Operating Systems
Process managment

Christian Khoury

Objectives

- Understand how the system handles processes
  - What a process structure is
  - How processes are identified
  - How processes are scheduled
  - Thread/Process differentiation
  - How inter-process synchronization is done
  - What a deadlock is and how it could be handled

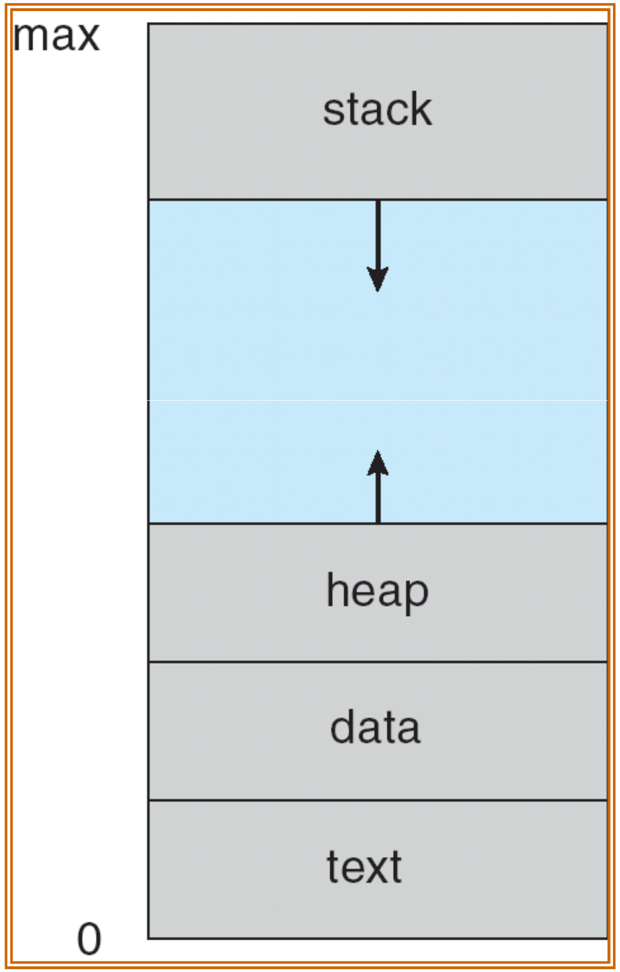
Contents

- **Process Concept**
- Process Scheduling
- Threads Vs Processes
- Process Synchronization
- Deadlocks

Definitions

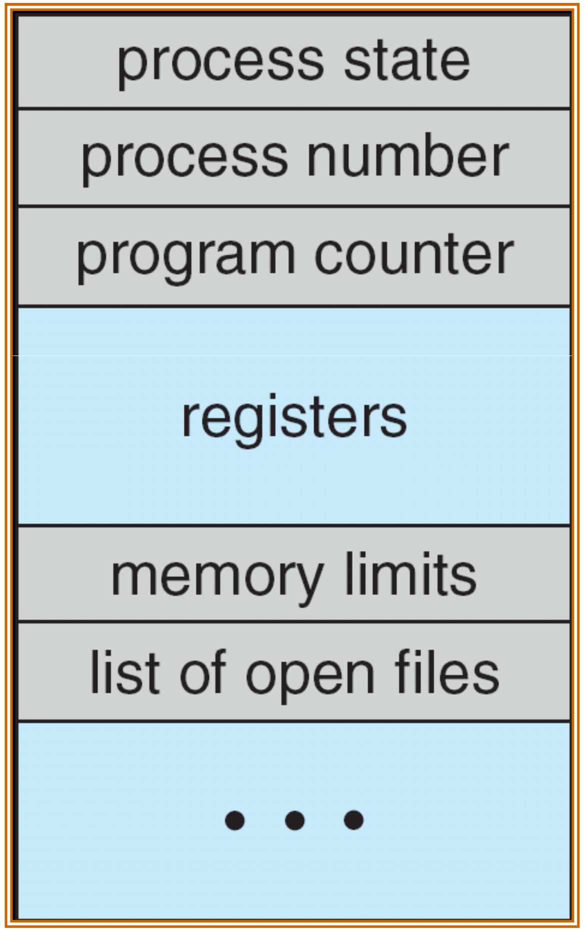
- Program = source code/binary code
  - Static !
- Process = Executed program
  - The only entity recognized by the OS
  - Dynamic !
- Other words that you may encounter
  - Job (batch systems), task, thread

