Topics: Process (CPU) Scheduling
(SGG 5.1-5.3 (if time permits 5.4-5.5))

CS 3733 Operating Systems

Instructor: Dr. Turgay Korkmaz
Department Computer Science
The University of Texas at San Antonio

Office:	NPB 3.330
Phone:	(210) 458-7346
Fax:	(210) 458-4437
e-mail:	korkmaz@cs.utsa.edu
web:	www.cs.utsa.edu/~korkmaz

These slides are prepared based on the materials provided by Drs. Robbins, Zhu, and Liu. Thanks to all.

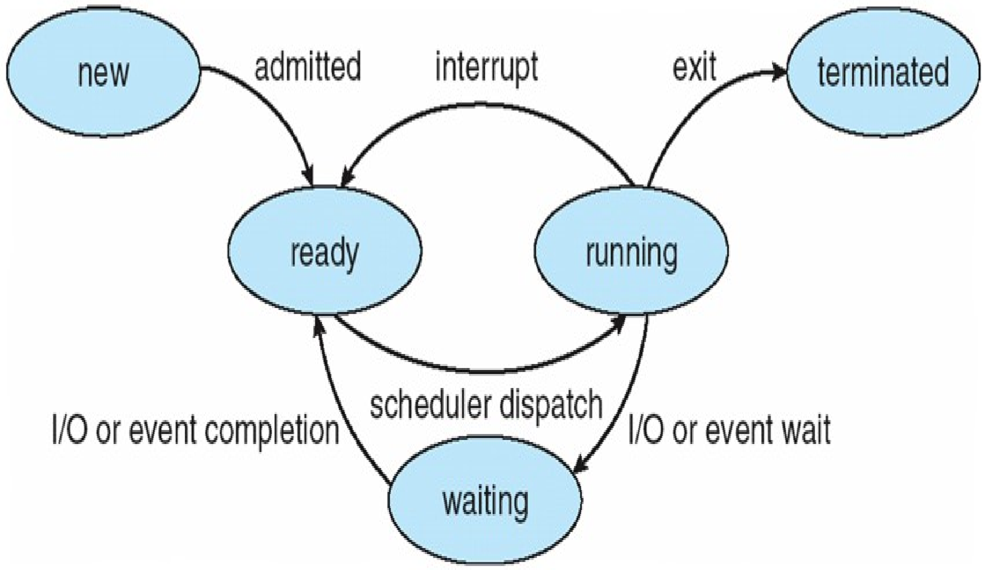
Outline

- Reviews on processes and their states
- Process queues and scheduling events
- Different levels of schedulers
  - CPU Scheduler
  - Preemptive vs. non-preemptive
  - Context switches and dispatcher
  - Models and assumptions for CPU scheduling
  - CPU-bound and IO-bound processes
- Performance criteria
  - Fairness, efficiency, waiting time, response time, throughput, and turnaround time;
- Classical schedulers: FIFO, SFJ, RR, and PR
- CPU Gantt chart vs. process Gantt charts

Reviews

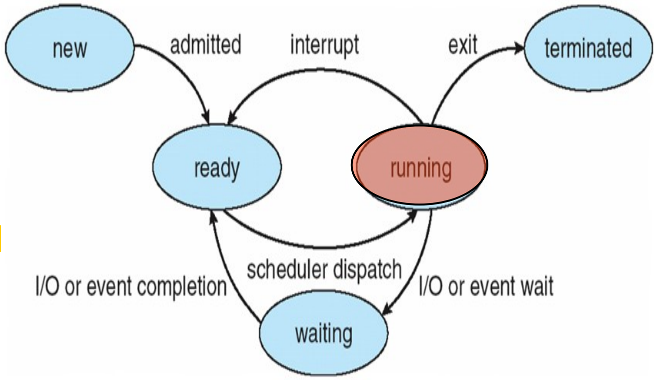
- Process
  - Execution of program

- States of Process
  - **New** and **terminated**
  - **Running (R)** ☾ using CPU
  - **ready (r)** ☾ in memory, ready for the CPU
  - **waiting (w)** ☾ waiting for I/O device or interrupt



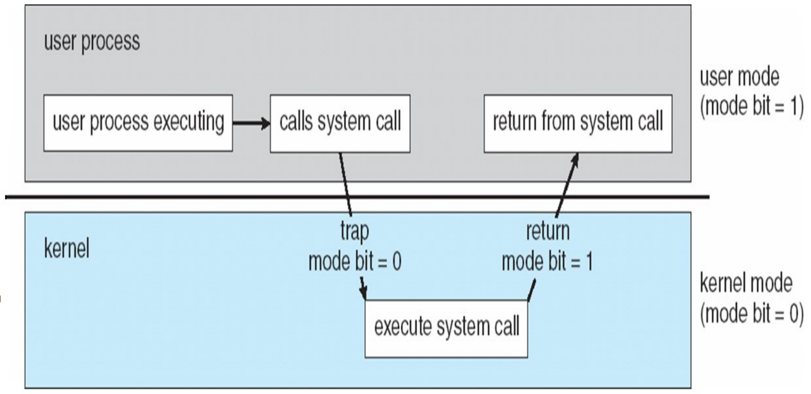
What may cause a process to move out of the CPU?

When to move out of CPU?

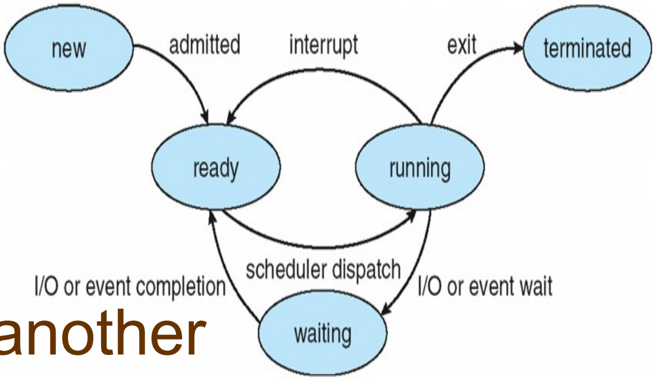


- I/O request
  - Need to read/write data from/to a file (on disk)
- Invoke a system call (e.g., **`exit, fork, wait)`**
  - **`fork()`** creates a new process
  - **`fork()`** returns to both parent/child processes with different return values;
    - ✓ Parent gets child process ID while Child gets 0;
- An interrupt or signal
  - Timer interrupt (quantum time)
  - Signal
  - Other synchronization like wait/joir

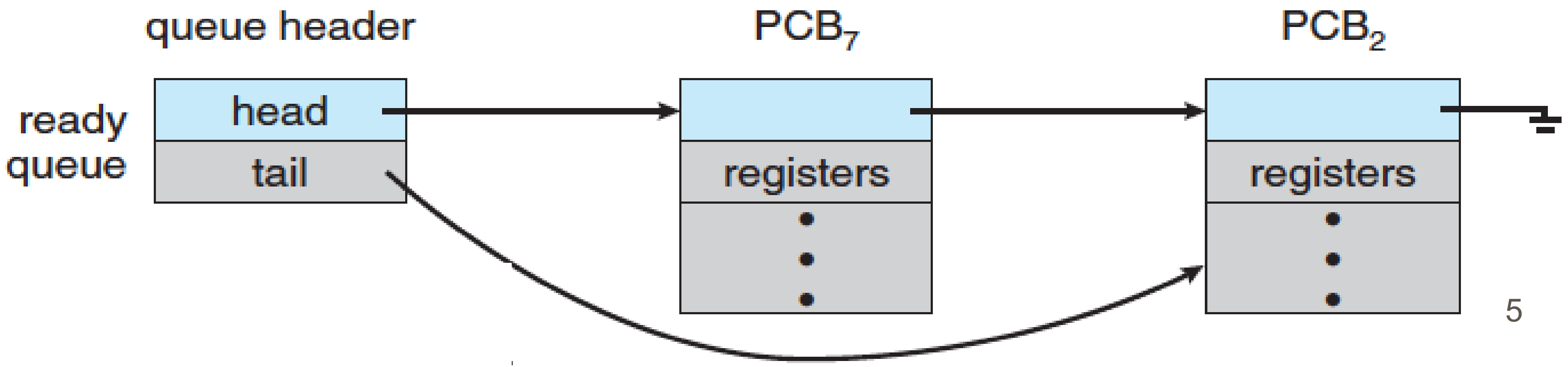
Not every system call results in context-switch



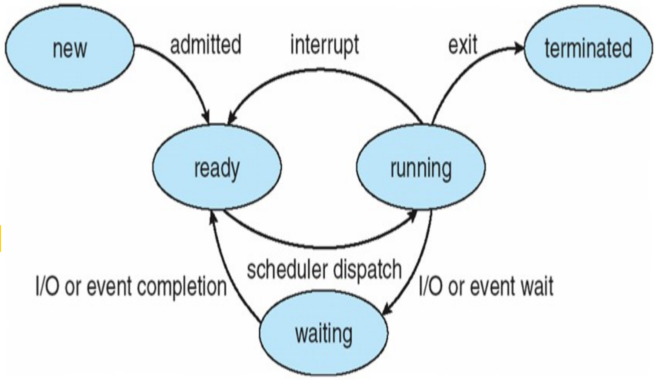
Process Queues



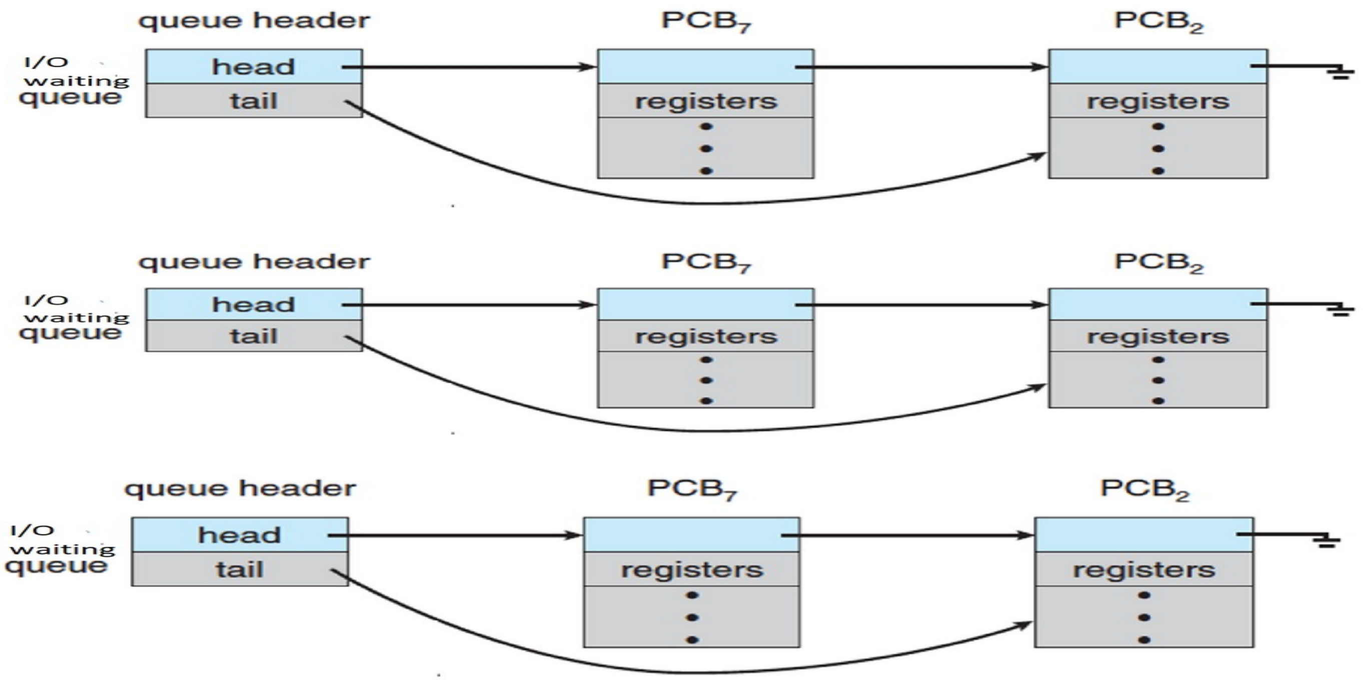
- Process state transition from one queue to another
- **Job queue** (before process gets into main memory)
  - All processes in mass storage (disk) waiting for allocation of memory (non-demand-paging system)
- **Ready queue**
  - In main memory waiting for the CPU
  - Usually a **linked list** is used to manage PCBs of all processes in the ready queue



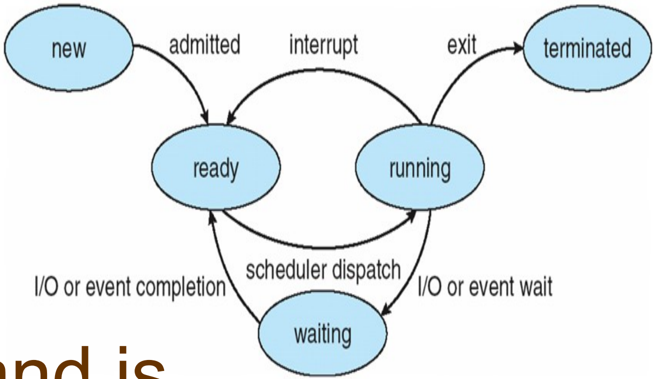
Process Queues (cont.)



- Device queues
  - One for each device containing all processes waiting for it
  - Disk drives, tape drives, and terminals
  - Shareable devices (disk drives): may have processes
  - Dedicated devices (tape drives): have at most one process



Process Queues (cont.)



