

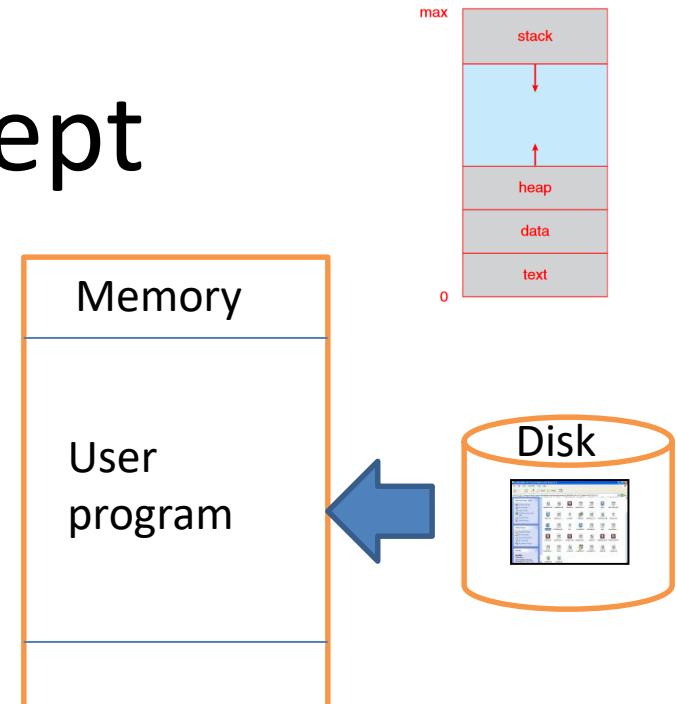
# Process management

## *What are we going to learn?*

- **Processes** : Concept of processes, process scheduling, co-operating processes, inter-process communication.
- **CPU scheduling** : scheduling criteria, preemptive & non-preemptive scheduling, scheduling algorithms (FCFS, SJF, RR, priority), algorithm evaluation, multi-processor scheduling.
- **Process Synchronization** : background, critical section problem, critical region, synchronization hardware, classical problems of synchronization, semaphores.
- **Threads** : overview, benefits of threads, user and kernel threads.
- **Deadlocks** : system model, deadlock characterization, methods for handling deadlocks, deadlock prevention, deadlock avoidance, deadlock detection, recovery from deadlock.

# Process concept

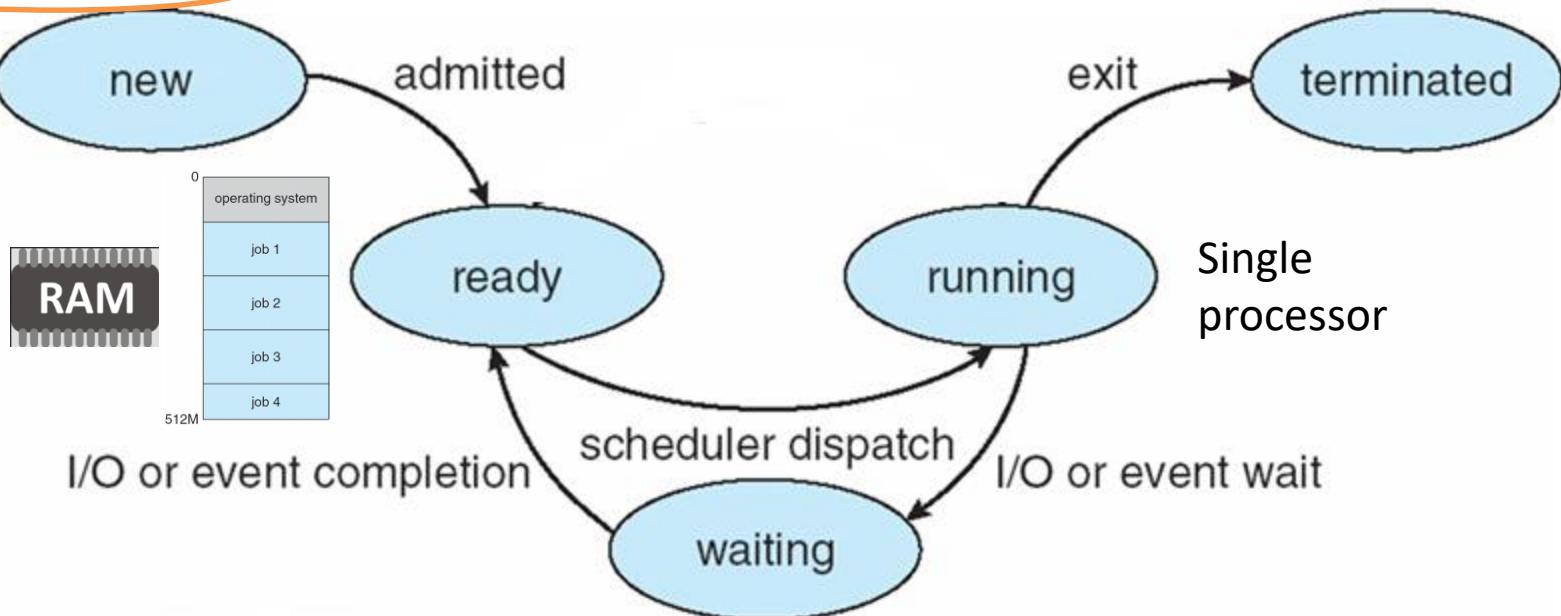
- Process is a dynamic entity
  - Program in execution
- Program code
  - Contains the text section
- Program becomes a process when
  - executable file is loaded in the memory
  - Allocation of various resources
    - Processor, register, memory, file, devices
- One program code may create several processes
  - One user opened several MS Word
  - Equivalent code/text section
  - Other resources may vary



# Process State

- As a process executes, it changes *state*
  - **new**: The process is being created
  - **ready**: The process is waiting to be assigned to a processor
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **terminated**: The process has finished execution

# Process State diagram

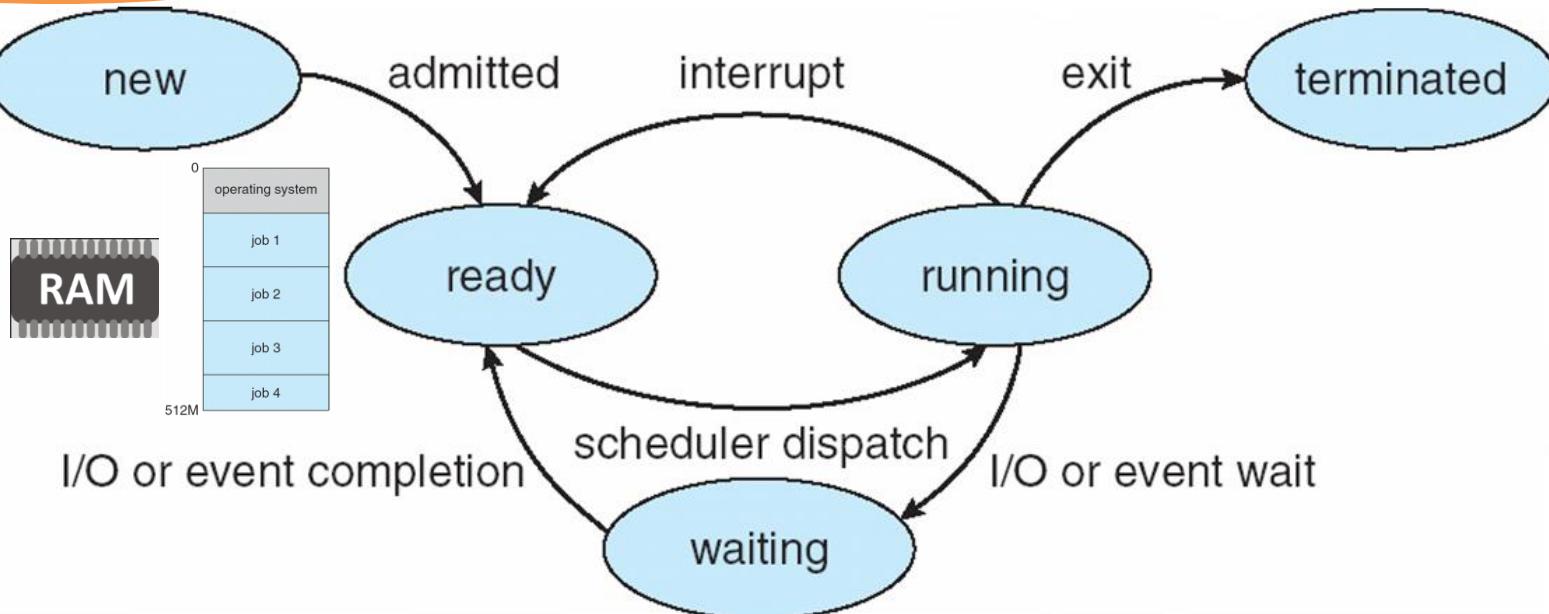


## Multiprogramming

As a process executes, it changes *state*

- **new**: The process is being created
- **running**: Instructions are being executed
- **waiting**: The process is waiting for some event to occur
- **ready**: The process is waiting to be assigned to a processor
- **terminated**: The process has finished execution

Job pool



## Multitasking/Time sharing

As a process executes, it changes *state*

- **new**: The process is being created
- **running**: Instructions are being executed
- **waiting**: The process is waiting for some event to occur
- **ready**: The process is waiting to be assigned to a processor
- **terminated**: The process has finished execution

# How to represent a process?

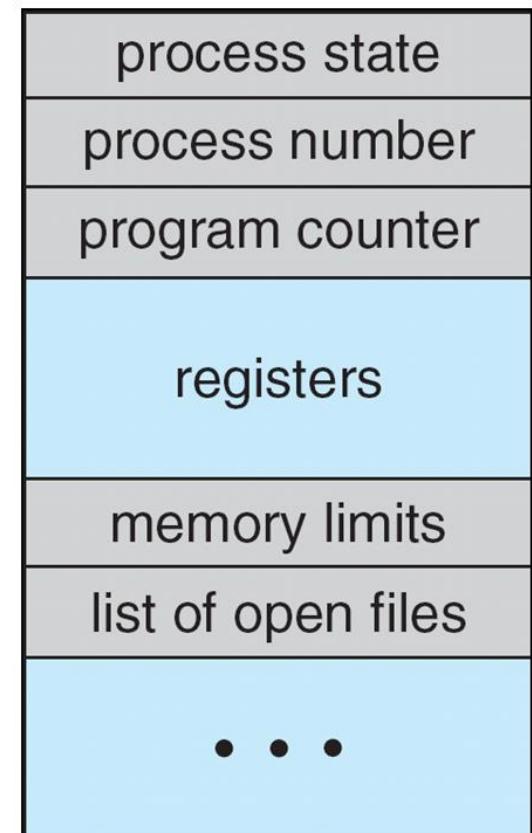
- Process is a dynamic entity
  - Program in execution
- Program code
  - Contains the text section
- Program counter (PC)
- Values of different registers
  - Stack pointer (SP) (maintains process stack)
    - Return address, Function parameters
  - Program status word (PSW)  
  - General purpose registers
- Main Memory allocation
  - Data section
    - Variables
  - Heap
    - Dynamic allocation of memory during process execution

# Process Control Block (PCB)

- Process is represented in the operating system by a Process Control Block

Information associated with each process

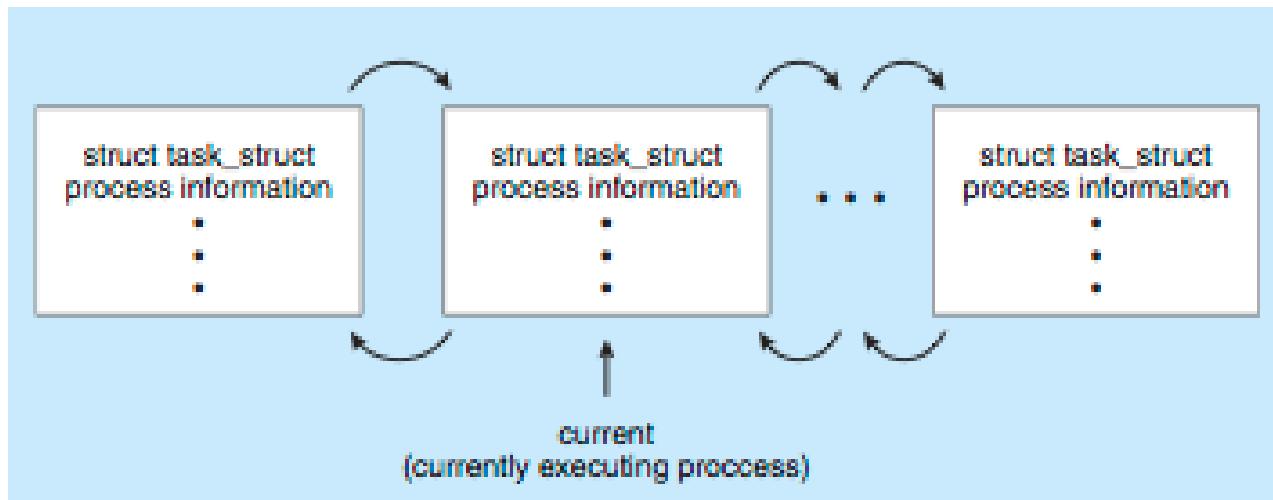
- Process state
- Program counter
- CPU registers
  - Accumulator, Index reg., stack pointer, general Purpose reg., Program Status Word (PSW)
- CPU scheduling information
  - Priority info, pointer to scheduling queue
- Memory-management information
  - Memory information of a process
  - Base register, Limit register, page table, segment table
- Accounting information
  - CPU usage time, Process ID, Time slice
- I/O status information
  - List of open files=> file descriptors
  - Allocated devices



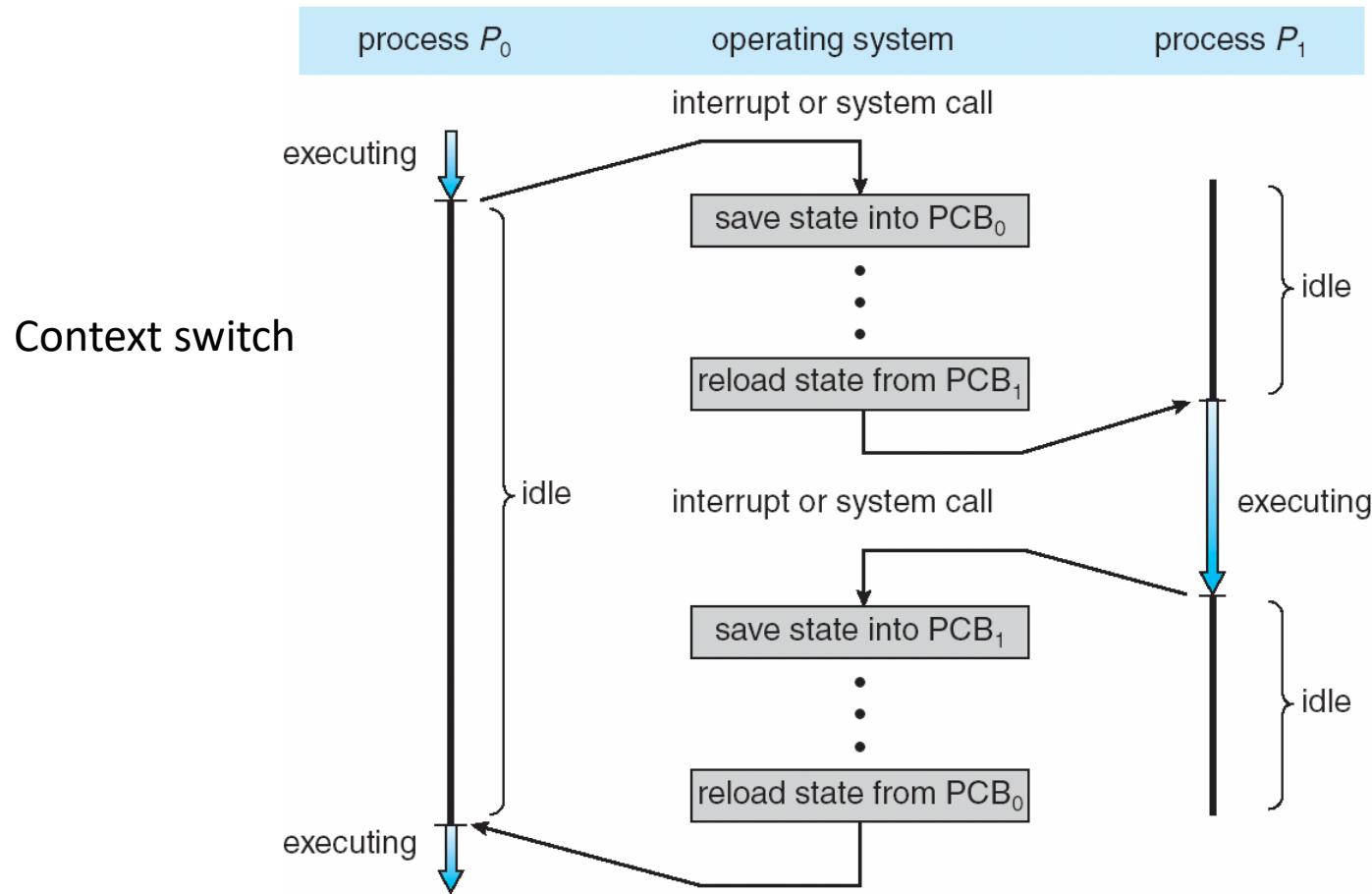
# Process Representation in Linux

Represented by the C structure `task_struct`

```
pid_t pid; /* process identifier */  
long state; /* state of the process */  
unsigned int time_slice /* scheduling information */  
struct task_struct *parent; /* this process's parent */  
struct list_head children; /* this process's children */  
struct files_struct *files; /* list of open files */  
struct mm_struct *mm; /* address space of this pro */
```

