Chapter 6 CPU Scheduling

A/Prof Jian Zhang

School of Computing and Communication

University of Technology, Sydney

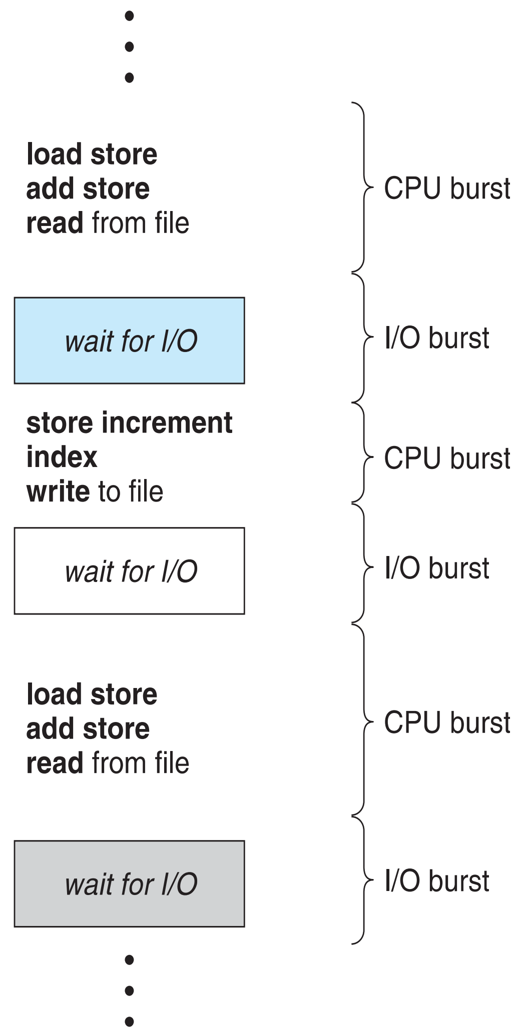
Ref: A. Silberschatz, P. B. Galvin & G. Gagne

Chapter 6: CPU Scheduling

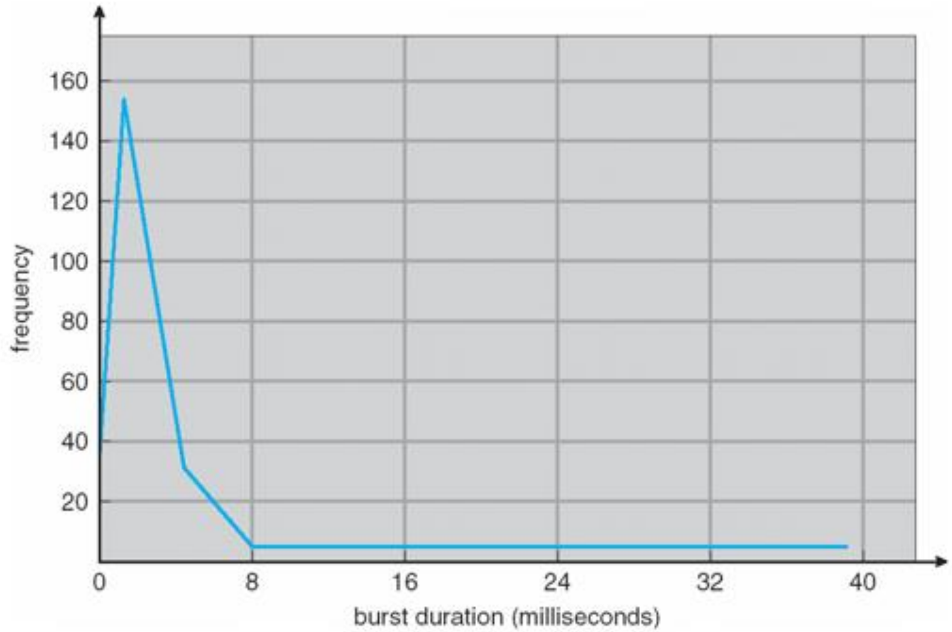
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation

