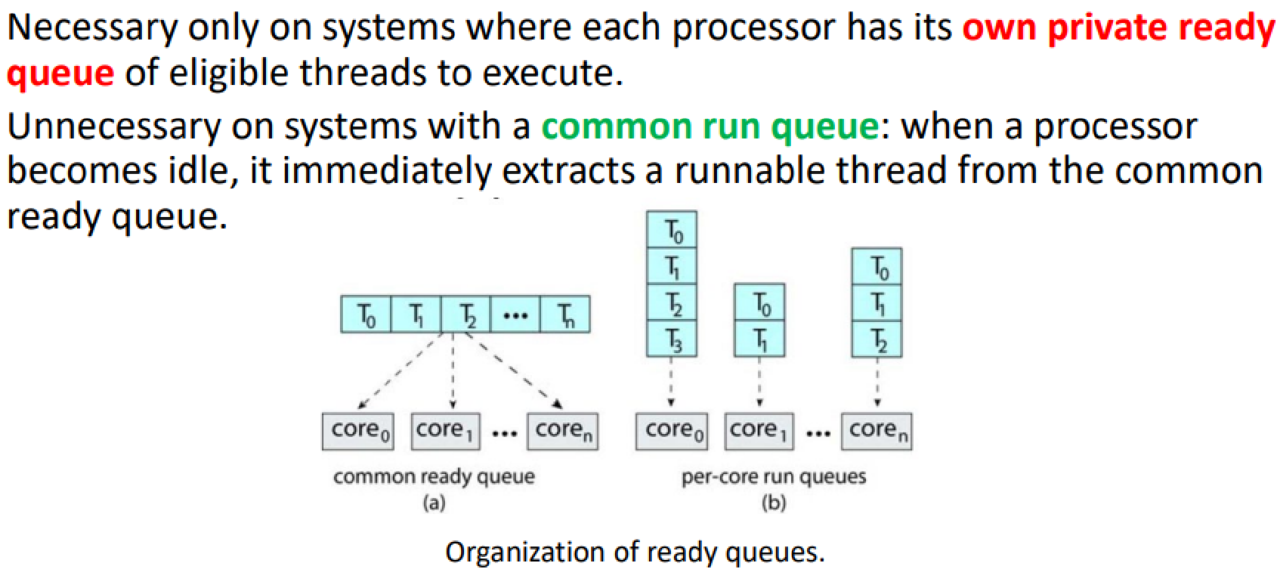
1. The operating system deciding which software thread to run on a logical CPU.
   - Scheduling algorithms.
2. How each core decides which hardware thread to run on the physical core.
   - RR, Priority, …

Two levels of scheduling.

choose a chip then choose a thread

Load Balancing

a concept for symmetric multiprocessing systems



Necessary only on systems where each processor has its **own private ready queue** of eligible threads to execute.

Unnecessary on systems with a **common run queue**: when a processor becomes idle, it immediately extracts a runnable thread from the common ready queue.

T_0 T_1 T_2 ••• T_n
core_0 core_1 ••• core_n
common ready queue
(a)
T_0 T_1 T_2 T_3
T_0 T_1
T_0 T_1 T_2
core_0 core_1 ••• core_n
per-core run queues
(b)

Organization of ready queues.

---

push migration

- periodic task checks load on each processor
- imbalance found → task pushed from overloaded CPU to other CPUs

pull migration

- idle processors pull a waiting task from a busy processor

Processor Affinity

When a thread has been running on one processor, the cache contents of that processor stores the memory accesses by that thread.

Successive memory accesses by thread are often satisfied in cache memory (aka “warm cache”).

```
Prof uses an example of repititive actions on a data structure, something like a for-
loop iterating through an array or spreadsheet.

We start to move things to the cache to speed things up
```

If load balancing causes a thread to migrate to another processor:

- contents of cache memory must be invalidated for the 1st processor
- cache for 2nd processor must be repopulated
- ***high cost***

This high cost is undesireable so OS and SMP try to avoid migrating a thread from one processor to another.

Proceesor Affinity:

- a process has an affinity for the processor on which it is currently running
- the warm cache is there for the process to take advantage of