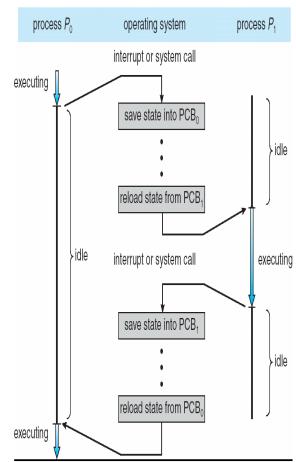


# CPU Switch From Process to Process



# Context Switch

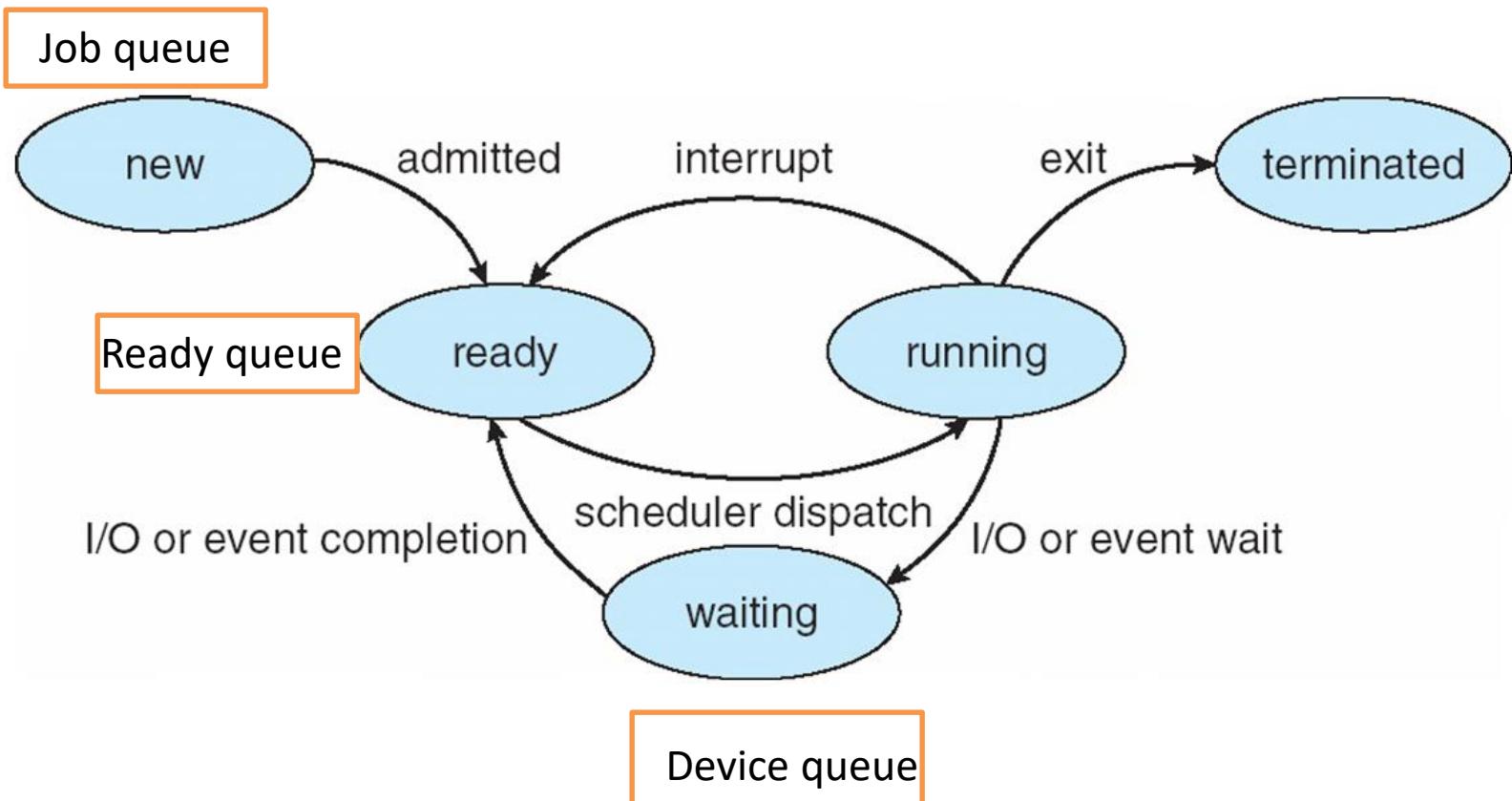
- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a **context switch**.
- **Context** of a process represented in the PCB
- Context-switch time is overhead; the system does not do useful work while switching
  - The more complex the OS and the PCB -> longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU -> multiple contexts loaded at once



# Scheduling queues

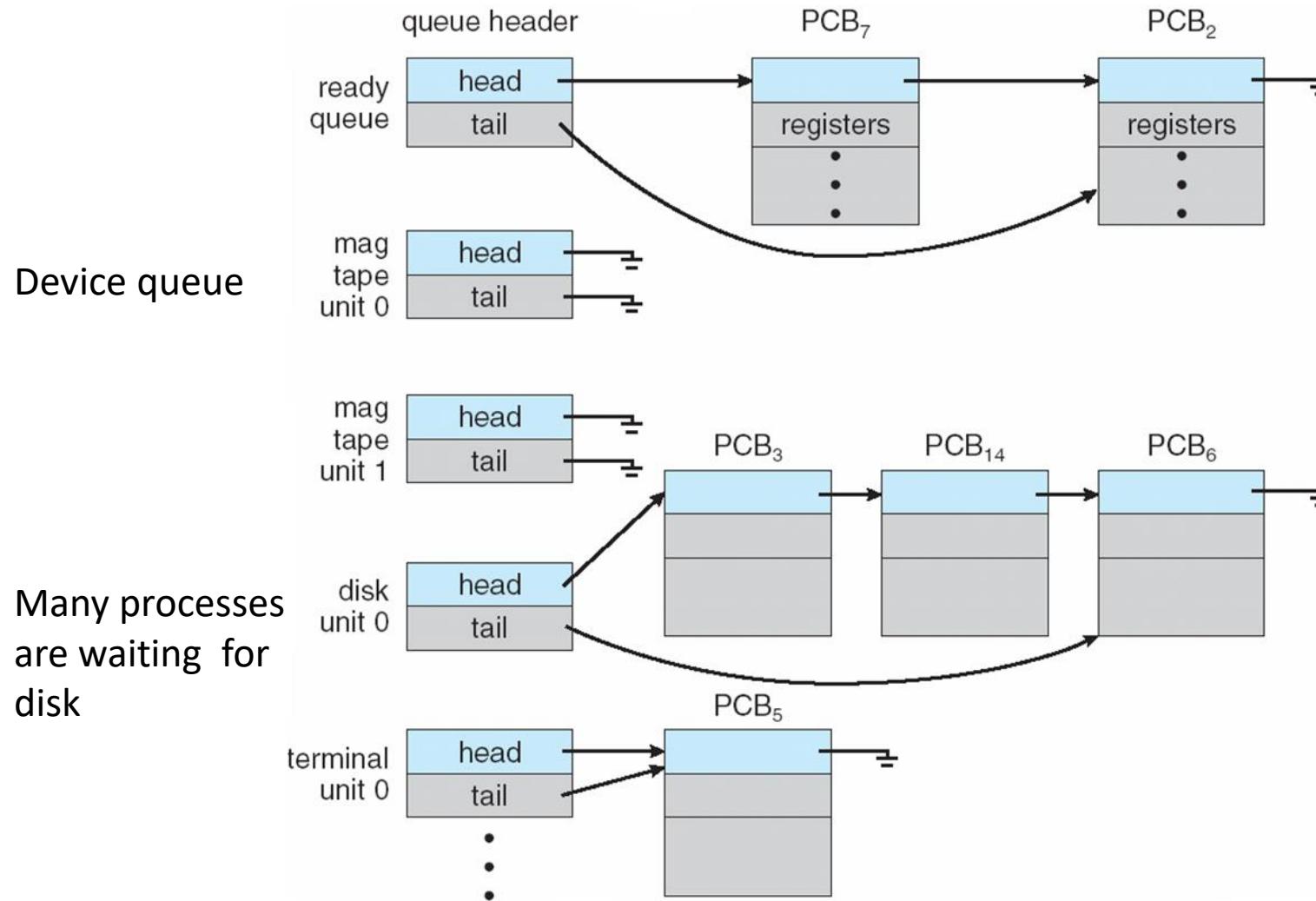
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues

# Scheduling queues



# Ready Queue And Various I/O Device Queues

Queues are linked list of PCB's

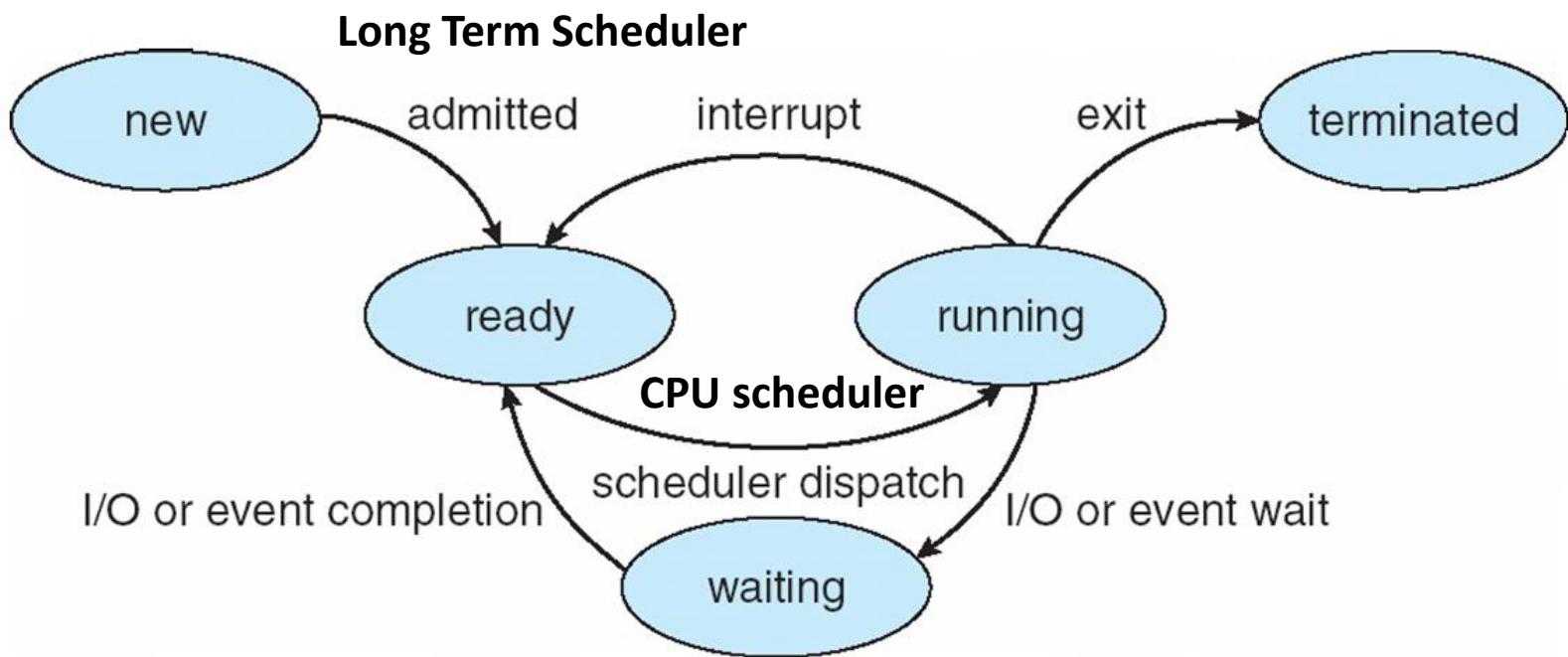


# Process Scheduling

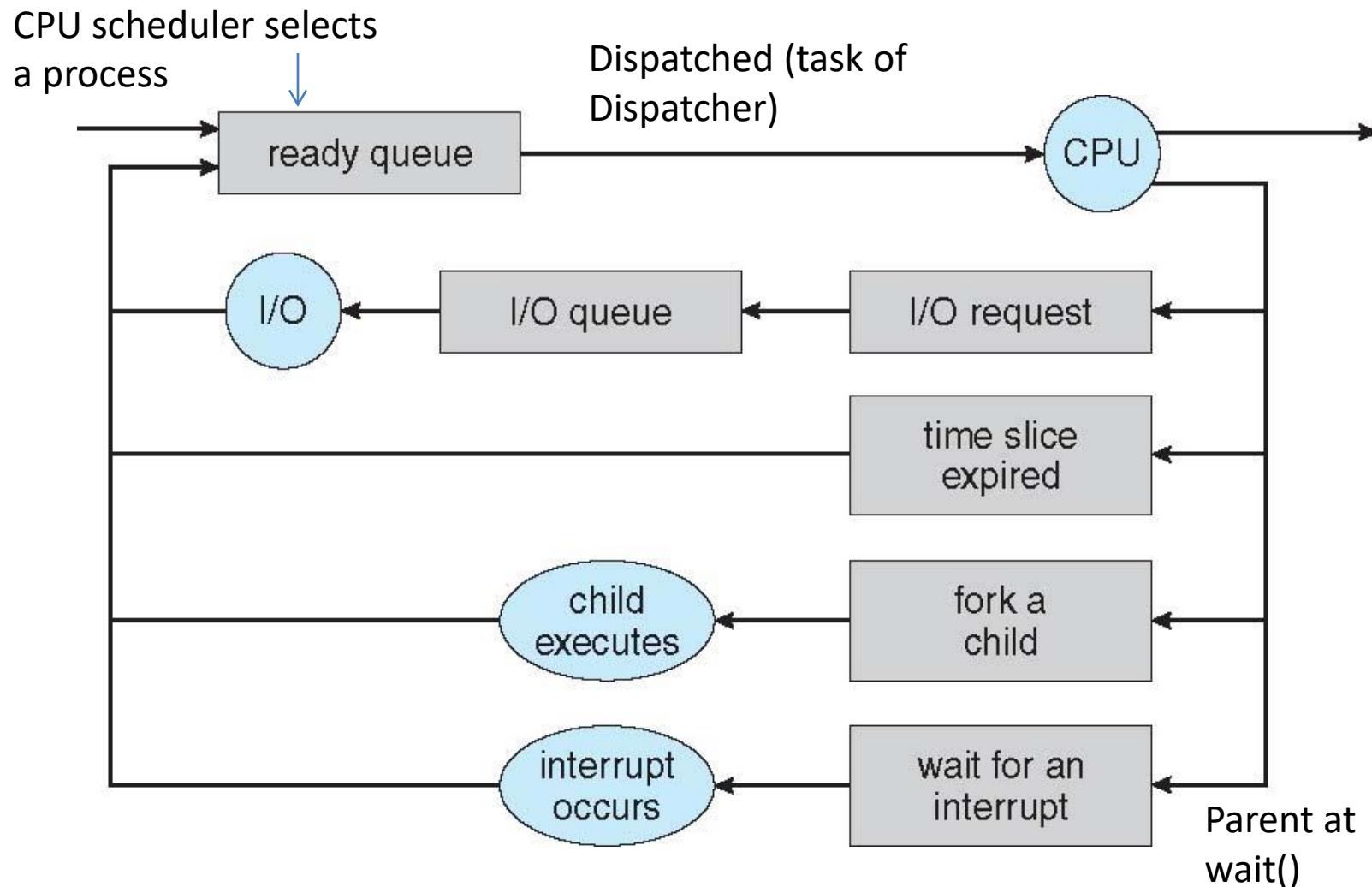
- We have various queues
- Single processor system
  - Only one CPU=> only one running process
- Selection of one process from a group of processes
  - **Process scheduling**

# Process Scheduling

- Scheduler
  - Selects a process from a set of processes
- Two kinds of schedulers
  1. Long term schedulers, job scheduler
    - A large number of processes are submitted (more than memory capacity)
    - Stored in disk
    - Long term scheduler selects process from job pool and loads in memory
  2. Short term scheduler, CPU scheduler
    - Selects one process among the processes in the memory (ready queue)
    - Allocates to CPU



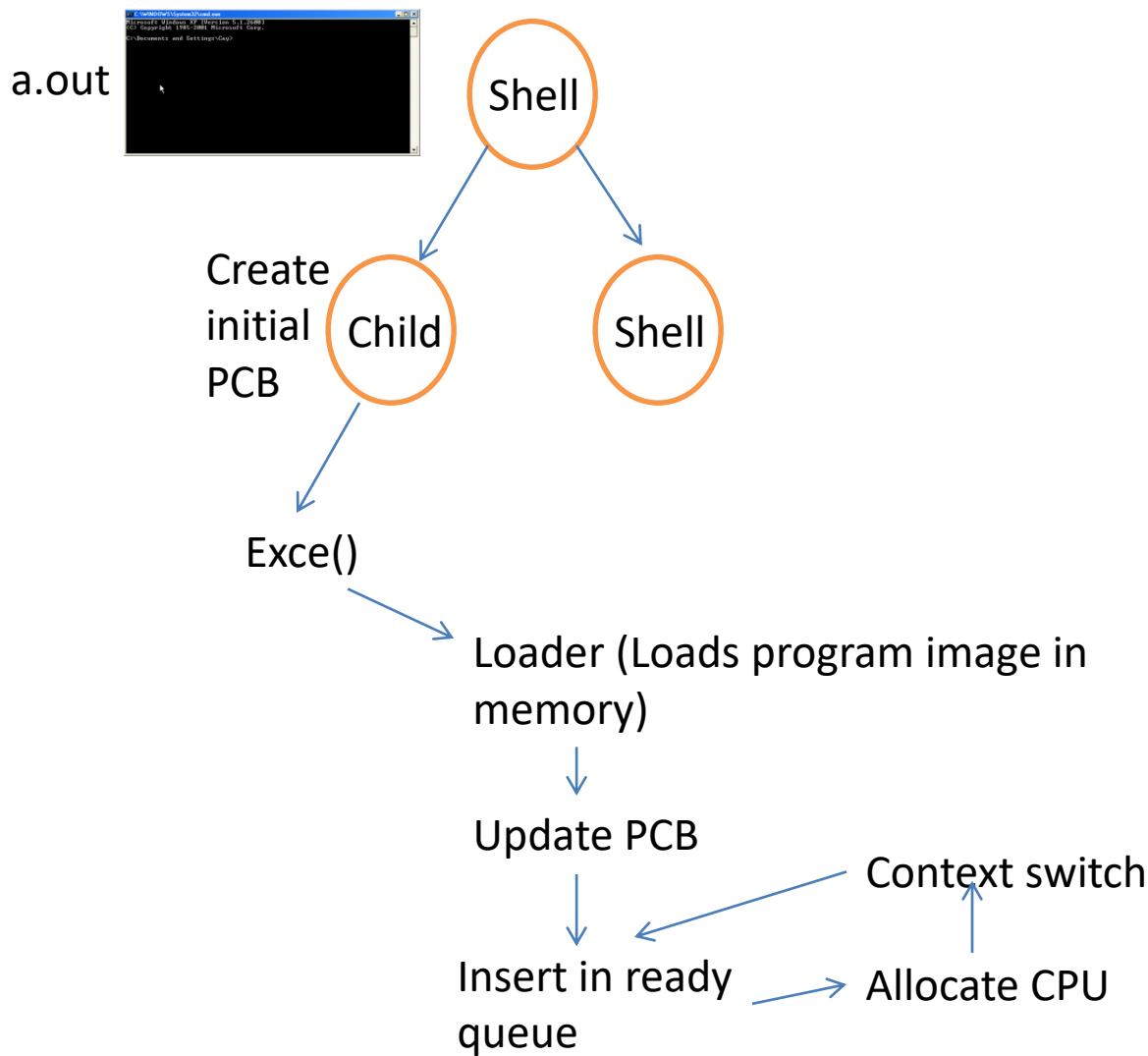
# Representation of Process Scheduling



# Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running

# Creation of PCB



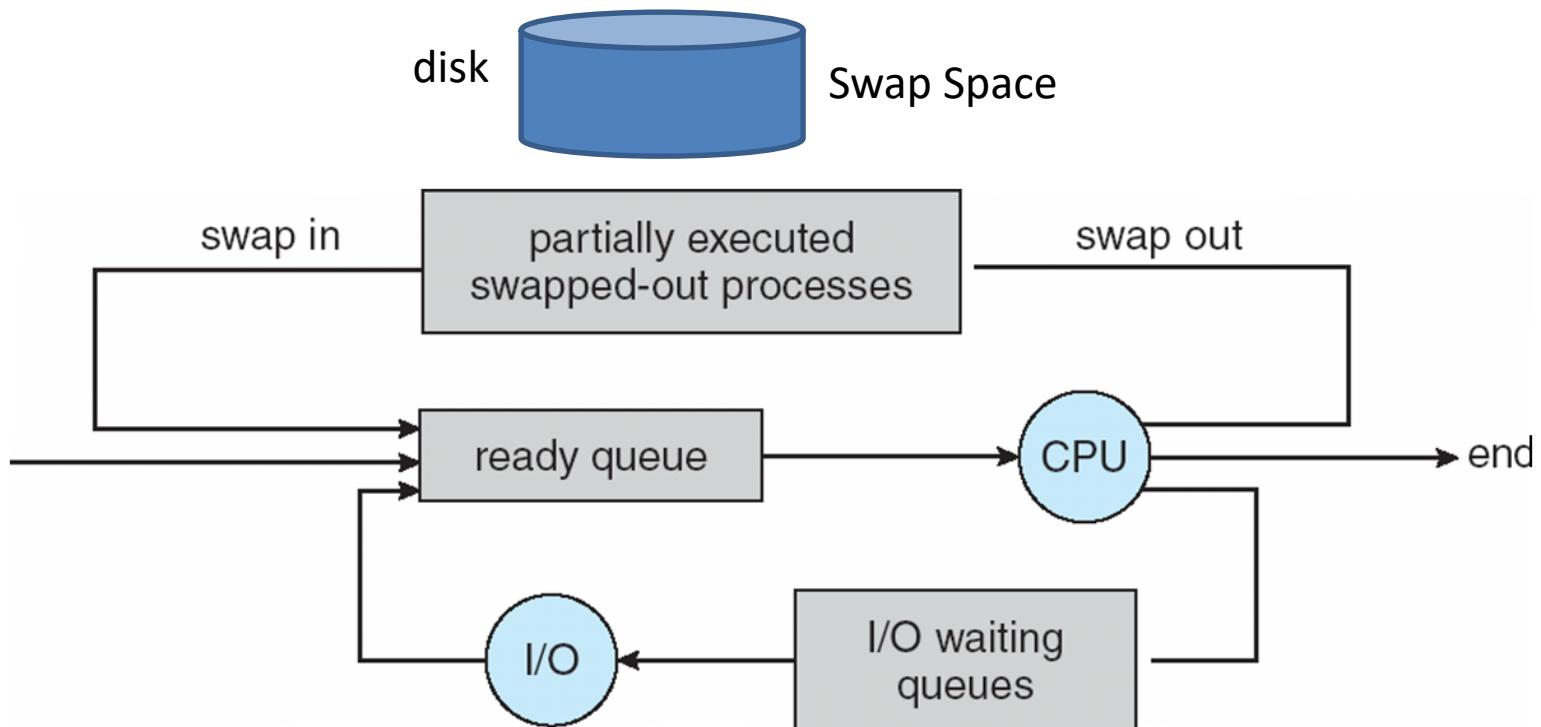
# Schedulers

- **Scheduler**
  - Selects a process from a set
- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system

