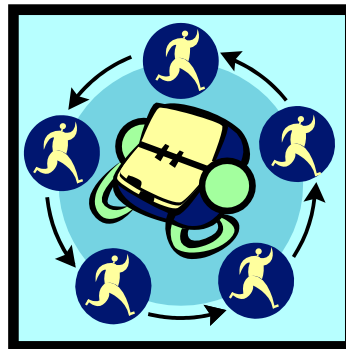


# CIS 520, *Operating Systems Concepts*

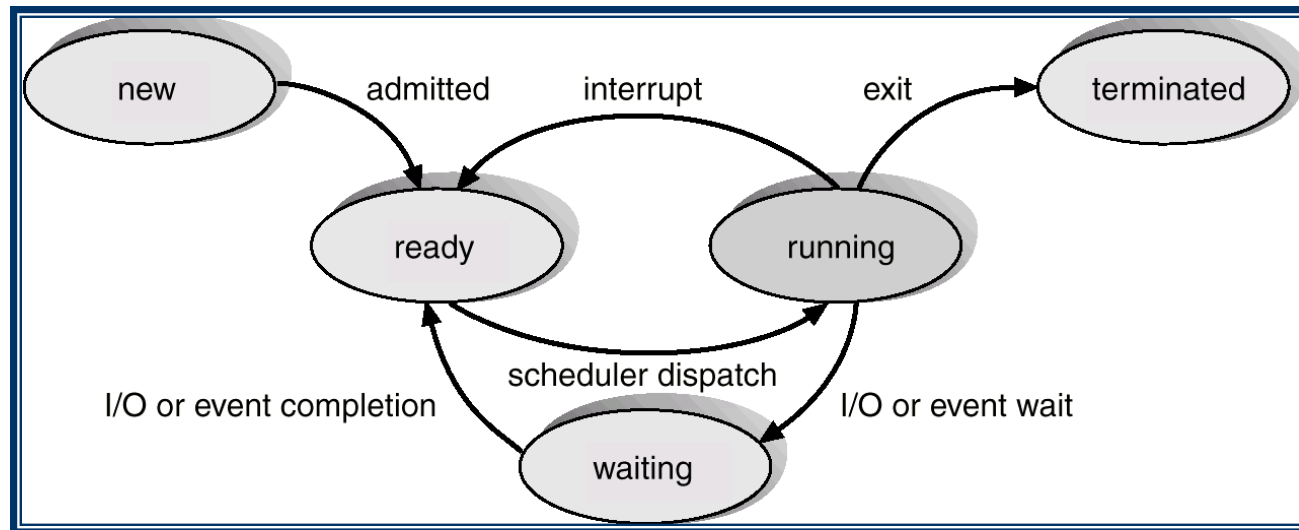
## Lecture 2 Process (CPU) Scheduling



# Why “Process (CPU)”?

- ◆ Well, we really schedule *processes* and by doing so, we *allocate* a CPU to each process
- ◆ We can also say that we schedule the CPU time among the processes
- ◆ So, both terms *process scheduling* and *CPU scheduling*, mean the same thing
- ◆ We will use both terms interchangeably

# Remember the Process States?



# How Many Processes are in the *Running* State?

- ◆ Typically, there are about as many as there are CPUs to execute them (although one CPU could be kept just for executing the operating system)
- ◆ If there is one CPU—our assumption for the rest of the day—only one process can be running

# So, What Causes a Transition out of *Running*?

- ◆ In older systems, only three things:
  1. Waiting for an I/O [transition to *waiting*]
  2. Executing an illegal instruction or accessing non-existing address, which resulted in the termination of the process [transition to *terminated*]
  3. Finishing execution (natural transition to *terminated*)
- ◆ But waiting could be—and was—generalized to waiting for any condition (which is what semaphores do)
- ◆ In systems that do only data-processing (which is I/O-heavy), just that could be sufficient, but what to do with a process that solves a linear programming problem for 17 hours and then just prints the solution?

# A Note on Interrupt Processing

- ◆ And what happens when a *running* process is interrupted—say, by an I/O device which completed the action that another process was waiting for?
  - In some systems, the other process would transition to *ready* and move to the *ready* queue, but the CPU would be returned to the process that was running
  - Yet, another solution may be to transition the *running* process to *ready* and then select the new running process from the ones that have been ready.

# Preemptive schemes (Time Slicing)

- ◆ Back to the *CPU-bound* processes: to co-exist among themselves (as well as with the *I/O-bound* ones) they must be *preempted* when they exceed their time quota (*time slices*)
- ◆ For that, hardware must support a *timer* (or an *alarm*), which would issue an appropriate interrupt

# Time Slicing (cont.)

- ◆ Preemptive schemes require more frequent context switches, so they come with a cost!
- ◆ In general, time slicing is needed to ensure fairness on mainframes and public servers, but it was not *that* much needed in PCs
- ◆ Hence, earlier personal computers' operating systems (Windows 3.1 and Apple Macintosh) did not support time slicing



# Scheduling

- ♦ *Q*: What do we do when there are more than one *ready* process?
- ♦ *A*: We queue them (in the *Ready Queue*) and then select the one to run (for each available CPU—but let us assume for now that we have one CPU), according to a *scheduling algorithm*
- ♦ *Q*: What are we trying to achieve in such an algorithm?

# Scheduling Criteria

(some at cross-purposes with others!)

- ◆ Maximize *CPU utilization* (should be between 40% and 90%)
- ◆ Maximize *I/O device utilization* (should be between 40% and 90%)
- ◆ Maximize *throughput* (*processes/sec*)
- ◆ Minimize average (or maximum) *turnaround* time
- ◆ Minimize average (or maximum) *response* time—in interactive systems) and its variance
- ◆ Minimize average (or maximum) *waiting time* (the time spent in the *ready* queue)

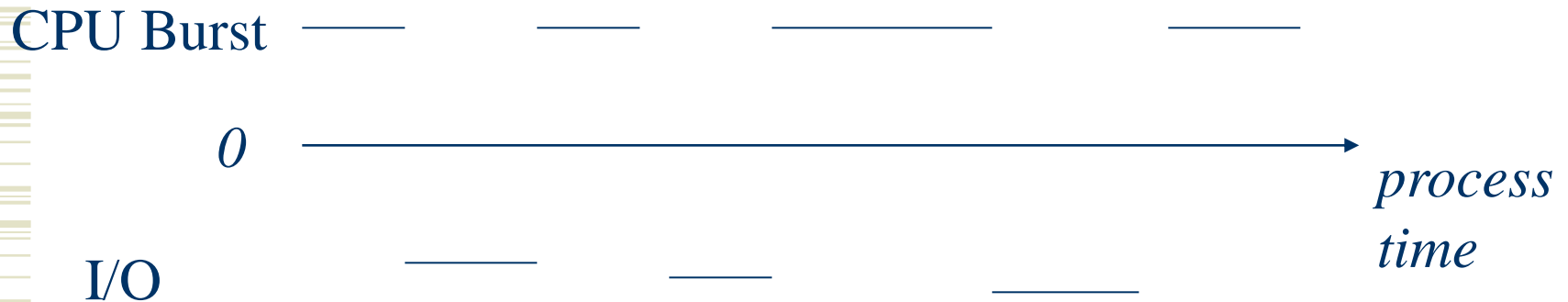
# Long Term/Short Term

In the (old) batch systems user jobs were read from a device (a tape- or card reader) and stored on the disk