Process Structure

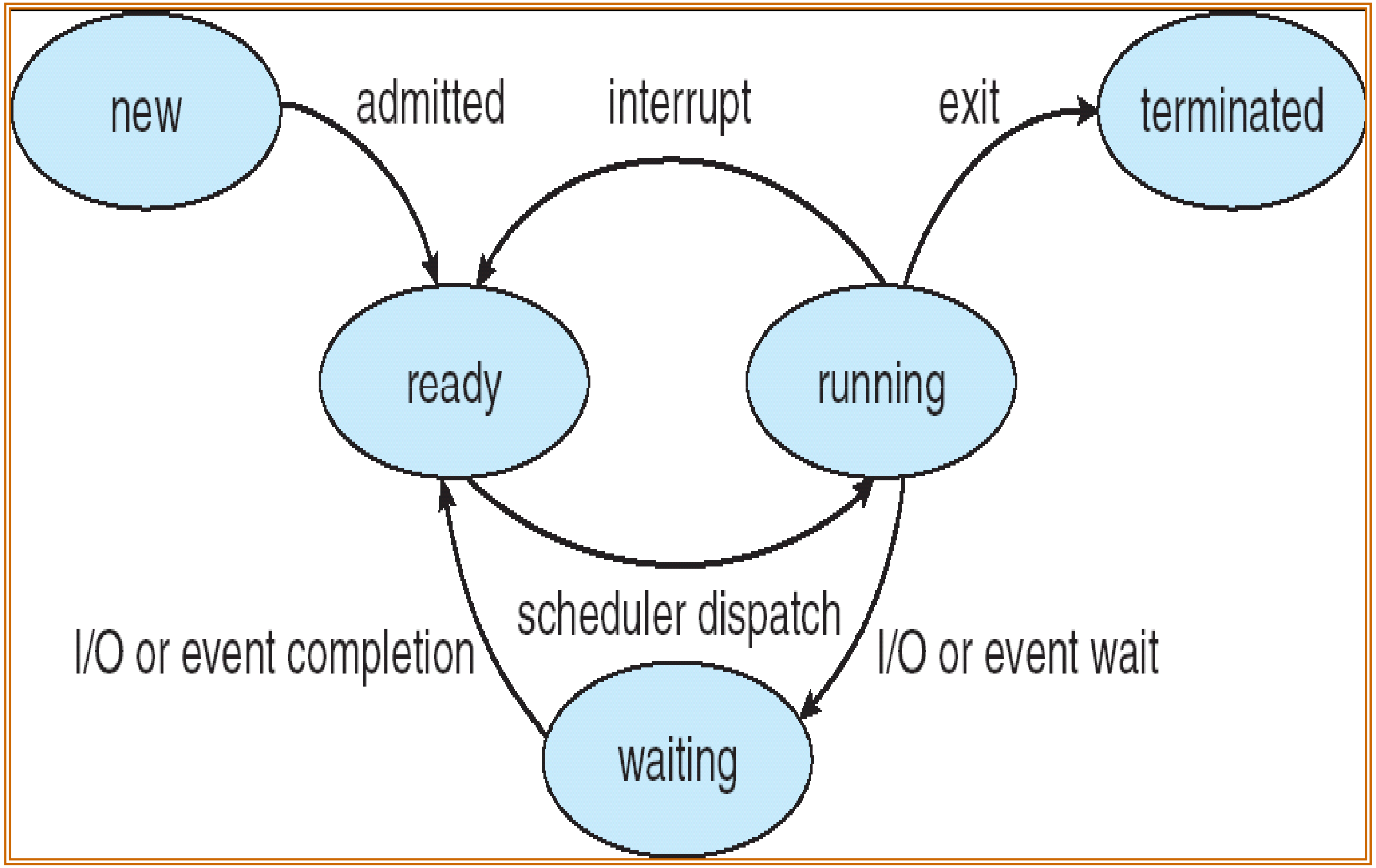


Process Control Block

Contains all the information the system has about a process

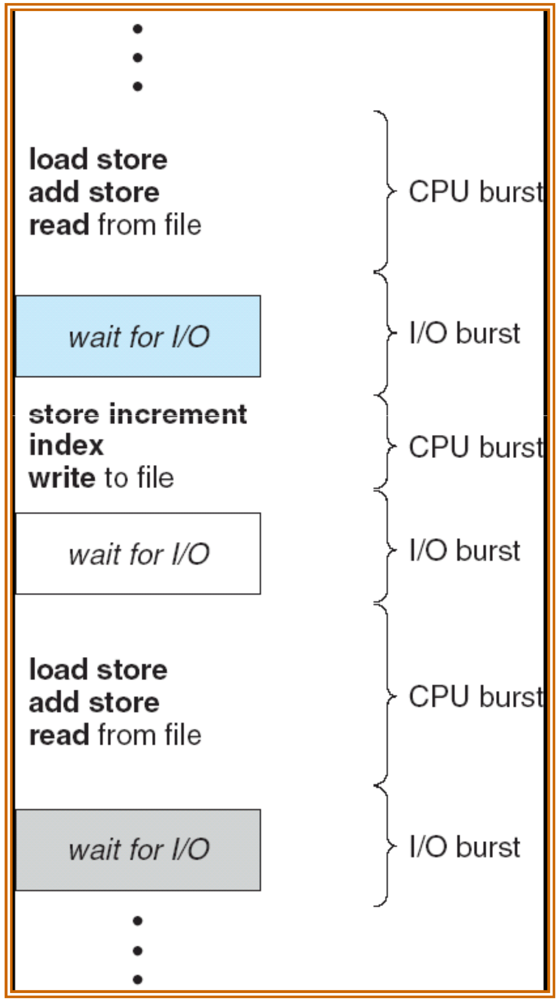


Process State



Process Model

- Any process is an alternating sequence of computations and I/O bursts
  – I/O-bound processes do lots of I/O (e.g., interactive processes)
  – CPU-bound processes do more computation (e.g., scientific calculations)



Contents

- Process Concept
- **Process Scheduling**
- Threads Vs Processes
- Process Synchronization
- Deadlocks

Scheduling Goals

- **Benefit from available resources as much as possible to provide the user with the best possible services (as in any system!)**
  - Maximum CPU utilization
  - Maximum throughput
  - Minimum response time
  - Minimum turnaround time
  - Minimum waiting time

Different Schedulers

- Short-term/CPU scheduler
  - Selects new processes for the CPU
- Long-term scheduler
  - Executes much less frequently
  - Controls the degree of multiprogramming (number of processes in memory)
- Medium-term scheduler
  - It’s the **swapping** scheduler
  - Reduces the degree of multiprogramming