# Schedulers: frequency of execution

- Short-term scheduler is invoked **very frequently** (milliseconds)  
⇒ (must be fast)
  - After a I/O request/ Interrupt
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (may be slow)
  - The long-term scheduler controls the *degree of multiprogramming*
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
    - Ready queue empty
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
    - Devices unused
- Long term scheduler ensures good process mix of I/O and CPU bound processes.

# Addition of Medium Term Scheduling



## Swapper

# ISR for context switch

```
Current <- PCB of current process
```

```
Context_switch()
```

```
{
```

```
    Disable interrupt;  
    switch to kernel mode  
    Save_PCB(current);  
    Insert(ready_queue, current);  
    next=CPU_Scheduler(ready_queue);  
    remove(ready_queue, next);  
    Dispatcher(next);  
    switch to user mode;  
    Enable Interrupt;
```

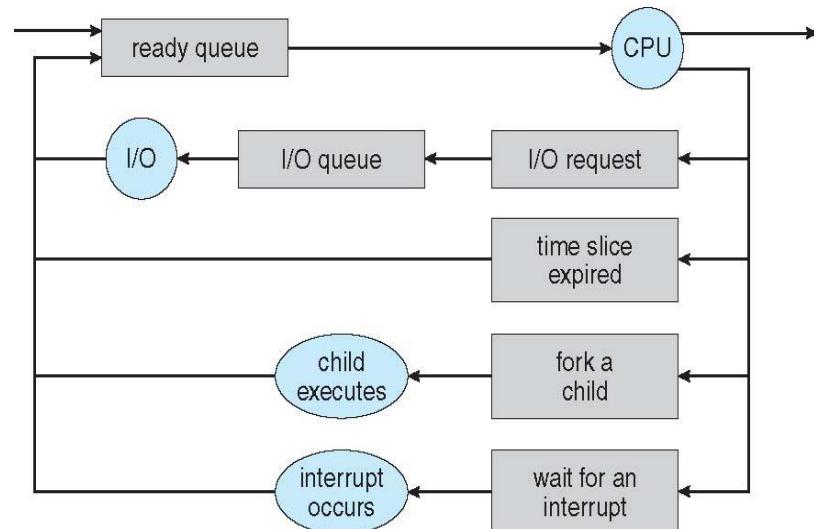
```
}
```

```
Dispatcher(next)
```

```
{
```

```
    Load_PCB(next); [update PC]
```

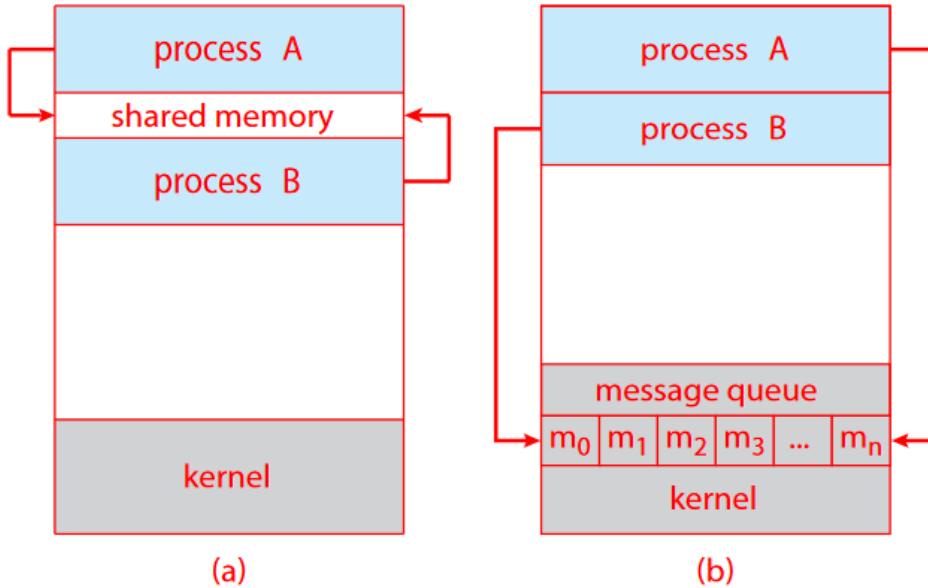
```
}
```



# Interprocess Communication

- Processes within a system may be **independent** or **cooperating**
- Cooperating process can affect or be affected by other processes, including sharing data
- Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data— that is, send data to and receive data from each other.
- Cooperating processes need **interprocess communication (IPC)**
- Two models of IPC
  - Shared memory
  - Message passing

# Interprocess Communication



In the **shared-memory model**, a **region of memory** that is shared by the cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.

In the **message-passing model**, communication takes place by means of messages exchanged between the cooperating processes (Kernel involvement, slow)

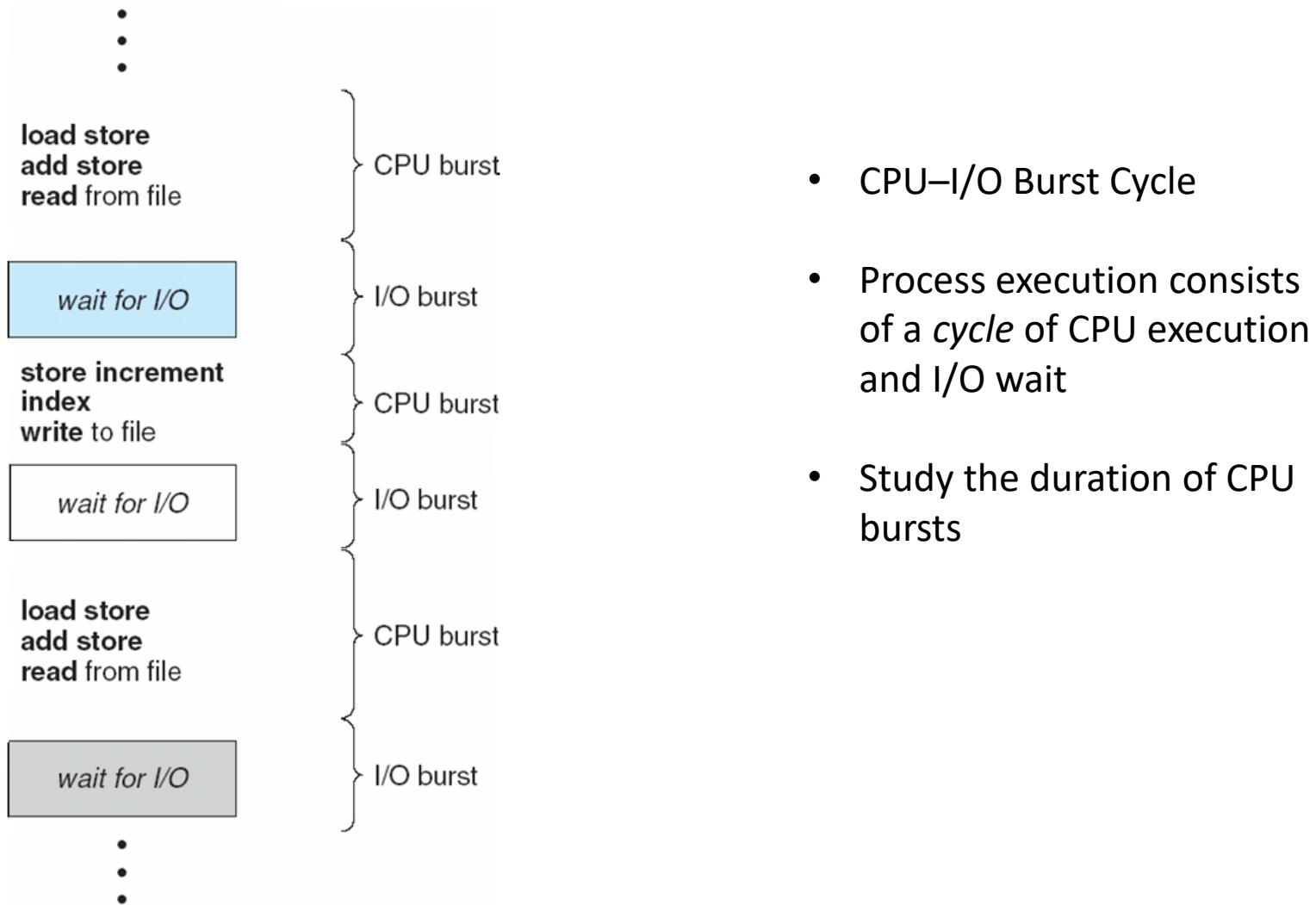
# CPU Scheduling

- Describe various CPU-scheduling algorithms
- Evaluation criteria for selecting a CPU-scheduling algorithm for a particular system

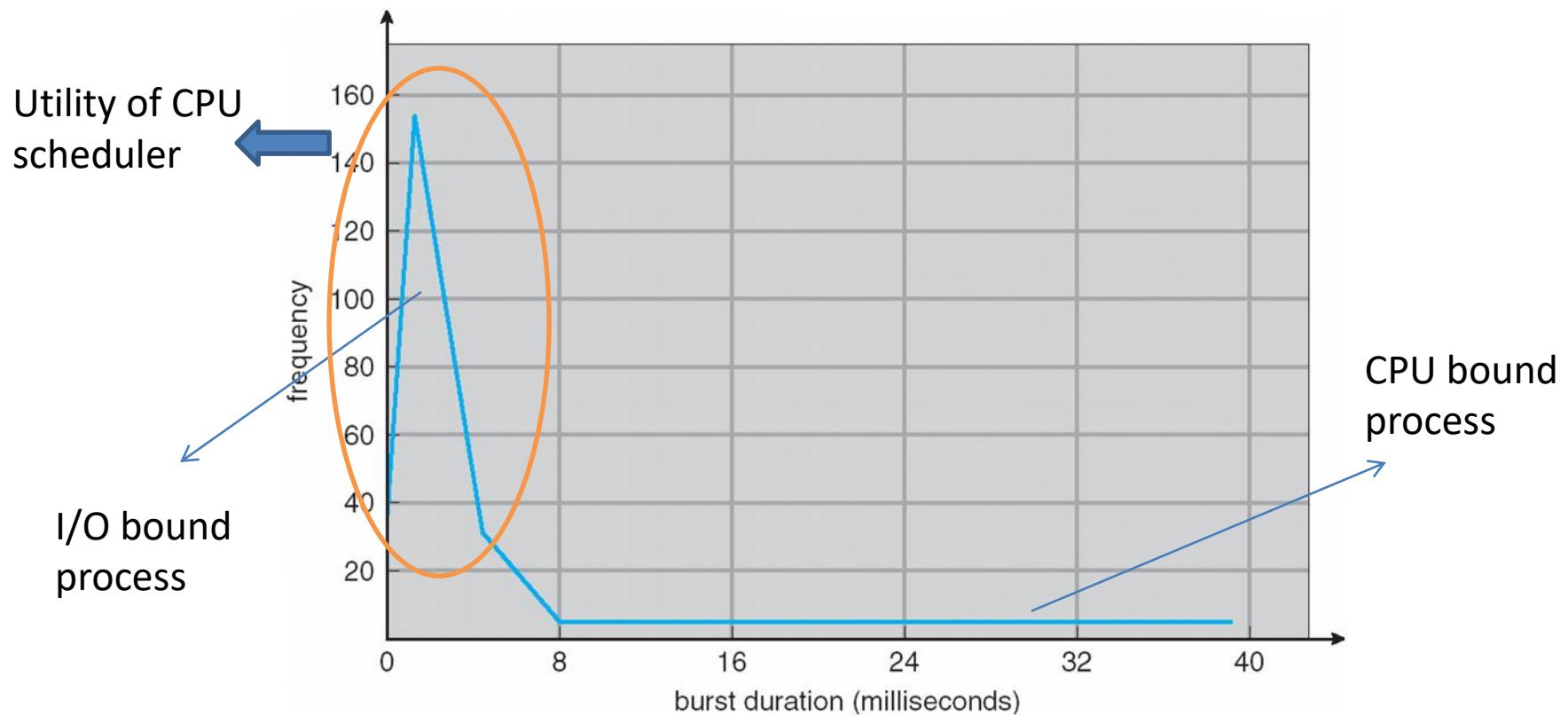
# Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
  - Several processes in memory (ready queue)
  - When one process requests I/O, some other process gets the CPU
  - Select (schedule) a process and allocate CPU

# Observed properties of Processes



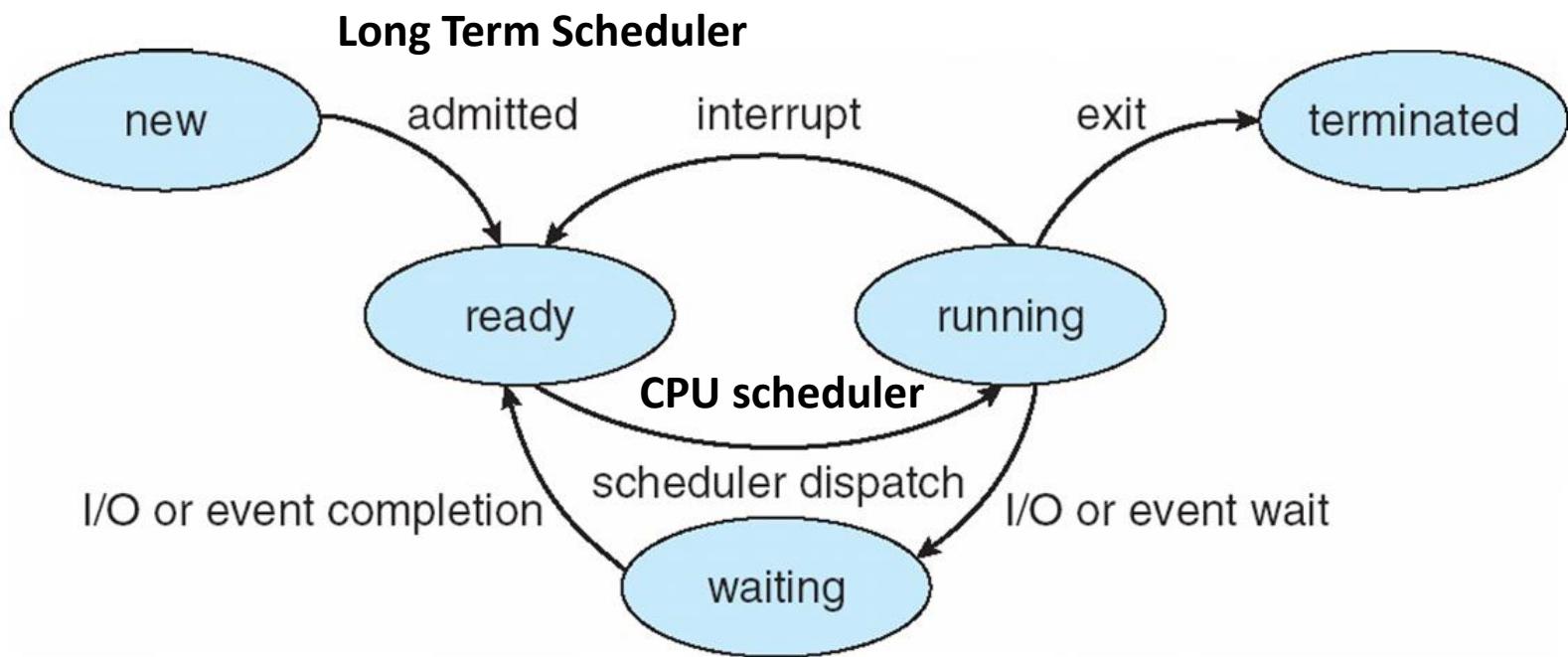
# Histogram of CPU-burst Times



Large number of short CPU bursts and small number of long CPU bursts

# Preemptive and non preemptive

- Selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways (not necessarily FIFO)
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**



# Preemptive scheduling

**Preemptive scheduling**

**Results in cooperative processes**

**Issues:**

- Consider access to shared data
  - Process synchronization
- Consider preemption while in kernel mode
  - Updating the ready or device queue
  - Preempted and running a “ps -el”

**Race condition**

# Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # 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

# Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time
- Mostly optimize the average
- Sometimes optimize the minimum or maximum value
  - Minimize max response time
- For interactive system, variance is important
  - E.g. response time
- System must behave in predictable way

# Scheduling algorithms

- First-Come, First-Served (FCFS) Scheduling
- Shortest-Job-First (SJF) Scheduling
- Priority Scheduling
- Round Robin (RR)

# First-Come, First-Served (FCFS) Scheduling

- Process that requests CPU first, is allocated the CPU first
- Ready queue=>FIFO queue
- Non preemptive
- Simple to implement

## Performance evaluation

- Ideally many processes with several CPU and I/O bursts
- Here we consider only one CPU burst per process

## First-Come, First-Served (FCFS) Scheduling

Process Burst Time

$P_1$	24
$P_2$	3
$P_3$	3

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



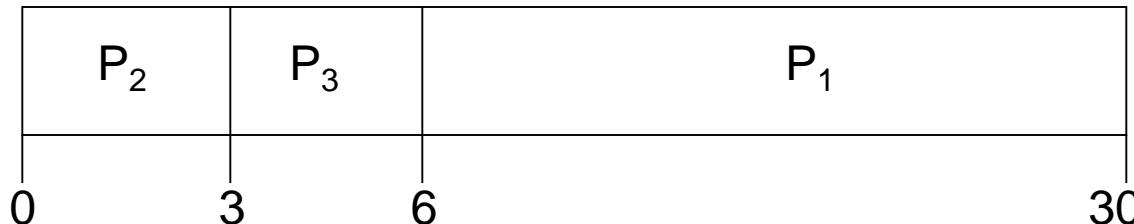
- Waiting time for  $P_1 = 0$ ;  $P_2 = 24$ ;  $P_3 = 27$
- Average waiting time:  $(0 + 24 + 27)/3 = 17$

# FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

$$P_2, P_3, P_1$$

- The Gantt chart for the schedule is:



- Waiting time for P<sub>1</sub> = 6; P<sub>2</sub> = 0, P<sub>3</sub> = 3
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- Average waiting time under FCFS heavily depends on process arrival time and burst time
- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

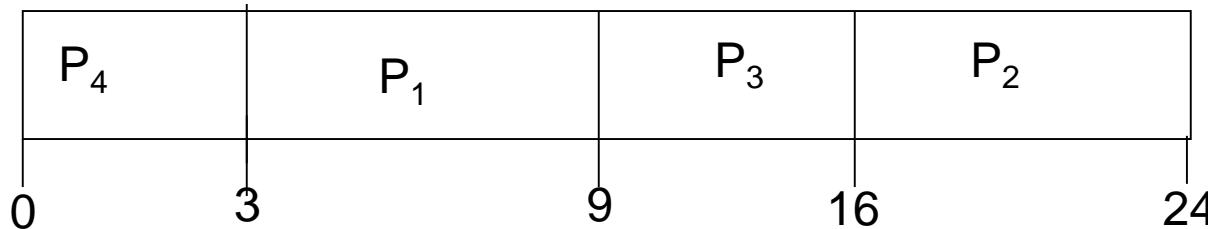
# Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Allocate CPU to a process with the smallest next CPU burst.
  - Not on the total CPU time
- Tie=>FCFS

# Example of SJF

<u>Process</u>	<u>Burst Time</u>
$P_1$	6
$P_2$	8
$P_3$	7
$P_4$	3

- SJF scheduling chart



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

Avg waiting time for FCFS?