- Selecting the jobs to load into memory is a matter of *long-term scheduling*
- Selecting an in-memory job to run is a matter of *short-term scheduling*

# CPU Burst/I/O Burst

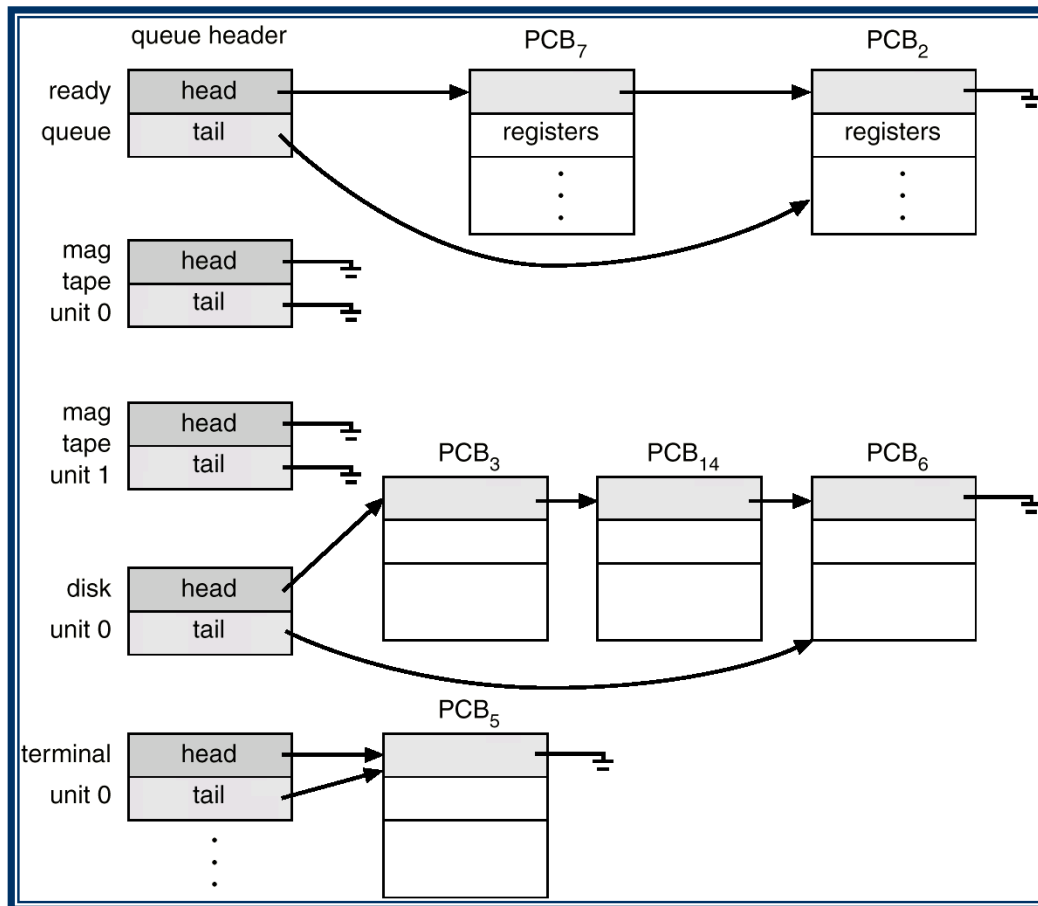
- ◆ Overall, a *useful* life of an average process is a repetition of CPU bursts followed by I/O bursts:



# The process information is maintained in the Process Control Block (PCB)

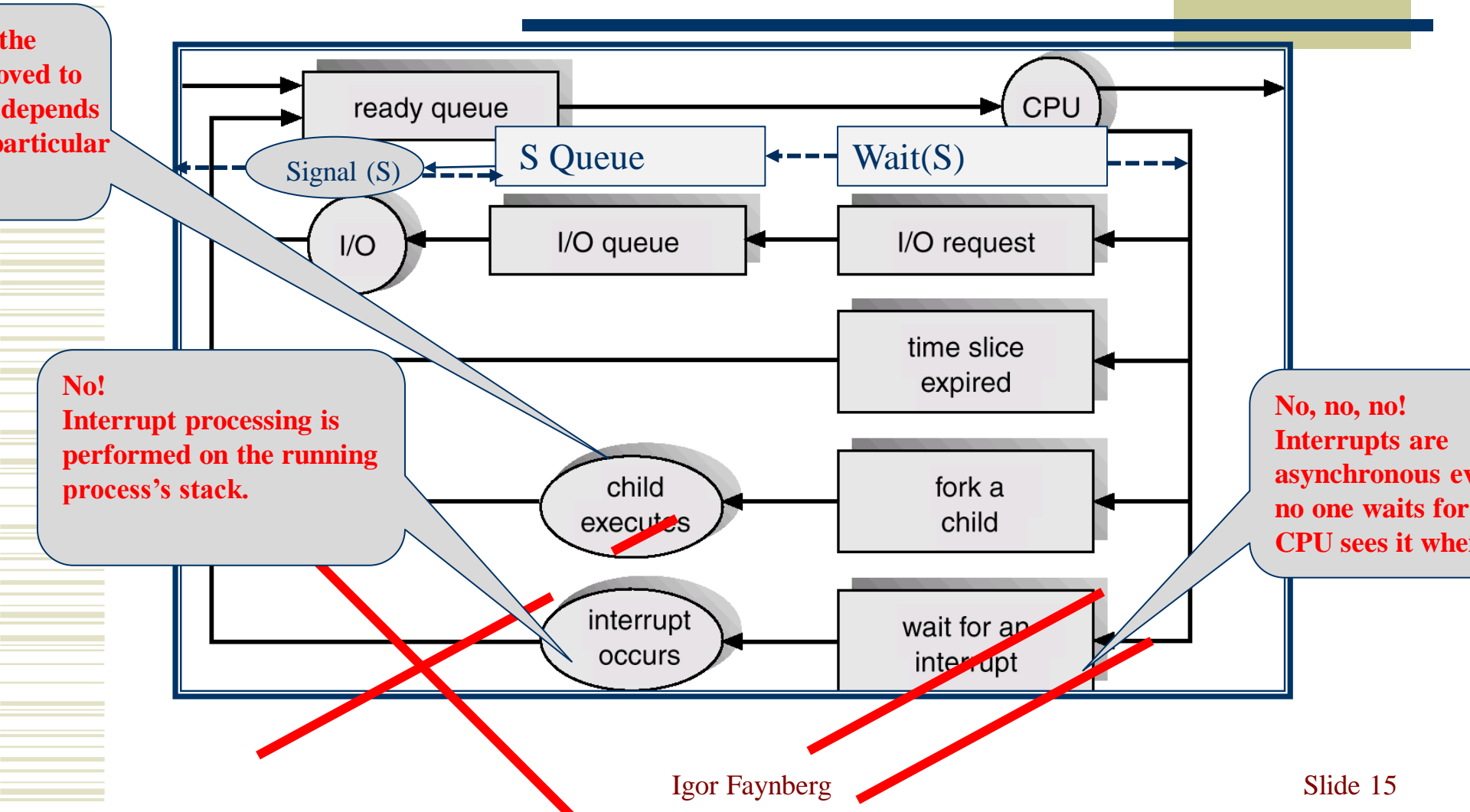
Process ID
Process state ( <i>ready, running, etc.</i> )
All the registers (including PC and SPs)
Memory management information
File information
Children processes information
... many other things ....