Soft Affinity

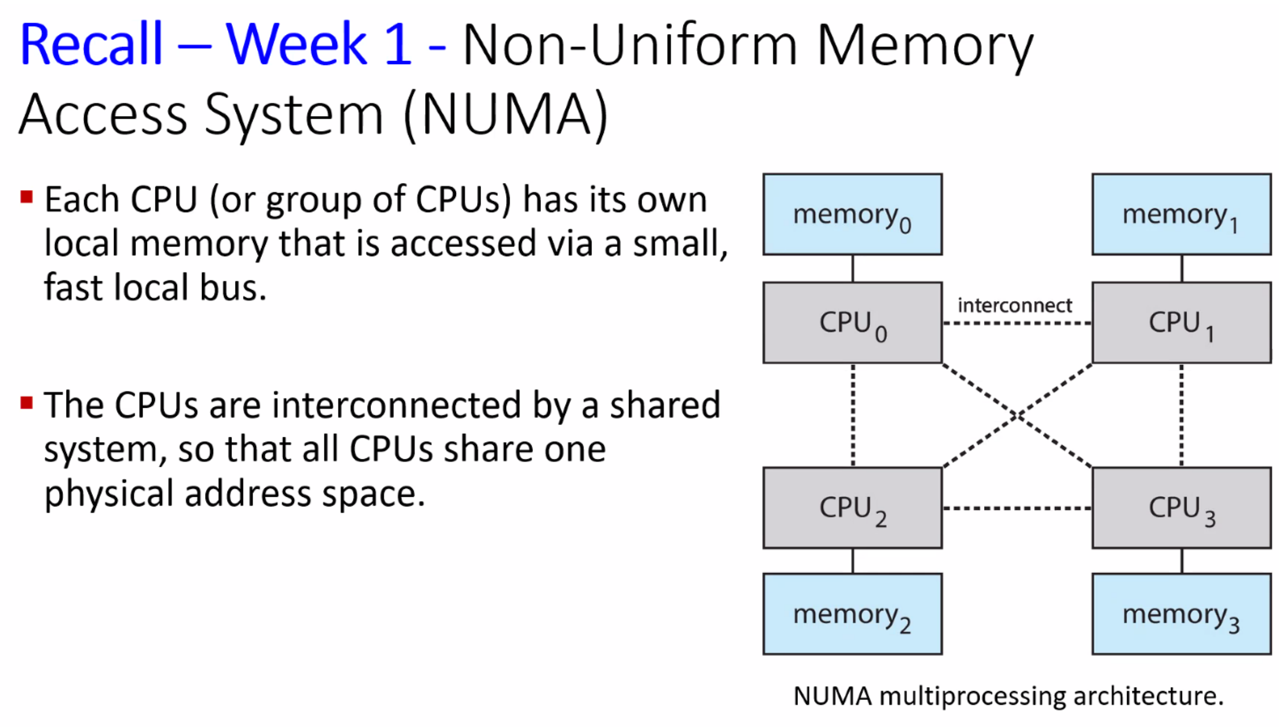
- OS attempts to keep a thread running on the same processor
- no guarantees

Hard Affinity

- allows a process to specify a set of processor it may run on

```
initially the cache is empty (usually almost all 16 kilobytes)

when the process starts it misses over and over again but then starts filling out the
cache

processor starts prepopulating the cache in order to help it out.
seeing what to remove and what to get rid of.

for hard affinity the process decides on a number of CPUs.

It can choose CPUs to benefit from a shared L2 cache
```

NUMA Systems



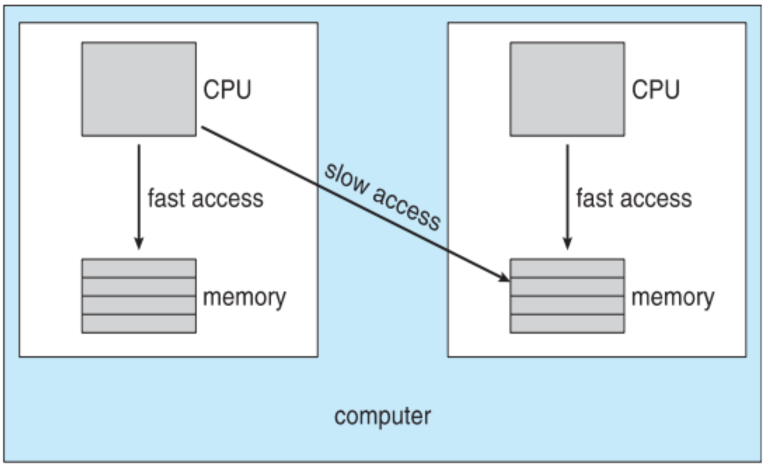
Recall – Week 1 - Non-Uniform Memory Access System (NUMA)

- Each CPU (or group of CPUs) has its own local memory that is accessed via a small, fast local bus.
- The CPUs are interconnected by a shared system, so that all CPUs share one physical address space.

memory$_0$
memory$_1$
CPU$_0$
interconnect
CPU$_1$
CPU$_2$
CPU$_3$
memory$_2$
memory$_3$

NUMA multiprocessing architecture.

If the operating system is NUMA-aware, then it will allocate memory closest to the CPU where the thread is running, thus providing the thread the fastest possible memory access -> processor affinity.



NUMA and CPU scheduling.

Hetereogeneous Multiprocessing

so far we have assumed all processors are identical so we can allow any thread to run on any processing core.

We only differed based on load balancing, processor affinity policies, and NUMA systems

Some mobile systems are now designed using cores that run the same instruction set but vary in terms of their clocks peed and power management.