# SJF

- SJF is optimal – gives minimum average waiting time for a given set of processes  
**(Proof: home work!)**
- The difficulty is knowing the length of the next CPU request
- Useful for Long term scheduler
  - Batch system
  - Could ask the user to estimate
  - Too low value may result in “time-limit-exceeded error”

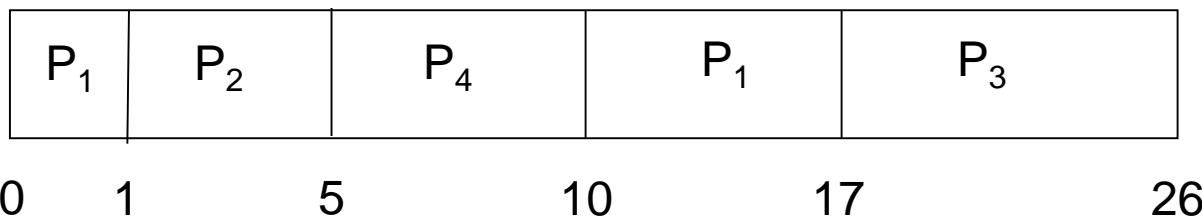
# Preemptive version

## Shortest-remaining-time-first

- Preemptive version called **shortest-remaining-time-first**
- Concepts of varying arrival times and preemption to the analysis

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
$P_1$	0	8
$P_2$	1	4
$P_3$	2	9
$P_4$	3	5

- *Preemptive SJF Gantt Chart*



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

Avg waiting time for non preemptive?

# Determining Length of Next CPU Burst

- Estimation of the CPU burst length – should be similar to the previous burst
  - Then pick process with shortest predicted next CPU burst
- Estimation can be done by using the length of previous CPU bursts, using time series analysis

1.  $t_n$  = actual length of  $n^{th}$  CPU burst

2.  $\tau_{n+1}$  = predicted value for the next CPU burst

3.  $\alpha, 0 \leq \alpha \leq 1$

4. Define :

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

Boundary  
cases  $\alpha=0, 1$

- Commonly,  $\alpha$  set to  $\frac{1}{2}$

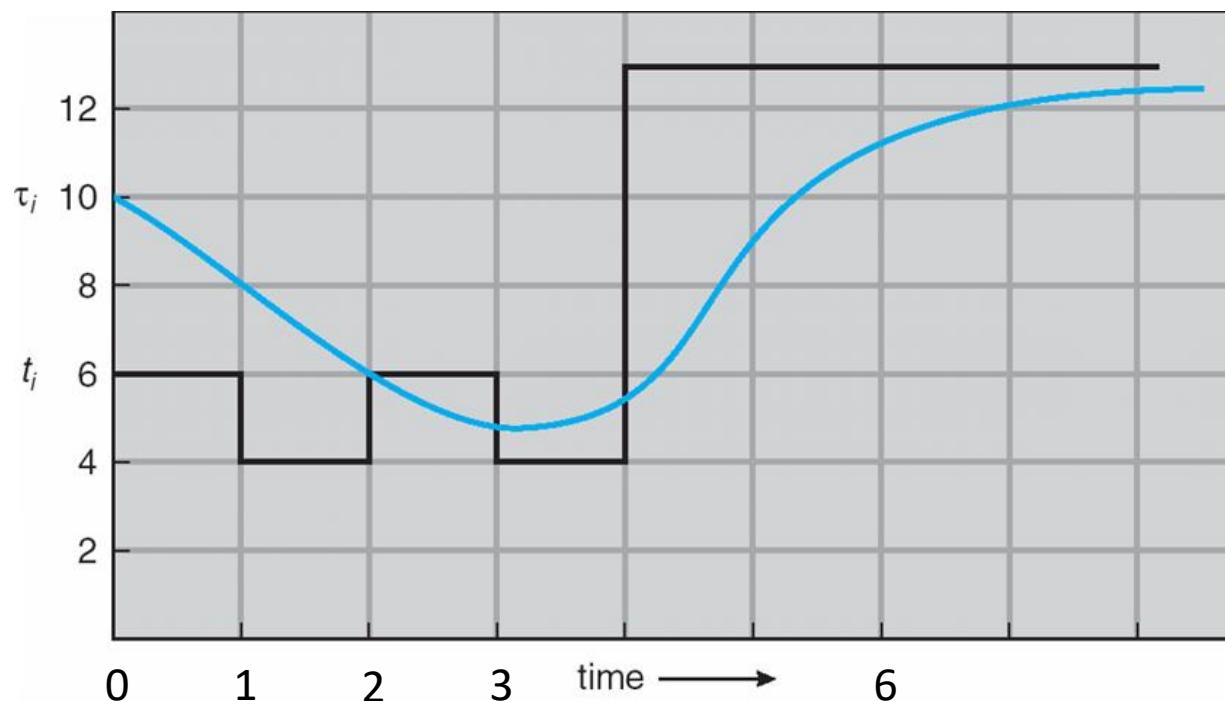
# Examples of Exponential Averaging

- $\alpha = 0$ 
  - $\tau_{n+1} = \tau_n$
  - Recent burst time does not count
- $\alpha = 1$ 
  - $\tau_{n+1} = t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

$$\begin{aligned}\tau_{n+1} &= \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \dots \\ &\quad + (1 - \alpha)^j \alpha t_{n-j} + \dots \\ &\quad + (1 - \alpha)^{n+1} \tau_0\end{aligned}$$

- Since both  $\alpha$  and  $(1 - \alpha)$  are less than or equal to 1, each successive term has less weight than its predecessor

# Prediction of the Length of the Next CPU Burst



CPU burst ( $t_i$ )      6      4      6      6      4      13      13      13      ...  
"guess" ( $\tau_i$ )      10      8      6      6      5      9      11      12      ...

# Priority Scheduling

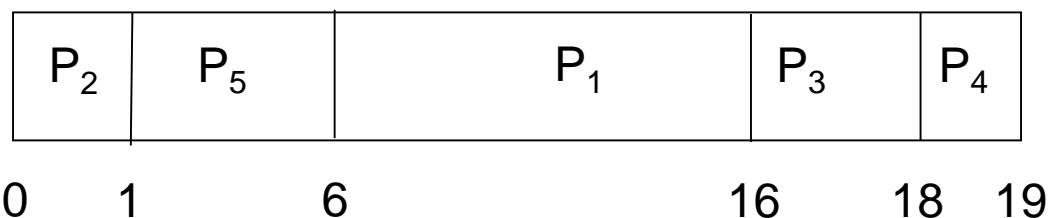
- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer  $\equiv$  highest priority)
- Set priority value
  - Internal (time limit, memory req., ratio of I/O Vs CPU burst)
  - External (importance, fund etc)
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Two types
  - Preemptive
  - Nonpreemptive
- Problem  $\equiv$  **Starvation** – low priority processes may never execute
- Solution  $\equiv$  **Aging** – as time progresses increase the priority of the process

nice

# Example of Priority Scheduling

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
$P_1$	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

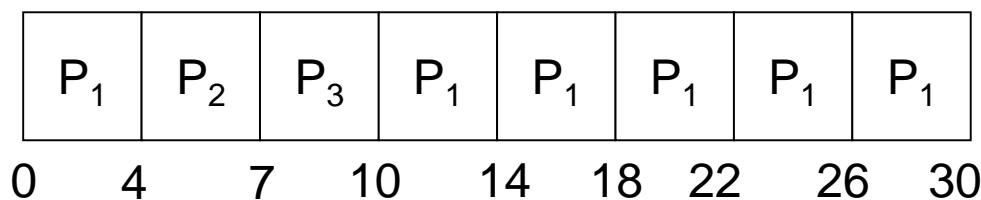
# Round Robin (RR)

- Designed for time sharing system
- Each process gets a small unit of CPU time (**time quantum q**), usually 10-100 milliseconds.
- After this time has elapsed, the process is preempted and added to the end of the ready queue.
- Implementation
  - Ready queue as FIFO queue
  - CPU scheduler picks the first process from the ready queue
  - Sets the timer for 1 time quantum
  - Invokes despatcher
- If CPU burst time < quantum
  - Process releases CPU
- Else Interrupt
  - Context switch
  - Add the process at the tail of the ready queue
  - Select the front process of the ready queue and allocate CPU

# Example of RR with Time Quantum = 4

<u>Process</u>	<u>Burst Time</u>
$P_1$	24
$P_2$	3
$P_3$	3

- The Gantt chart is:



- Avg waiting time =  $((10-4)+4+7)/3=5.66$

# Round Robin (RR)

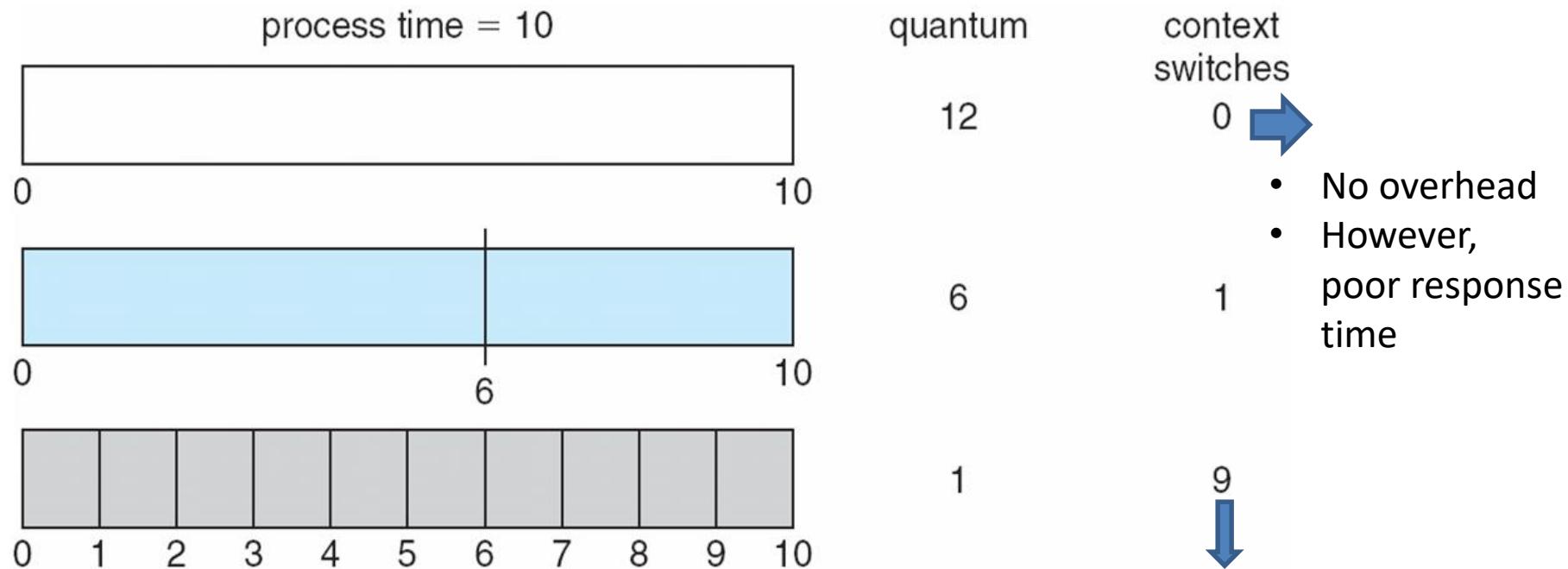
- Each process has a time quantum  $T$  allotted to it
- Dispatcher starts process  $P_0$ , loads a external counter (timer) with counts to count down from  $T$  to 0
- When the timer expires, the CPU is interrupted
- The context switch ISR gets invoked
- The context switch saves the context of  $P_0$ 
  - PCB of  $P_0$  tells where to save
- The scheduler selects  $P_1$  from ready queue
  - The PCB of  $P_1$  tells where the old state, if any, is saved
- The dispatcher loads the context of  $P_1$
- The dispatcher reloads the counter (timer) with  $T$
- The ISR returns, restarting  $P_1$  (since  $P_1$ 's PC is now loaded as part of the new context loaded)
- $P_1$  starts running

# Round Robin (RR)

- 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  $(n-1)q$  time units.
- Timer interrupts every quantum to schedule next process
- Performance depends on time quantum  $q$ 
  - $q$  large  $\Rightarrow$  FIFO
  - $q$  small  $\Rightarrow$  Processor sharing ( $n$  processes has own CPU running at  $1/n$  speed)

# Effect of Time Quantum and Context Switch Time

## Performance of RR scheduling

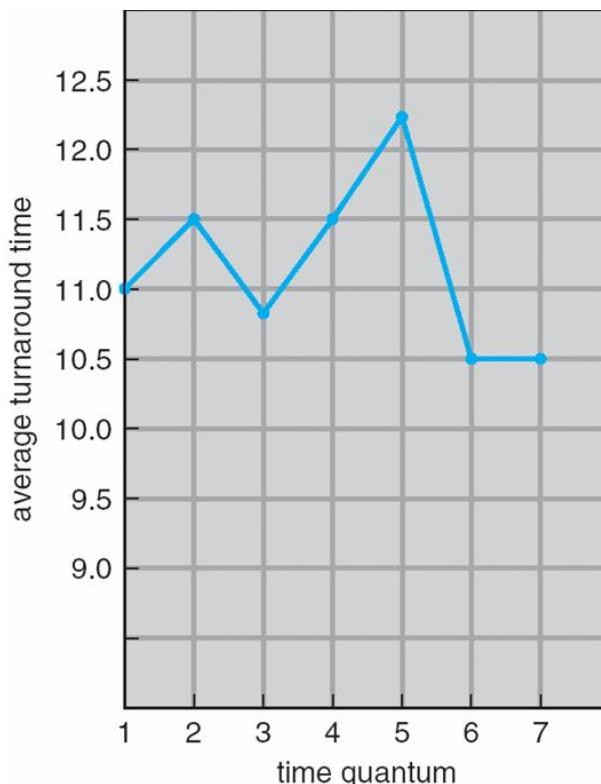


- $q$  must be large with respect to context switch, otherwise overhead is too high
- $q$  usually 10ms to 100ms, context switch < 10 microsec

- No overhead
  - However, poor response time
- Too much overhead!
  - Slowing the execution time

## Effect on Turnaround Time

- TT depends on the time quantum and CPU burst time
  - Better if most processes complete their next CPU burst in a single q



process	time
$P_1$	6
$P_2$	3
$P_3$	1
$P_4$	7

- Large q=> processes in ready queue suffer
- Small q=> Completion will take more time

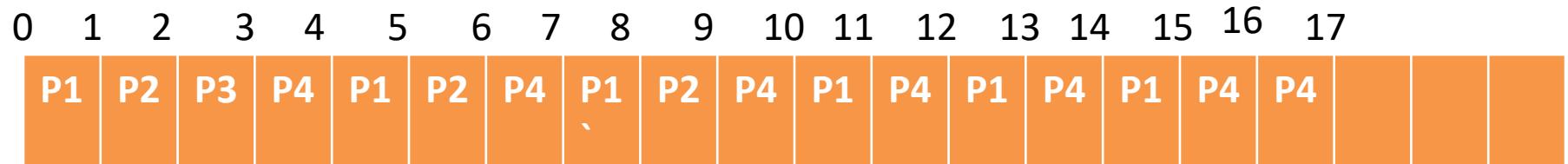
80% of CPU bursts  
should be shorter than  
q

### Response time

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

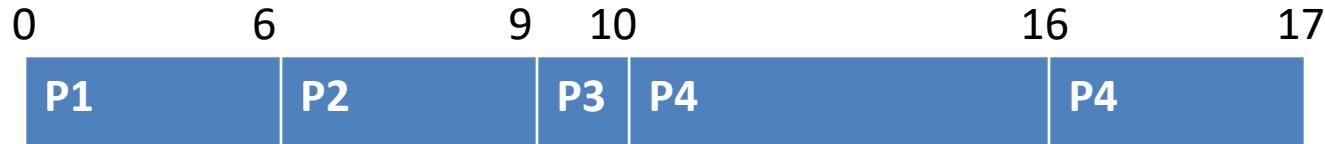
# Turnaround Time

$q=1$



$$\text{Avg Turnaround time} = \\ (15+9+3+17)/4=11$$

process	time
$P_1$	6
$P_2$	3
$P_3$	1
$P_4$	7



$$(6+9+10+17)/4=10.5$$

$q=6$

# Process classification

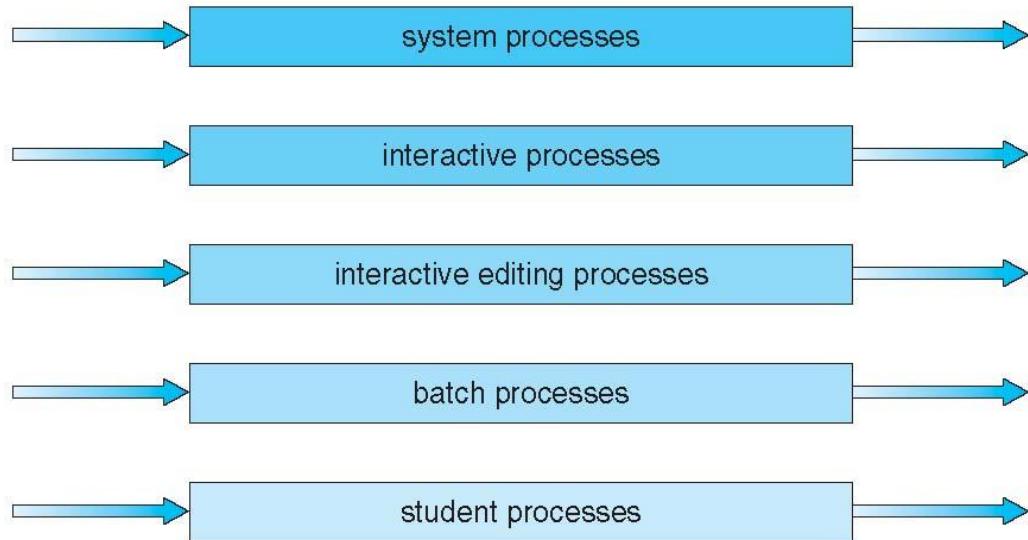
- Foreground process
  - Interactive
  - Frequent I/O request
  - Requires low response time
- Background Process
  - Less interactive
  - Like batch process
  - Allows high response time
- Can use different scheduling algorithms for two types of processes ?

# Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - foreground (interactive)
  - background (batch)
- Process permanently assigned in a given queue
  - Based on process type, priority, memory req.
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background).
  - Possibility of starvation.

# Multilevel Queue Scheduling

highest priority



- No process in batch queue could run unless upper queues are empty
- If new process enters
  - Preempt

## Another possibility

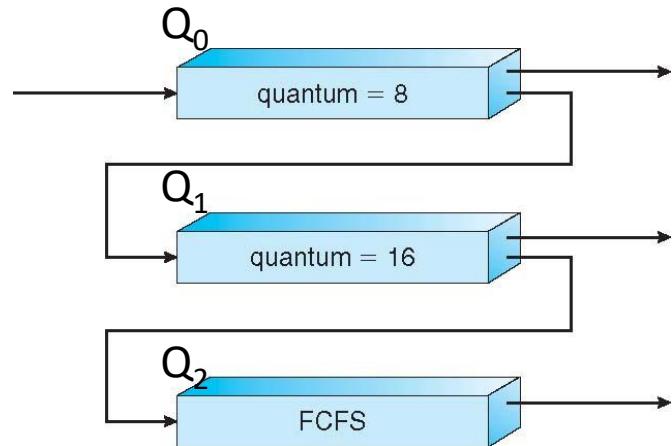
- Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
- 20% to background in FCFS

# Multilevel Feedback Queue

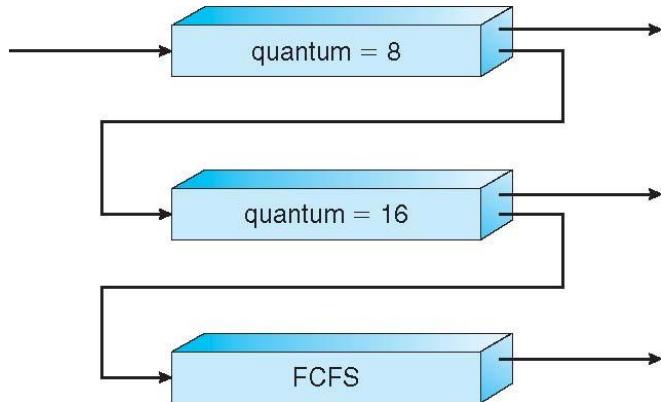
- So a process is permanently assigned a queue when they enter in the system
  - They do not move
- **Flexibility!**
  - Multilevel-feedback-queue scheduling
- A process can move between the various queues;
- Separate processes based of the CPU bursts
  - Process using too much CPU time can be moved to lower priority
  - Interactive process => Higher priority
- Move process from low to high priority
  - Implement aging

# 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  $Q_0$ 
    - 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 receives 16 milliseconds
    - If it still does not complete, it is preempted and moved to queue  $Q_2$



# Multilevel Feedback Queues



- Highest Priority to processes CPU burst time <8 ms
- Then processes >8 and <24

- 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

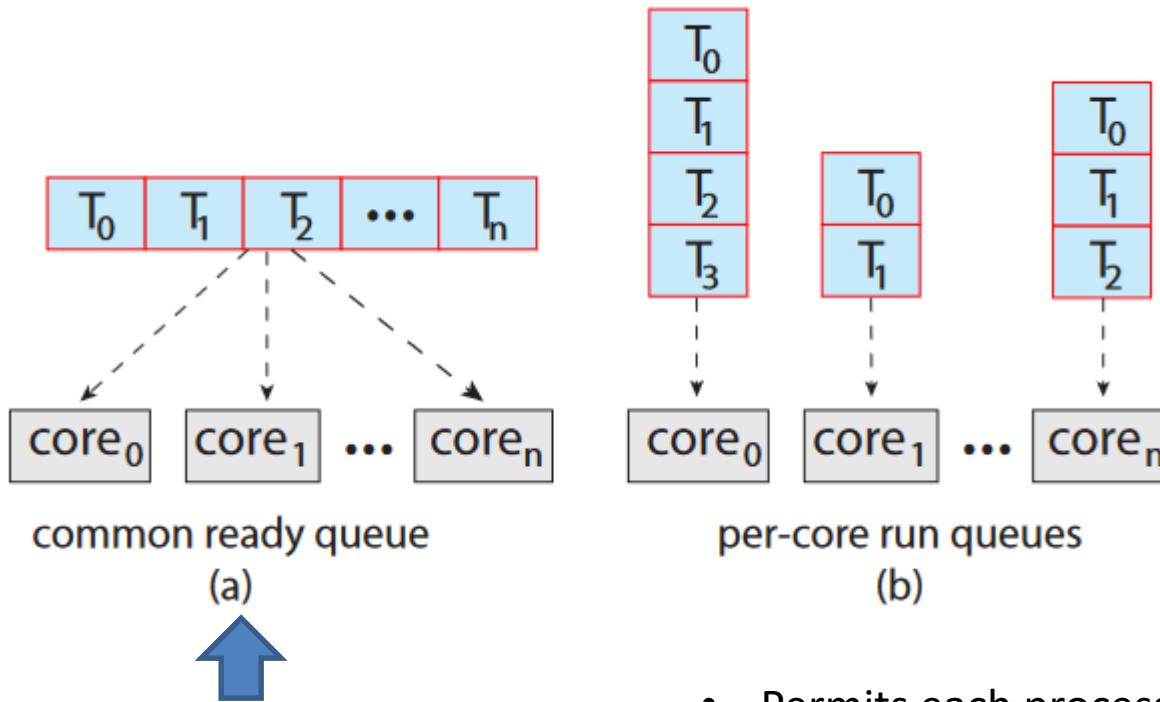
# Multiple-Processor Scheduling

- If multiple CPUs are available, multiple processes may run in parallel
- However **scheduling** issues become correspondingly more **complex**.
- Many possibilities have been tried
- As we saw with CPU scheduling with a single-core CPU
  - there is no one best solution

# Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** –
  - Master server
  - only **one processor** accesses the **system data structures**, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling,
- all processes in **common ready queue**, or each has its **own private queue** of ready processes
- Scheduler for each processor **examine the ready queue**
  - select a process to run.

# Multiple-Processor Scheduling



- We have a possible **race condition**
- **Locking** to protect the common ready queue from this race condition.
- Accessing the shared queue would likely be a **performance bottleneck**

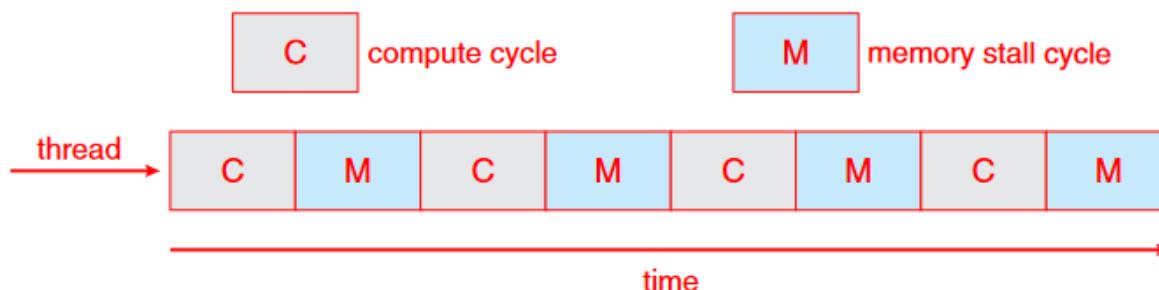
- Permits each processor to schedule process from its **private ready queue**
- **Does not suffer** from the possible **performance** problems
- Most common approach on systems supporting SMP.
- **Load balancing**

# Multi core processors

- SMP systems have allowed several processes to run **in parallel** by providing **multiple physical processors**.
- Recently, **multiple computing cores** on the **same physical chip**, resulting in a **multicore processor**.
- **Each core** maintains its **architectural state** and thus appears to the operating system to be a separate **logical CPU**
- SMP systems that use multicore processors are **faster and consume less power**
  - than systems in which each CPU has its own physical chip

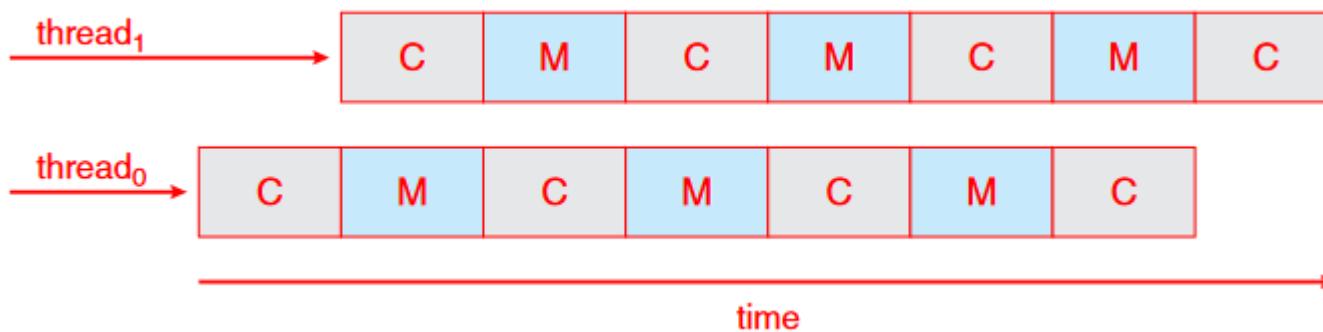
# Challenge: Memory stall

- When a processor **accesses memory**, it spends a significant amount of **time waiting** for the data to become available.
- This situation, known as a **memory stall**, occurs primarily because
  - modern processors operate at much faster speeds than memory.
  - For cache miss

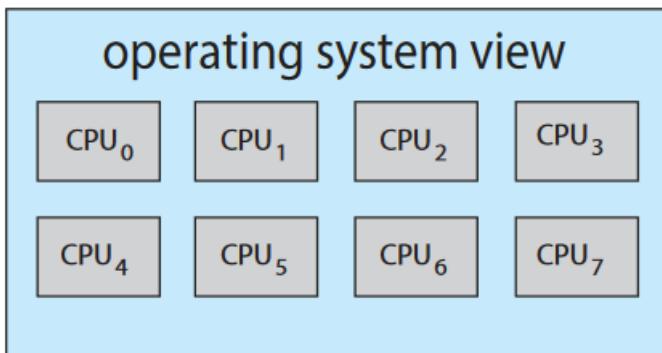
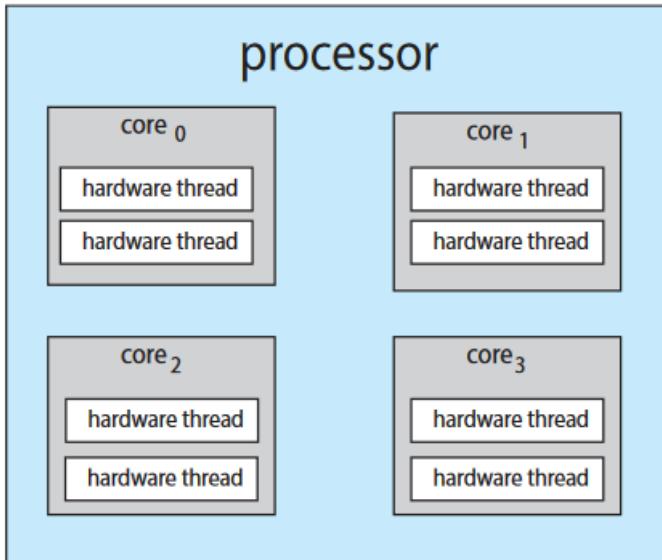


# Solution: Hardware threads

- Recent hardware designs have implemented **multithreaded processing cores** in which two (or more) hardware threads are assigned to **each core**.
- That way, if **one hardware thread stalls** while waiting for memory, the core can **switch** to another thread.



# Hyper-threading



- **Each hardware thread** maintains its **architectural state**, such as **instruction pointer and register set**,
- Thus appears as a **logical CPU** that is available to run a **software process**.

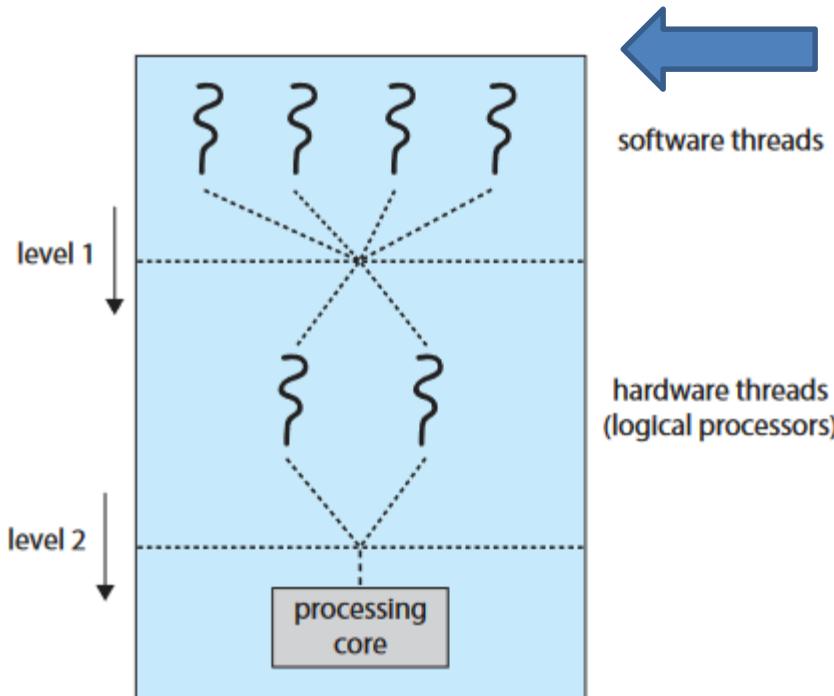
Contemporary Intel processors—such as the **Intel i7**—support **two threads per core**,

**Oracle Sparc M7** processor supports **eight threads per core**, with eight cores per processor, thus providing the operating system with 64 logical CPUs

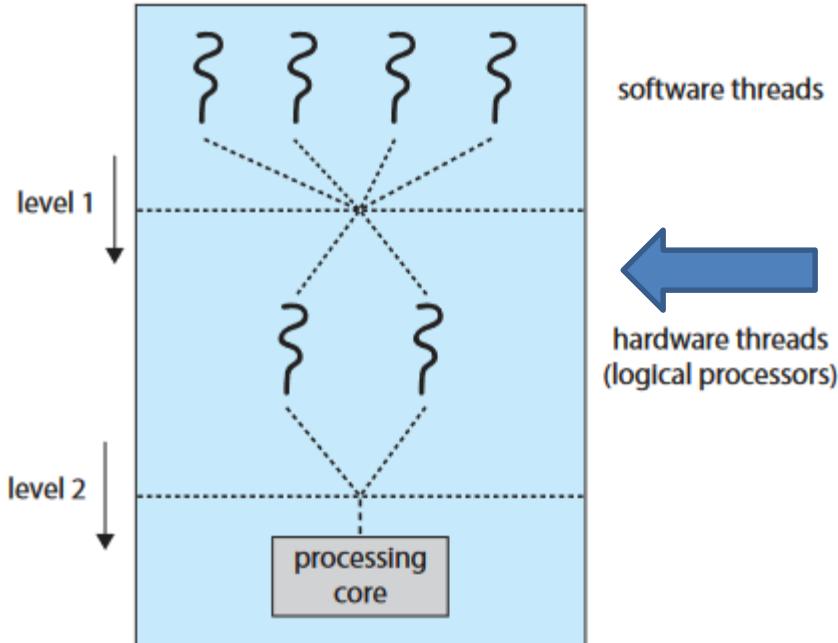
# Dual scheduling

- It is important to note that
- a **processing core** can only execute **one hardware thread** at a time.
  - **resources of the physical core** (such as caches and pipelines) must be shared among its **hardware threads**
- Multithreaded, multicore processor actually requires **two different levels of scheduling**

# Dual scheduling



- The **scheduling decisions** that must be made by the operating system as it **chooses which software process** to run on each **hardware thread** (logical CPU).
- For this level of scheduling, the operating system may choose **any scheduling algorithm**



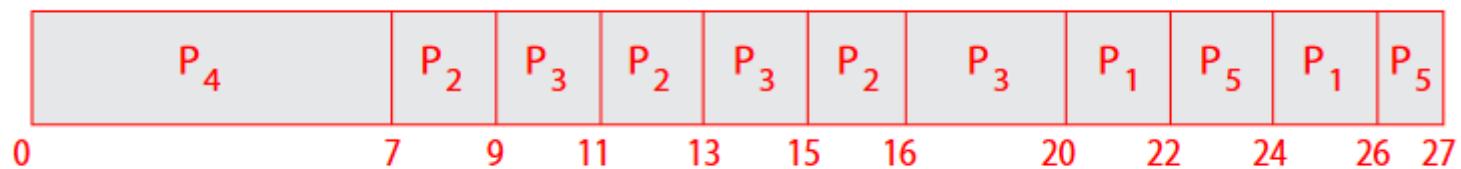
- A second level of scheduling specifies **which hardware thread to run on a core**.
- One approach is to use a **simple round-robin algorithm** to schedule a hardware thread to the processing core.
- This is the approach adopted by the UltraSPARC T3.
- Another approach is used by the Intel Itanium
- Assigned to each hardware thread is a dynamic urgency value ranging from 0 to 7

# Problem 1

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
$P_1$	4	3
$P_2$	5	2
$P_3$	8	2
$P_4$	7	1
$P_5$	3	3

Combine round-robin and priority scheduling in such a way that the system executes the highest-priority process and runs processes with the same priority using round-robin scheduling ( $q=2$ ).