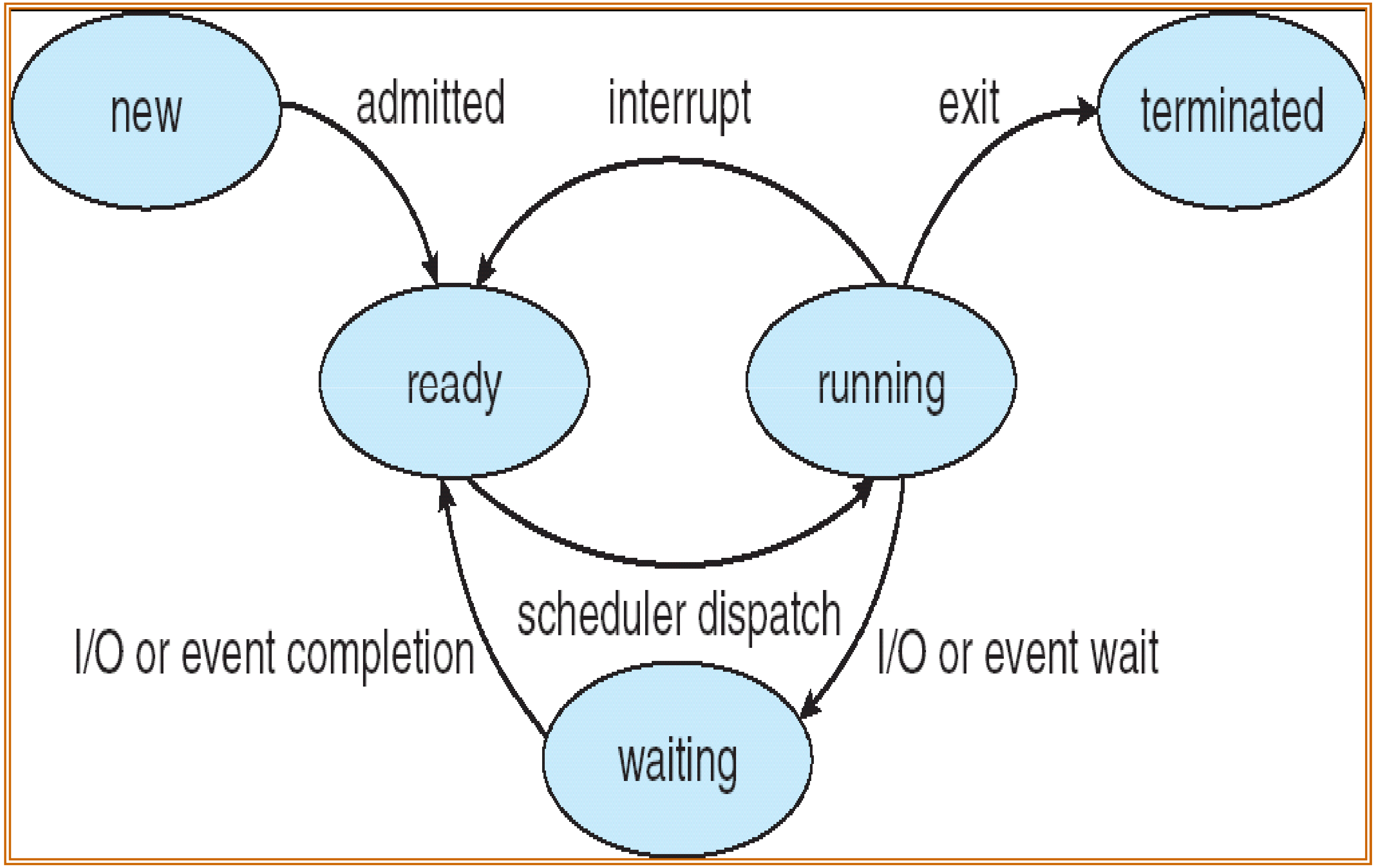
Short-Term Scheduling

1. When the current process awaits an event/IO
2. When a process terminates

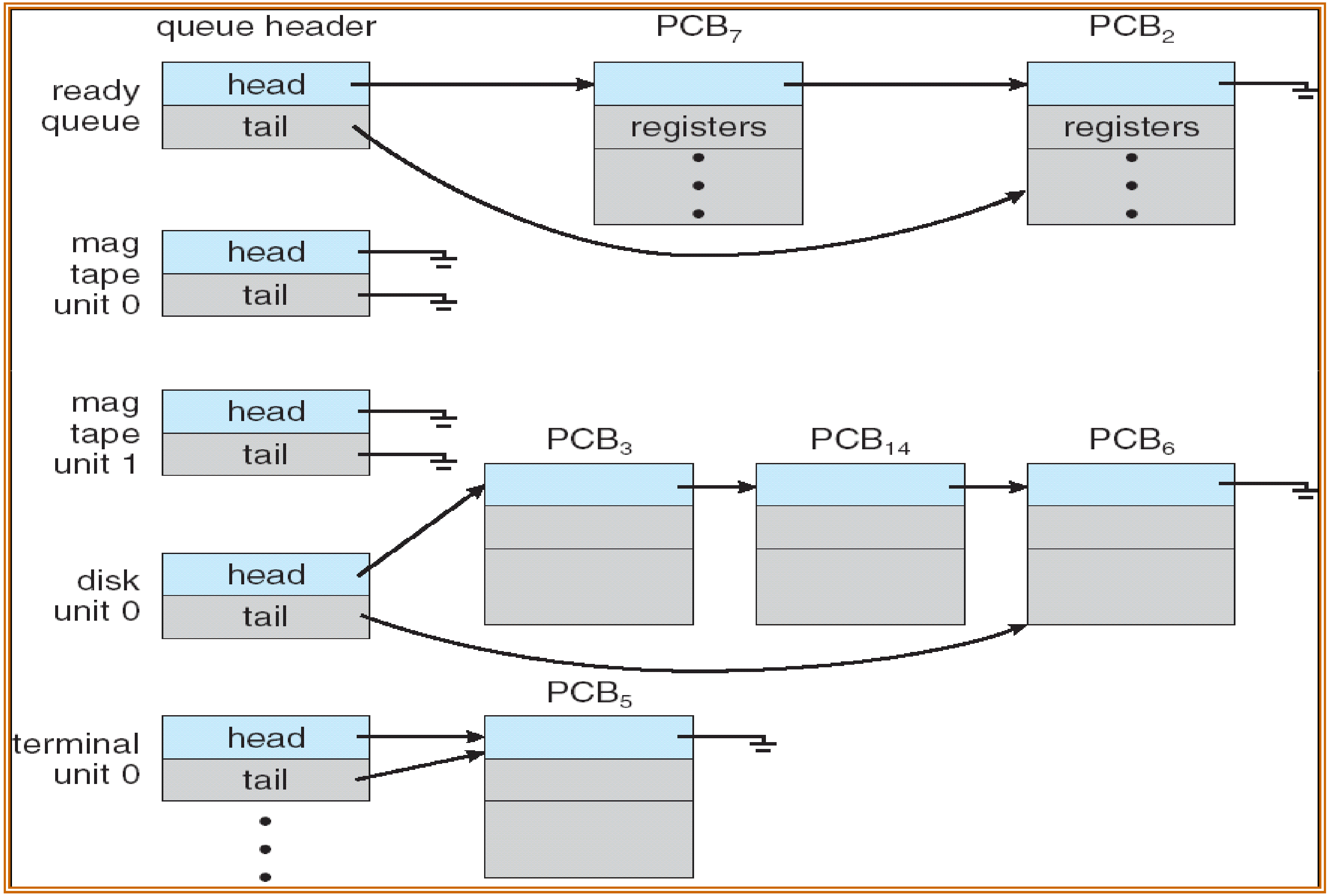
[Preemptive]