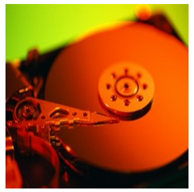
- Once the process is allocated the CPU and is executing, one of several events could occur:
  - I/O requests (e.g., read/write data from/to files)
  - System calls (e.g., create a new child process and wait for it)
  - Interrupts (forcibly removed from the CPU)
- A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

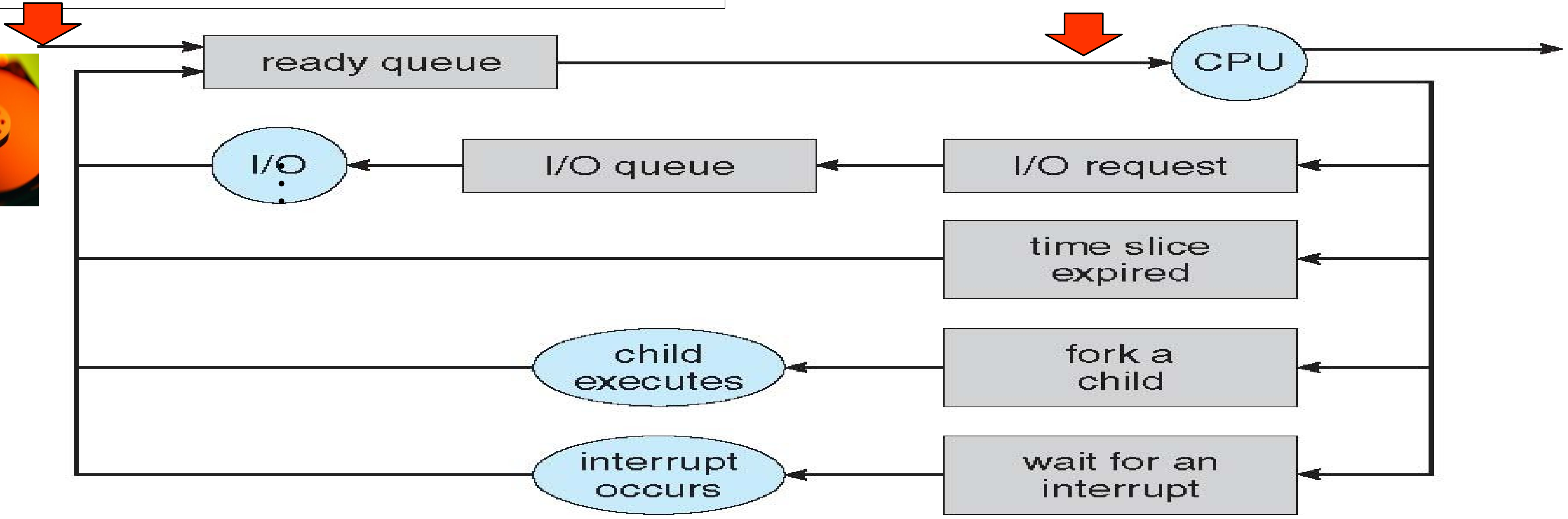
Which process to select next?

Different Levels of Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
  - Less frequent
  - Controls degree of multiprogramming

- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU
  - More frequent (e.g., every 100 ms)
  - Must be fast (if it takes 10ms, then we have ~10% performance degradation)



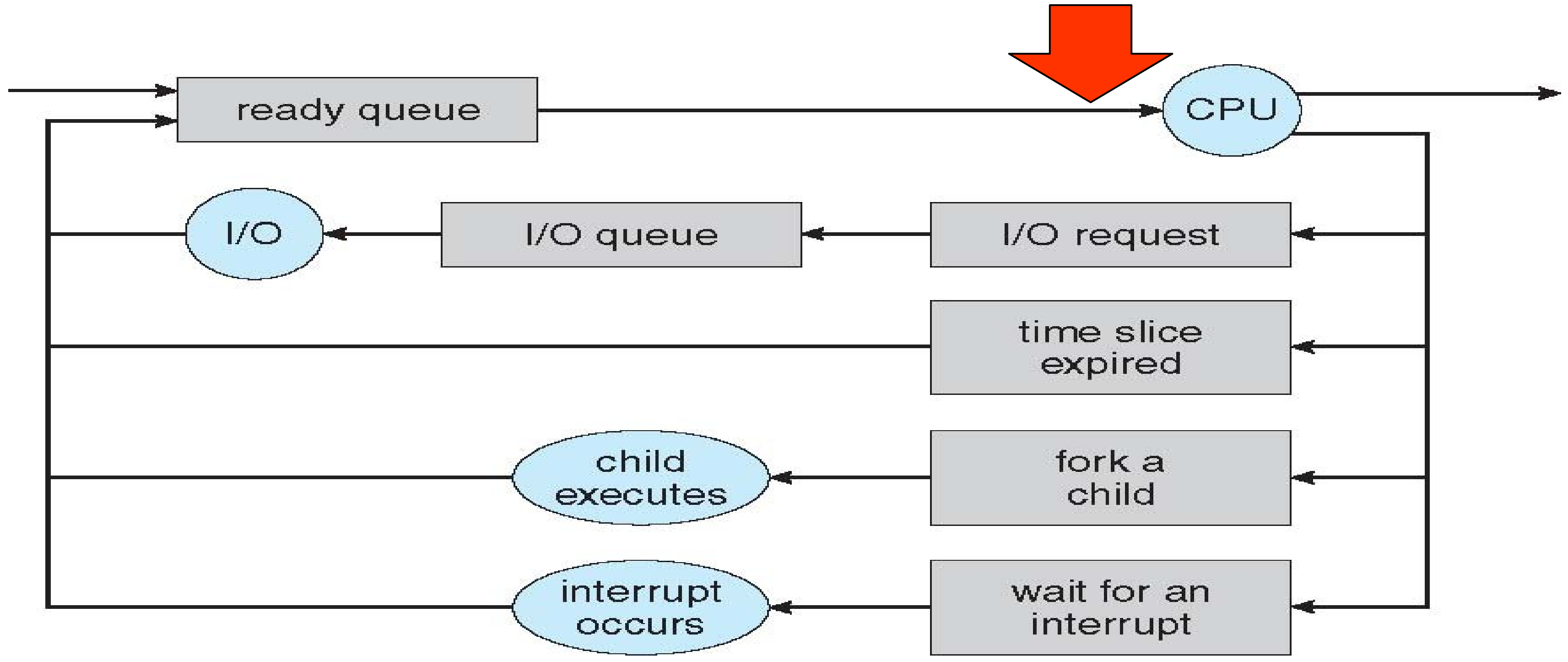


Different Levels of Schedulers

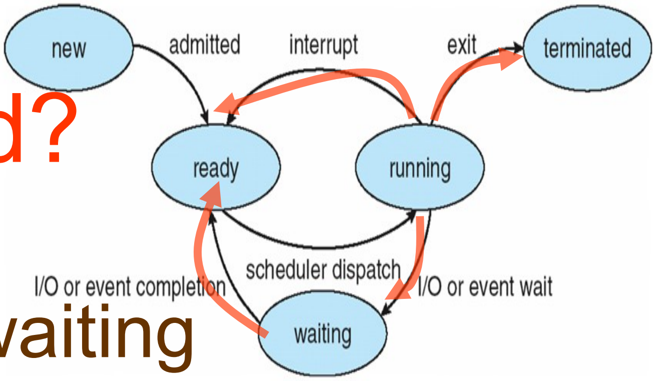
- **Short-term (CPU) scheduler**
  - **Selects which process from ready queue(s);**
  - **Must operate frequently and fast, several times a second**
- Medium-term scheduler (for time-sharing systems)
  - Swapping --- moving processes in and out of memory
  - Too many ☾ lots of paging with decreased performance
- Long-term (job) scheduler
  - Decide which processes are admitted to the system
  - Determines the **degree of multiprogramming**
  - In stable conditions: invoke only when a process terminates
  - May take long time; for batch systems; may not be good for time-sharing systems;

CPU Scheduler

- A process is migrated among various queues
- Operating system must select (schedule) processes from these queues in some fashion.
  - The selection process is called as "Scheduler".



When is CPU Scheduler Invoked?



- 1. Process switches from running to waiting
  - Requests for I/O operations
- 2. Process switches from waiting to ready
  - I/O completed and interrupt
- 3. Process switches from running to ready
  - Due to timer interrupt
- 4. Process terminates
  - When a process complete its work

Non-Preemptive vs. Preemptive

- Non-preemptive scheduling: **voluntarily give up CPU**
  - The process having the CPU uses it until finishing or needing I/O
  - Not suitable for time-sharing
  - Only IO (case 1) and process termination (case 4) can cause scheduler action

- Preemptive scheduling: **system forcibly gets CPU back**
  - Process may be taken off CPU **non-voluntarily**
  - Time-sharing systems have to be **preemptive**!
  - Both occasions 2 and 3 may cause scheduler action

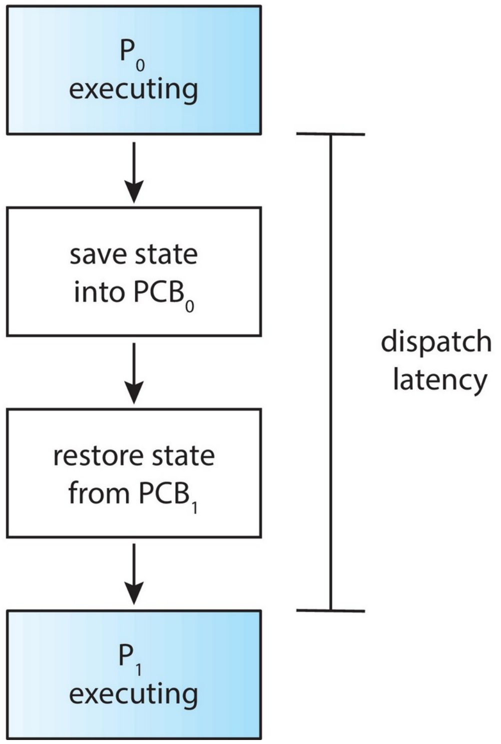
May result in race conditio

- Virtually all modern operating systems including Windows, MacOS, Linux, and UNIX use preemptive scheduling algorithms.

Dispatcher and Context Switch

- **Dispatcher**: gives control of CPU to selected process
  - **Context switch**
  - Setting the program counter (PC), and …
- **Context Switch**: switch CPU to another process
  - Save **running state** of the old process: **where to save** ?
  - Load the **saved state** of the new process: **from where** ?
  - A few microsecond to 100's of microseconds depending on **hardware support**

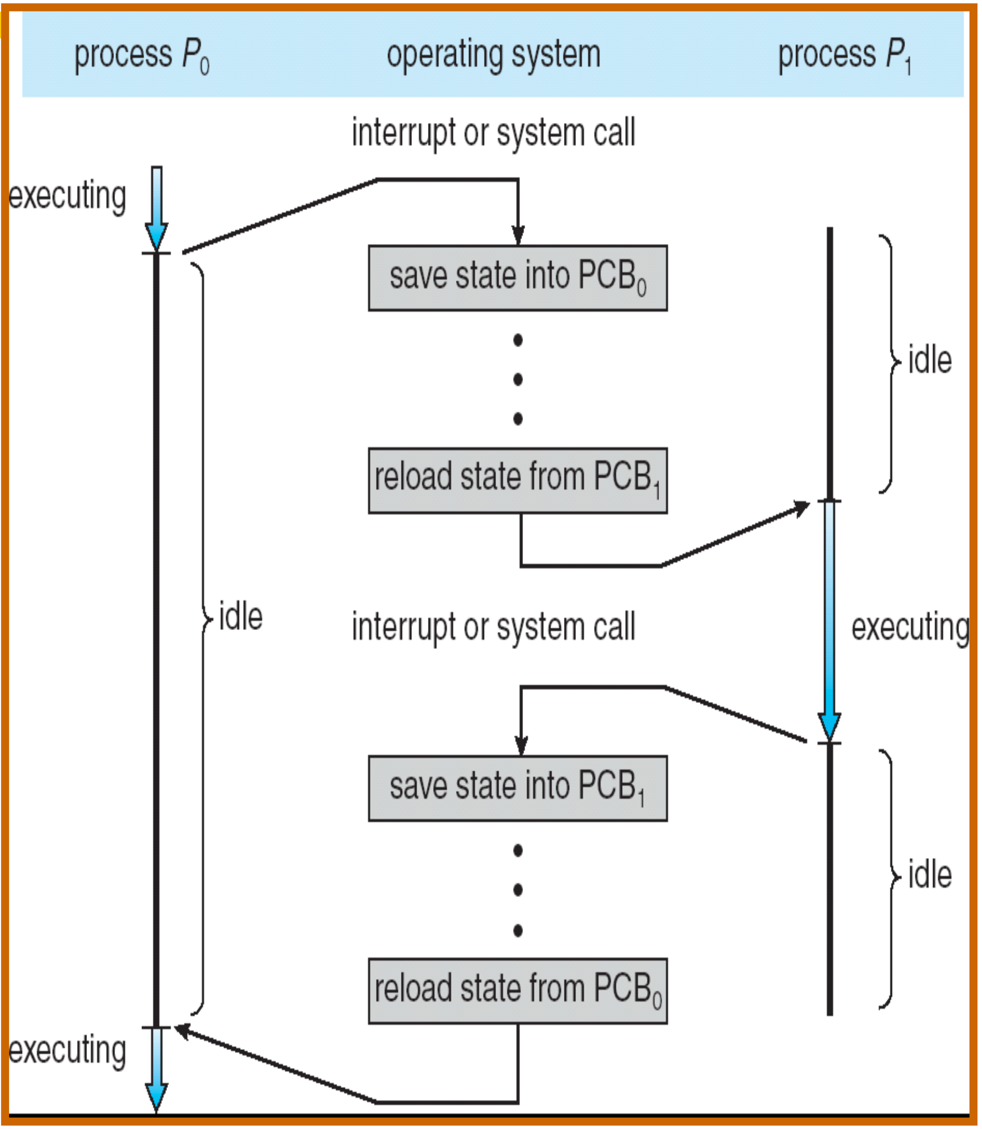
PBC



Context Switch (recall)