# Solution 1



# Problem 2

Consider three processes (process id 0, 1, 2 respectively) with compute time bursts 2, 4 and 8 time units. All processes arrive at time zero. Consider the longest remaining time first (**LRTF**) scheduling algorithm. In LRTF ties are broken by giving priority to the process with the lowest process id. Compute average turn around time

Process	AT	BT	TAT
P0	0	2	
P1	0	4	
P2	0	8	

# Solution 2

2. Consider three processes (process ID 0,1,2 respectively) with compute time bursts 2,4 and 8 time units. All processes arrive at time zero. Consider **the longest remaining time first (LRTF) scheduling algorithm**. In LRTF, ties are broken by giving priority to the process with the lowest process ID. The **average turnaround time** is:

P2	P2	P2	P2	P1	P2	P1	P2	P0	P1	P2	P0	P1	p2
----	----	----	----	----	----	----	----	----	----	----	----	----	----

PID	A.T	B.T	C.T	T.A.T	W.T
P0	0	2	12	12	10
P1	0	4	13	13	9
P2	0	8	14	14	6
<b>TOTAL</b>				<b>39</b>	<b>25</b>

A.T. Arrival Time

B.T. Burst Time

C.T. Completion Time.

T.A.T. Turn Around Time

W.T. Waiting Time.

**Average TAT = 39/3 = 13 units**

# Problem 3

Process 0: CPU-bound (each CPU burst is 100ms)

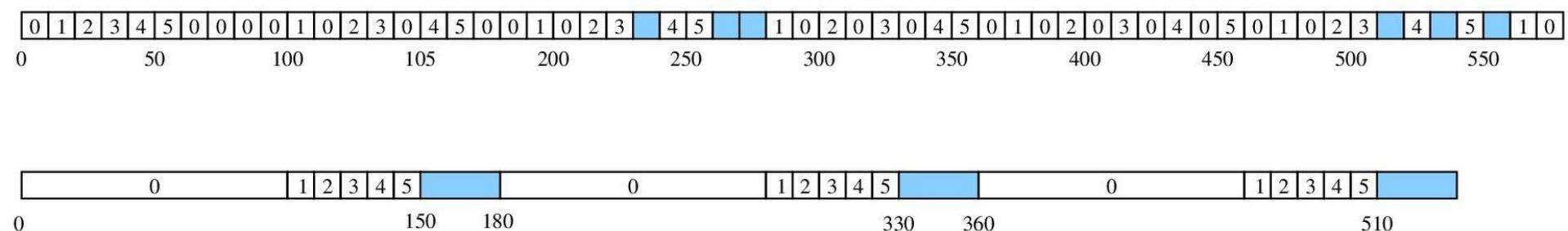
Processes 1--5: IO bound (10ms CPU burst),IO time: 80ms

All processes are available at  $t = 0$ .

FCFS: find out when CPU becomes idle for the first time.

RR: Take time quantum  $q = 10\text{ms}$ . Again find out when the CPU becomes idle for the first time.

# Solution 3



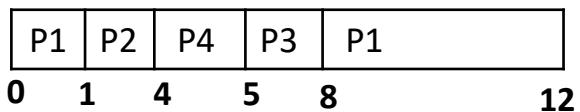
# Problem 4

Consider the following set of processes, with the arrival times and the CPU-burst times given in milliseconds. What is the average turnaround time for these processes with the **preemptive shortest remaining processing time first (SRPT) algorithm**?

3. Consider the following set of processes, with the arrival times and the CPU-burst times given in milliseconds. What is the average turnaround time for these processes with the **preemptive shortest remaining processing time first (SRPT) algorithm**?

Process	Arrival Time	Burst Time
P1	0	5
P2	1	3
P3	2	3
P4	4	1

Answer:



$$\text{Average turnaround time} = \frac{12 + 3 + 6 + 1}{4} = \frac{22}{4} = 5.5$$

Process	Waiting Time = (Turnaround Time – Burst Time)	Turnaround Time = (Completion Time – Arrival Time)
P1	7	12
P2	0	3
P3	3	2
P4	0	1

Queue 1: SJF

Queue 2: FCFS

Queue 3: RR with  $Q=2$

$\{ \begin{matrix} Q_1 > Q_2 > Q_3 \\ (\text{priority}) \end{matrix} \}$

PID	A.T	B.T	Queue
P <sub>1</sub>	0	4	1
P <sub>2</sub>	0	3	1
P <sub>3</sub>	0	9	3
P <sub>4</sub>	9	5	2
P <sub>5</sub>	11	7	1
P <sub>6</sub>	8	2	3

A

Q1: P<sub>1</sub>, P<sub>2</sub>, P<sub>5</sub>

Q2: P<sub>4</sub>

Q3: P<sub>3</sub>, P<sub>6</sub>

P2	P1	P3	P4	P5	P4	P3	P6	P3	P3	P3	P3
0	3	7	9	11	18	21	23	25	27	29	30

# Interprocess Communication

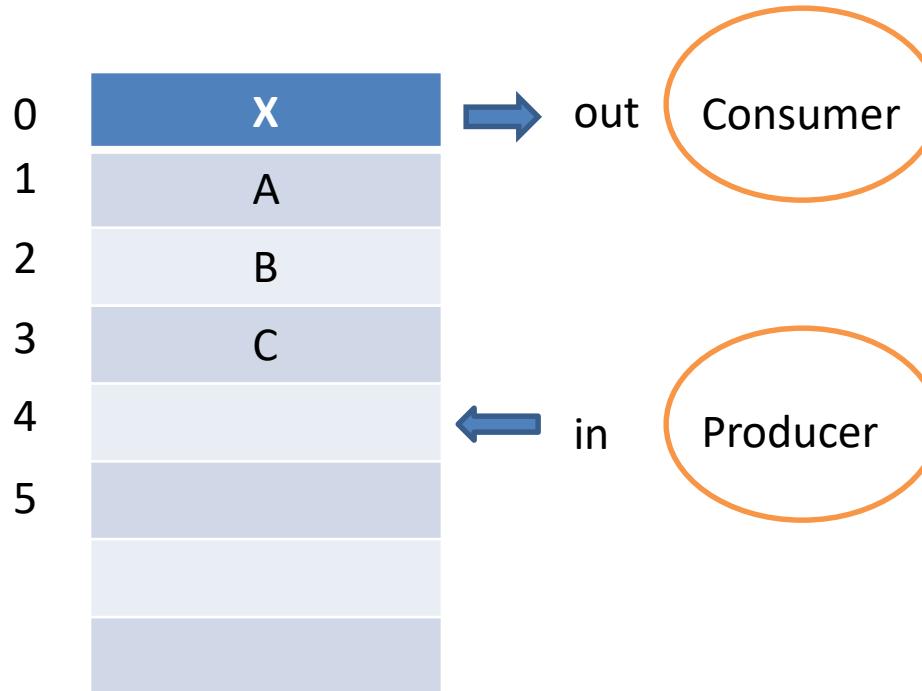
- Processes within a system may be **independent or cooperating**
- Cooperating process can affect or be affected by other processes, including sharing data
- Cooperating processes need **interprocess communication (IPC)**
- Two models of IPC
  - Shared memory
  - Message passing

# Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process

Buffer empty=>  
in=out

Buffer full=>  
 $(in+1) \% \text{size} = \text{out}$



- unbounded-buffer* places no practical limit on the size of the buffer
- bounded-buffer* assumes that there is a fixed buffer size

# Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
    ...
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

# Bounded-Buffer – Producer

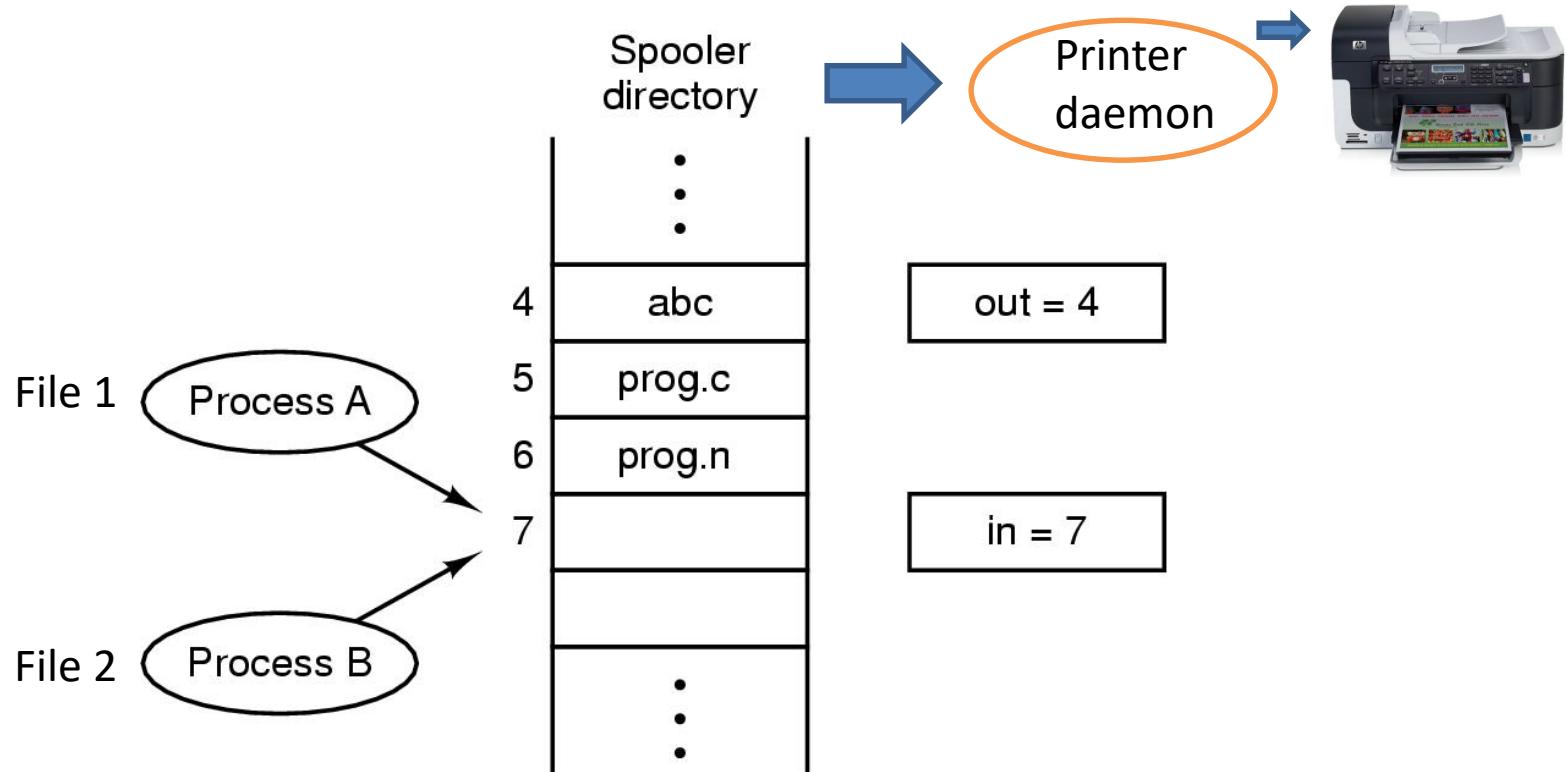
```
while (true) {
    /* Produce an item */
    while (((in + 1) % BUFFER SIZE) == out)
        ; /* do nothing -- no free buffers */
    buffer[in] = item;
    in = (in + 1) % BUFFER SIZE;
}
```

# Bounded Buffer – Consumer

```
while (true) {  
    while (in == out)  
        ; // do nothing -- nothing to consume  
  
    // remove an item from the buffer  
    item = buffer[out];  
    out = (out + 1) % BUFFER SIZE;  
    return item;  
}
```

# Interprocess Communication

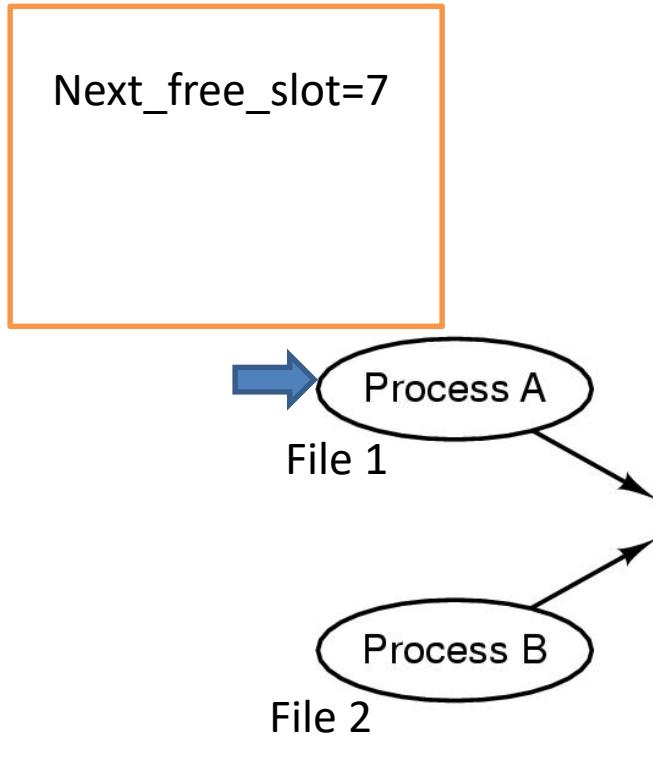
## Race Conditions



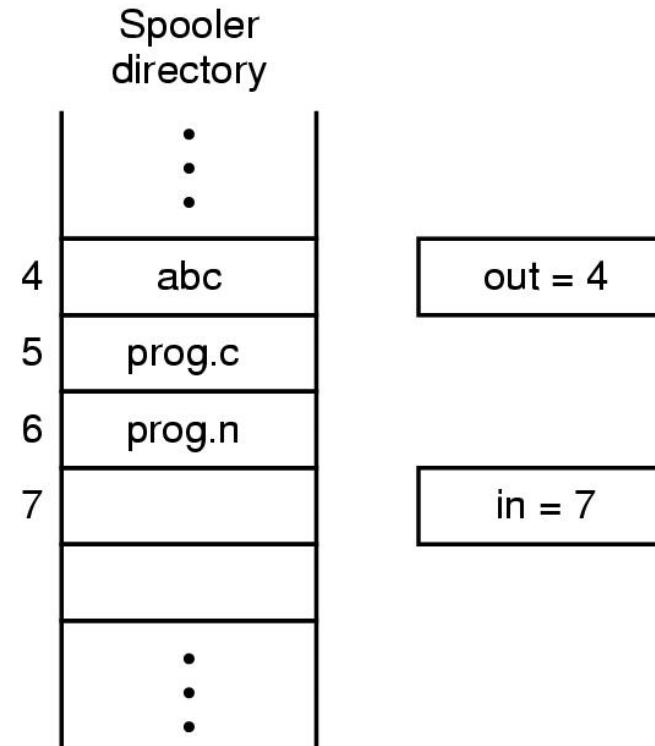
Two processes want to access shared memory at same time

# Interprocess Communication

Process A



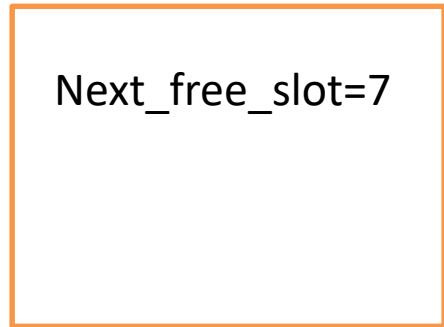
Race Conditions



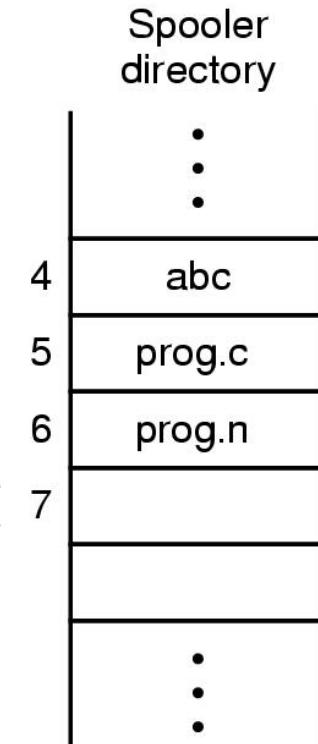
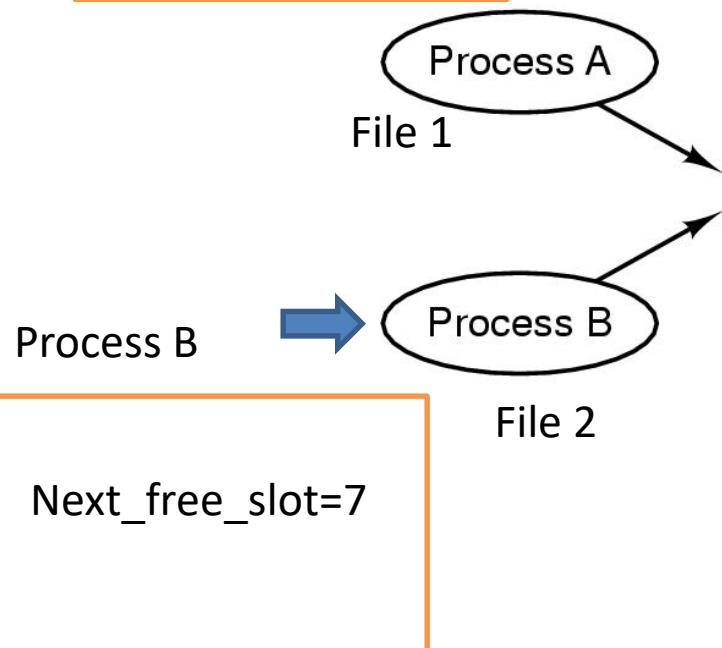
Two processes want to access shared memory at same time

# Interprocess Communication

Process A



Race Conditions



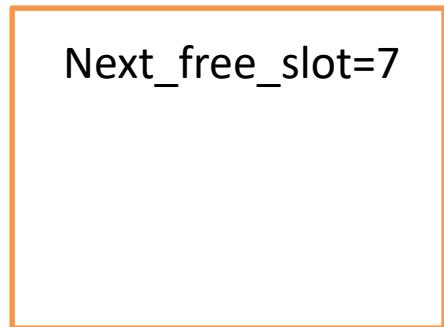
out = 4

in = 7

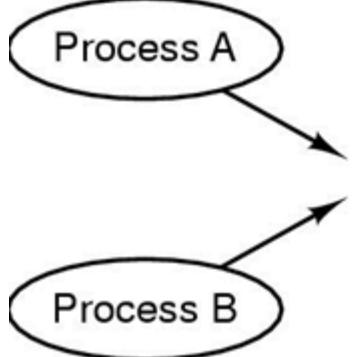
Two processes want to access shared memory at same time