3. When a process goes from running state to ready state
4. When a process goes from waiting state to ready state

Process State



Scheduling Queues



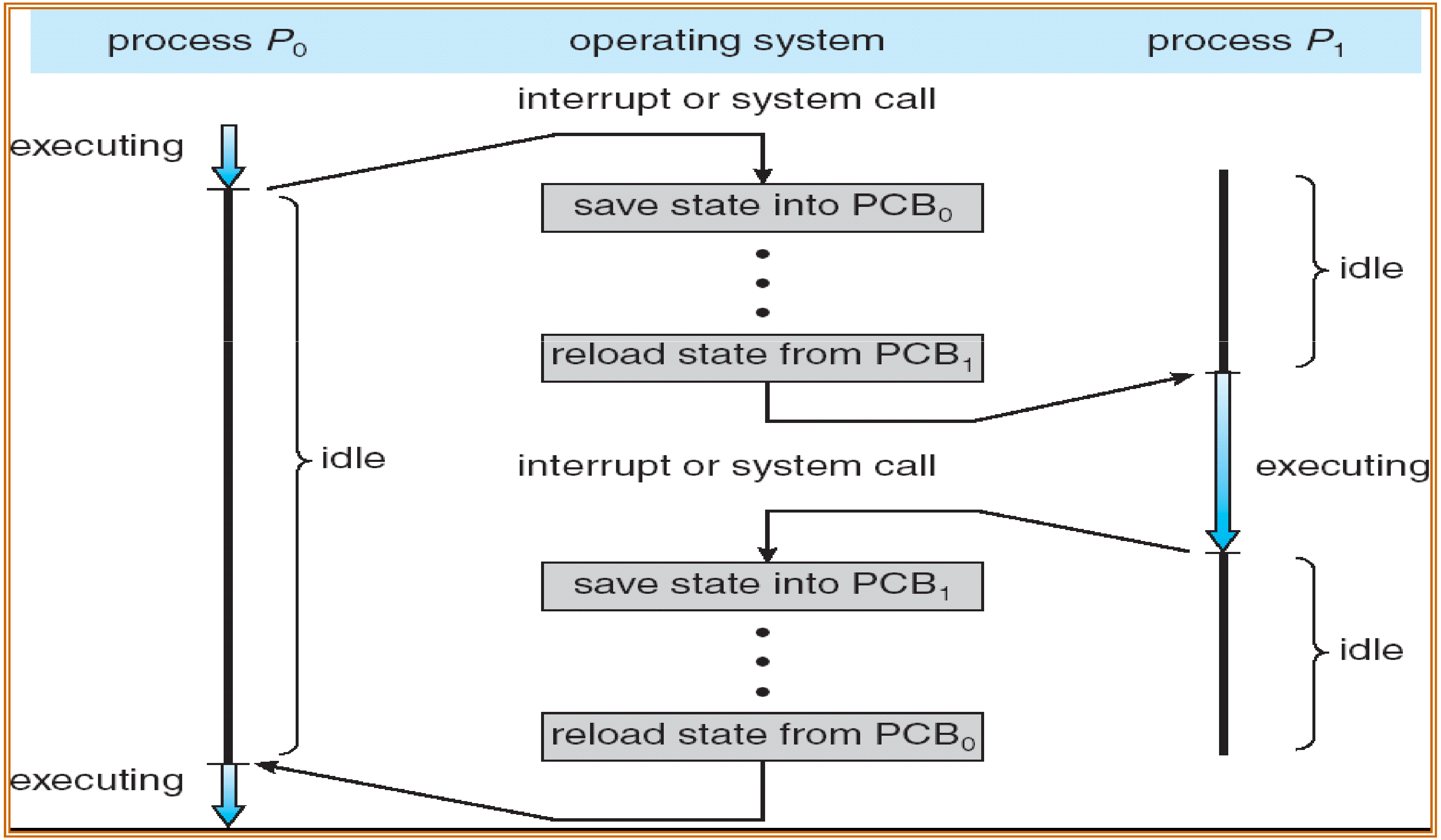
Context Switch (1/2)

- Occurs when a new process is selected by the scheduler

- General context switch process
  1. Save state of current process int its PCB
  2. Change its state to « ready » (add it to the ready queue)
  3. Reload state of the selcted process
  4. Change its state to « running »

- Context switch is overhead

Context Switch (2/2)



Scheduling Algorithms

First-Come, First Served (1/2)

- Jobs are selected for execution as they come (FIFO)
- Non-preemptive algorithm

- (P1, 24), (P2, 3), (P3, 3)
  - Waiting Time : P1(0), P2(24), P3(27)
  - Average Waiting Time = (0 + 24 + 27)/3 = 17

First-Come, First Served (2/2)

- (P2, 3), (P3, 3), (P1, 24)
  – Waiting Time : P2(0), P3(3), P1(6)
  – Average Waiting Time : (0 + 3 + 6)/3 = 3

Shortest Job First (1/2)

- This is a priority based algorithm
  - Priority = execution time
  - Preeptive and non-preemptive
  - It's provable that it gives minimum average time for a given set of processes

- (P1, 24), (P2, 3), (P3, 3)
- Non-preemptive
  - Waiting Time : P2(0), P3(3), P1(6)
  - Average Waiting Time : (0 + 3 + 6)/3 = 3