They can also adjust the power draw of a core to the point of idling.

These are known as Heterogeneous Multiprocessing (HMP).

This type of architecture is known as `big.LITTLE`:

- `big` cores are higher-performance cores
  - greater power draw → used for short periods of time
  - interactive tasks
- `LITTLE` are energy efficient cores
  - use less energy → used for longer periods of time
  - background tasks

Windows 10 supports HMP scheduling by allowing a thread to select a scheduling policy that best supports its power management

---

knowledge check

- ___ allows a thread to run on only one processor
  - a. processor affinity
- two general approaches to load balancing are ___ and ___
  - d. push migration, pull migration
  - course grained is when a process keeps executing on a core until there's a huge stall event
  - fine grained is when we are switching on a very fine level of granularity between processes
    - the boundary of an instruction cycle
      - fetch and execute then now we switch to another thread
  - each comes with their pros and cons with relation to process switching

Real-Time CPU Scheduling

can present special issues

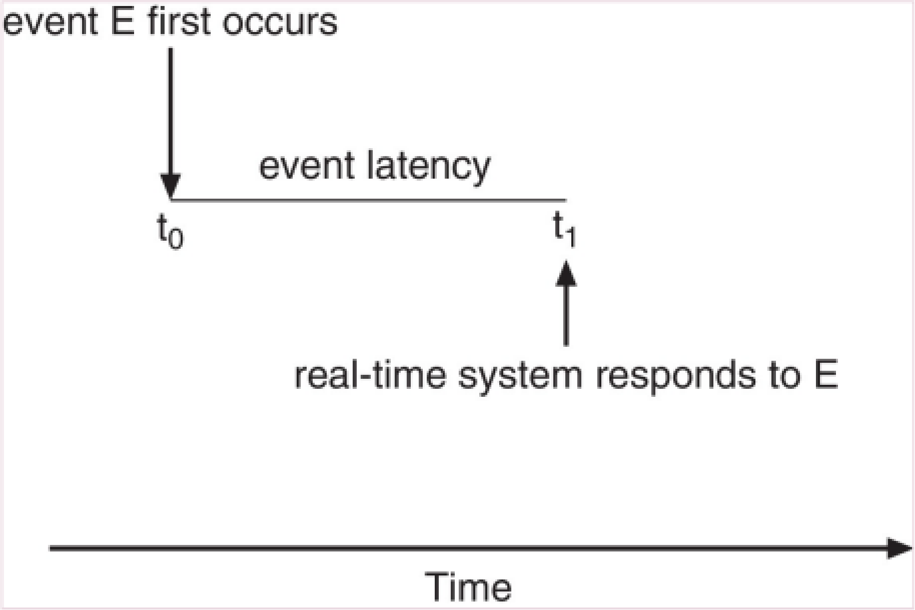
2 types

- soft real-time systems
  - critical real-time tasks have the highest priority
  - no guarantee as to when tasks will be scheduled
- hard real-time systems
  - task must be serviced by its deadline
  - service after deadlin has expired
    - same as no service at all

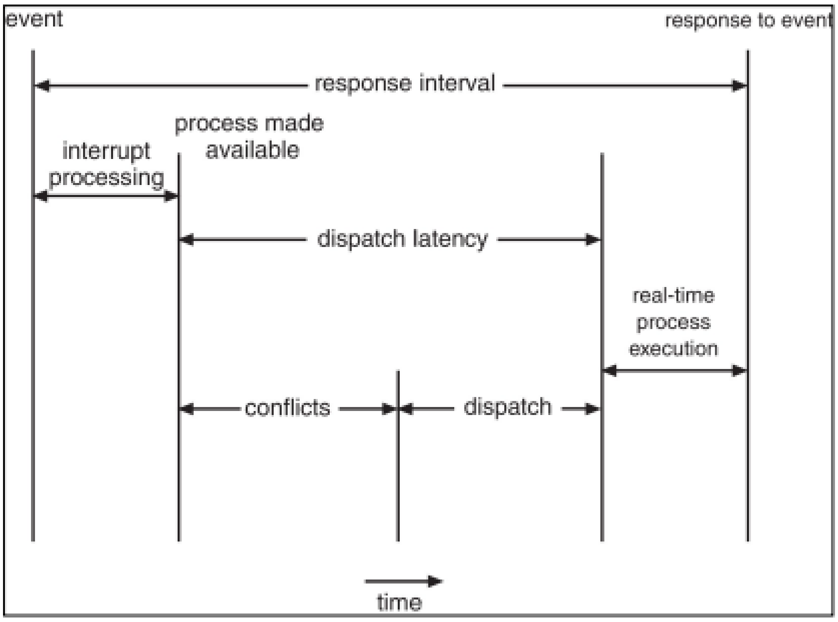
event latency

- amount of time that elapses from when an event occurs to when it is serviced
- 2 types that affect performance
  - interrupt
  - dispatch

    - time for schedule to take current process off CPU and switch to another



Event latency.