CPU–I/O Burst Cycle – Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern



CPU–I/O Burst Cycle



- This distribution can be important in the selection of an appropriate CPU-scheduling algorithm.
  - <u>An I/O-bound program typically has many short CPU bursts.</u>
  - <u>A CPU-bound program might have a few long CPU bursts.</u>
  - <u>By giving higher priority to I/O-bound programs and allow them to execute ahead of the CPU-bound programs</u>

CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
- 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**
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities
- **Long-term scheduler** to make decisions on which job(s) to move from Job pool to ready queue and moves them

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 (for time-sharing environment)
- **Scheduling Algorithm Optimization Criteria**
  - Max CPU utilization
  - Max throughput
  - Min turnaround time
  - Min waiting time
  - Min response time

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

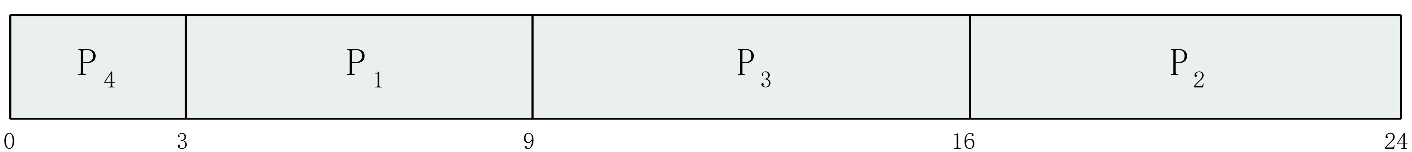
- Waiting time for P_1 = 6; P_2 = 0; P_3 = 3
- Average waiting time: (6 + 0 + 3)/3 = 3
- **Much better than previous case**
- **Convoy effect** - short process behind long process
  - as all the other processes wait for the one big process to get off the CPU
  - This effect results in lower CPU and device utilization.

Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is how to know the length of the next CPU request in advance

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



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

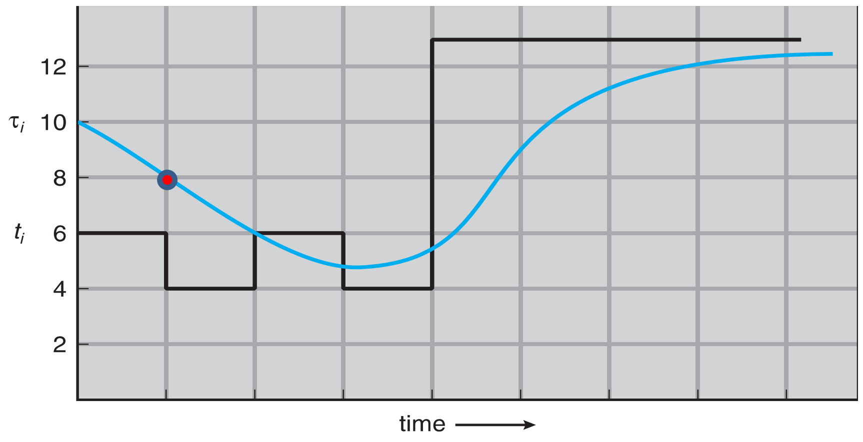
Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous one
  - Then pick the process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging

1. $t_n =$ actual length of n^{th} CPU burst
2. $\tau_{n+1} =$ predicted value for the next CPU burst
3. $\alpha, 0 \le \alpha \le 1$
4. Define: $\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n.$

- Commonly, α set to ½

Prediction of the Length of the Next CPU Burst



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

Examples of Exponential Averaging

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

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

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\alpha\, t_{n-1} + \ldots + (1 - \alpha)^j \alpha\, t_{n-j} + \ldots + (1 - \alpha)^{n+1} \tau_0$$