Shortest Job First (2/2)

- Preemptive
  - Shortest Remaining Time First
  - (P1, 0, 10), (P2, 2, 4), (P3, 4, 1), (P5, 5, 4)
  - Waiting Time : P1(9), P2(1), P3(0), P4(2)
  - Average waiting time : (9 + 1 + 0 + 2)/4 = 3

General Priority Algorithm

- Premptive or non-preemptive
- Not all processes have the same priority (system, batch, interactive, …)
- The process with the highest priority is selected
- Problem
  – Starvation is possible; lower priority processes may never execute
- Solution
  – Aging; increase priority with « age » (time)

Round Robin (1/2)

- Previous algorithms did not share the cpu equitably !
- This is a time-sharing algorithm
  - Each process executes for a time-quantum (10-100 milliseconds)
  - After it finishes its time quantum, it is preempted and pushed to the back of the ready queue
  - With n processes in front of a process, it would have to wait a maximum of « nq » time units
    - NO STARVATION POSSIBLE !

Round Robin (2/2)

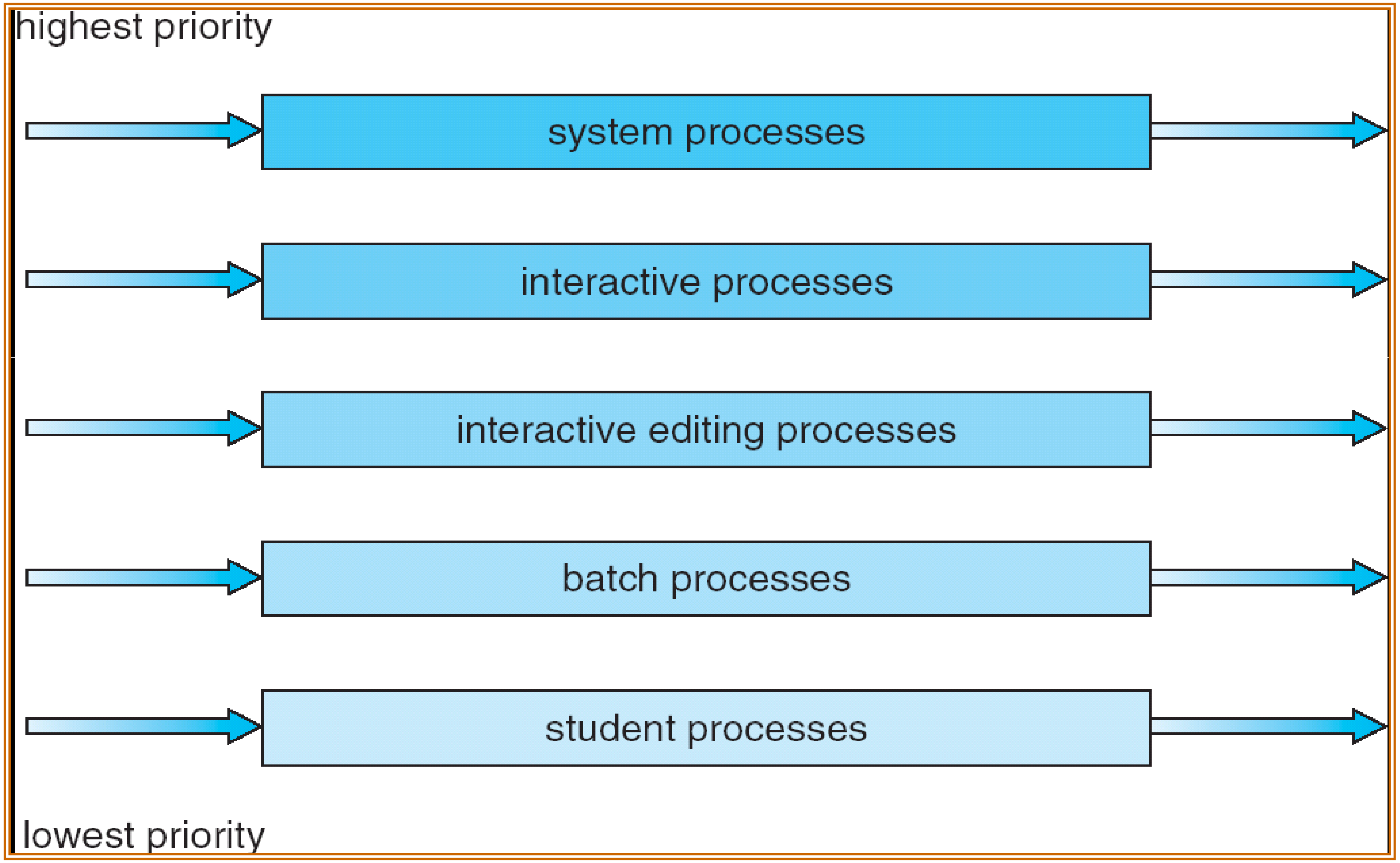
- Large q => FIFO like
  - Not good for interactive processes
- Small q => context switch overhead is too high

- q value
  - should be large enough to make execution time (useful time) higher than the context switch overhead
  - Should be low enough to handle properely interactive processes

Multilevel Queues (1/4)

- Ready queue is partitioned into separate queues:
  - foreground (interactive)
  - background (batch)
- 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.
  - 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