# Where is This Process?

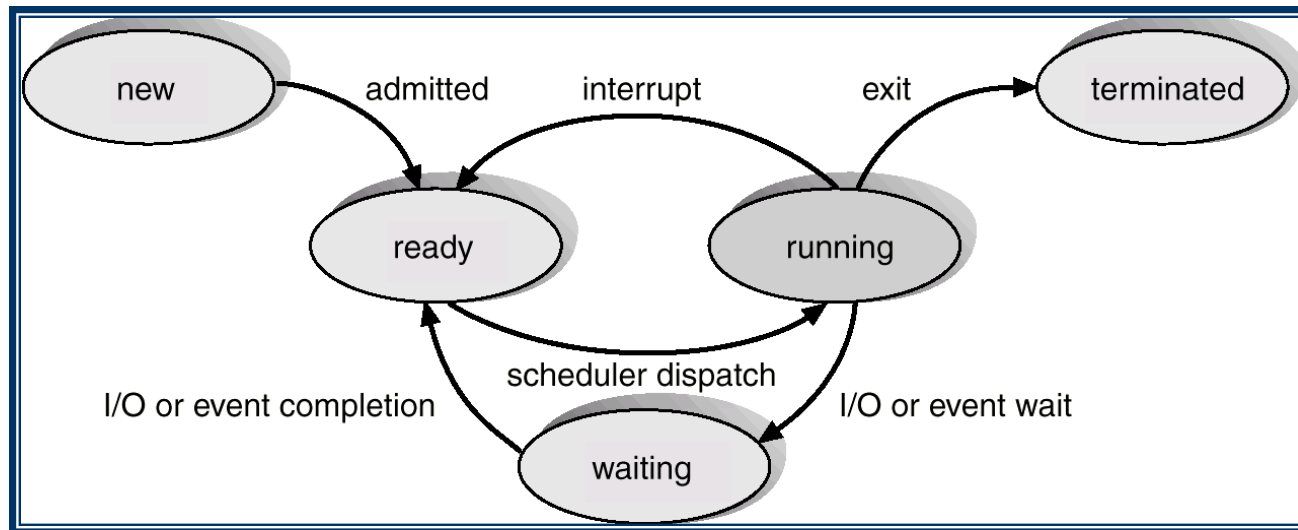


# Process Scheduling

(from the Book, with corrections and additions)



# Back to the state machine

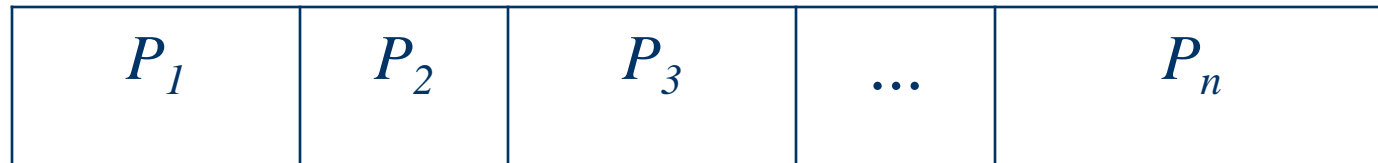




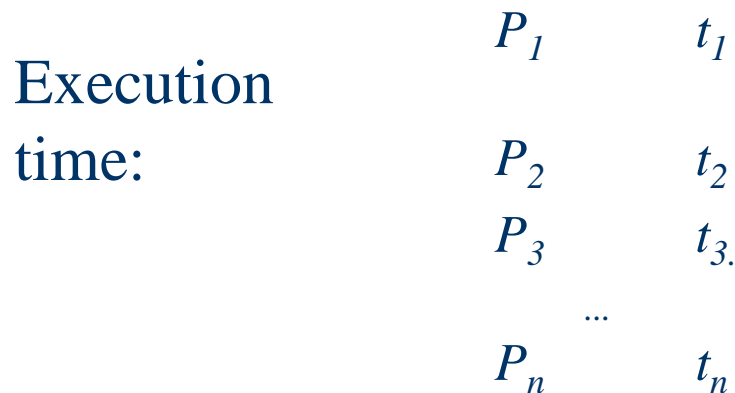
# Scheduling Algorithms

- ◆ First-Come, First-Served (FCFS) Scheduling
- ◆ Priority Scheduling
  - Special Case: Shortest-Job-First (SJF) (or rather *Shortest Next CPU Burst First*)
- ◆ Round Robin Scheduling
- ◆ Multilevel Queue Scheduling
  - Multilevel Queue Feedback Scheduling

# A Management Tool: *Gantt Chart*



$$\begin{array}{ccccccc}
 0 & & t_1 & & t_1 + t_2 & & t_1 + t_2 + t_3 \dots & & \sum_{i=1}^n t_i
 \end{array}$$



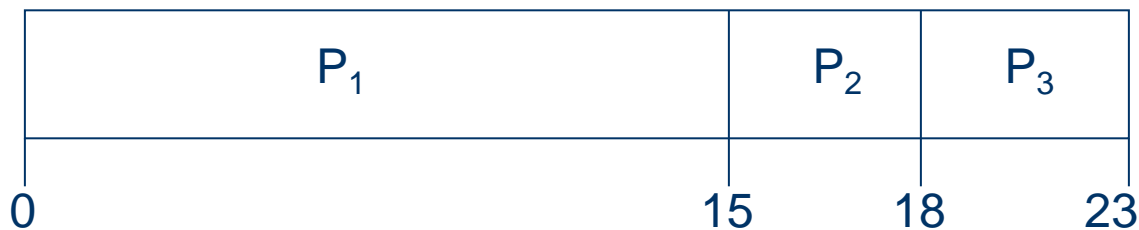
Average  
Waiting  
Time:

$$\frac{1}{n} \sum_{i=1}^{n-1} (n-i) t_i$$

# First-Come, First-Served (FCFS)

<u>Process</u>	<u>Burst Time (in milliseconds)</u>
$P_1$	15
$P_2$	3
$P_3$	5

- Suppose that the processes arrive in the order:  $P_1, P_2, P_3$   
The Gantt Chart for the schedule is:

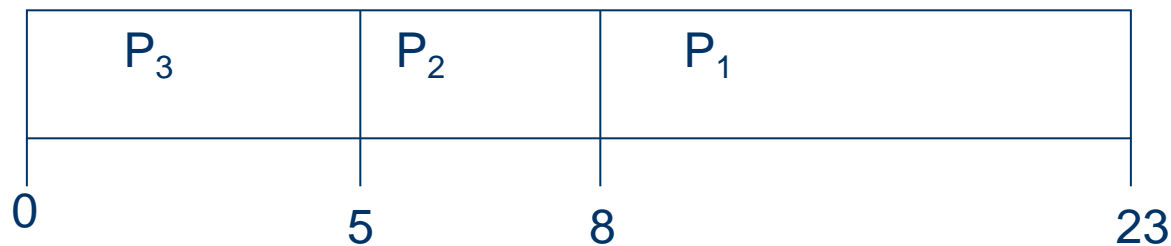


- Waiting time for  $P_1 = 0$ ,  $P_2 = 15$ ;  $P_3 = 3 + 15 = 18$
- Average waiting time:  $(15 + 18)/3 = 11$

# Let us try to rearrange the processes...

<u>Process</u>	<u>Burst Time</u>
$P_3$	5
$P_2$	3
$P_1$	15

Then the Gantt Chart for the schedule is:

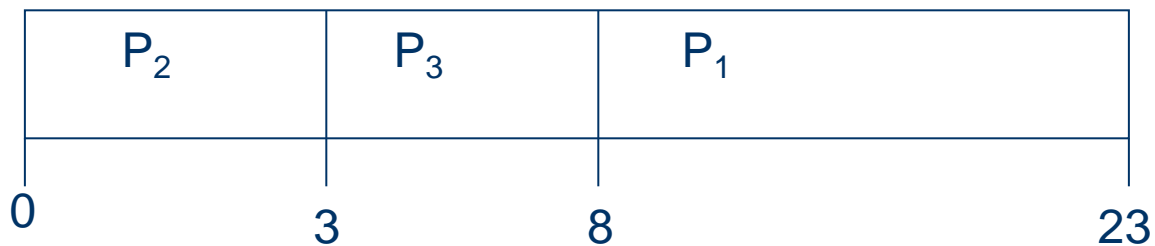


- ♦ Waiting time for  $P_3 = 0$ ,  $P_2 = 5$ ;  $P_1 = 8$
- ♦ Average waiting time:  $(5 + 8)/3 = 4 \frac{1}{3}$

# Or, better yet...

<u>Process</u>	<u>Burst Time</u>
$P_2$	3
$P_3$	5
$P_1$	15

Then the Gantt Chart for the schedule is:



- ♦ Waiting time for  $P_2 = 0$ ,  $P_3 = 3$ ;  $P_1 = 8$
- ♦ Average waiting time:  $(3 + 8)/3 = 3 \frac{2}{3}$

# FCFS Drawbacks:

- ◆ Allows CPU-bound processes to hug the CPU
- ◆ May have a terrible effect on I/O utilization
- ◆ In the absence of pre-emption is unacceptable in a multi-user environment

# Shortest Job First (SJF)

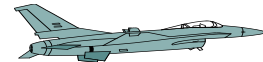
- ◆ This algorithm predicts the *next* CPU burst of a ready process and builds the queue sorted in the ascending order of the *expected* CPU bursts:



3



2



1

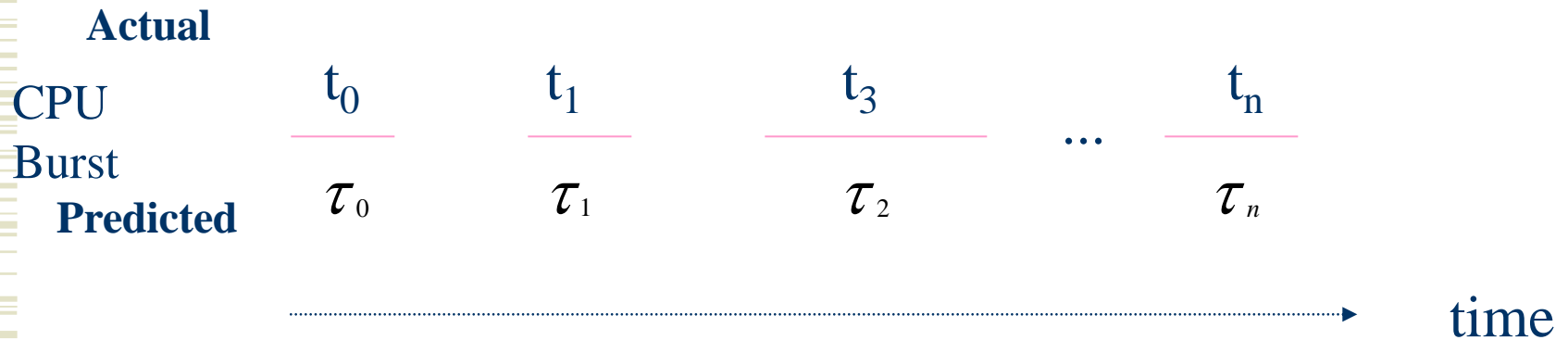
# SJF Issues

- ◆ It is optimal as far as waiting is concerned: it minimizes the average waiting time for a given set of processes (as you will prove when doing homework!)
- ◆ But then it depends on predicting the CPU burst times



# But *how* do we predict the next burst? (An **important** technique to remember!)

## ◆ *Exponential Averaging*:



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

$$0 \leq \alpha \leq 1$$

$\tau_0$  is guessed or selected randomly

# Exponential Averaging

- ◆ Is a *smoothing* mechanism
- ◆ Is applicable to many domains [e.g., signal processing, operating systems, Internet transport (TCP), and Internet QoS in routers]
- ◆ Stores the past history:  $\{t_k\}_{k=1}^{n-1}$
- ◆ Assigns lower weights to more remote terms:

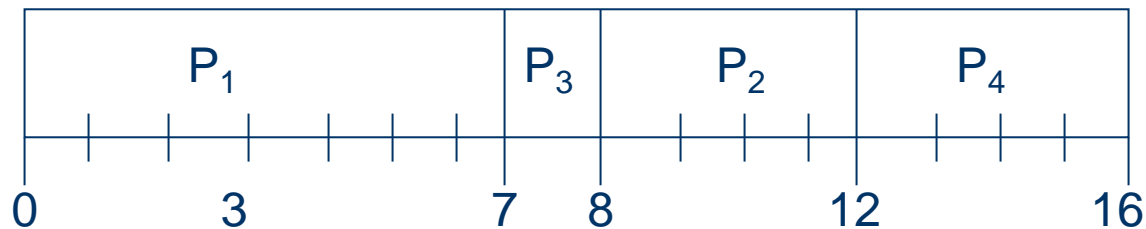
$$\tau_{n+1} = (1 - \alpha)^{n+1} \tau_0 + \alpha \sum_{k=0}^n (1 - \alpha)^{n-k} t_k$$