Dispatch latency.

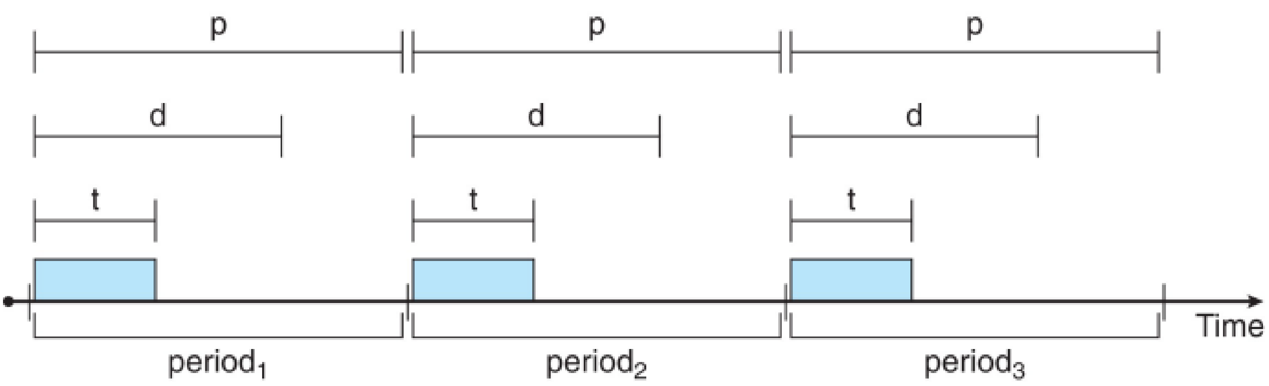
conflict phase of dispatch latency:

1. preemption of any process running in kernel mode
2. release by low-priority process of resources needed by high-preiority processes

Priority Based Scheduling

for soft real-time scheduling, scheduler must support preemptive, priority-based scheduling.

For hard real-time we must also provide the ability to meet deadlines which requires additional scheduling features.



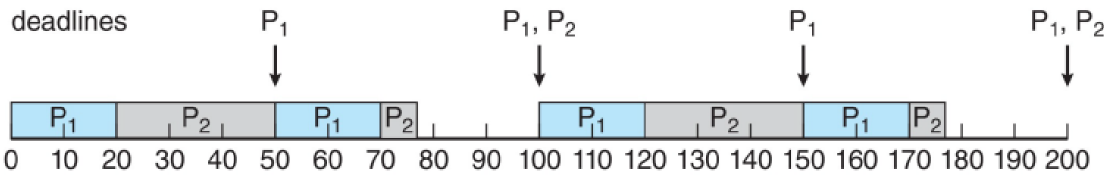
Periodic task.

Processes have new characteristics: **periodic** ones require CPU at constant intervals.

- Has a fixed processing time ***t***, deadline ***d*** by which it must be serviced by, and a period ***p***.
- $0 \le t \le d \le p$
- **Rate** (aka **frequency**) of periodic task is $1/p$
- **Utilization (u)** = t/p

Rate Monotonic Scheduling (RMS)

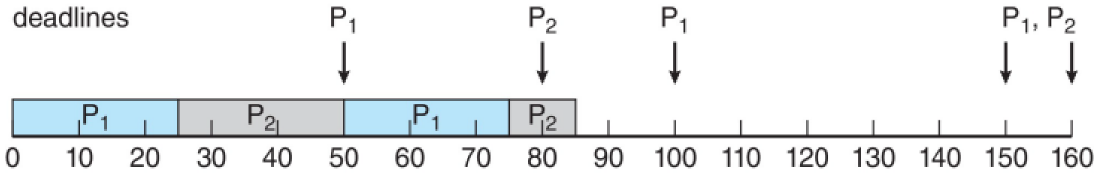
- A priority is assigned based on the inverse of its period.
  - Shorter periods = higher priority;
  - Longer periods = lower priority
- Example:
  - $P_1=50$, $t_1=20$, $P_2=100$, $t_2=35$
  - P_1 is assigned a higher priority than P_2 as it has the shorter period.
  - $u_1=20/50 = 0.4$, $u_2=35/100 = 0.35 \rightarrow u_t = 0.75$



Rate-monotonic scheduling.

Missed Deadlines with Rate Monotonic Scheduling

- P_1=50, t_1=25, P_2=80, t_2=35
- P_1 is assigned a higher priority than P_2 as it has the shorter period.
- u_1=25/50 = 0.5, u_2=35/80 = 0.44 → $\mathbf{u_t = 0.94}$



Missing deadlines with rate-monotonic scheduling.