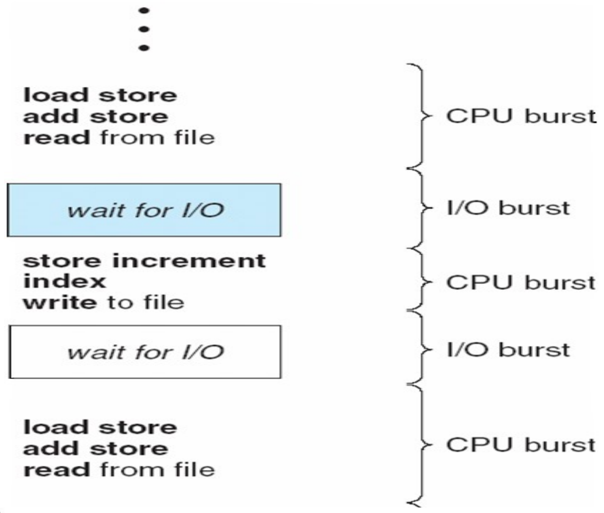
Not every system call results in context-switch

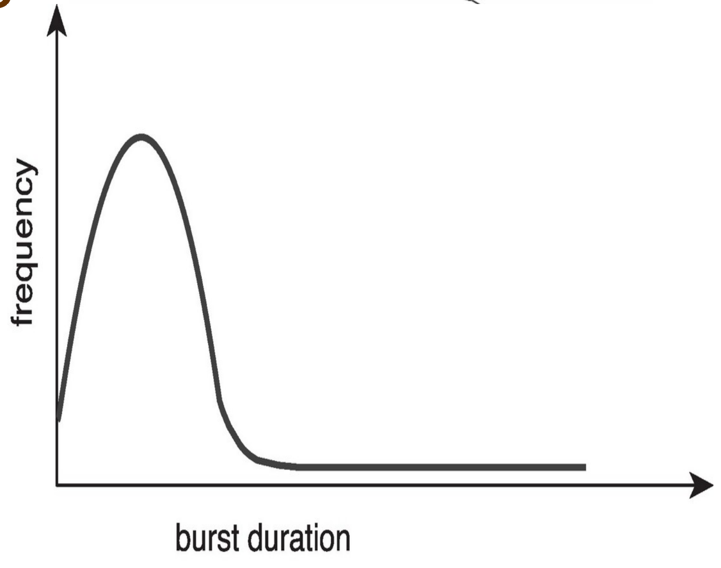
- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is **overhead**; the system does no useful work while context switching
- Hardware support
  - Multiple set of registers then just change pointers
- Other performance issues/problems
  - Cache content: locality is lost
  - TLB content: may need to flush

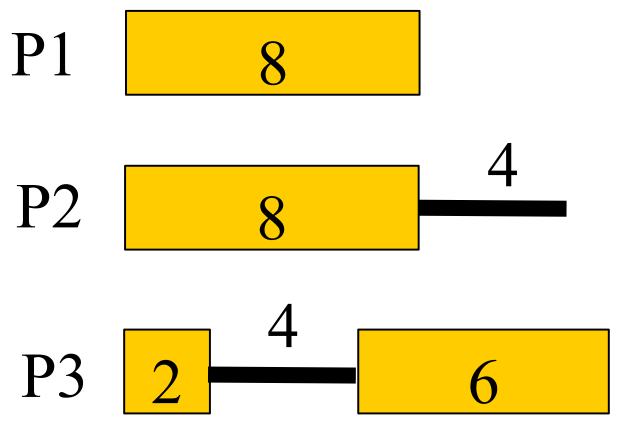


Process Model for CPU Scheduling

- Model of Process
  - Cycle of (interleaving) CPU and I/O operations
- **CPU bursts**: time to use CPU
- **I/O bursts**: time to use I/O devices

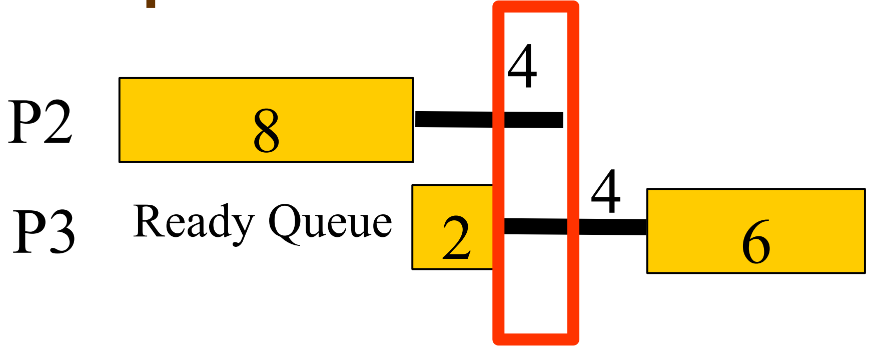






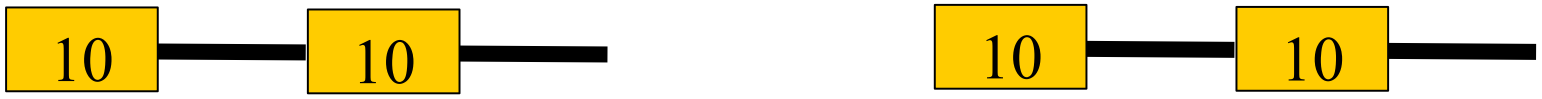
Assumptions for CPU Scheduling

- CPU model
  - By default, assume **only a single CPU core**
  - **Exclusive use of CPU**: only one process can use CPU
- I/O model
  - Multiple I/O devices (**no** waiting in IO queues, so just IO time!)
    - ✓ !NOT IN PRACTICE!
  - Processes can access/request different I/O devices
  - I/O **operation time** of different processes can overlap

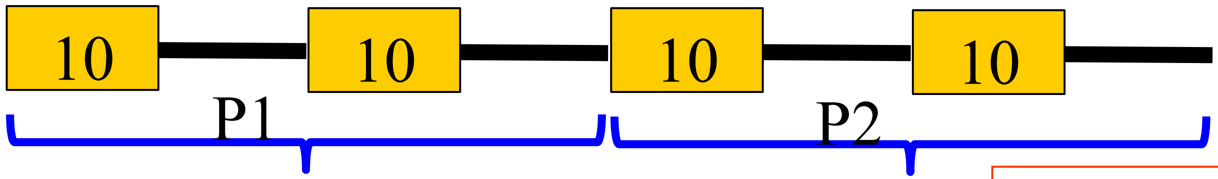


An Example: No Multiprogramming

- Suppose 2 processes, where each process
  - Requires 20 seconds of CPU time
  - Waits 10 second for I/O for every 10 seconds execution



- How will they run without multiprogramming?
  - Run one after another

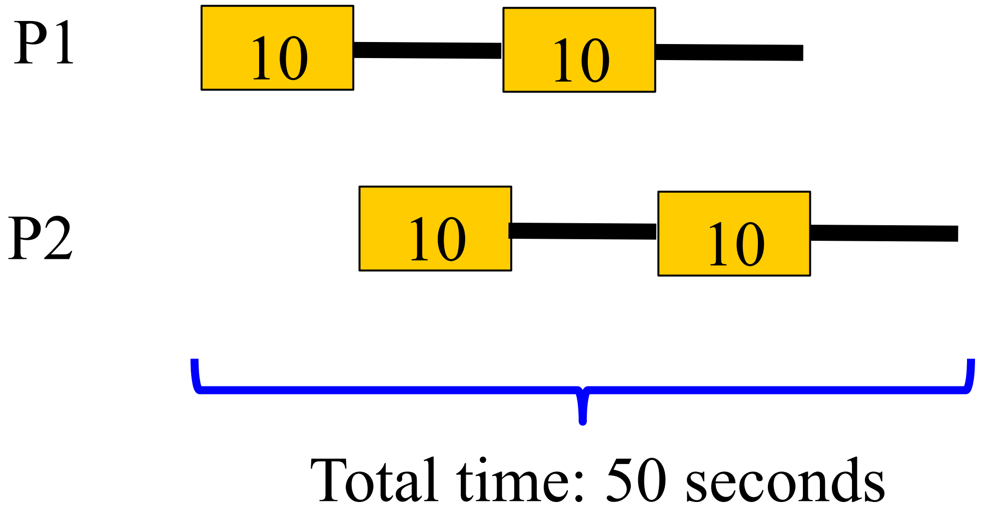


- How well do we utilize our CPU?
  - Out of 80 sec, CPU was busy for 40 sec
  - **CPU utilization is about 40/80*100 = 50%**

$$CPU\_util = \frac{CPU\_busy\_time}{Total\_time}$$

An Example: with Multiprogramming

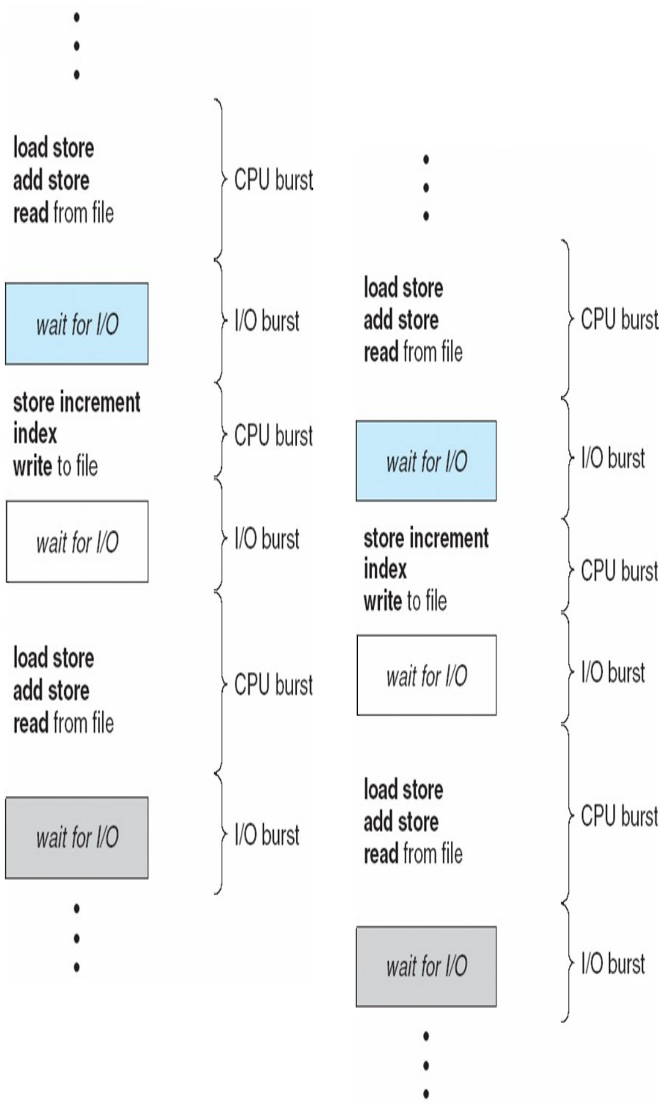
- Multiprogramming: both processes run **together (concurrently)**
  - The first process finishes in 40 seconds
  - The second process uses CPU (I/O) alternatively with first one and finishes 10 second later ☾ **50 seconds**



CPU utilization = ?

40/50*100 = 80%

Multiprogramming



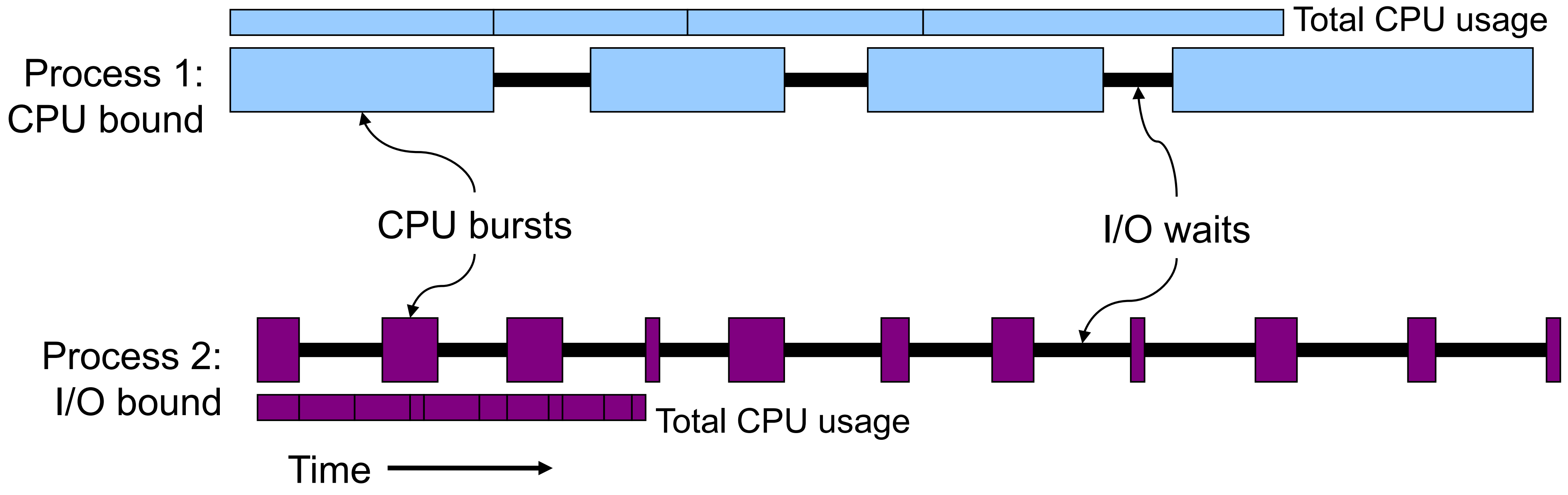
- Multiprogramming is a form of <u>parallel processing</u> in which several programs are run at the same time (**concurrently**) on a uniprocessor.