# Interprocess Communication

Process A

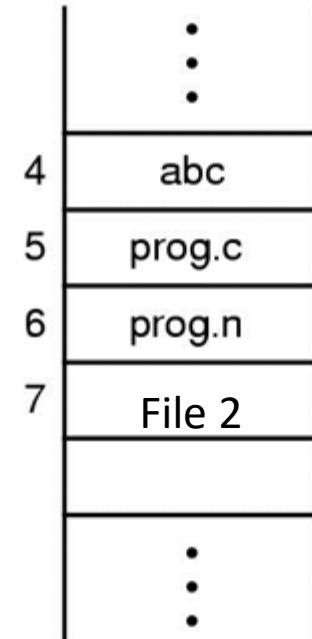


File 1



Race Conditions

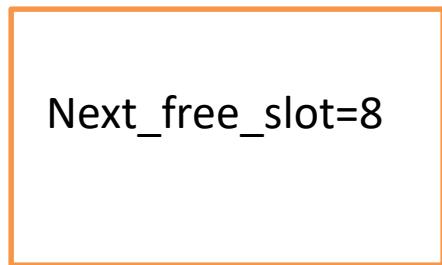
Spooler  
directory



out = 4

in = 8

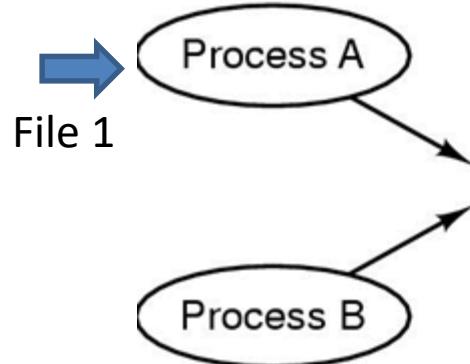
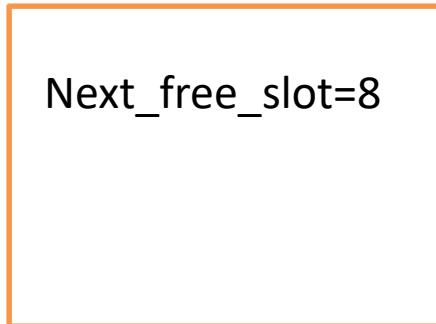
Process B



Two processes want to access shared memory at same time

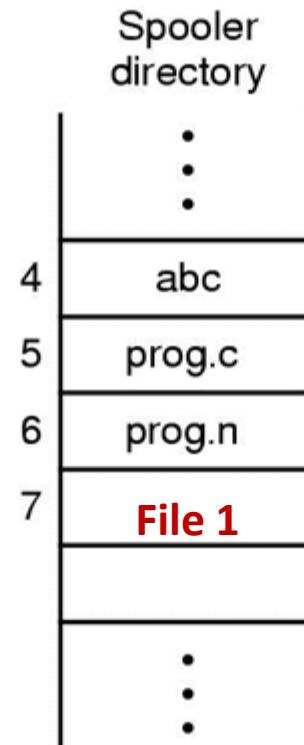
# Interprocess Communication

Process A



Next\_free\_slot=8

Race Conditions



out = 4

in = 8

Race condition

Two processes want to access shared memory at same time

# Race condition

- Race condition
  - Two or more processes are reading or writing some shared data and the final result depends on who runs precisely when
  - In our former example, the possibilities are various
  - Hard to debug

# Critical Section Problem

- Critical region
  - Part of the program where the shared memory is accessed
- Mutual exclusion
  - Prohibit more than one process from reading and writing the shared data at the same time

# Critical Section Problem

- Consider system of n processes  $\{p_0, p_1, \dots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- Critical section problem is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

# Critical Section Problem

do {

*entry section*

critical section

*exit section*

remainder section

} while (TRUE);

General structure of a typical process  $P_i$

# Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process  $P_i$  is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** –
  - If no process is executing in its critical section
  - and there exist some processes that wish to enter their critical section
  - then only the processes outside remainder section (i.e. the processes competing for critical section, or exit section) can participate in deciding which process will enter CS next
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
  - Assume that each process executes at a nonzero speed
  - No assumption concerning relative speed of the n processes

# Critical Section Problem

do {

*entry section*

critical section

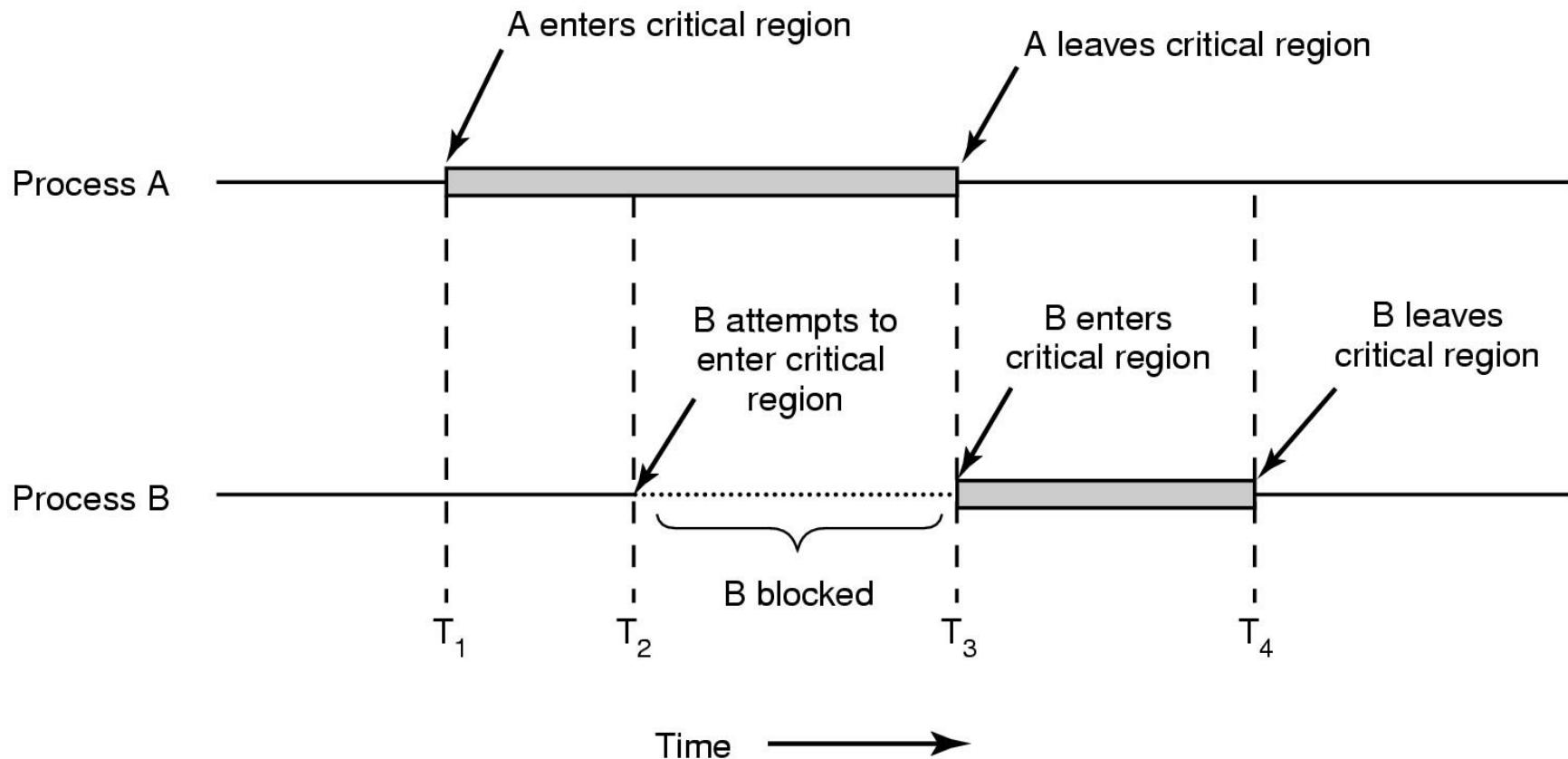
*exit section*

remainder section

} while (TRUE);

General structure of a typical process  $P_i$

# Critical Section Problem



Mutual exclusion using critical regions

# Mutual Exclusion

- Disable interrupt
  - After entering critical region, disable all interrupts
  - Since clock is just an interrupt, no CPU preemption can occur
  - Disabling interrupt is useful for OS itself, but not for users...

# Mutual Exclusion with busy waiting

- Lock variable
  - A software solution
  - A single, shared variable (lock)
  - before entering critical region, programs test the variable,
  - if 0, enter CS;
  - if 1, the critical region is occupied

– **What is the problem?**

```
While(true)
{
    while(lock!=0);
    Lock=1
    CS()
    Lock=0
    Non-CS()
}
```

# Concepts

- Busy waiting
  - Continuously testing a variable until some value appears
- Spin lock
  - A lock using busy waiting is call a spin lock
- CPU time wastage!

# Mutual Exclusion with Busy Waiting : strict alternation

```
while (TRUE) {  
    while (turn != 0)      /* loop */ ;  
    critical_region( );  
    turn = 1;  
    noncritical_region( );  
}
```

(a)

```
while (TRUE) {  
    while (turn != 1)      /* loop */ ;  
    critical_region( );  
    turn = 0;  
    noncritical_region( );  
}
```

(b)

Proposed solution to critical region problem

(a) Process 0.      (b) Process 1.

1. Mutual exclusion is preserved? **Y**
2. Progress requirement is satisfied? **N**
3. Bounded-waiting requirement is met? **N**

# Peterson's Solution

- Two process solution
- The two processes share two variables:
  - int **turn**;
  - Boolean **interested [2]**
- The variable **turn** indicates whose turn it is to enter the critical section
- The **interested** array is used to indicate if a process is interested to enter the critical section.
- **interested[i]** = true implies that process  $P_i$  is interested!

# Mutual Exclusion with Busy Waiting (2) : a workable method

```
#define FALSE 0
#define TRUE 1
#define N      2          /* number of processes */

int turn;                  /* whose turn is it? */
int interested[N];         /* all values initially 0 (FALSE) */

void enter_region(int process); /* process is 0 or 1 */
{
    int other;              /* number of the other process */

    other = 1 - process;    /* the opposite of process */
    interested[process] = TRUE; /* show that you are interested */
    turn = process;         /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process) /* process: who is leaving */
{
    interested[process] = FALSE; /* indicate departure from critical region */
}
```

Peterson's solution for achieving mutual exclusion

# Algorithm for Process $P_i$

```
do {  
    interested[i] = TRUE;  
    turn = j ;  
    while (interested[j] && turn == j);  
        critical section  
    interested[i] = FALSE;  
        remainder section  
} while (TRUE);
```

Provable that

1. Mutual exclusion is preserved
2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

**Does this alter the sequence?**

# Hardware Instruction Based Solutions

## Multiprocessor system

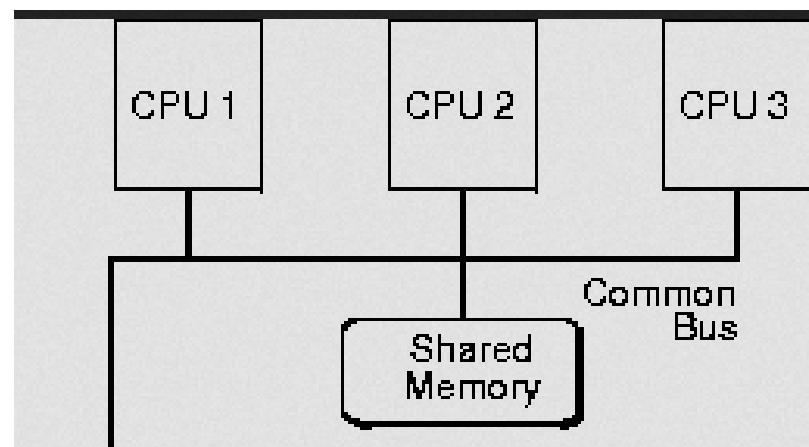
- Some architectures provide special instructions that can be used for synchronization
- **TSL**: Test and modify the content of a word **atomically**

*TSL Reg, lock*

{

*Reg=lock;  
lock = true;*

}



# Hardware Instruction Based Solutions

enter\_region:

TSL REGISTER,LOCK

| copy lock to register and set lock to 1

CMP REGISTER,#0

| was lock zero?

JNE enter\_region

| if it was non zero, lock was set, so loop

RET | return to caller; critical region entered

leave\_region:

MOVE LOCK,#0

| store a 0 in lock

RET | return to caller

**Does it satisfy all the conditions?**

Entering and leaving a critical region using the  
TSL instruction

# System call version

- Special system call that can be used for synchronization
- **TestAndSet**: Test and modify the content of a word **atomically**

```
boolean TestAndSet (boolean &target) {  
    boolean v = target;  
    target = true;  
    return v;  
}
```

# Mutual Exclusion with Test-and-Set

- Shared data:  
*boolean lock = false;*      **Does it satisfy all the conditions?**
- Process  $P_i$   
*do {*  
    *while (TestAndSet(lock)) ;*  
    *critical section*  
    *lock = false;*  
    *remainder section*  
*}*

# Swap Instruction

---

```
int compare_and_swap(int *value, int expected, int new_value) {  
    int temp = *value;  
  
    if (*value == expected)  
        *value = new_value;  
  
    return temp;  
}
```

---

```
while (true) {  
    while (compare_and_swap(&lock, 0, 1) != 0)  
        ; /* do nothing */  
  
    /* critical section */  
  
    lock = 0;  
  
    /* remainder section */  
}
```

---

# Concepts

- Busy waiting
  - Continuously testing a variable until some value appears
- Spin lock
  - A lock using busy waiting is call a spin lock
- CPU time wastage!

- Drawback of Busy waiting
  - A lower priority process has entered critical region
  - A higher priority process comes and preempts the lower priority process, it wastes CPU in busy waiting, while the lower priority don't come out
  - Priority inversion problem

# Producer-consumer problem

- Two processes share a common, fixed-sized buffer
- Producer puts information into the buffer
- Consumer takes information from buffer
- A simple solution

# Sleep and Wakeup

```
#define N 100                                     /* number of slots in the buffer */
int count = 0;                                    /* number of items in the buffer */

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();                    /* repeat forever */
        if (count == N) sleep();                  /* generate next item */
        insert_item(item);                      /* if buffer is full, go to sleep */
        count = count + 1;                      /* put item in buffer */
        if (count == 1) wakeup(consumer);        /* increment count of items in buffer */
                                                /* was buffer empty? */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        if (count == 0) sleep();                /* repeat forever */
        item = remove_item();                  /* if buffer is empty, got to sleep */
        count = count - 1;                     /* take item out of buffer */
                                                /* decrement count of items in buffer */
        if (count == N - 1) wakeup(producer);   /* was buffer full? */
        consume_item(item);                  /* print item */
    }
}
```

# Producer-Consumer Problem

```
#define N 100          /* number of slots in the buffer */
int count = 0;         /* number of items in the buffer */

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();           /* repeat forever */
        if (count == N) sleep();         /* generate next item */
        insert_item(item);              /* if buffer is full, go to sleep */
        count = count + 1;              /* put item in buffer */
        if (count == 1) wakeup(consumer); /* increment count of items in buffer */
        /* was buffer empty? */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        if (count == 0) sleep();         /* repeat forever */
        item = remove_item();           /* if buffer is empty, got to sleep */
        consume_item(item);            /* take item out of buffer */
        count = count - 1;              /* decrement count of items in buffer */
        if (count == N - 1) wakeup(producer); /* was buffer full? */
        /* print item */
    }
}
```

- What can be the problem?
- Signal missing
  - Shared variable: counter
  - When consumer read count with a 0 but didn't fall asleep in time
  - then the signal will be lost

Producer-consumer problem with fatal race condition

# Tasks

- We must ensure proper process synchronization
  - Stop the producer when buffer full
  - Stop the consumer when buffer empty
- We must ensure mutual exclusion
  - Avoid race condition
- Avoid busy waiting

# Semaphore

- Widely used synchronization tool
- Does not require **busy-waiting**
  - CPU is not held unnecessarily while the process is waiting
- A Semaphore  $S$  is
  - A data structure with an integer variable  $S.value$  and a queue  $S.list$  of processes (shared variable)
  - The data structure can only be accessed by two **atomic** operations,  $\text{wait}(S)$  and  $\text{signal}(S)$  (also called  $\text{down}(S)$ ,  $P(S)$  and  $\text{Up}(s)$ ,  $V(S)$ )
- Value of the semaphore  $S$  = value of the integer  $S.value$

```
typedef struct {  
    int value;  
    struct process *list;  
} semaphore
```

# Semaphore

## Wait(S)      S<= semaphore variable

- When a process P executes the wait(S) and finds
  - $S == 0$
  - Process must wait => block()
  - Places the process into a waiting queue associated with S
  - Switch from running to waiting state
- Otherwise decrement S

## Signal(S)

When a process P executes the signal(S)

- Check, if some other process Q is waiting on the semaphore S
- Wakeup(Q)
- Wakeup(Q) changes the process from waiting to ready state
- Otherwise increment S

# Semaphore (wait and signal)

- Implementation of wait:

```
wait(semaphore *S) {  
    S->value--;  
    if (S->value < 0) {  
        add this process to S->list;  
        block();  
    }  
}
```

List of PCB

- Implementation of signal:

```
signal(semaphore *S) {  
    S->value++;  
    if (S->value <= 0) {  
        remove a process P from S->list;  
        wakeup(P);  
    }  
}
```

Atomic/  
Indivisible

**Note:** which process is picked  
for unblocking may depend on  
policy.

# Usage of Semaphore

- **Counting** semaphore – integer value can range over an unrestricted domain
  - Control access to a shared resource with finite elements
  - Wish to use => wait(S)
  - Releases resource=>signal(S)
  - Used for synchronization
- **Binary** semaphore – integer value can range only between 0 and 1
  - Also known as **mutex locks**
  - Used for mutual exclusion

# Ordering Execution of Processes using Semaphores (Synchronization)

- Execute statement  $B$  in  $P_j$  only after statement  $A$  executed in  $P_i$
- Use semaphore  $flag$  initialized to 0
- Code:

$P_i$	$P_j$
:	:
Stmt. A	<i>wait(flag)</i>
<i>signal(flag)</i>	Stmt. B

- Multiple such points of synchronization can be enforced using one or more semaphores

# Semaphore: Mutual exclusion

- Shared data:

```
semaphore mutex;           /* initially mutex = 1 */
```

- Process P<sub>i</sub>:

```
do {  
    wait(mutex);
```

*critical section*

```
    signal(mutex);
```

*remainder section*

```
} while (1);
```

# Producer-consumer problem

## : Semaphore

- Solve producer-consumer problem
  - Full: counting the slots that are full; initial value 0
  - Empty: counting the slots that are empty, initial value N
  - Mutex: prevent access the buffer at the same time, initial value 1 (**binary semaphore**)
  - Synchronization & mutual exclusion

# Semaphores

```
#define N 100                                     /* number of slots in the buffer */
typedef int semaphore;                           /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                            /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);                         /* TRUE is the constant 1 */
        down(&mutex);                          /* generate something to put in buffer */
        insert_item(item);                     /* decrement empty count */
        up(&mutex);                           /* enter critical region */
        up(&full);                            /* put new item in buffer */
        up(&full);                           /* leave critical region */
        up(&full);                           /* increment count of full slots */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);                         /* infinite loop */
        down(&mutex);                        /* decrement full count */
        item = remove_item();                 /* enter critical region */
        up(&mutex);                          /* take item from buffer */
        up(&empty);                           /* leave critical region */
        up(&empty);                          /* increment count of empty slots */
        consume_item(item);                  /* do something with the item */
    }
}
```

$P_0$

wait (S);

wait (Q);

.

.

.

signal (S);

signal (Q);

$P_1$

wait (Q);

wait (S);

.

.

.

signal (Q);

signal (S);

Let **S** and **Q** be two semaphores initialized to 1

# Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let **S** and **Q** be two semaphores initialized to 1

$P_0$

```
wait (S);  
wait (Q);  
.  
.  
.  
signal (S);  
signal (Q);
```

$P_1$

```
wait (Q);  
wait (S);  
.  
.  
.  
signal (Q);  
signal (S);
```

- **Starvation** – indefinite blocking
  - LIFO queue
  - A process may never be removed from the semaphore queue in which it is suspended

# Tutorial problems

# Problem 1

```
flag[i] = true;  
turn = i;  
while ((flag[j] == true) && (turn == j)) { }  
/* CRITICAL SECTION */  
flag[i] = false;  
/* Remainder section */
```

Does this solution work?