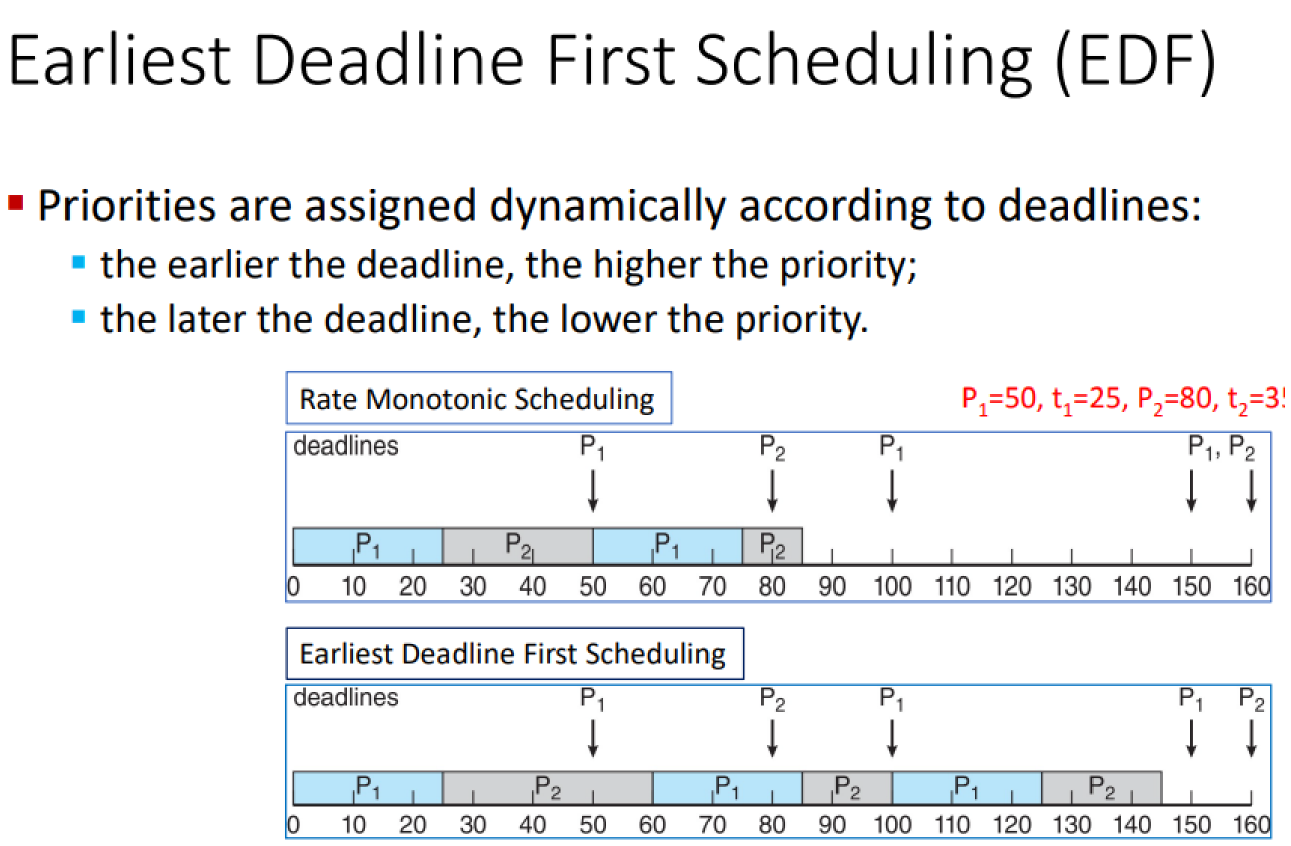
- Process P2 misses finishing its deadline at time 80.

Missed Deadlines with Rate Monotonic Scheduling

- Max utilization when having N processes in the system:

$$N(2^{1/N} - 1)$$

- With two processes, CPU utilization is bounded at about 83%.
- With one process in the system, CPU utilization is 100%, but it falls to approximately 69% as the number of processes approaches infinity.



Earliest Deadline First Scheduling (EDF)

- Priorities are assigned dynamically according to deadlines:
  - the earlier the deadline, the higher the priority;
  - the later the deadline, the lower the priority.

---

knowledge check

- the reate of a periodic tasks in a hard real-time system is ___, where p is a period and t is the processing time.
  - a. 1/p
- which of the following is true of the rate-monotonic scheduling algo?
  - a is for first scheduling
  - rate-monotonic scheduling does not have a dynamic priority policy
  - it has to be preemptive
  - answer: c. cpu utilization is bounded when using this algo
- which of the following is true of earliest-deadline-first (EDF) scheduling algo?
  - the rule for `b` is reversed
  - priorities aren't assigned statically
  - priorities are not fixed
  - answer: a. when a proces becomes runnable, it must announce its deadline

Algo Eval

How to select CPU-scheduling algo for an OS?