Why? Objective?

Maximize CPU utilization.
When a process waits for IO, CPU time is wasted and no useful work is accomplished.
So give it to another process!

CPU-bound vs. IO-Bound

CPU-bound: spend more time on CPU, high CPU utilization



I/O-bound: spend more time on IO, low CPU utilization

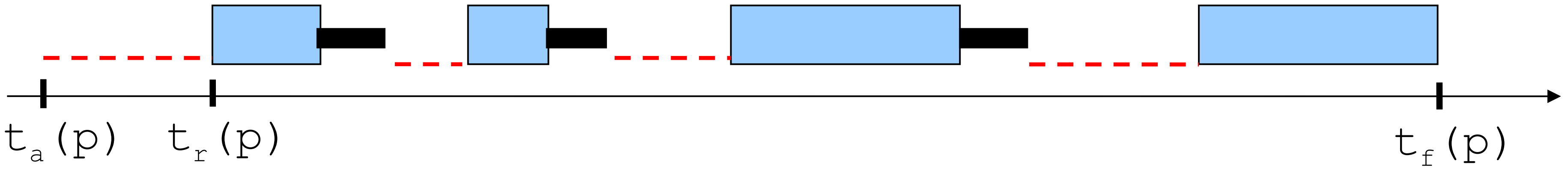
Outline

- Reviews on processes and their states
- Process queues and scheduling events
- Different levels of schedulers
  - CPU Scheduler
  - Preemptive vs. non-preemptive
  - Context switches and dispatcher
  - Models and assumptions for CPU scheduling
  - CPU-bound and IO-bound processes
- Performance criteria
  - Fairness, efficiency (CPU Utilization), waiting time in ready queue, response time, throughput, and turnaround time;
- Classical schedulers: FIFO, SFJ, RR, and PR
- CPU Gantt chart vs. process Gantt charts

Performance Measures per process p

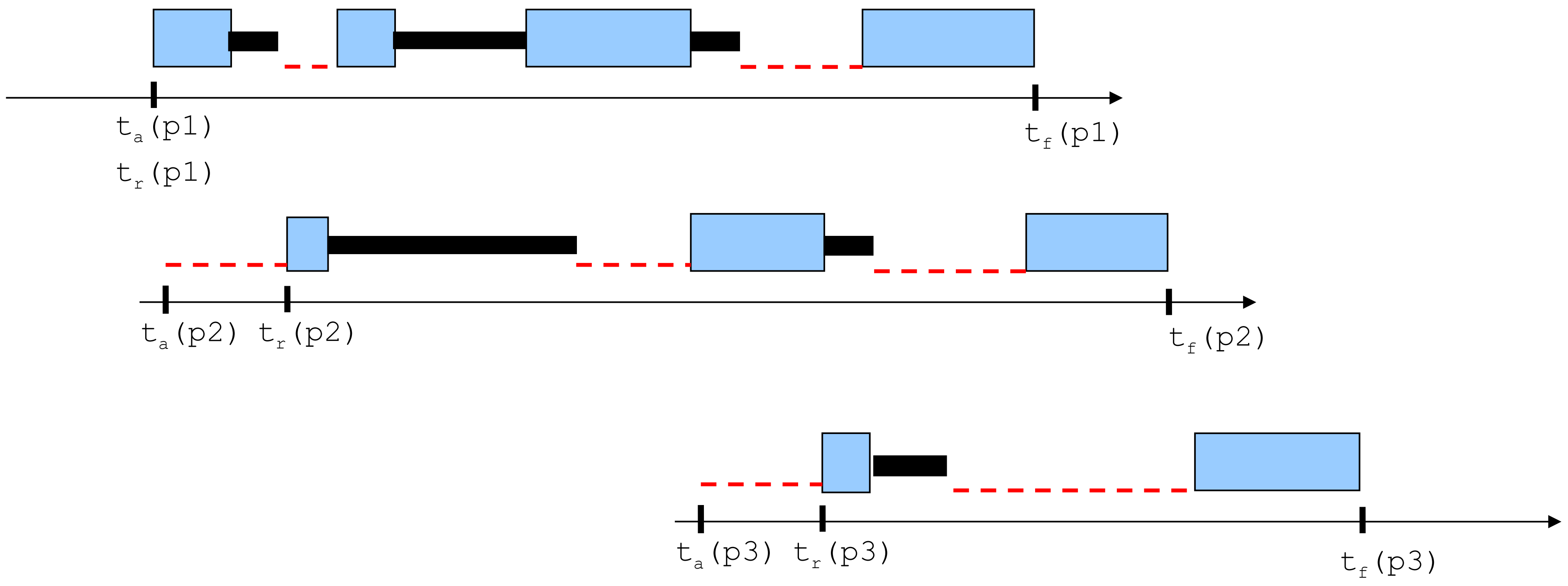
We can measure the followings about each process p

- Arrival time: $t_a(p)$
- First time to respond: $t_r(p)$
- Finish time: $t_f(p)$



- Turn around time (from submission to termination) $t_{turn\_around}(p) = t_f(p) - t_a(p)$
- Response time $t_{response\_time}(p) = t_r(p) - t_a(p)$
- Total waiting time in Ready queue $t_{wait\_ready}(p)$ $\boxed{\mathrel{+}= (t_{p\_taken\_from\_r\_queue} - t_{p\_put\_into\_r\_queue})}$
- Total waiting time in IO queue $t_{wait\_io}(p)$ (0, we have multiple I/O !!!)
- Total CPU burst time: $t_{cpu}(p)$
- Total I/O burst time: $t_{io}(p)$

Performance Measures for the system



- We can measure
  - Total time to finish all processes t_{total}
  - Total time CPU was idle t_{idle}
  - Total time spend for context-switch $t_{dispatch}$

Performance Metrics
involving all processes

- Fairness
  - Each process gets a fair share of CPU in Multiprogramming
- Efficiency: **CPU Utilization**
  - Percentage of time CPU is busy;

$$util = \frac{\sum t_{cpu}(p)}{t_{total}}$$

- Throughput
  - Number of processes completed per unit time (e.g., 10 jobs per second)

$$Throughput = \frac{\#of\_processes}{t_{total}}$$

- Average Turnaround Time
  - Time from submission to termination

$$\frac{\sum \text{all } t_{turn\_arround}(p)}{\#of\_processes}$$

Performance Metrics
involving all processes (cont.)

- Average Waiting time in Ready queue
  - Time for a process waiting for CPU in a ready queue

$$\frac{\sum \text{all } t_{wait\_ready}(p)}{\#of\_processes}$$

- Average Response time
  - Time between submission and the first response
  - Good metric for interactive systems
  - Response time variance
    - ✓ For interactive systems, response time should **NOT** vary too much

$$\frac{\sum \text{all } t_{response\_time}(p)}{\#of\_processes}$$

Scheduling Goals

- All systems
  - **Fairness**: give each process a fair share of the CPU
  - **Balance**: keep all parts of the system busy; CPU vs. I/O
  - Enforcement: ensure that the stated policy is carried out
- Batch systems
  - **Throughput**: maximize jobs per unit time (hour)
  - Turnaround time: minimize time users wait for jobs
  - CPU utilization: CPU time is precious → keep the CPU as busy as possible
- Interactive systems
  - **Response/wait time**: respond quickly to users' requests
  - Proportionality: meet users' expectations
- Real-time systems: correctness and in-time processing
  - Meet **deadlines:** deadline miss → system failure!
  - Hard real-time vs. soft real-time: self-driving car control system vs. DVD player
  - Predictability: timing behaviors is predictable

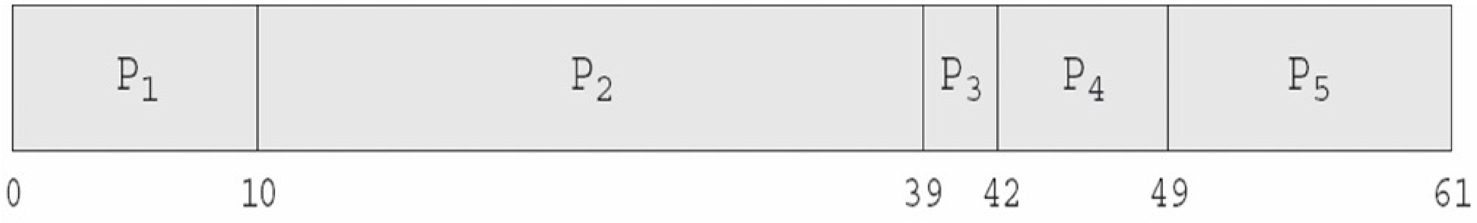
Max CPU utilization
Max throughput
Min turnaround time
Min waiting time
Min response time

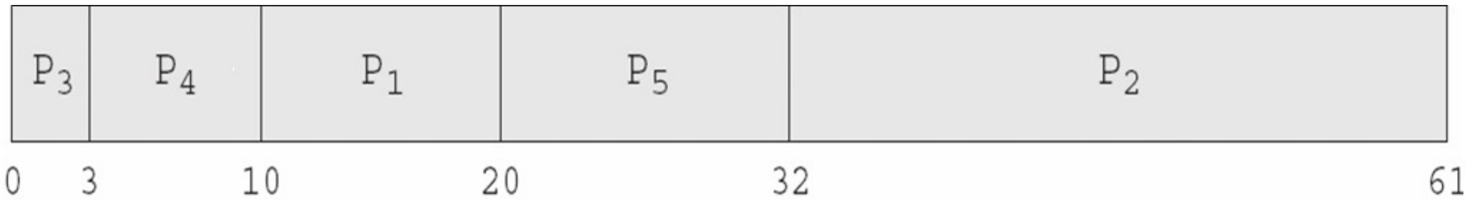
Usually NOT possible to optimize for *all* metrics with the same scheduling algorithm.
So, focus on different goals under different systems

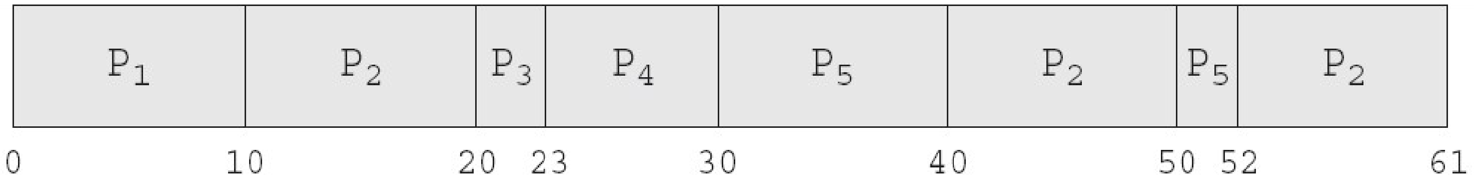
Exercise: Compute performance metrics

Suppose all processes arrived at the same time t=0! And NO I/O

Find turnaround, waiting, response time for each process. Then take their averages…







- **CPU utilization** : What percent of the time the CPU is used
- **Throughput** : Number of processes that are completed per time unit
- **Turnaround time :** Amount of time to complete a process
- **Waiting time in Ready queue:** Amount of time a process has been waiting in the ***ready queue***
- **Response time :** Amount of time it takes from when a request was submitted until the first response is produced

Outline

- Reviews on processes and their states
- Process queues and scheduling events
- Different levels of schedulers
  - CPU Scheduler
  - Preemptive vs. non-preemptive
  - Context switches and dispatcher
  - Models and assumptions for CPU scheduling
  - CPU-bound and IO-bound processes
- Performance criteria
  - Fairness, efficiency, waiting time, response time, throughput, and turnaround time;
- Classical schedulers: FIFO, SFJ, RR, and PR
- CPU Gantt chart vs. process Gantt charts

FIFO or	(First In First Out)	
FCFS	(First Come First Serve)	: non-preemptive, based on arrival time
SJF	(Shortest Job First)	: preemptive & non-preemptive
PR	(PRiority-based)	: preemptive & non-preemptive
RR	(Round-Robin)	: preemptive

SCHEDULING ALGORITHMS

Deciding which of the processes
to select from the ready queue!

Scheduling Policy vs. Mechanism

- Separate ***what*** should be done from ***how*** it is done
  - **Policy** sets **what** priorities are assigned to processes
  - Mechanism allows
    - ✓ Priorities to be assigned to processes
    - ✓ CPU to select processes with high priorities
- Scheduling algorithm parameterized
  - Mechanism in the kernel
  - Priorities assigned in the kernel or by users
- Parameters may be set by user processes
  - Don’t allow a user process to take over the system!
  - Allow a user process to voluntarily lower its own priority
  - Allow a user process to assign priority to its threads

First In First Out (FIFO)
First-Come First Served (FCFS)

- Ready Queue is strictly managed as FIFO
  - *processes are added to queue based on arrival time*
  - *And executed in a non-preemptive manner.*
- Suppose 3 processes arrive at t=0 in order
  - P1: **24** (CPU burst time), P2: **3**, P3: **3**
  - **CPU Gantt chart**

```
 ------------------------ --- ---
|           P1           |P2 |P3 |
 ------------------------ --- ---
0                        24  27  30
```

shows which process uses CPU over time.

  - Average waiting time (in ready queue) =

(0 + 24 + 27)/3 = 17

- **CPU utilization** : What percent of the time the CPU is used
- **Throughput** : Number of processes that complete their execution per time unit
- **Turnaround time :** Amount of time to execute a particular process
- **Waiting time:** Amount of time a process has been waiting in the ready queue
- **Response time :** Amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

FIFO or FCFS: cont.