# Preemption in SJF: SRTF

- ◆ SJF can be either *preemptive* or *non-preemptive*
- ◆ When the process with a shorter CPU burst than what remains for the currently running process arrives into the ready queue, a choice can be made to preempt the running process. The resulting scheduling discipline is called *Shortest-Remaining-Time-First (SRTF)*
- ◆ ***NB: The decision is made only when 1) a new process arrives or 2) when a process has finished its execution.***

# Example of Non-Preemptive SJF

<u>Process</u>	<u>Arrival Time</u>	<u>Expected Burst Time</u>
$P_1$	0	7
$P_2$	2	4
$P_3$	4	1
$P_4$	6	4

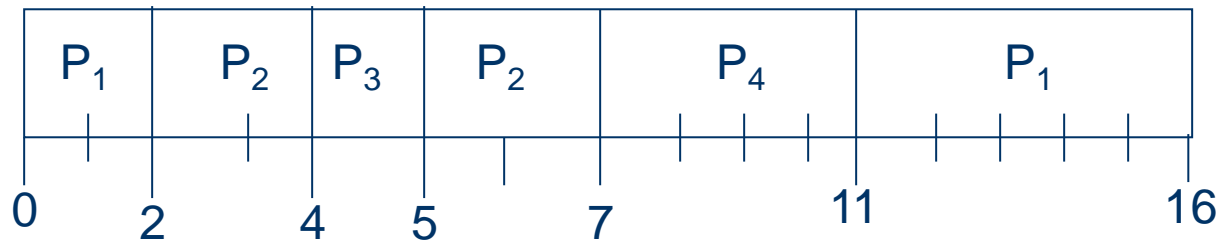


- ♦ Average waiting time =  $(0 + 6 + 3 + 6)/4 = 3 \frac{3}{4}$

# Example of SRTF

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
$P_1$	0	7
$P_2$	2	4
$P_3$	4	1
$P_4$	6	4

- ◆ The Gant chart:



- ◆ Average waiting time =  $(9 + 1 + 0 + 1)/4 = 2 \frac{3}{4}$

# Priority Scheduling

- ◆ Here, a number (*priority*) is assigned to each process, and the processes are scheduled so that the process with the highest priority runs first
- ◆ We *define* that a smaller number designates a higher priority (  $2 \succ 5$ ), but it could be defined the other way around
- ◆ SJF is a type of priority scheduling, where the expected burst time is used as the value of priority

# Assignment of Priorities

- ◆ Priorities can be assigned *internally* by the operating system (e.g., all OS activities: device managers, scheduler, etc. have higher priorities than user processes)
- ◆ Priorities can be also assigned *externally* (e.g., by a price a user pays or a user's relative importance)

# A problem with Priority Scheduling

*Starvation (indefinite blocking)*: A process is waiting for a CPU (in a *ready*) state, but it never gets it, because new processes with higher priority always come in

**A solution:** *Aging*, which increases the priority of a waiting process after a certain period of time it spent in a ready queue; thus, each process will eventually get a sufficiently high priority to run



# Time slicing: Round-Robin (RR)

- ◆ With the methods we dealt with so far, the preemption occurred *only* when a new process arrived
- ◆ With *time slicing*, a process may be pre-empted after it ran for a defined period of time—a *time slice* (also called a *quantum*)
- ◆ When a pre-empted process is simply put back into the *ready* queue, the scheduling algorithm is called *Round Robin (RR)*

# Example of RR with Time Quantum = 20 msec

<u>Process</u>	<u>Burst Time</u>	<u>Runs</u>
$P_1$	53	53, 33, 13
$P_2$	18	18
$P_3$	59	59, 39, 19
$P_4$	43	43, 23, 3

- ♦ The Gantt chart:

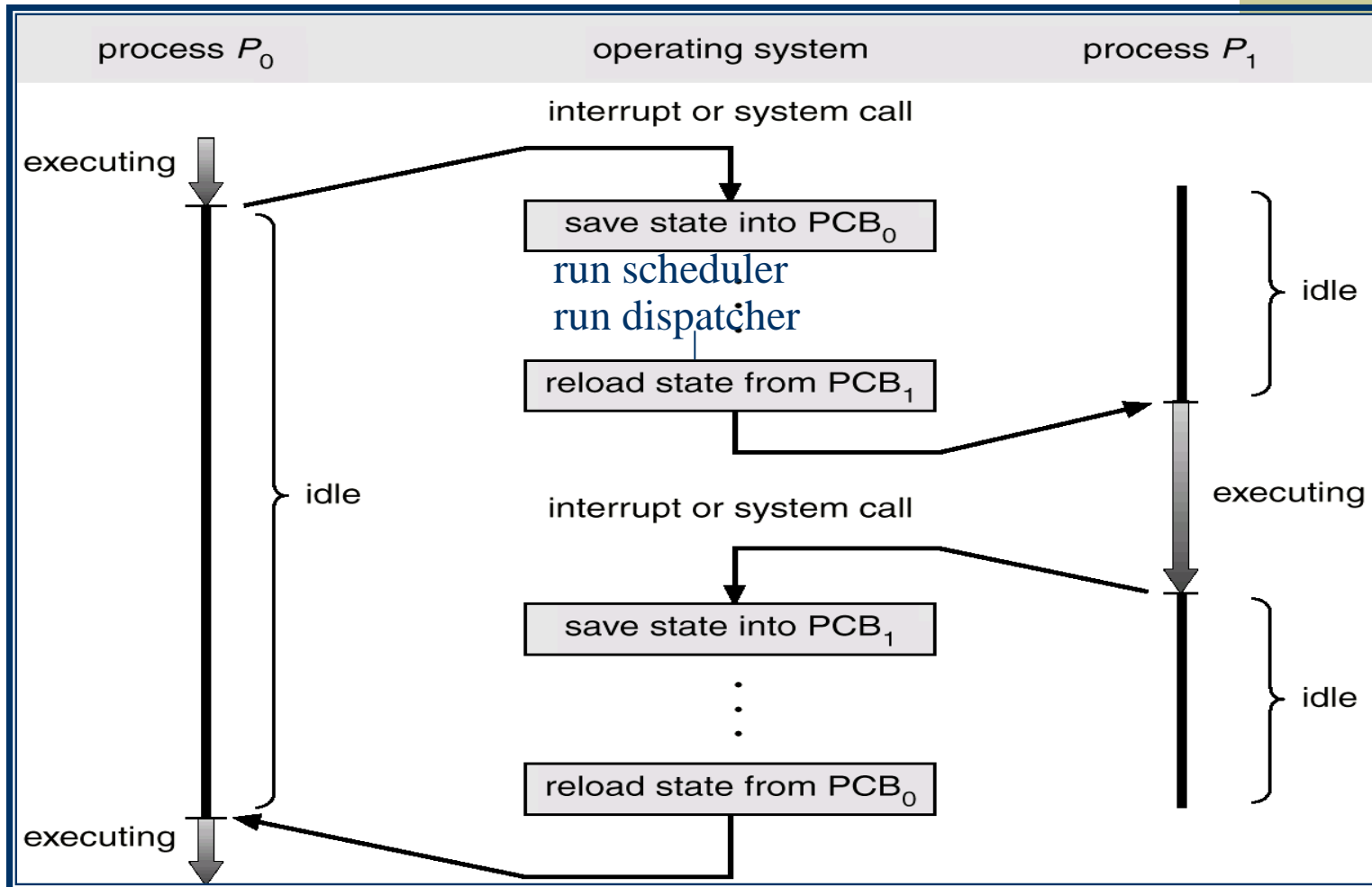
	<div><div>P<sub>1</sub></div><div>P<sub>2</sub></div><div>P<sub>3</sub></div><div>P<sub>4</sub></div><div>P<sub>1</sub></div><div>P<sub>3</sub></div><div>P<sub>4</sub></div><div>P<sub>1</sub></div><div>P<sub>3</sub></div><div>P<sub>4</sub></div></div>										
<i>t</i> :	0	20	38	58	78	98	118	138	151	170	173
Time left:		33	0	39	23	13	19	3	0	0	0