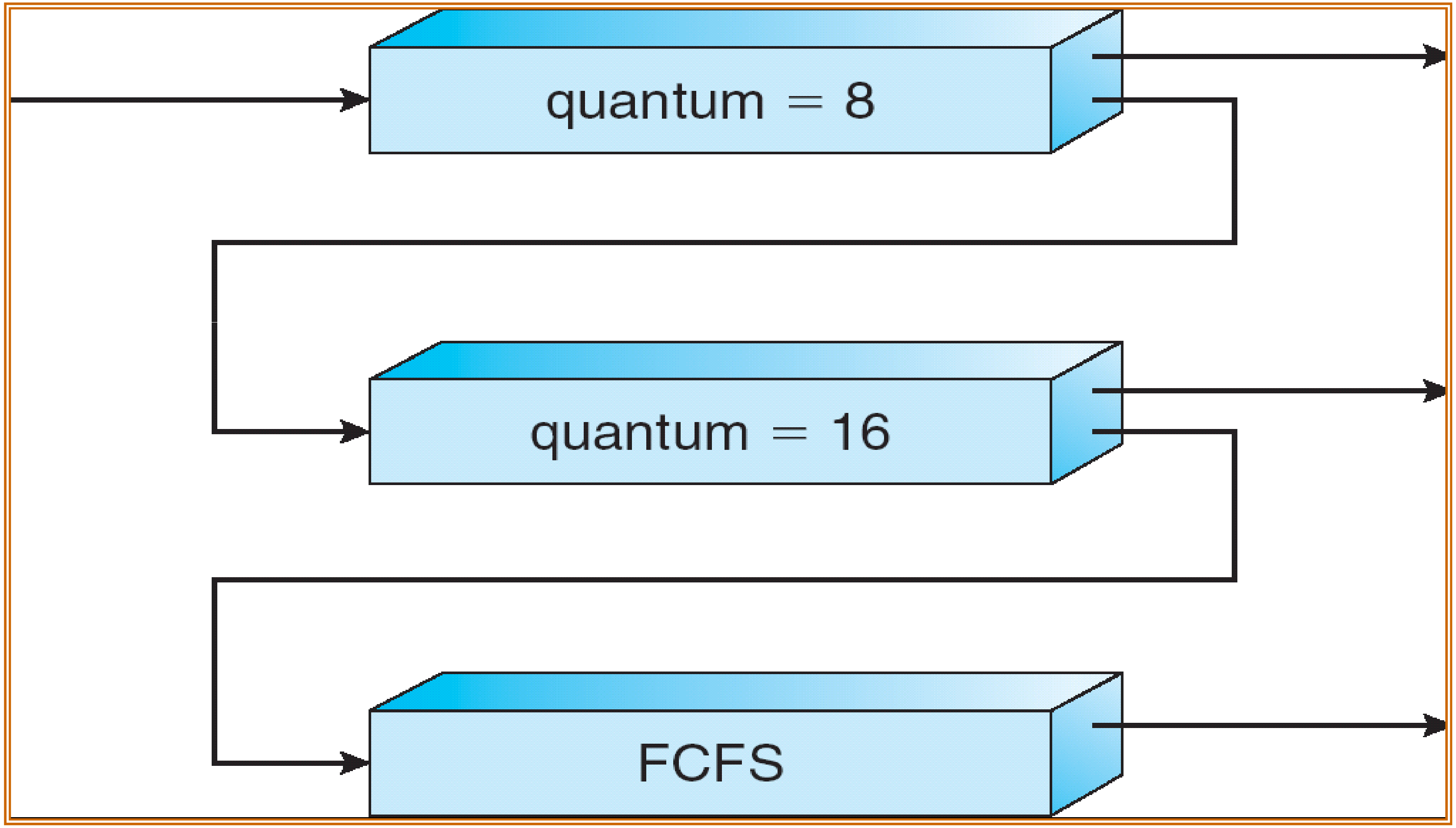
Multilevel Queues (2/4)



Multilevel Feedback Queues (3/4)

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

Multilevel Feedback Queues (4/4)