- What if the 3 processes arrive in a **different order**
  - P2:**3**, P3:**3**, and P1: **24** (CPU burst time)
  - CPU Gantt chart

```
 --- --- -------------------------
|P2 |P3 |            P1            |
 --- --- -------------------------
0   3   6                        30
```

  - Avg waiting time (AWT) = (0 + 3+ 6)/3 = 3 !!!
  - **Big improvement of AWT over the previous case**!

Problem of FCFS: long jobs delay every job after them. Many processes may wait for a single long job.

Convoy effect: short process behind long process

Process Gantt Chart vs. CPU Gantt chart

- For each process, show its state at any time
- For the last example with the original order
  - CPU Gantt Chart

```
 --------------------------- --- ---
|            P1             |P2 |P3 |
 --------------------------- --- ---
0                          24  27  30
```

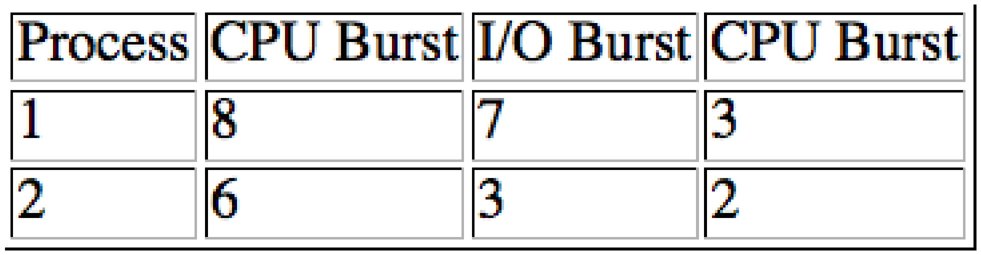
  - Process Gantt chart

```
P1: RRRRRRRRRRRRRRRRRRRRRRRR
P2: rrrrrrrrrrrrrrrrrrrrrrrrRRR
P3: rrrrrrrrrrrrrrrrrrrrrrrrrrrRRR
```

R (Running), r (ready), w (waiting)

Another Example: CPU and I/O Bursts

- Two processes



Process	CPU Burst	I/O Burst	CPU Burst
1	8	7	3
2	6	3	2

- Process Gantt chart for FCFS - FIFO

```
P1:  RRRRRRRRwwwwwwwRRR
P2:  rrrrrrrrRRRRRRwwwrRR
```

- AWT in ready queue = (0+9) / 2 = 4.5
- Notes:
  - **Waiting time in ready queue** for process is the number of **r**'s in the string
  - AWT is total number of r's divided by number of processes
  - **CPU utilization** is the total number of R's divided by the total time (the length of the longest string)

Shortest Job First (SJF)

- SJF: run the job with the shortest CPU burst first!
- Example of 4 processes
  - CPU Gantt chart vs. Process Gantt chart

```
 --- ------- -------- ---------
|P4 |  P1   |  P3    |   P2    |
 --- ------- -------- ---------
0   3       9        16        24
```

Prcocess	Burst Time
1	6
2	8
3	7
4	3

  - AWT in ready queue = (0+3+9+16)/4 = 7
- Which job has the shortest CPU burst in practice?
  - Use past history to predict:
  - Suppose that
    τ is predicted time,
    and t is actual run time
  - τ _(n+1) = a*t_n + (1-a)* τ_n

exponential average

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

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

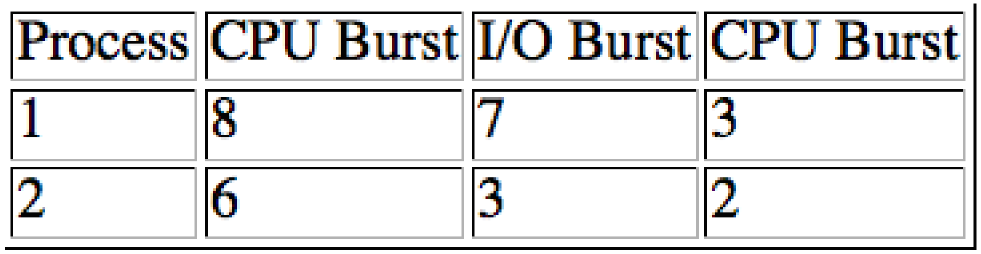
$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + (1 - \alpha)^2\tau_{n-1}$$

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + (1 - \alpha)^2\alpha t_{n-2} + (1 - \alpha)^3\tau_{n-2}$$

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \ldots + (1 - \alpha)^j\alpha t_{n-j} + \ldots (1 - \alpha)^n\alpha t_0$$

Shortest Job First (SJF): cont.

- Example 3: SJF
  - Process Gantt chart



Process	CPU Burst	I/O Burst	CPU Burst
1	8	7	3
2	6	3	2

```
P1: rrrrrrRRRRRRRRwwwwwwwRRR
P2: RRRRRRwwwrrrrrRR
```

- Average waiting time:
  - P1: waits for **6 (r)** units first, then CPU, then I/O, and CPU
  - P2: runs for 6, does I/O, waits for **5 (r)** units, and CPU
  - AWT = (6 + 5)/2 = 5.5

SJF is optimal means that it gives minimum average waiting time for a given set of processes

Preemptive Shortest Job First (**PSJF**)

Also called **shortest-remaining-time-first**

- So far, we considered **non-preemptive** SJF
- But it can be **Preemptive**, too! How?
  - If a new process enters the ready queue that has a shorter next CPU burst compared to **what is expected to be left (remaining)** of the currently executing process, then
  - Current running job will be replaced by the new one
- *Shortest-**remaining**-time-first* scheduling
- Example 3
  - Process Gantt chart

Process	CPU Burst	I/O Burst	CPU Burst
1	8	7	3
2	6	3	2

```
P1: rrrrrrRRRrrRRRRRwwwwwwwRRR
P2: RRRRRRwwwRR
```

- **AWT = (8+0)/2=4**

Round Robin Scheduling (RR)

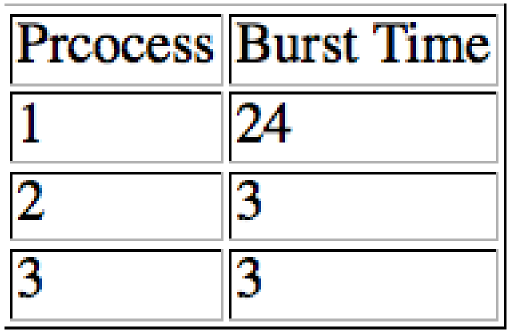
- **Non-preemptive**: process keeps CPU until it terminates or requests I/O
- **Preemptive**: allows higher priority process preempt an executing process (if its priority is lower)
- Time sharing systems
  - Need to avoid CPU intensive processes that occupy the CPU too long

- Round Robin scheduler (RR)
  - **Quantum**: a small unit of time (10 to 100 milliseconds)
  - Processes take turns to run/execute for a quantum of time
  - Take turns are done in FIFO or FCFS

RR Examples

- RR with quantum of 4

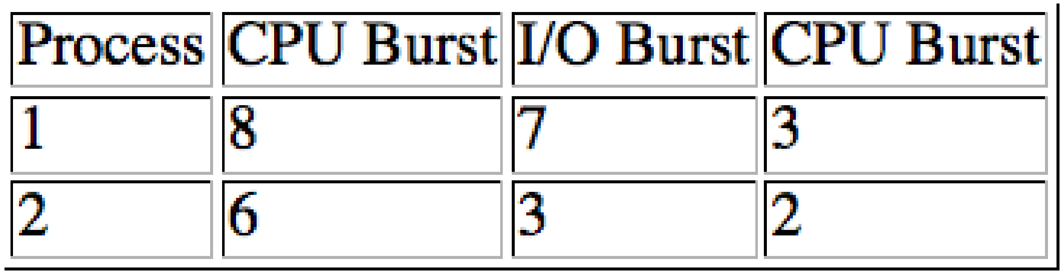
```
P1: RRRRrrrrrrRRRRRRRRRRRRRRRRRRRR
P2: rrrrRRR
P3: rrrrrrrRRR
```



Prcocess	Burst Time
1	24
2	3
3	3

  - AWT = (6+4+7)/3 = 17/3 = 5.67

- RR with quantum of 3



Process	CPU Burst	I/O Burst	CPU Burst
1	8	7	3
2	6	3	2

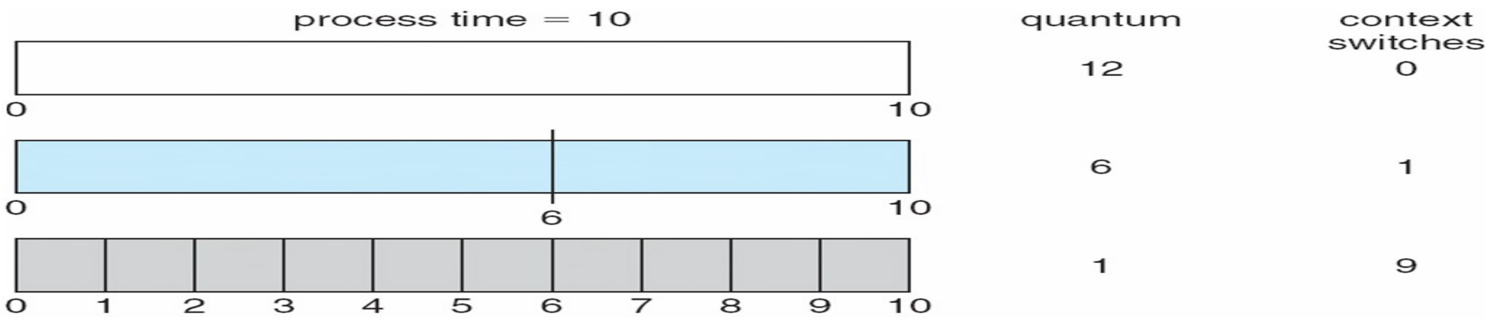
  - AWT = (6+6)/2 = 6.0

```
P1: RRRrrrRRRrrrRRwwwwwwwRRR
P2: rrrRRRrrrRRRwwwRR
```

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

RR: Round Robin Scheduling + Quantum

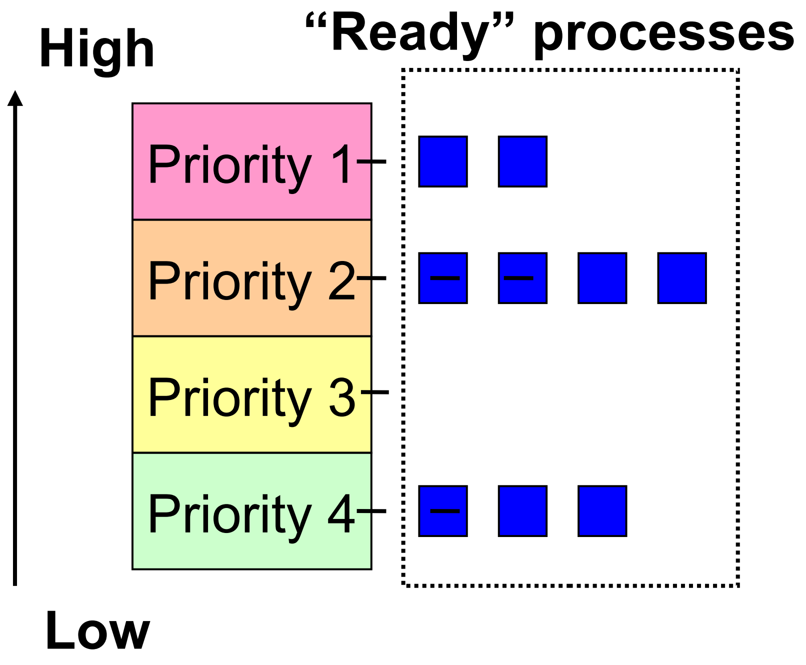
- If quantum is small enough
  - For n processes, each appears to have ***its own CPU*** that is ***1/n of CPU's original speed***
  - *What could be the poblem with too small quantum?*
- Quantum: determines **efficiency** & **response time**
- How to decide quantum size? ☾ **10 to 100 ms**
  - Too small ☾ too many context switches ☾ no useful work
  - Too long ☾ response time suffers
  - Rule of thumb: 80% of CPU bursts <= quantum



quantum	context switches
12	0
6	1
1	9

PR (PRiority Based Scheduling)

- Assign a priority to each process
  - "Ready" process with highest priority allowed to run
  - Can be preemptive or non-preemptive
  - Same priority: use FIFO



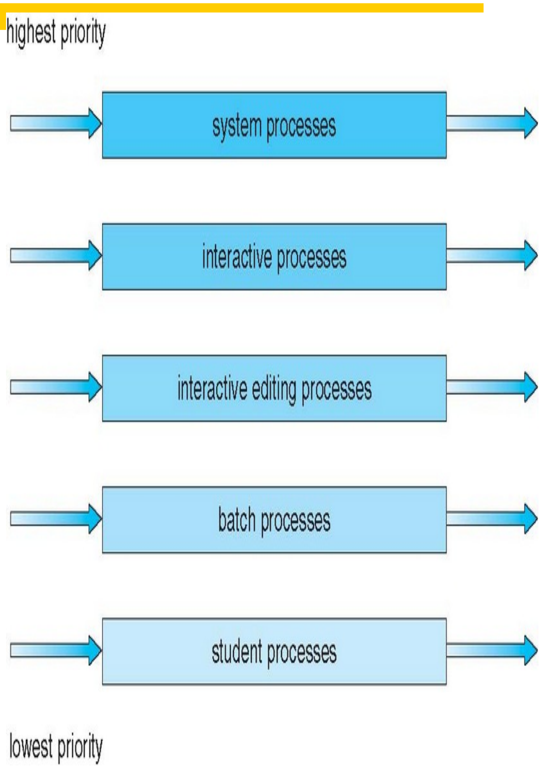
- Priorities may be assigned dynamically
  - Reduced when a process uses CPU time
  - Increased when a process waits for I/O