# Some observations about RR

- ◆ If a slice is large (i.e., larger than the what it takes for the longest process to execute, RR is equivalent to FCFS)
- ◆ If there are  $n$  processes in the ready queue, and the quantum,  $q$  msec, is much smaller than the shortest process' execution time, then each process gets  $1/n$  of the CPU time and waits for its next turn for  $(n-1)q$  msec
- ◆ But...

# What is involved in preemption?

After the Book



# Scheduling and Context Switching Time vs. Quantum

- ◆ Scheduling and Context Switching (SCS) take time (which is why they must be programmed as efficiently as possible!)
- ◆ If the time slice is comparable to that of SCS, the overhead is unbearable:
  - Say, one process needs time  $T$  to execute
  - $q$  is the quantum,  $c$  is the context switching time
  - $T + \frac{T}{q} c$  is needed. If  $c = q$ , 50% of CPU is wasted!

# Context Switch Time

- ◆ Typically varies from 1 to 1000 microseconds
- ◆ May be significantly decreased by the hardware (CDC and *Sun UltraSPARC* CPUs provide multiple sets of registers)
- ◆ Can be further decreased using *threads*

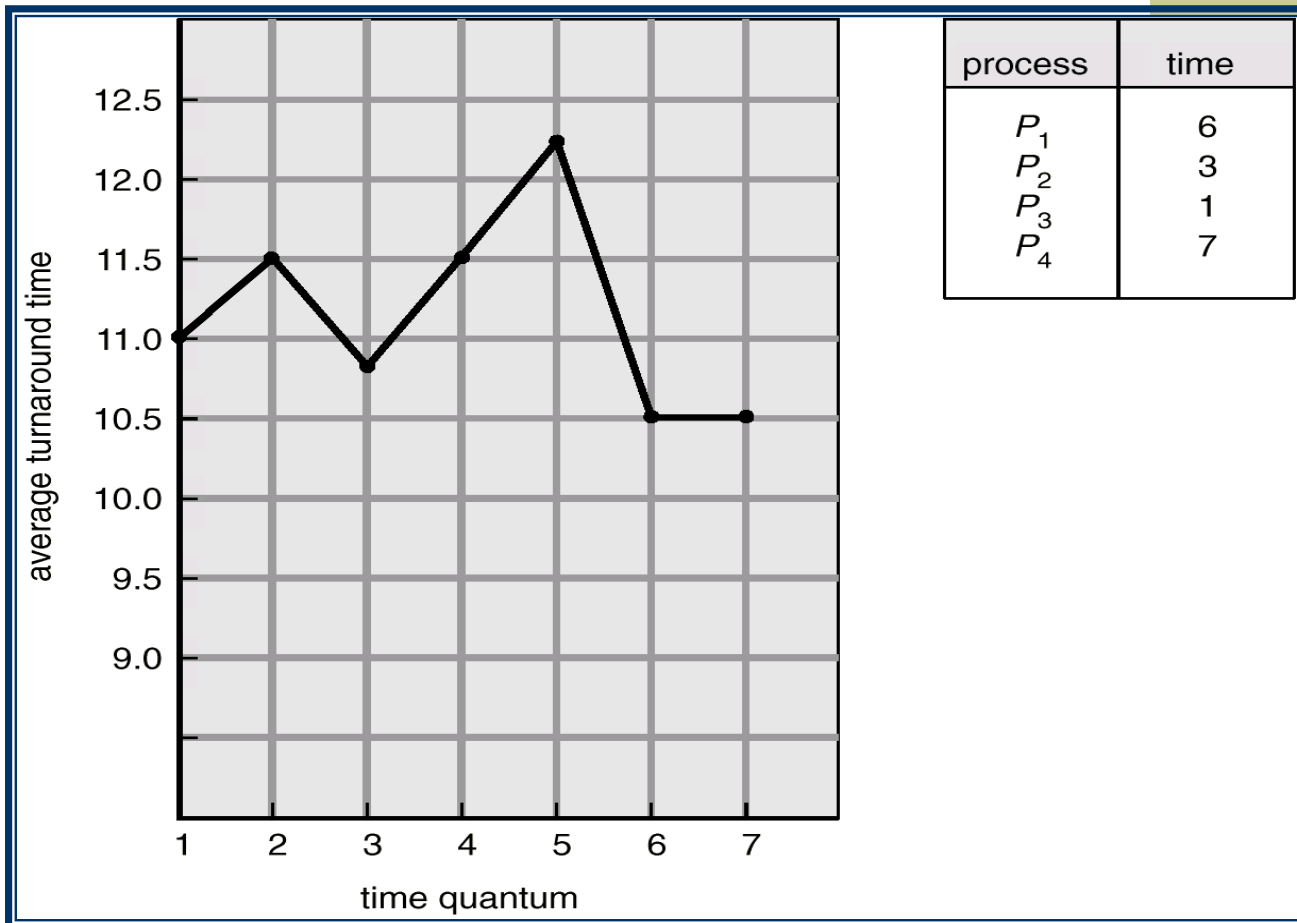
# The Rule of Thumb

- ◆ The scheduling and context switching time (SCS) time is known, so the quantum should be selected appropriately
- ◆ Yet it may not be too large, or RR will degenerate into FCFS!