Determine criteria, then eval algos

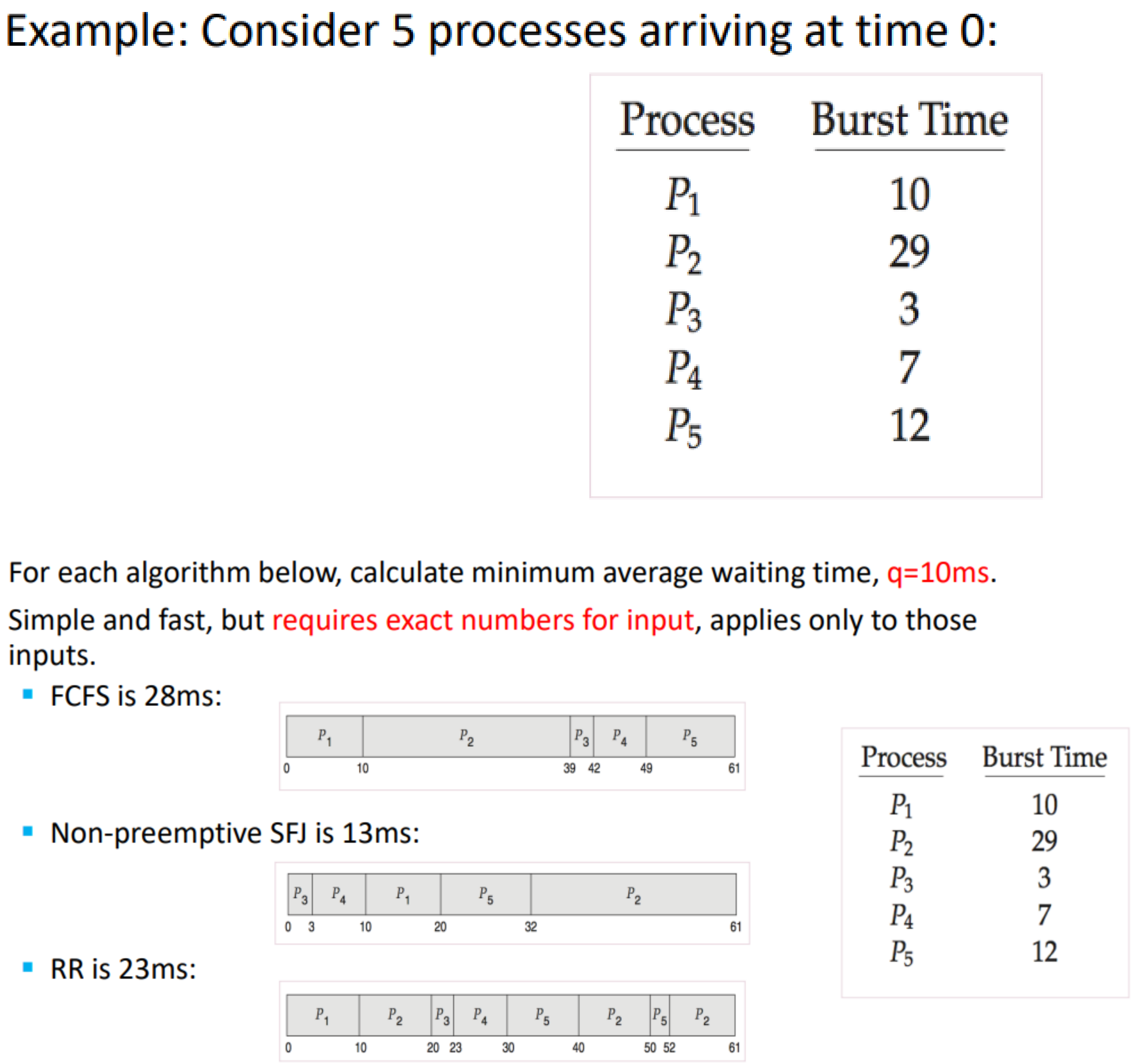
example of criteria:

- maximizing utilization under the constratint that the maximum response time is 300 milliseconds
- maximizing throughput such that turnaround time is (on average) linearly proportional to total execution time

Deterministic

Deterministic Modelling:

- type of analytic evaluation
- takes a particular predetermined workload and defines the performance of each algo for that workload



Example: Consider 5 processes arriving at time 0:

Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

For each algorithm below, calculate minimum average waiting time, q=10ms.

Simple and fast, but requires exact numbers for input, applies only to those inputs.