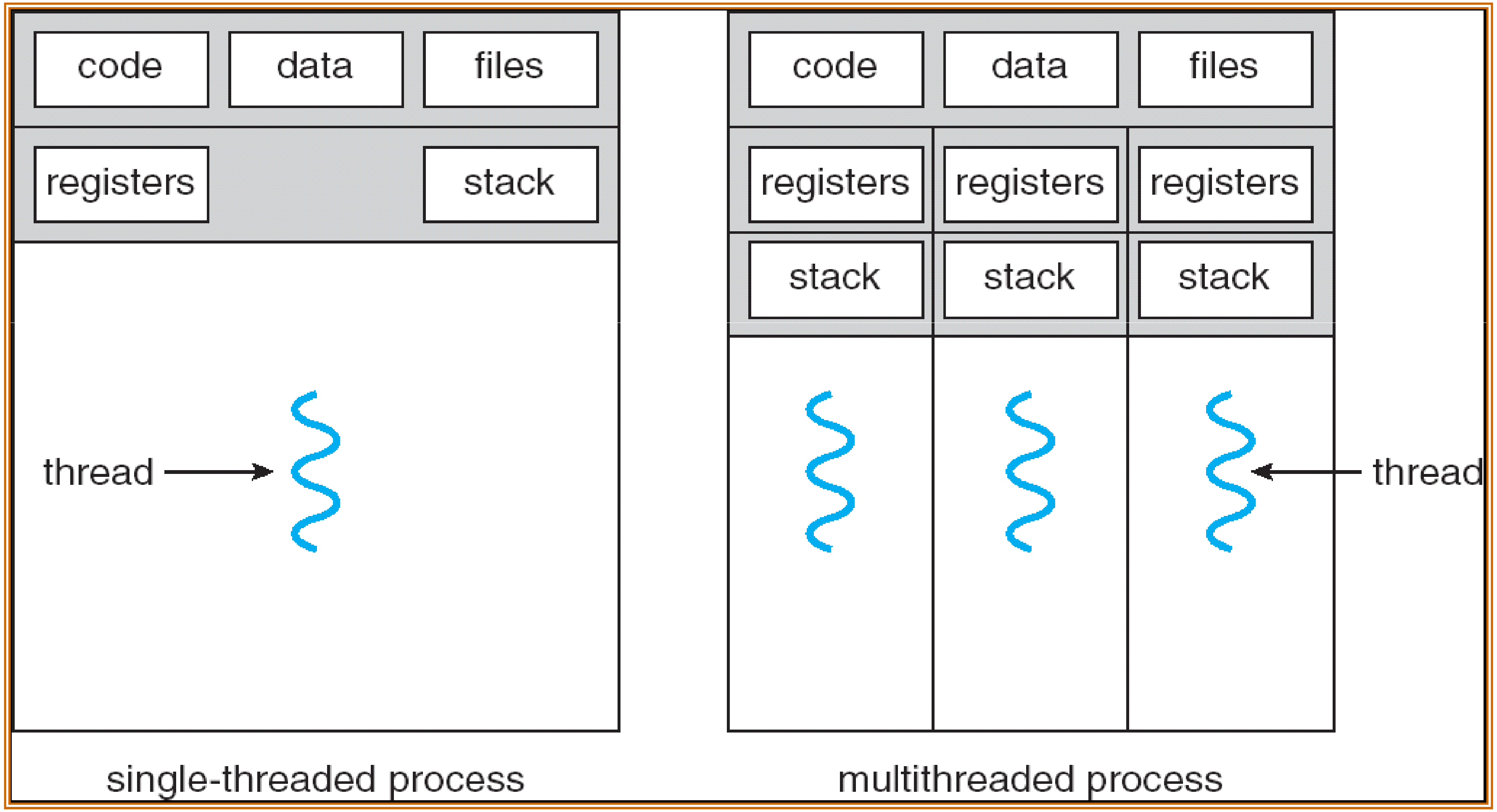
Contents

- Process Concept
- Process Scheduling
- **Threads Vs Processes**
- Process Synchronization
- Deadlocks

Why Threads ?

- Because naturally we do things simultaneously when possible
  - That’s more efficient !

- Example
  - (a+b) – (c*d)/(e-f)


code
data
files
registers
stack
thread
code
data
files
registers
registers
registers
stack
stack
stack
thread
single-threaded process
multithreaded process

Benefits

- Context switch is faster
  - Code/Text and data are shared by threads of the same process
  - Only registers and the stack should be saved and replaced by the new thread
    - Code and data form usually the bigger part of any process !

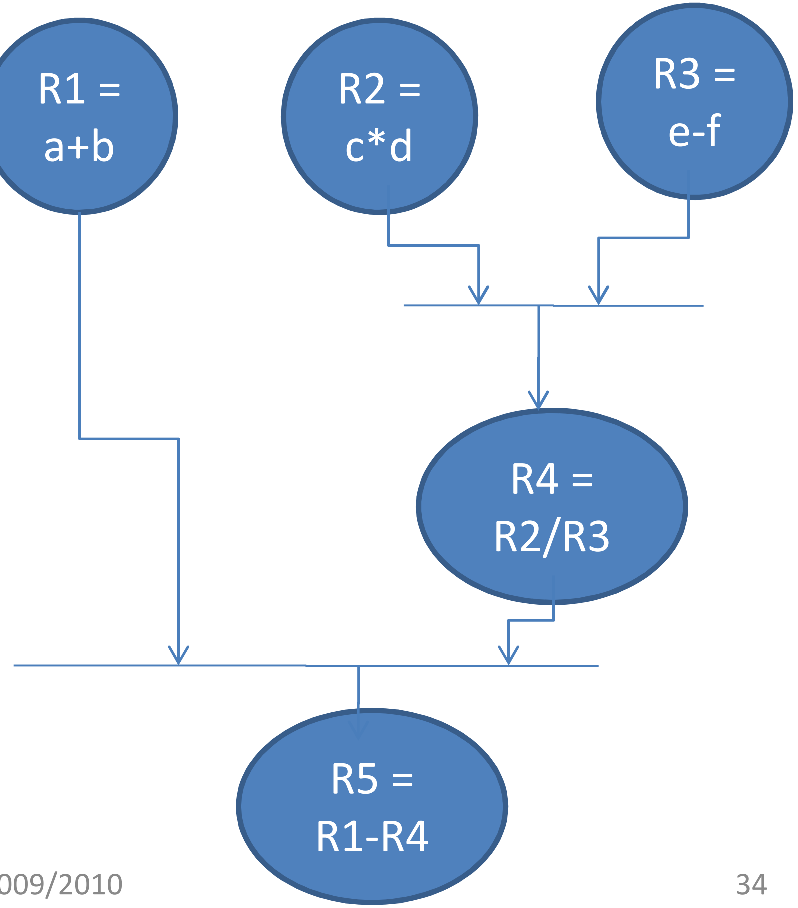
- Issues
  - Should scheduling be done locally (between threads of the same process) or globally (between all threads) ?

Contents

- Process Concept
- Process Scheduling
- Threads Compared to Processes
- **Process Synchronization**
- Deadlocks

Why Synchronize ? (1/4)

- Sometimes processes need to cooperate
- Example
  - (a+b) - (c*d)/(e-f)



Why Synchronize ? (2/4)

- Race Problems
  - Let i be a shared varaible initialized to 5
  - P1 : i++
  - P2 : i--

- You would expect that any possible execution of P1 and P2 would end with i = 5 !!!

Why Synchronize ? (3/4)

<u>P1</u>

(1.1) Reg <- i

(1.2) Inc Reg

(1.3) I <- Reg

<u>P2</u>

(2.1) Reg <- i

(2.2) Dec Reg

(2.3) I <- Reg

- Examine this particular execution
  - 1.1, 2.1, 2.2, 2.3, 1.2, 1.3
  - End result : i = 6
- What if it was : 1.1, 2.1, 1.2, 1.3, 2.2, 2.3 ?

Why Synchronize ? (4/4)

- Because of concurrency problems
  - P1 and P2 modifying the same data « simultaneously » (i++ and i--)
  - These code sections are called « critical »
- Particular solution
  - Execute the increment and decrement operations sequentially (serialized executions)
- General solution
  - Enforce mutual exclusion in critical sections
  - Only one process at a time has the permission to modify shared data

Algorithmic solutions

- Peterson

```
while (true) {
        flag[i] = TRUE;
        turn = j;
        while ( flag[j] && turn == j);

            CRITICAL SECTION

        flag[i] = FALSE;

            REMAINDER SECTION

}
```

- Shortfall
  – Busy waiting !