Summary of CPU Scheduling Algorithms

- **FIFO** or **FCFS** *: non-preemptive, based on arrival time*
  - - Long jobs delay everyone else
- **SJF** *: preemptive & non-preemptive*
  - + Optimal in terms of waiting time
- **PR** *: preemptive & non-preemptive*
  - + Real-time systems: earliest deadline first (EDF)
- **RR** *: preemptive*
  - + fairness, Processes take turns with fixed time quantum (e.g., 10ms)
- Multi-level queue (priority classes)
  - System processes > faculty processes > student processes
- Multi-level feedback queues: change queues
  - short ☾ long quantum

Multilevel Queues (Express lanes)

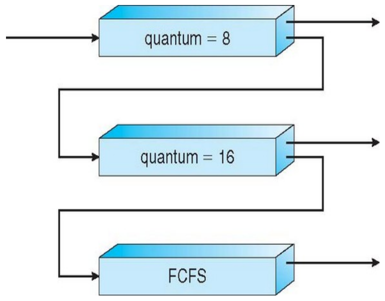
- Ready queue is partitioned into separate queues: **foreground** (interactive), **background** (batch)
- Each queue has its own scheduling algorithm
  - foreground – RR
  - background – FCFS
- How to do scheduling between the queues?
  - Fixed priority scheduling;
    - ✓ Serve all from foreground then from background
    - ✓ Possibility of starvation.
  - Time slice (Weighted Queuing (WQ))
    - ✓ Each queue gets a certain amount of CPU time which it can schedule among its processes;
      - 80% to foreground in RR
      - 20% to background in FCFS



+ simple
- Not flexible
- Starvation if PR sch is used

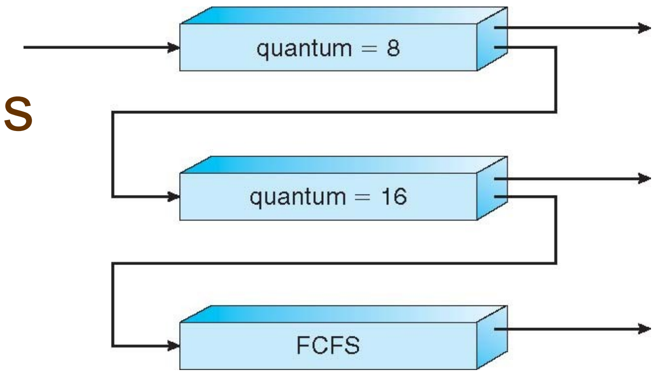
Multilevel Feedback Queues

- A process can move between the various queues based on CPU burst characteristics
  - CPU bound ☾ move into low priority queue
  - I/O bound ☾ move into high priority queue
- Aging can be implemented this way to avoid starvation
- 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
- Most flexible and general, but hard to configure



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 job enters queue Q_0 which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1.
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.
  - Processes requiring less than 8 ms will be served quickly…

Exercises: CPU Scheduling … see psq07 for more

a. **[12 points]** Consider two processes as in Assignment 2/Quiz 4, where each process has two CPU bursts with one I/O burst in between on a single core CPU. Suppose P1 and P2 have the following life-cycles:

P1 has x1=**8**, y1=7, z1=**3** units for the first CPU burst, I/O burst, second CPU burst, respectively.

P2 has x2=**6**, y2=**1**, z2=**2** units for the first CPU burst, I/O burst, second CPU burst, respectively.

Both arrives at the same time (in case of ties, pick P1) and there is no other processes in the system.

For each of the scheduling algorithms below, create Gantt charts as you did for the Quiz 4. Fill each box with the state of the corresponding process. Use **R** for **R**unning, **W** for **W**aiting, and **r** for **r**eady. Calculate the waiting times and CPU utilization (as a fraction) for each process and fill in the table below.

Gantt Charts for SJF (Shortest Job First, non-preemptive) [4pt]

a)	SJF					5					10					15					20					25					30				
P1	▨																																		
P2	▨																																		

Gantt Charts for PSJF (Preemptive SJF) [4pt]

b)	PSJF					5					10					15					20					25					30				
P1	▨																																		
P2	▨																																		

Waiting time and CPU utilizations [4pt]

Algorithm	Waiting times in ready queue			Finish time		Longest Schedule length	CPU utilization
	Process 1	Process 2	average	Process 1	Process 2		
b) SJF							
c) PSJF							

Exercise: multilevel queue

b. **[8 points]** Suppose we have a system using **multilevel queuing**. Specifically there are two queues and each queue has its own scheduling algorithm: QueueA uses RR with quantum 3 while QueueB uses FCFS. CPU simply gets processes form these two queues in a weighted round robin manner with 2:1 ratio (i.e. it gets **two** processes from QueueA then gets **one** process from QueueB, and then gets **two** processes from QueueA then gets **one** process from QueueB, and so on), But when it gets a process from QueueA, it applies RR scheduling with quantum 3. When it gets a process from QueueB, it applies FCFS scheduling.

Draw the Gantt charts (**5pt**) and compute waiting times (**3pt**) for the following four processes: P1, P2, P3, P4 on a single core CPU. Assume these processes arrived at the same time and in that order. Each process has a single CPU burst time of 5 units. There is no other processes or IO bursts.

	ratio						5					10					15					20	
QueueA	2	P1																					
RR q=3		P2																					
QueueB	1	P3																					
FCFS		P4																					

Compute Waiting times in ready queue				
P1	P2	P3	P4	average

Summary

- Reviews on processes and their states
- Process queues and scheduling events
- Different levels of schedulers
  - CPU Scheduler
  - Preemptive vs. non-preemptive
  - Context switches and dispatcher
  - Models and assumptions for CPU scheduling
  - CPU-bound and IO-bound processes
- Performance criteria
  - Fairness, efficiency, waiting time, response time, throughput, and turnaround time;
- Classical schedulers: FIFO, SFJ, RR, and PR
- CPU Gantt chart vs. process Gantt charts

IF TIME PERMITS, we may re-visit the followings at the end of the semester

THREAD SCHEDULING
MULTIPLE-PROCESSOR SCHEDULING
REAL-TIME CPU SCHEDULING

Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as **process-contention scope** (**PCS**) since scheduling competition is within the process
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is **system-contention scope** (**SCS**) – competition among all threads in system

Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
  - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
- Can be limited by OS – Linux and macOS only allow PTHREAD_SCOPE_SYSTEM

Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
   int i, scope;
   pthread_t tid[NUM THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
         printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
         printf("PTHREAD_SCOPE_SYSTEM");
      else
         fprintf(stderr, "Illegal scope value.\n");
   }
```

Pthread Scheduling API

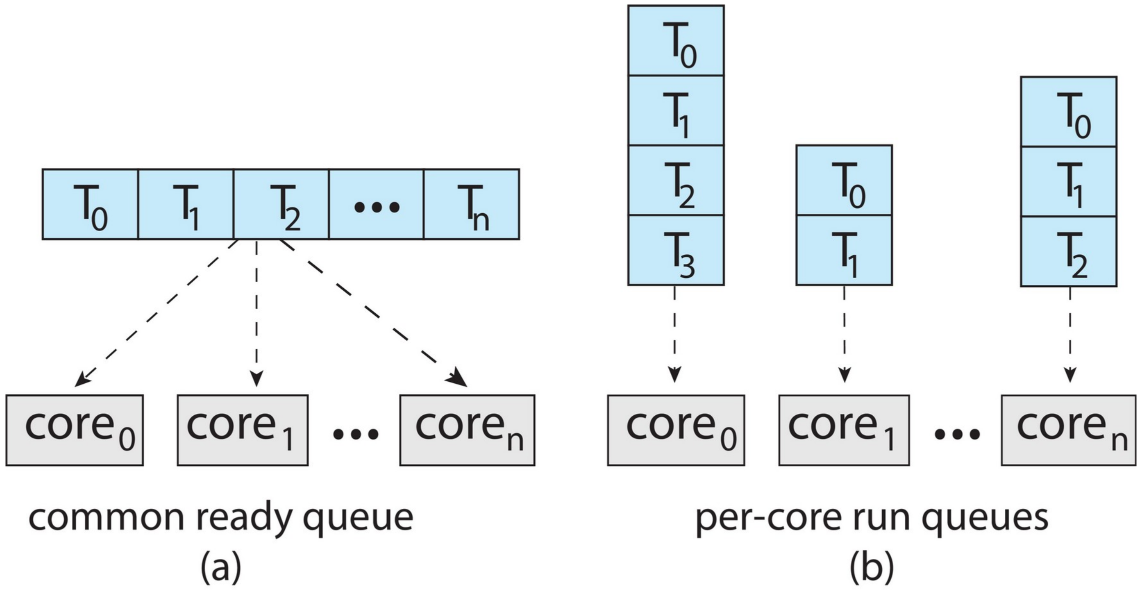
```
   /* set the scheduling algorithm to PCS or SCS */
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_create(&tid[i],&attr,runner,NULL);
   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
   /* do some work ... */
   pthread_exit(0);
}
```

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- Multiprocess may be any one of the following architectures:
  - Multicore CPUs
  - Multithreaded cores
  - NUMA systems
  - Heterogeneous multiprocessing

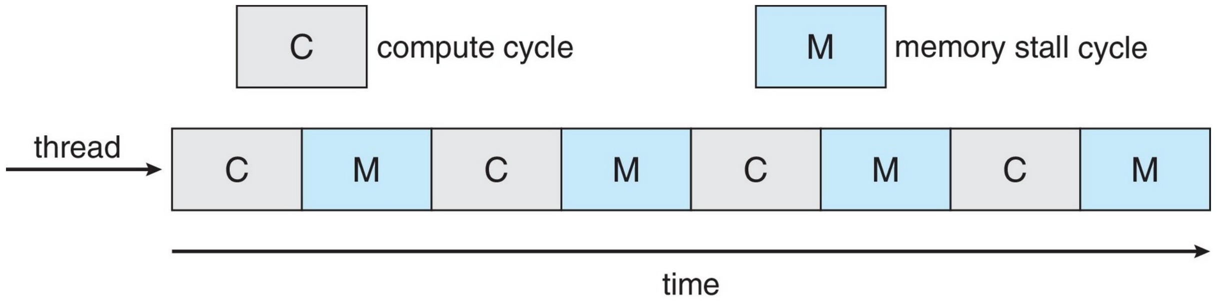
Multiple-Processor Scheduling

- Symmetric multiprocessing (SMP) is where each processor is self scheduling.
- All threads may be in a common ready queue (a)
- Each processor may have its own private queue of threads (b)



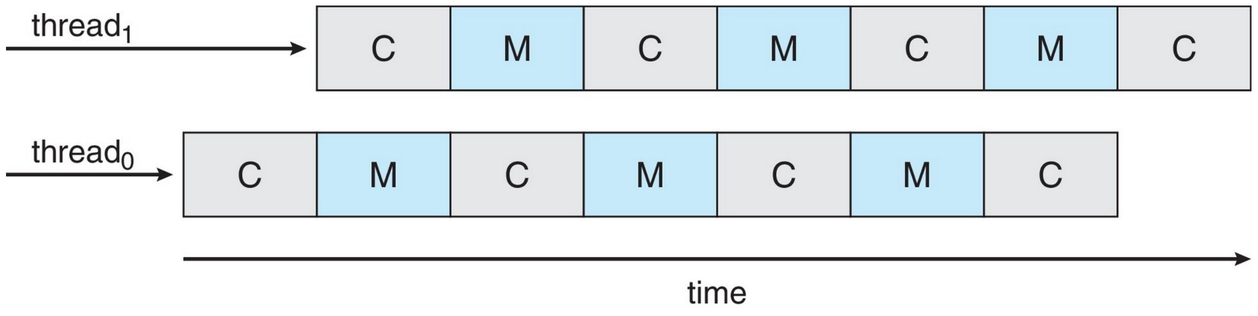
Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens
- Figure



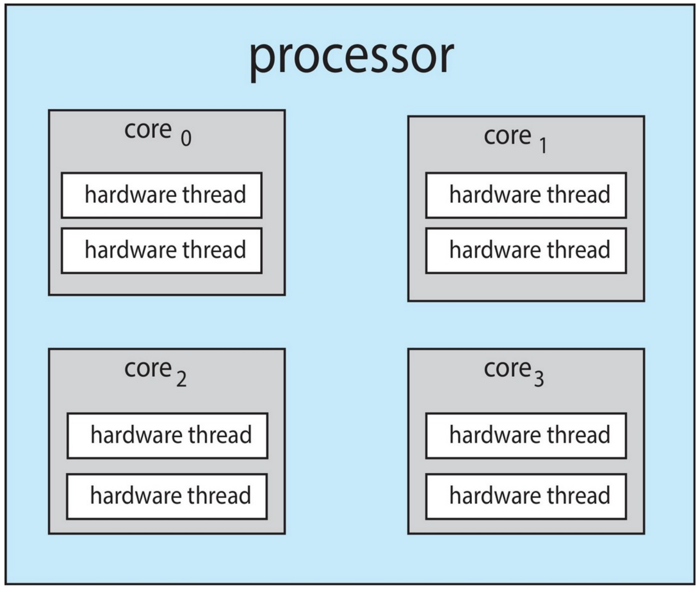
Multithreaded Multicore System

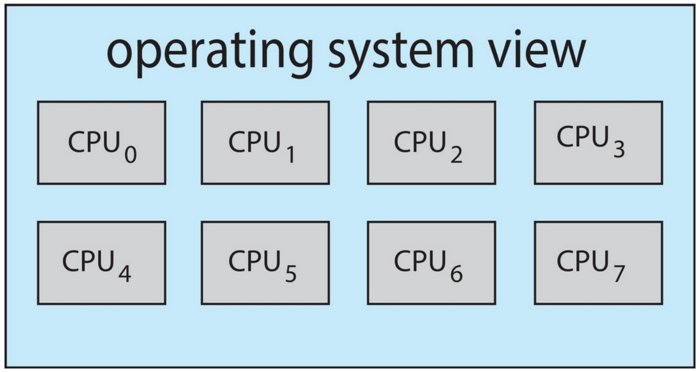
- Each core has > 1 hardware threads.
- If one thread has a memory stall, switch to another thread!
- Figure



Multithreaded Multicore System

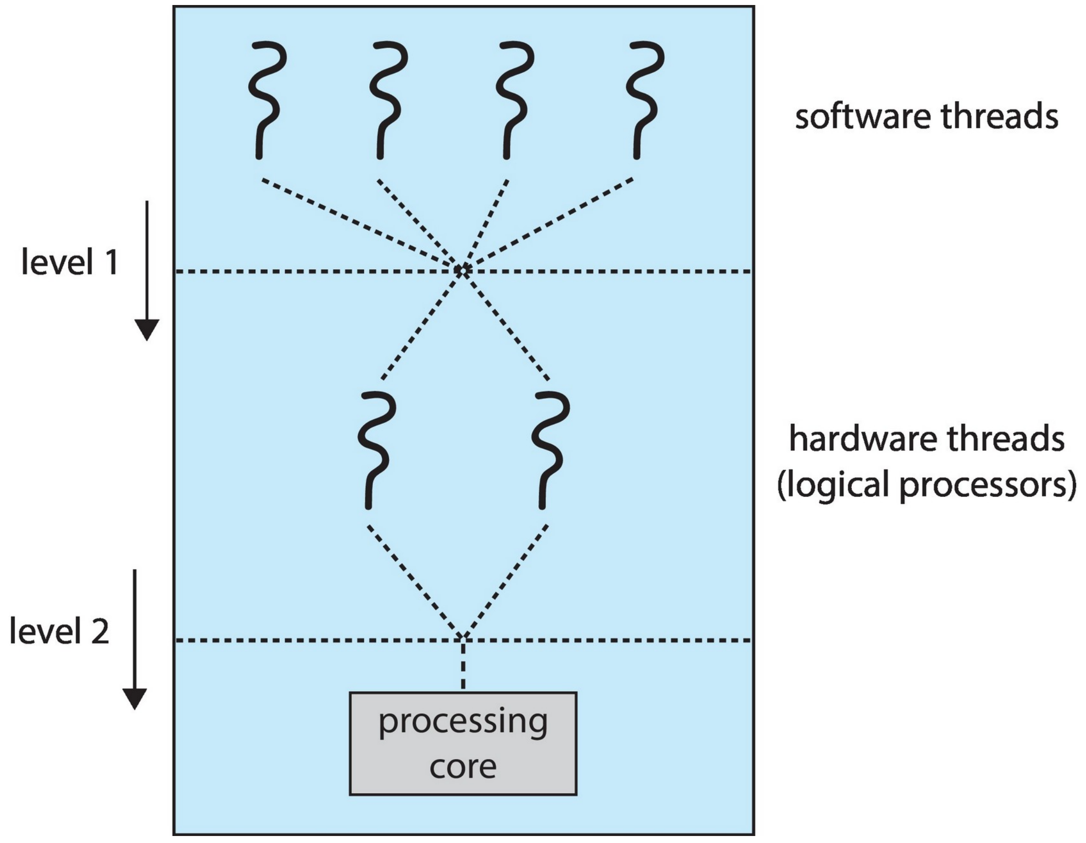
- **Chip-multithreading** (CMT) assigns each core multiple hardware threads. (Intel refers to this as **hyperthreading**.)
- On a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors.





Multithreaded Multicore System

- Two levels of scheduling:
  1. The operating system deciding which software thread to run on a logical CPU
  2. How each core decides which hardware thread to run on the physical core.



Multiple-Processor Scheduling – Load Balancing

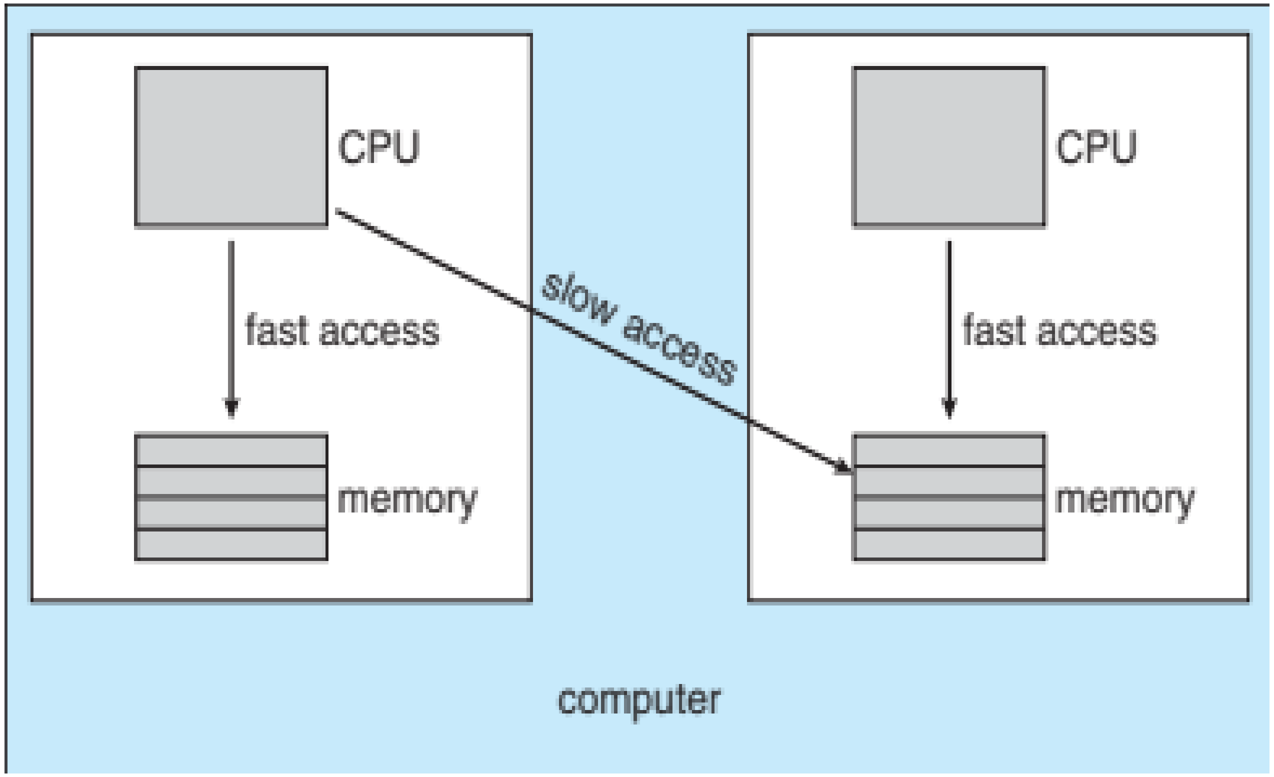
- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pulls waiting task from busy processor

Multiple-Processor Scheduling – Processor Affinity

- When a thread has been running on one processor, the cache contents of that processor stores the memory accesses by that thread.
- We refer to this as a thread having affinity for a processor (i.e., “processor affinity”)
- Load balancing may affect processor affinity as a thread may be moved from one processor to another to balance loads, yet that thread loses the contents of what it had in the cache of the processor it was moved off of.
- **Soft affinity** – the operating system attempts to keep a thread running on the same processor, but no guarantees.
- **Hard affinity** – allows a process to specify a set of processors it may run on.

NUMA and CPU Scheduling

If the operating system is **NUMA-aware**, it will assign memory closes to the CPU the thread is running on.

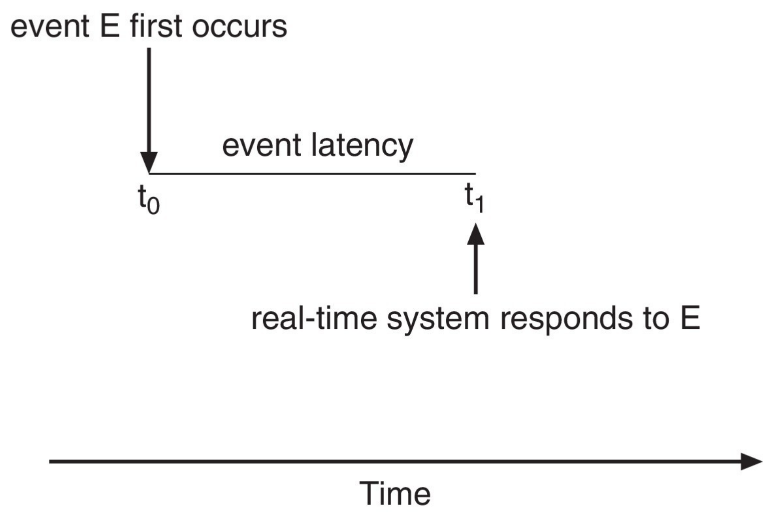


Real-Time CPU Scheduling

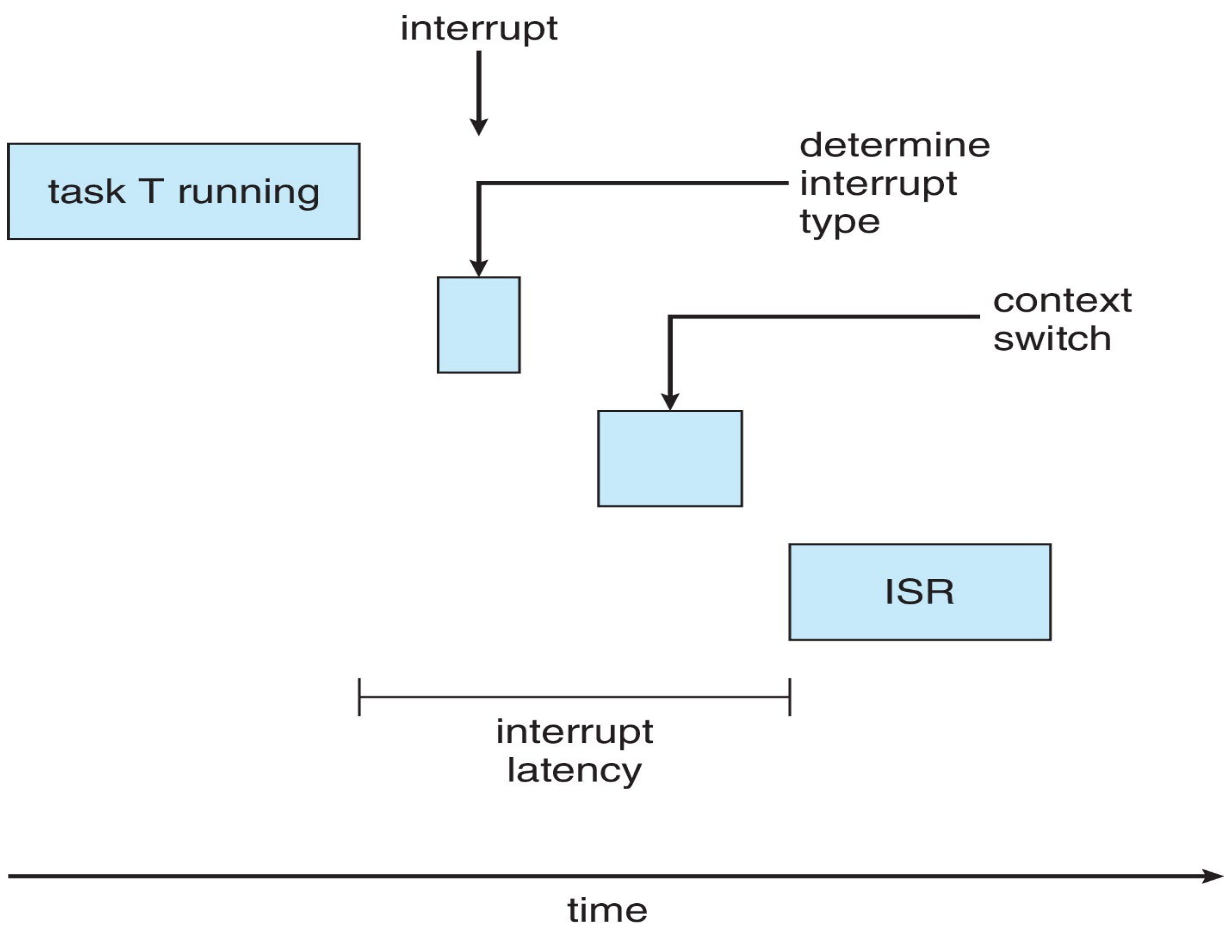
- Can present obvious challenges
- **Soft real-time systems** – Critical real-time tasks have the highest priority, but no guarantee as to when tasks will be scheduled
- **Hard real-time systems – task must be serviced by its deadline**

Real-Time CPU Scheduling

- Event latency – the amount of time that elapses from when an event occurs to when it is serviced.
- Two types of latencies affect performance
  1. **Interrupt latency** – time from arrival of interrupt to start of routine that services interrupt
  2. **Dispatch latency** – time for schedule to take current process off CPU and switch to another

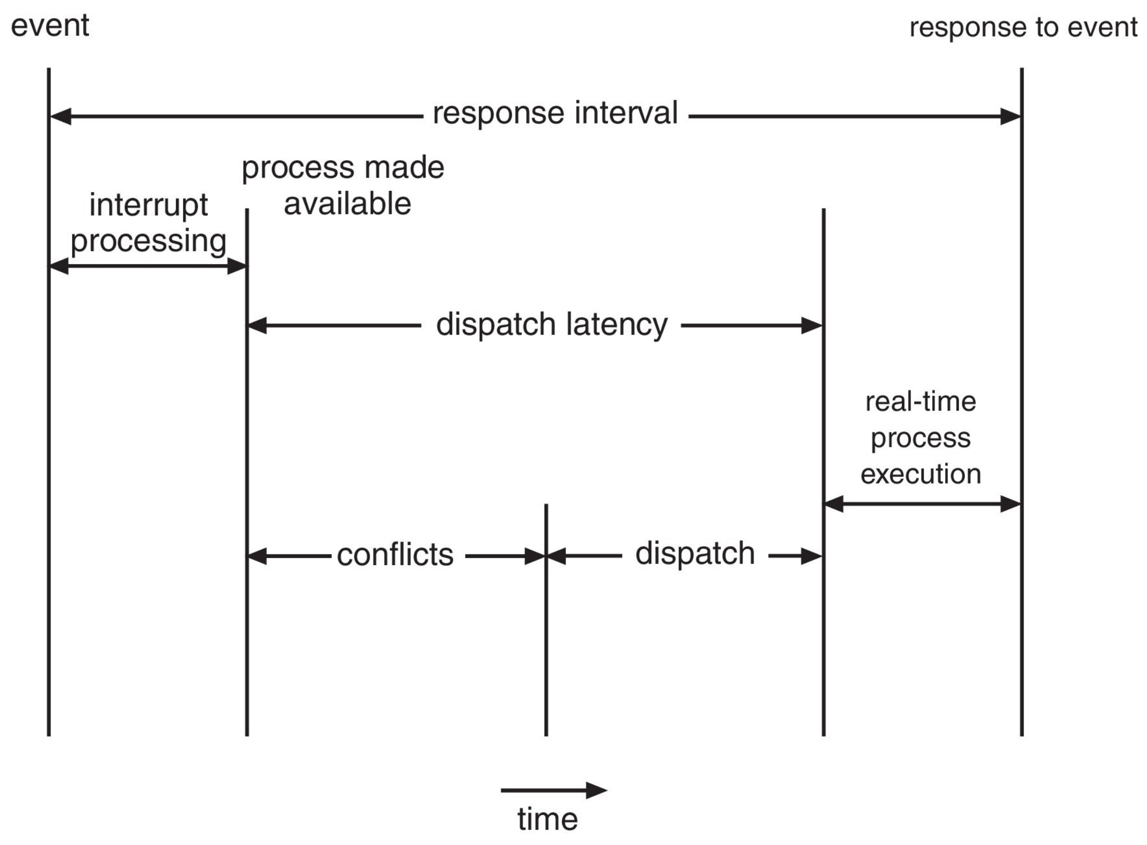


Interrupt Latency



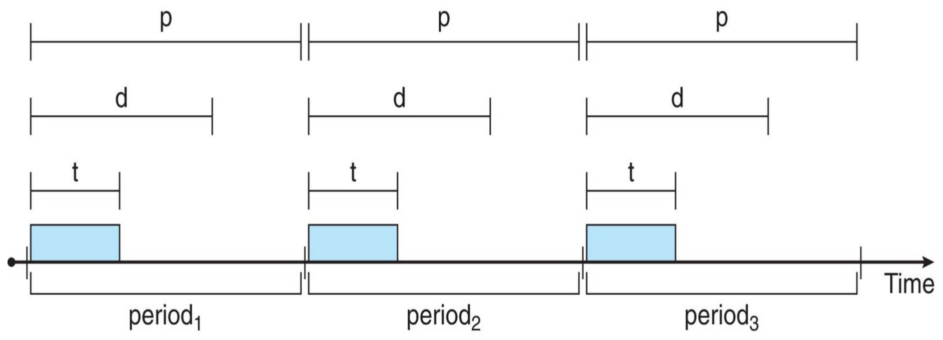
Dispatch Latency

- 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-priority processes



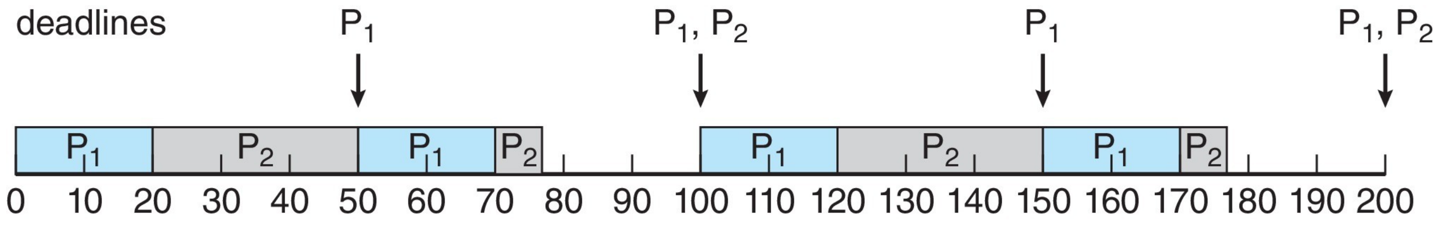
Priority-based Scheduling

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: **periodic** ones require CPU at constant intervals
  - Has processing time *t*, deadline *d*, period *p*
  - $0 \leq t \leq d \leq p$
  - **Rate** of periodic task is $1/p$



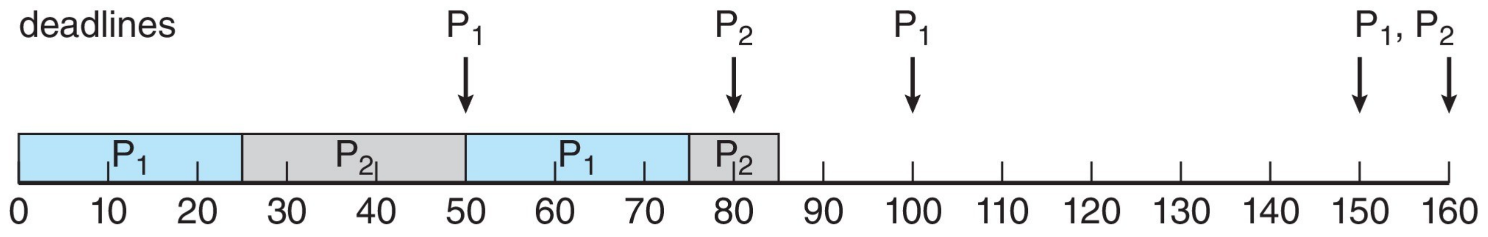
Rate Monotonic Scheduling

- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority;
- Longer periods = lower priority
- P_1 is assigned a higher priority than P_2.



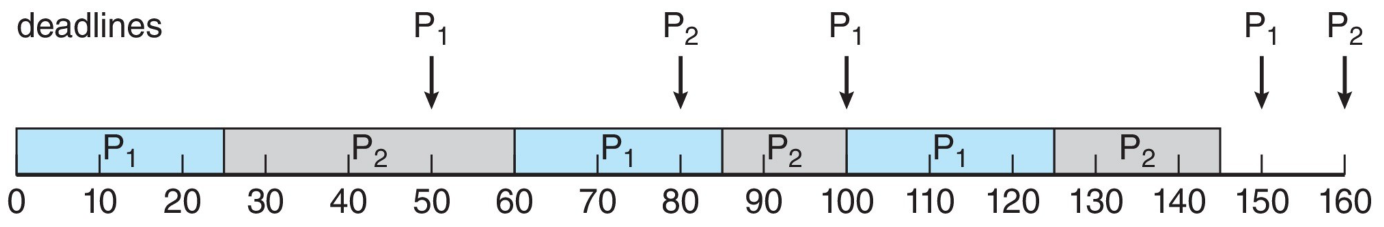
Missed Deadlines with Rate Monotonic Scheduling

- Process P_2 misses finishing its deadline at time 80
- Figure



Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:
  - The earlier the deadline, the higher the priority
  - The later the deadline, the lower the priority
- Figure



Proportional Share Scheduling

- T shares are allocated among all processes in the system
- An application receives N shares where $N < T$
- This ensures each application will receive ***N*** / *T* of the total processor time

POSIX Real-Time Scheduling

- The POSIX.1b standard
- API provides functions for managing real-time threads
- Defines two scheduling classes for real-time threads:
  1. SCHED_FIFO - threads are scheduled using a FCFS strategy with a FIFO queue. There is no time-slicing for threads of equal priority
  2. SCHED_RR - similar to SCHED_FIFO except time-slicing occurs for threads of equal priority
- Defines two functions for getting and setting scheduling policy:
  1. `pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`
  2. `pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)`

POSIX Real-Time Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[])
{
   int i, policy;
   pthread_t tid[NUM_THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* get the current scheduling policy */
   if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
      fprintf(stderr, "Unable to get policy.\n");
   else {
      if (policy == SCHED_OTHER) printf("SCHED_OTHER\n");
      else if (policy == SCHED_RR) printf("SCHED_RR\n");
      else if (policy == SCHED_FIFO) printf("SCHED_FIFO\n");
   }
```