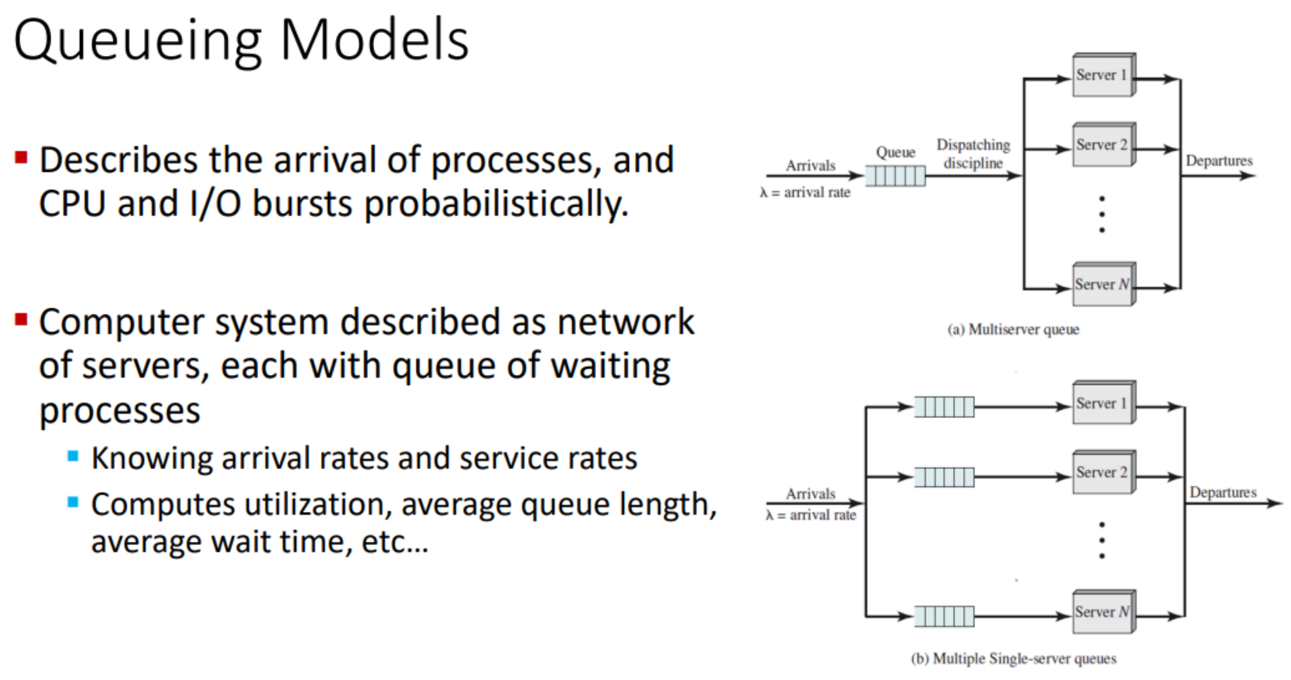
- FCFS is 28ms:

- Non-preemptive SFJ is 13ms:

- RR is 23ms:

Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Queueing Models

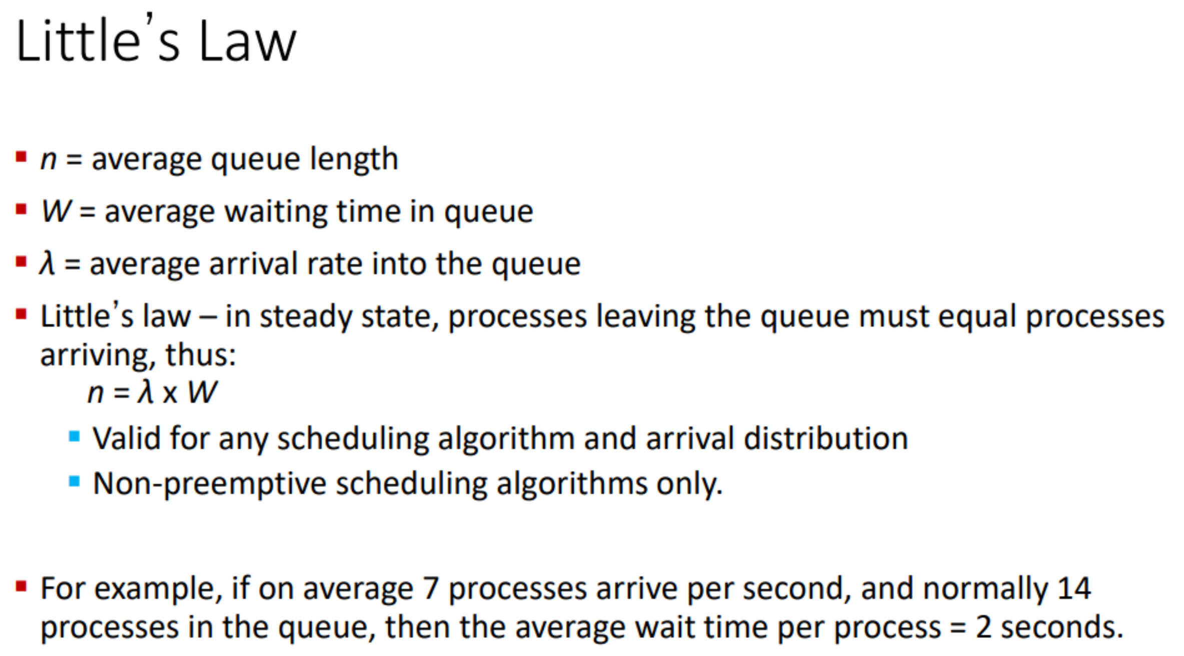


Queueing Models

- Describes the arrival of processes, and CPU and I/O bursts probabilistically.
- Computer system described as network of servers, each with queue of waiting processes
  - Knowing arrival rates and service rates
  - Computes utilization, average queue length, average wait time, etc…