Semaphores (1/5)

- User's viewpoint
  - Semaphores manage resources
- Structure
  - Counter
  - Queue for blocked processes
- Behaviour
  - P (wait) and V (signal)

Semaphores (2/5)

ACQUIRE a RESOURCE

P(S)

counter --

If (counter < 0)

push process on the queue

FREE a RESOURCE

V(S)

counter ++

If (counter <= 0)

wakeup processes on the queue

Sempahores (3/5)

- P and V operations are atomic/indivisible !
- Sempahores are used as a general synchronization tool

Mututal Exclusion with Semaphores (4/5)

- The resource any process would like to acquire is « entering the critical section »
- And how many processes should be able to do that ? Just ONE (mutual exclusion)
  - 1 resource => S.counter = 1

- Before entering the critical section, any process should acquire « it »
  - P(S)
- On leaving the critical section, it should release the resource
  - V(S)

Semaphores (5/5)

- Problems
  - Incorrect use of semaphore operations:
    - V (mutex) …. P (mutex)
    - P (mutex) … P (mutex)
    - Omitting of P (mutex) or V (mutex) (or both)
- Higher level tools
  - Monitors

Contents

- Process Concept
- Process Scheduling
- Threads Compared to Processes
- Process Synchronization
- **Deadlocks**

Deadlocks

- 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.

- In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting

System Model

- Resource types $R_1, R_2, \ldots, R_m$

  CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - request
  - use
  - release

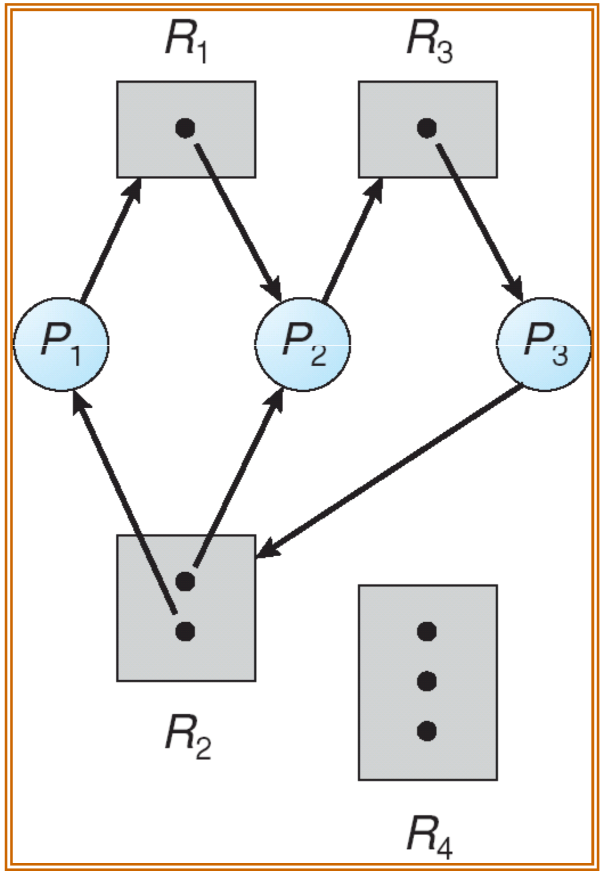
Necessary Conditions for Deadlocks

- **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 $\{P_0, P_1, \ldots, P_0\}$ 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_0 is waiting for a resource that is held by P_0.

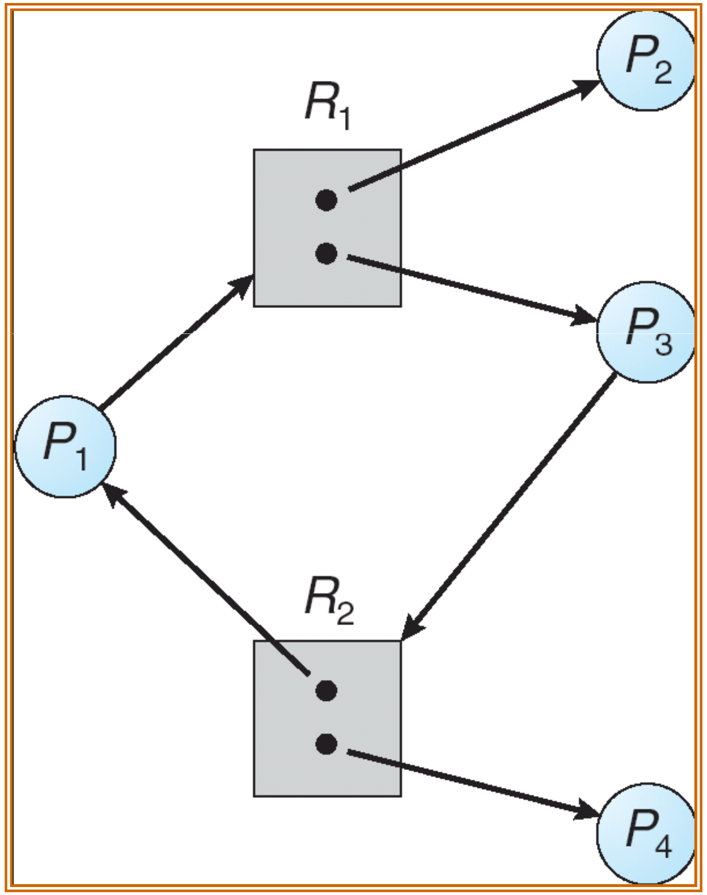
Deadlock Graph

- Resource-Allocation Graph with
  - Vertices : Processes and resources
  - Edges
    - Request edge : P1 -> R1
    - Allocation edge : R1 -> P1

Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



Handling Deadlocks (1/3)

- Prevention/Avoidance

- Detect and Recover

- Pretend that deadlocks never occur !

Deadlock Prevention/Avoidance (2/3)

- Prevention
  - Restrain the ways requests can be made
  - Ensure that at least one of the necessary conditions for deadlocks cannot hold !
- Avoidance
  - Use an algorithm to ensure allocations cannot lead to a deadlock
  - Banker's Algorithm

Deadlock Detection(3/3)

- An algorithm that examines the state of the system to determine whether a deadlock has occurred

- An algorithm to recover from the deadlock