# Problem 2

```
flag[i] = true;  
turn = i;  
while ((flag[j] == true) || (turn == j)) { }  
/* CRITICAL SECTION */  
flag[i] = false;  
/* Remainder section */
```

Does this solution work?

# Algorithm for Process $P_i$

```
do {  
    interested[i] = TRUE;  
    turn = j ;  
    while (interested[j] && turn == j);  
        critical section  
    interested[i] = FALSE;  
        remainder section  
} while (TRUE);
```

Provable that

1. Mutual exclusion is preserved
2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

**Does this alter the sequence?**

# CASwap Instruction

---

```
int compare_and_swap(int *value, int expected, int new_value) {  
    int temp = *value;  
  
    if (*value == expected)  
        *value = new_value;  
  
    return temp;  
}
```

---

```
while (true) {  
    while (compare_and_swap(&lock, 0, 1) != 0)  
        ; /* do nothing */  
  
    /* critical section */  
  
    lock = 0;  
  
    /* remainder section */  
}
```

---

# Problem 3: Atomic increment using CAS

Implement atomic increment using CAS

```
increment(&sequence);
```

# Atomic increment using CAS

```
void increment	atomic_int *v)
{
    int temp;

    do {
        temp = *v;
    }
    while (temp != compare_and_swap(v, temp, temp+1));
}
```

# Problem 4: Mutex Lock

```
while (true) {
```

```
    acquire lock
```

critical section

```
    release lock
```

remainder section

```
}
```

```
    acquire() {
        while (!available)
            ; /* busy wait */
        available = false;
    }
```

```
    release() {
        available = true;
    }
```

```
available = 1
acquire()
{ while(CAS(available, 1, 0)==0);
}
```

# Problem 5: Does TSL and CAS Satisfies all properties of critical section solution?

```
do {  
    while (TestAndSet(lock));  
    critical section  
    lock = false;  
    remainder section  
}
```

---

```
while (true) {  
    while (compare_and_swap(&lock, 0, 1) != 0)  
        ; /* do nothing */  
  
    /* critical section */  
  
    lock = 0;  
  
    /* remainder section */  
}
```

---

# System call version

- Special system call that can be used for synchronization
- **TestAndSet**: Test and modify the content of a word **atomically**

```
boolean TestAndSet (boolean &target) {  
    boolean v = target;  
    target = true;  
    return v;  
}
```

# Mutual Exclusion with Test-and-Set

- Shared data:  
*boolean lock = false;*      **Does it satisfy all the conditions?**
- Process  $P_i$   
*do {*  
    *while (TestAndSet(lock)) ;*  
    *critical section*  
    *lock = false;*  
    *remainder section*  
*}*

# Swap Instruction

---

```
int compare_and_swap(int *value, int expected, int new_value) {  
    int temp = *value;  
  
    if (*value == expected)  
        *value = new_value;  
  
    return temp;  
}
```

---

```
while (true) {  
    while (compare_and_swap(&lock, 0, 1) != 0)  
        ; /* do nothing */  
  
    /* critical section */  
  
    lock = 0;  
  
    /* remainder section */  
}
```

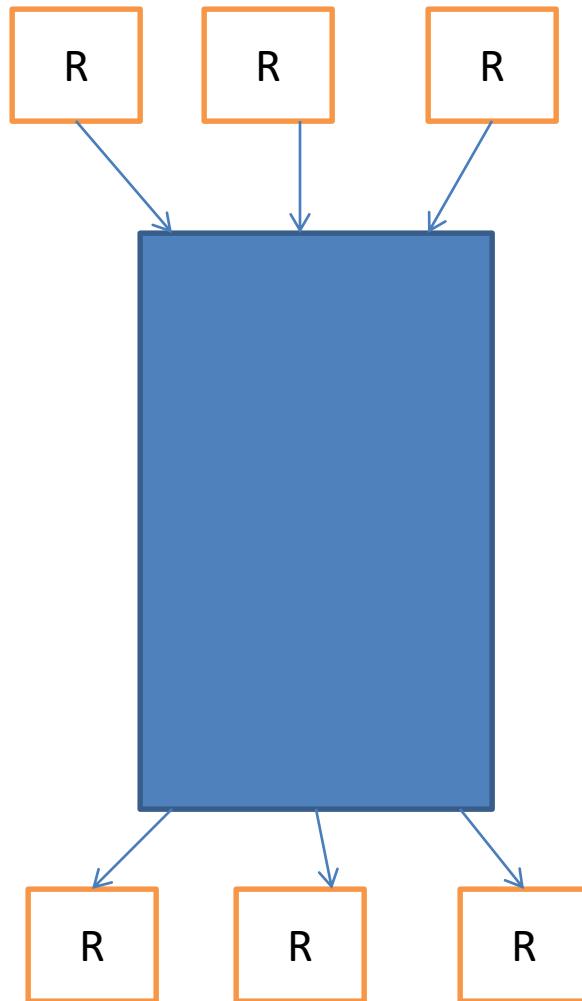
---

# Readers-Writers Problem

- A database is shared among a number of concurrent processes
  - Readers – only read the data set; they do **not** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are treated – all involve priorities
- Shared Data
  - Database
  - Semaphore **mutex** initialized to 1
  - Semaphore **wrt** initialized to 1
  - Integer **readcount** initialized to 0



# Readers-Writers Problem



## Writer

- Task of the writer
  - Just lock the dataset and write

## Reader

- Task of the **first** reader
  - Lock the dataset
- Task of the **last** reader
  - Release the lock
  - Wakeup the any waiting writer

# Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {  
    wait (wrt) ;  
  
    // writing is performed  
  
    signal (wrt) ;  
} while (TRUE);
```

# Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
    wait (mutex) ;
    readcount ++ ;
    if (readcount == 1)
        wait (wrt) ;
    signal (mutex)

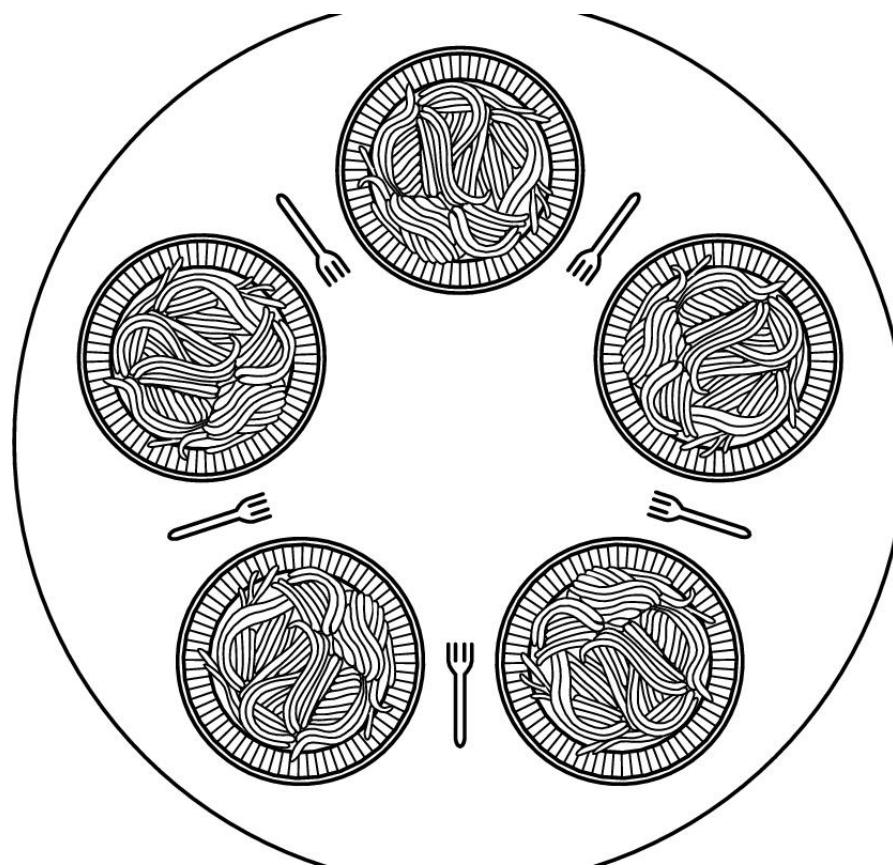
    // reading is performed

    wait (mutex) ;
    readcount -- ;
    if (readcount == 0)
        signal (wrt) ;
    signal (mutex) ;
} while (TRUE);
```

# Readers-Writers Problem (Cont.)

- Models database access
- Current solution
  - Reader gets priority over writer
- Home work
  - Writer gets priority

# Dining Philosophers Problem



# Dining Philosophers Problem

## First solution

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */

{
    while (TRUE) {
        think();
        take_fork(i);
        take_fork((i+1) % N);
        eat();
        put_fork(i);
        put_fork((i+1) % N);
    }
}
```

```
/* philosopher is thinking */
/* take left fork */
/* take right fork; % is modulo operator */
/* yum-yum, spaghetti */
/* put left fork back on the table */
/* put right fork back on the table */
```

- Take\_fork() waits until the fork is available
- Available? Then seizes it

- Ensure that two neighboring philosopher should not seize the same fork

# Dining Philosophers Problem

Each fork is implemented as a semaphore

- The structure of Philosopher  $i$ :      Semaphore **fork [5]**  
initialized to 1

```
do {  
    wait ( fork[i] );  
    wait ( fork[ (i + 1) % 5] );  
  
        // eat  
  
    signal ( fork[i] );  
    signal ( fork[ (i + 1) % 5] );  
  
        // think  
  
} while (TRUE);
```

Ensures no two neighboring  
philosophers can eat  
simultaneously

- What is the problem with this algorithm?

# Dining Philosophers Problem

## First solution

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */

{
    while (TRUE) {
        think();
        take_fork(i);
        take_fork((i+1) % N);
        eat();
        put_fork(i);
        put_fork((i+1) % N);
    }
}
```

- Take\_fork() waits until the fork is available
- Available? Then seizes it
- Suppose all of them take the left fork simultaneously
  - None of them will get the right fork
  - Deadlock

# Dining Philosophers Problem

## Second solution

- After taking the left fork, philosopher checks to see if right fork is available
  - If not, puts down the left fork

### Limitation

- All of them start simultaneously, pick up the left forks
- Seeing that their right forks are not available
  - Putting down their left fork
- Starvation
- Random delay (Exponential backoff) not going to help for critical systems

# Dining Philosophers Problem

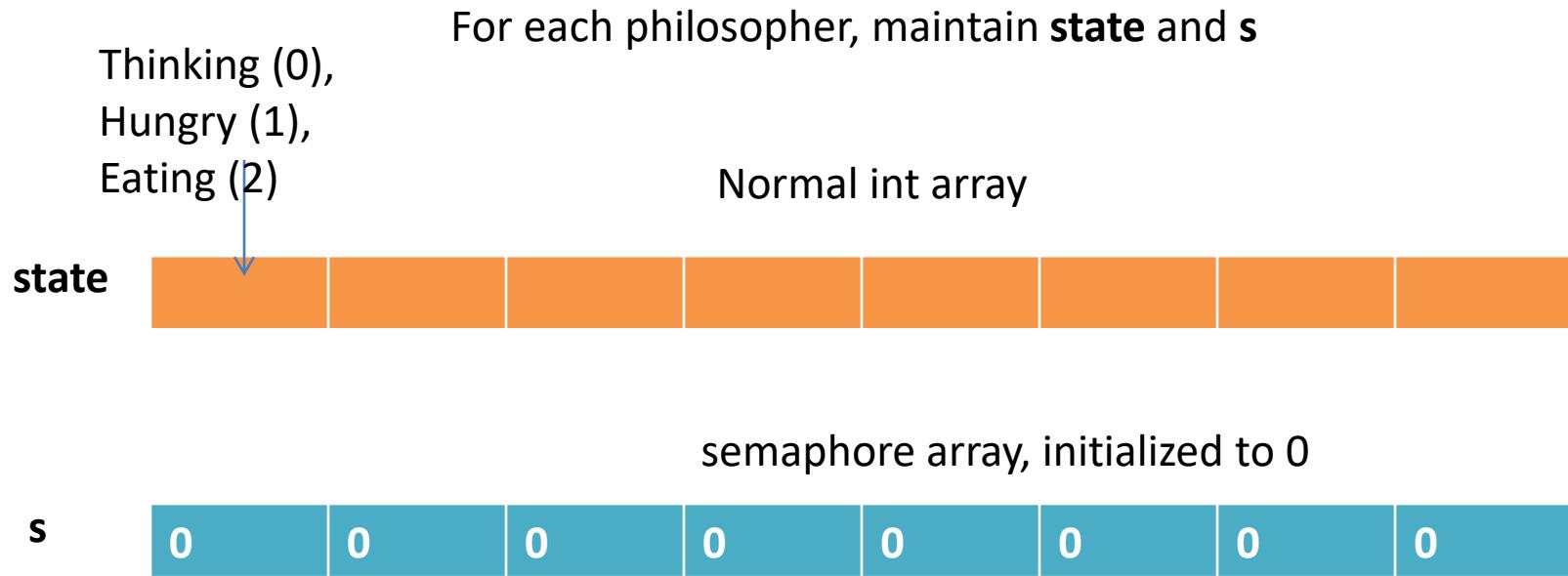
## Third solution

```
#define N 5                                     /* number of philosophers */  
  
void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */  
{  
    while (TRUE) {  
        think(); Wait(mutex);                  /* philosopher is thinking */  
        take_fork(i);                        /* take left fork */  
        take_fork((i+1) % N);                /* take right fork; % is modulo operator */  
        eat();                                /* yum-yum, spaghetti */  
        put_fork(i);                          /* put left fork back on the table */  
        put_fork((i+1) % N);                /* put right fork back on the table */  
        signal(mutex);  
    }  
}
```

Poor resource utilization

# Dining Philosophers Problem

## Final solution



- **State** takes care of acquiring the fork
- **s** stops a philosopher from eating when fork is not available

# Dining Philosophers Problem

## Final solution

```
#define N          5           /* number of philosophers */
#define LEFT        (i+N-1)%N   /* number of i's left neighbor */
#define RIGHT       (i+1)%N    /* number of i's right neighbor */
#define THINKING    0           /* philosopher is thinking */
#define HUNGRY      1           /* philosopher is trying to get forks */
#define EATING      2           /* philosopher is eating */
typedef int semaphore;
int state[N];
semaphore mutex = 1;
semaphore s[N];

void philosopher(int i)
{
    while (TRUE) {
        think();
        take_forks(i);
        eat();
        put_forks(i);
    }
}
```

/\* semaphores are a special kind of int \*/
/\* array to keep track of everyone's state \*/
/\* mutual exclusion for critical regions \*/
/\* one semaphore per philosopher \*/

/\* i: philosopher number, from 0 to N-1 \*/

/\* repeat forever \*/
/\* philosopher is thinking \*/
/\* acquire two forks or block \*/
/\* yum-yum, spaghetti \*/
/\* put both forks back on table \*/

# Dining Philosophers Problem

## Final solution

```
...  
void take_forks(int i)                                /* i: philosopher number, from 0 to N-1 */  
{  
    down(&mutex);  
    state[i] = HUNGRY;  
    test(i);  
    up(&mutex);  
    down(&s[i]);  
    /* enter critical region */  
    /* record fact that philosopher i is hungry */  
    /* try to acquire 2 forks */  
    /* exit critical region */  
    /* block if forks were not acquired */  
}  
...
```

# Dining Philosophers Problem

## Final solution

...

```
void put_forks(i)                                /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                               /* enter critical region */
    state[i] = THINKING;                      /* philosopher has finished eating */
    test(LEFT);                                /* see if left neighbor can now eat */
    test(RIGHT);                               /* see if right neighbor can now eat */
    up(&mutex);                               /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

# The Sleeping Barber Problem

```
#define CHAIRS 5

typedef int semaphore;
semaphore customers = 0;
semaphore barbers = 0;
semaphore mutex = 1;
int waiting = 0;

void barber(void)
{
    while (TRUE) {
        down(&customers);
        down(&mutex);
        waiting = waiting - 1;
        up(&barbers);
        up(&mutex);
        cut_hair();
    }
}

void customer(void)
{
    down(&mutex);
    if (waiting < CHAIRS) {
        waiting = waiting + 1;
        up(&customers);
        up(&mutex);
        down(&barbers);
        get_haircut();
    } else {
        up(&mutex);
    }
}

/* # chairs for waiting customers */
/* use your imagination */

/* # of customers waiting for service */
/* # of barbers waiting for customers */
/* for mutual exclusion */
/* customers are waiting (not being cut) */

/* go to sleep if # of customers is 0 */
/* acquire access to 'waiting' */
/* decrement count of waiting customers */
/* one barber is now ready to cut hair */
/* release 'waiting' */
/* cut hair (outside critical region) */

/* enter critical region */
/* if there are no free chairs, leave */
/* increment count of waiting customers */
/* wake up barber if necessary */
/* release access to 'waiting' */
/* go to sleep if # of free barbers is 0 */
/* be seated and be serviced */

/* shop is full; do not wait */
```

Barber sleeps on “Customer”  
Customer sleeps on “Barber”

**For Barber:** Checking the waiting room and calling the customer makes the **critical section**

**For customer:**  
Checking the waiting room and informing the barber makes its **critical section**

# Deadlock

# The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
- Example
  - System has 2 disk drives
  - $P_1$  and  $P_2$  each hold one disk drive and each needs another one
- Example
  - semaphores  $A$  and  $B$ , initialized to 1

$P_0$   
`wait (A);`  
`wait(B)`

$P_1$   
`wait (B);`  
`wait(A)`

# Introduction To Deadlocks

Deadlock can be defined formally as follows:

*A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.*

# System Model

- Resource types  $R_1, R_2, \dots, R_m$   
*CPU cycles, memory space, I/O devices*
- Each resource type  $R_i$  has  $W_i$  instances.
- Each process requests for an instance of a resource type
- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **release**

# Deadlock: necessary conditions

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** there exists a set  $\{P_0, P_1, \dots, P_n\}$  of waiting processes such that  $P_0$  is waiting for a resource that is held by  $P_1$ ,  $P_1$  is waiting for a resource that is held by  $P_2$ , ...,  $P_{n-1}$  is waiting for a resource that is held by  $P_n$ , and  $P_n$  is waiting for a resource that is held by  $P_0$ .

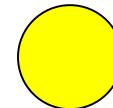
# Resource-Allocation Graph

A set of vertices  $V$  and a set of edges  $E$ .

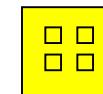
- $V$  is partitioned into two types:
  - $P = \{P_1, P_2, \dots, P_n\}$ , the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \dots, R_m\}$ , the set consisting of all resource types in the system
- **request edge** – directed edge  $P_i \rightarrow R_j$
- **assignment edge** – directed edge  $R_j \rightarrow P_i$

# Resource-Allocation Graph (Cont.)

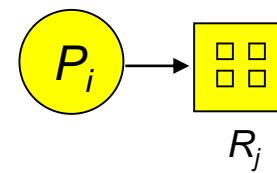
- Process



- Resource Type with 4 instances



- $P_i$  requests instance of  $R_j$



- $P_i$  is holding an instance of  $R_j$