(a) Multiserver queue

(b) Multiple Single-server queues



Little's Law

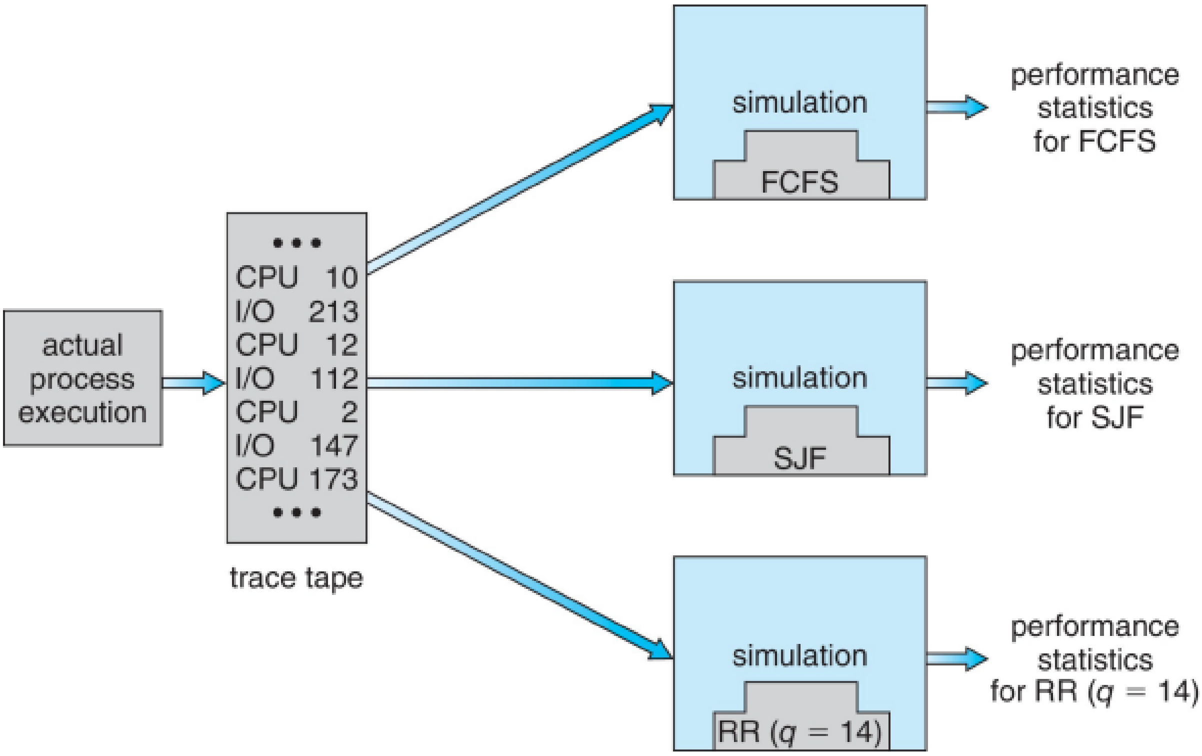
- n = average queue length
- W = average waiting time in queue
- λ = average arrival rate into the queue
- Little's law – in steady state, processes leaving the queue must equal processes arriving, thus:
  $n = \lambda \times W$
  - Valid for any scheduling algorithm and arrival distribution
  - Non-preemptive scheduling algorithms only.
- For example, if on average 7 processes arrive per second, and normally 14 processes in the queue, then the average wait time per process = 2 seconds.

Simulations

Queueing models are complicated and limited.

Simulations give more accurate evlaution of scheduling algos:

- programmed model of computer system
- clock is a variable
- gather stats indicating algo performance
- data to drive sim gathered via:
  - rng according ot probabilities
  - dists defined mathematically or empirically
  - trace files record sequences of real events in real systems



`I did not fucking hear anything she said about this`

Real Implementation

The best way is to just implement this stuff for real.

This is very costly.

In general, most flexible scheduling algos are those that can be altered by the system amangers or by the users so that they can be tuned for a specific (set of) app(s)

Envionments can vary however.

Knowledge Check

- in little's formula, λ, represents the ___:
  - b. average arrival time for new processes in the queue
-