POSIX Real-Time Scheduling API (Cont.)

```
    /* set the scheduling policy - FIFO, RR, or OTHER */
    if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
       fprintf(stderr, "Unable to set policy.\n");
    /* create the threads */
    for (i = 0; i < NUM_THREADS; i++)
       pthread_create(&tid[i],&attr,runner,NULL);
    /* now join on each thread */
    for (i = 0; i < NUM_THREADS; i++)
       pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this function */
void *runner(void *param)
{
    /* do some work ... */
    pthread_exit(0);
}
```

WE SKIP

REST is OPTIONAL

UNIX SCHEDULING

UNIX Scheduling Algorithm

- Also use multi-level feedback queues
- A runnable process get a number ☾ which queue
- Lower numbers ☾ higher priority
  - Negative numbers: system processes cannot be killed by signals
- First process in the lowest nonempty queue ☾ run
  - nice to reduce priority
- Time quantum of 0.1 second (100 milliseconds)
- Priorities are re-calculated once a second
- Non-preemptive except for quantum expiration

UNIX Scheduling Algorithm (cont.)

- Some flavors of Unix ☾ interactive process with window focus gets highest priority
- A process’ user-mode priority
  - p_usrpri = P_USER + .25 * p_cpu + 2 * p_nice
  - p_cpu : increased each time the system clock ticks and the process is running
  - p_cpu is adjusted once per second for ready processes using a digital decay filter

$$p_{cpu} = \frac{2\,\mathrm{load}}{2\,\mathrm{load} + 1} p_{cpu} + p_{nice}$$

  - Load ☾ sampled average length of run queue for previous 1 minute interval of system operations

UNIX Scheduling Algorithm (cont.)

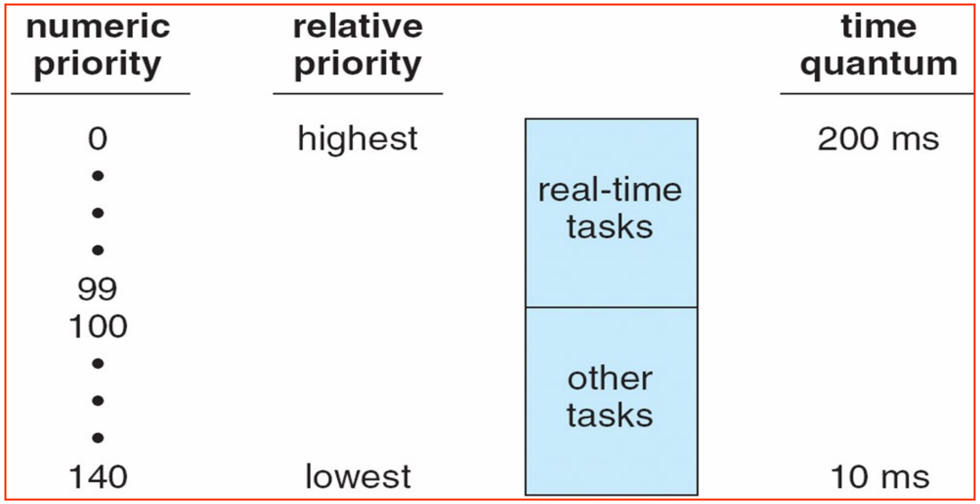
- When a process is blocked for an event, it cannot accumulate CPU time
- When a process sleeps for more than 1 second

$$p_{cpu} = \left[\frac{2\,\text{load}}{2\,\text{load} + 1}\right]^{p_{slptime}} p_{cpu}$$

  - p_slptime is an estimate of how long it is blocked
- **Non-preemptive** process running in **kernel** mode
  - **Not suitable for real-time systems**

Linux Scheduling Algorithm

- Use two separate scheduling algorithms
  - One for time-sharing with focus on fairness
  - One for real-time tasks with absolute priorities



Priorities and Time-slice length

- Time sharing processes: based on a credit system
  - Process has a fixed priority and variable number of credits
  - To choose a process ☾ the one with most credits to run
  - Running process loses one credit per timer interrupt, which is removed from CPU when its credits run out
  - If no process has any credit ☾ credits = credits/2 + priority

Linux Scheduling Algorithm (cont.)

- **Real-time processes have higher priority than time-sharing processes**
  - Time-sharing tasks run only if no runnable real-time tasks
- Real-Time Scheduling
  - Each process has a priority and a scheduling class
  - Scheduling class: can be FIFO = FCFS or RR
- The highest priority real-time task runs first
- For tasks with the same priority: FCFS
- For FCFS, a process runs until its I/O operation
- RT processes do NOT preempt other processes