👍  $q > 80\%$  of a CPU burst

# The effect of the quantum selection on Turnaround Time

*Note: The  
SCS time is 0*

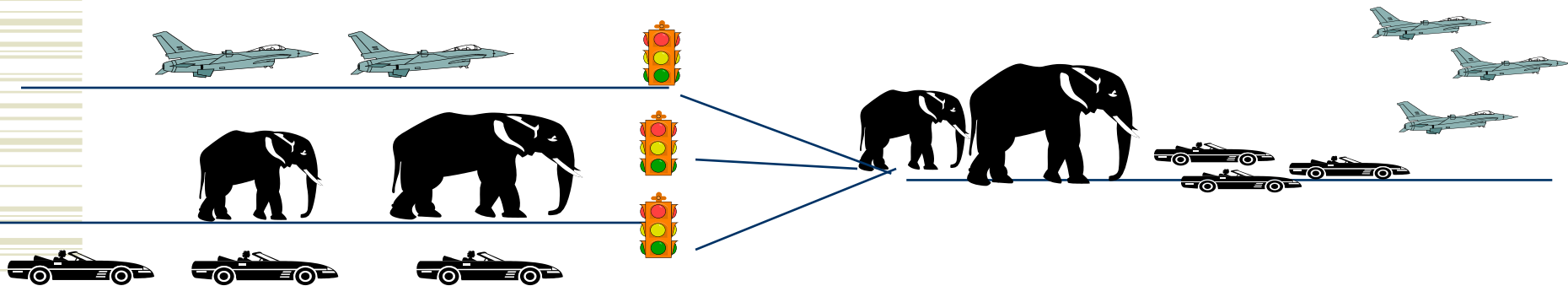




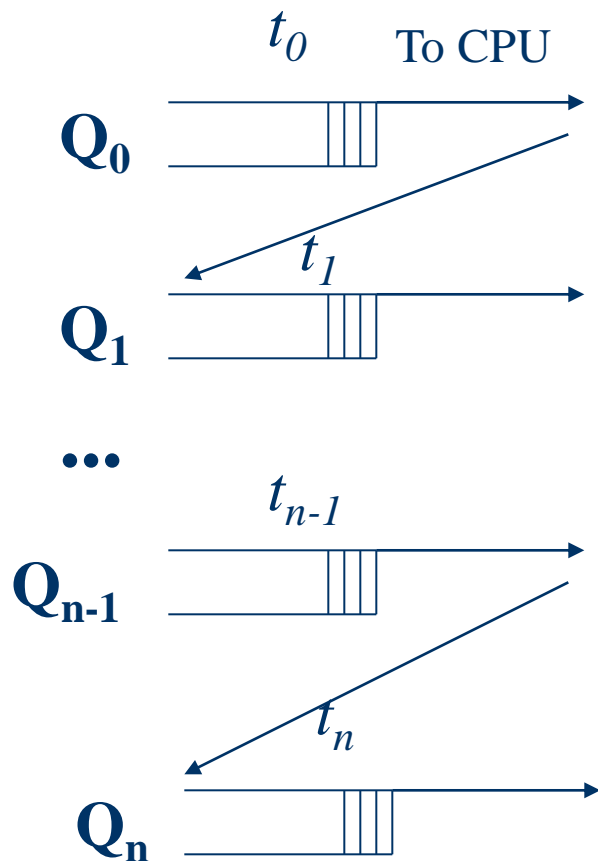
# Multilevel Queue Scheduling

- ◆ Ready queue is partitioned into separate queues:  
foreground (interactive)  
background (batch)
- ◆ Each queue has its own scheduling algorithm:  
foreground – typically, RR  
background – typically, FCFS
- ◆ The scheduling selects among the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background, which leaves the possibility of starvation for background processes.
  - Time slice – each queue gets a certain amount of CPU time which it can distribute among the processes (for example, 80% to foreground in RR and 20% to background in FCFS)

# Multilevel Queue Scheduling



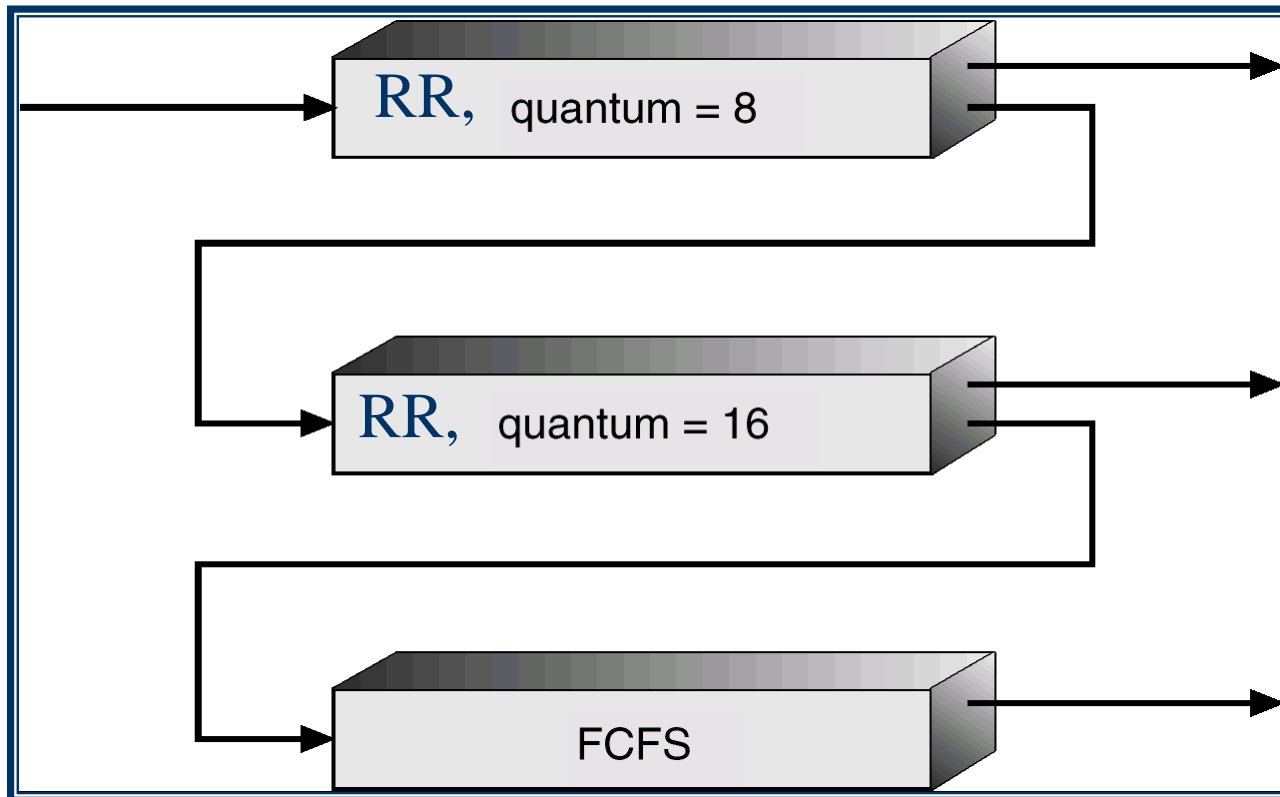
# Multilevel Feedback Queue Scheduling—a variation of aging



Processes in  $Q_i$  have a higher priority than those in  $Q_{i+1}$ . Until  $Q_i$  is empty,  $Q_{i+1}$  can not execute. But once a process has used its time slice,  $t_i$ , it is moved into  $Q_{i+1}$  instead of back to  $Q_i$ , and, of course,  $t_i < t_{i+1}$ .

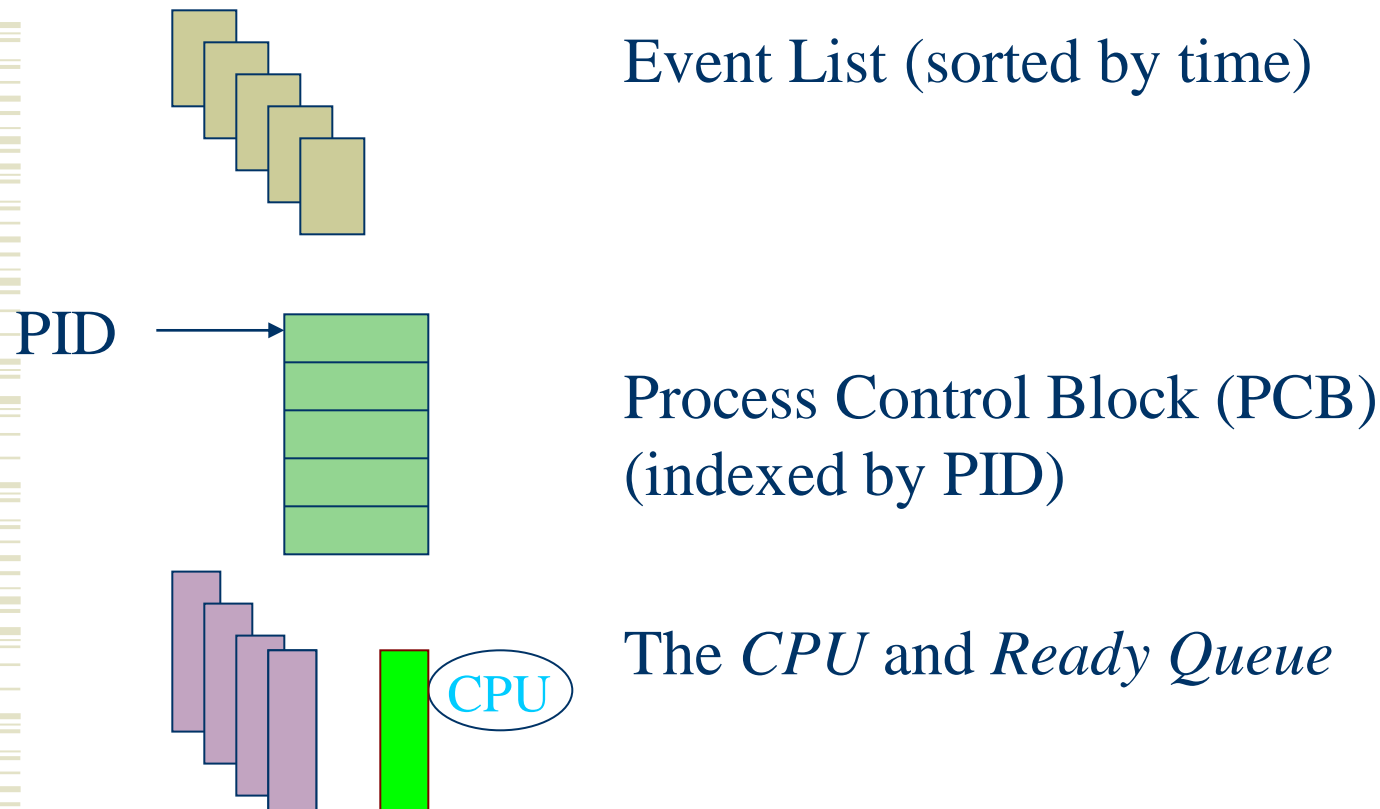
# An Example

After the Book



# Attachment

## Simulation (for HW #4): *Common Data Structures*



# Simulation: FCFS

- ◆ Two types of events: process *arrival* and process *completion*:

*arrival*:

```
{
    create a PCB entry;
    if CPU is free
    {
        mark it busy;
        PCB.state=running;
        schedule completion;
        ...
    }
    else
    {
        PCB.state=ready;
        Update the Ready Queue;
        ...
    }
```

# Simulation: FCFS (*cont.*)

- ◆ Two types of events: process *arrival* and process *completion*:

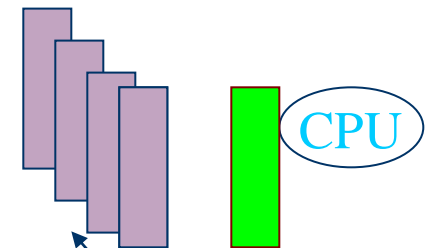
*completion*:

```
{
    destroy the PCB entry;
    if the Ready Queue is not empty
    {
        take the next process from the Ready Queue;
        select its PCB;
        PCB.state=running;
        schedule completion;
        ...
    }
    else
    {
        mark CPU free;
        ...
    }
}
```

# Simulation: SJF

- Everything is the same as with FCFS but *one thing*:

```
arrival:
{
    create a PCB entry;
    if CPU is free
    {
        mark it busy;
        PCB.state=running;
        schedule completion;
        ...
    }
    else
    {
        PCB.state=ready;
        Update the Ready Queue;
        ...
    }
}
```



**Sorted by  
completion  
times**

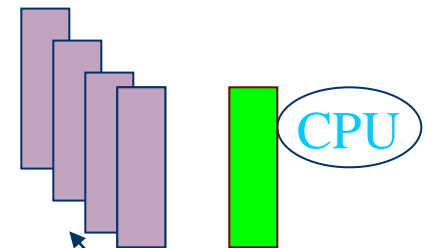


# Simulation: SRTF

- ◆ An **essential difference** because of **preemption** handling:

*arrival:*

```
{  
    create a PCB entry;  
    if CPU is free  
    {  
        mark it busy;  
        PCB.state=running;  
        schedule completion;  
        ...  
    }  
    else  
    {  
        New code;  
    }  
}
```



**Sorted by time  
remaining to completion**

# Simulation: SRTF (cont.)

- ◆ The new code for the arrival case when the CPU is busy:

```
insert the job in the Ready Queue (RQ);  
sort the queue by the remaining time;  
compute the remaining time of the job at the CPU;  
if it is greater than that of the first RQ entry then  
{  
    Update its remaining time in the PCB;  
    PCB.state=ready;  
    Remove its completion event from the Event str.;  
    Schedule the completion event for the first RQ  
    entry, change its PCB.state, etc.  
}
```

# Simulation: RR

- ♦ **Three** types of events: *arrival*, *completion*, and *timer interrupt*:

*arrival*:

```
{
    create a PCB entry;
    if CPU is free
    {
        mark it busy;
        PCB.state=running;
        schedule either completion or timer interrupt;
        ...
    }
    else
    {
        PCB.state=ready;
        Update the Ready Queue;
        ...
    }
}
```

# Simulation: RR (cont.)

*completion: (the same as in FCFS)*

*timer interrupt:*

```
{  
    Change the state and remaining time of the job  
    currently running;  
    Place it in the Ready Queue (FIFO in pure RR);  
    Remove the first entry in the Ready Queue;  
    Its PCB.state=running;  
    schedule either its completion or its  
    timer interrupt;  
    ...  
}
```