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

Example of Shortest-remaining-time-first

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

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

- ***Preemptive* SJF Gantt Chart**

P_1	P_2	P_4	P_1	P_3	
0	1	5	10	17	26

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

Example of Priority Scheduling

non-preemptive priority (a larger priority number implies a higher priority),

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

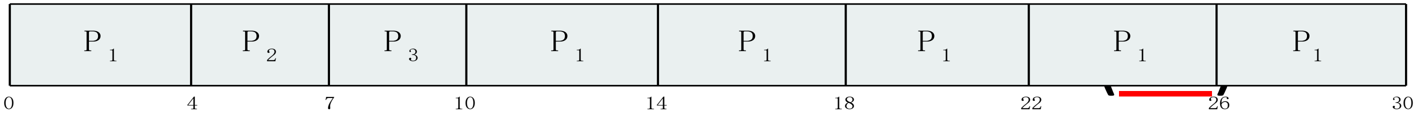
- Priority scheduling Gantt Chart

P_2	P_5	P_1	P_3	P_4	
0	1	6	16	18	19

- Average waiting time = (6+0+16+18+1)/5=8.2 msec

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

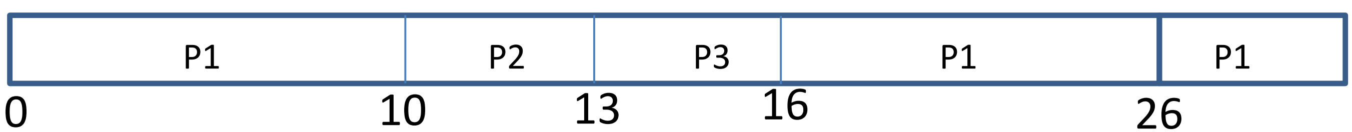


- Each process gets a small unit of CPU time (**time quantum** *q*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of 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
  - q large $\Rightarrow$ FIFO
  - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high

Process	Burst Time
P_1	24
P_2	3
P_3	3

Q=10 ms

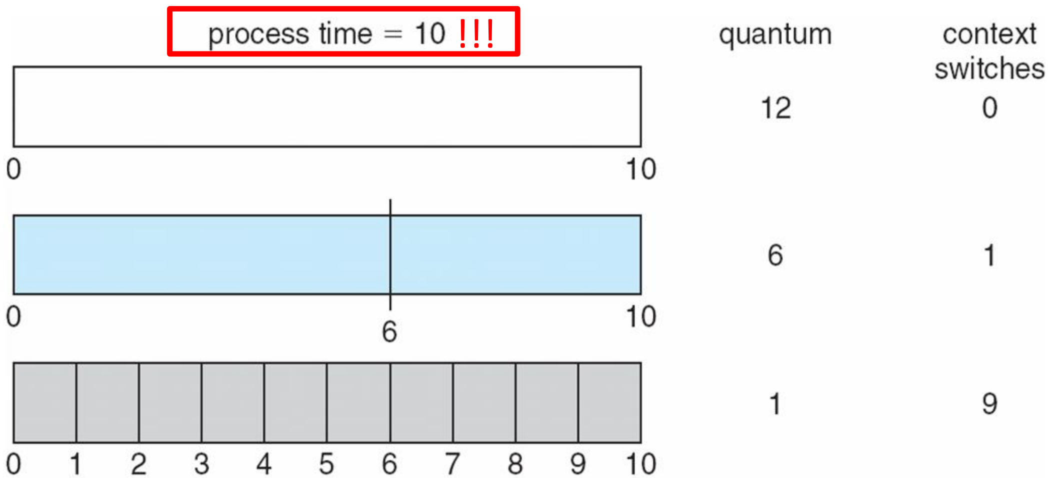
- The Gantt chart is:



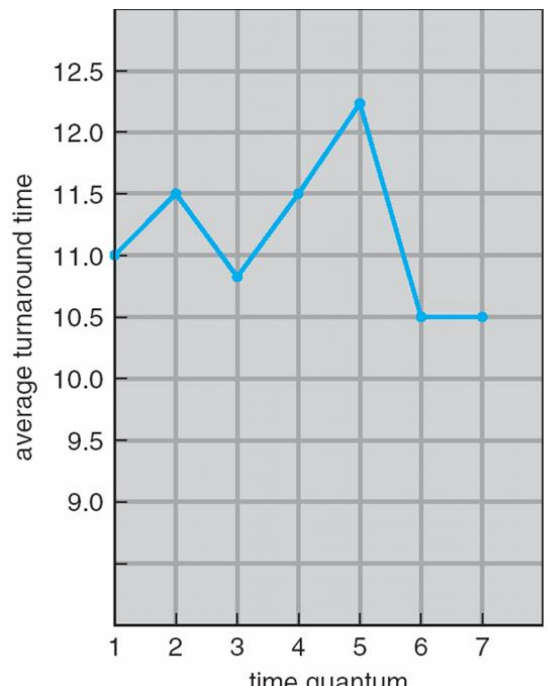
- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec

P1 = (0+16+26)/3 = 14, P2=10, P3= 13
AWT = (14+10+13)/3=12.3 msec

Time Quantum and Context Switch Time



Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

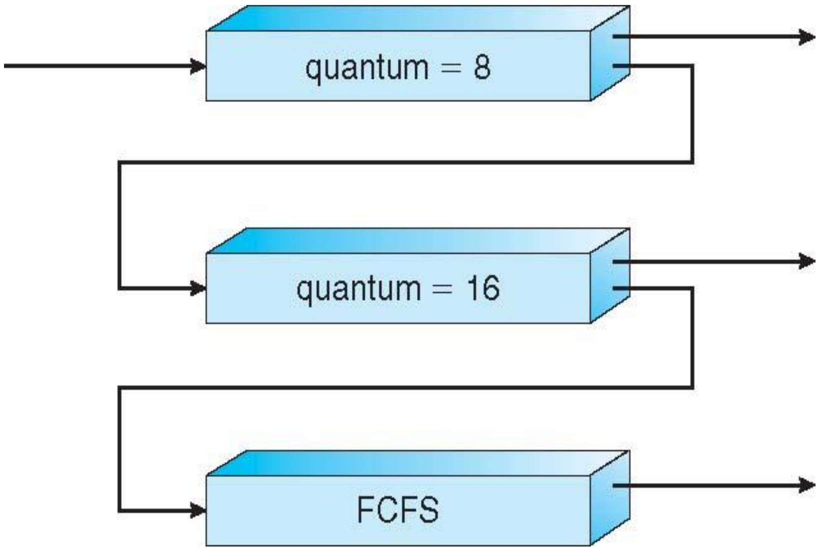
The Max performance is at the q=5!

Multilevel Queue Scheduling

- Ready queue is partitioned into separate queues, eg:
  - **foreground** (interactive)
  - **background** (batch)
- Processes are permanently assigned to a given queue when they enter the system.
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS (First come first sever)
- 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 and 20% to background in FCFS

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS



- Scheduling
  - A new job enters queue Q_0 which is served RR
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q_1
  - At Q_1 job is again served RR and receives 16 additional milliseconds. But the process are run only when queues 0 is empty
  - If it still does not complete, it is preempted and moved to queue Q2. The processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty.

Thread Scheduling

- Distinction between user-level and kernel-level threads
- On operating systems that support them, it is kernel-level threads—not processes—that are being scheduled by the operating system
- User level threads are managed by thread library
- Contention scope
  - Known as **process-contention scope** (**PCS**) since scheduling competition is within the process
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is **system-contention scope** (**SCS**) – competition among all threads in system

Pthread Scheduling

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

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – 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
  - Currently, most common
- **Processor affinity** – process has affinity for processor on which it is currently running
  - **soft affinity**
  - **hard affinity**
  - Variations including **processor sets**

Chapter 7 Deadlocks

A/Prof Jian Zhang
School of Computing and Communication
University of Technology, Sydney

Ref: A. Silberschatz, P. B. Galvin & G. Gagne

Deadlock Characterization

A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources that it has requested are held by other waiting processes. This situation is called a **deadlock**

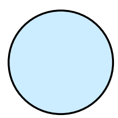
Deadlock can arise if four conditions hold simultaneously:

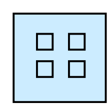
- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that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

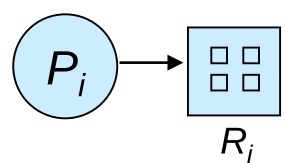
Resource-Allocation Graph

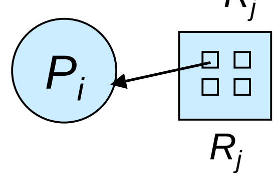
A set of vertices V and a set of edges E.

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

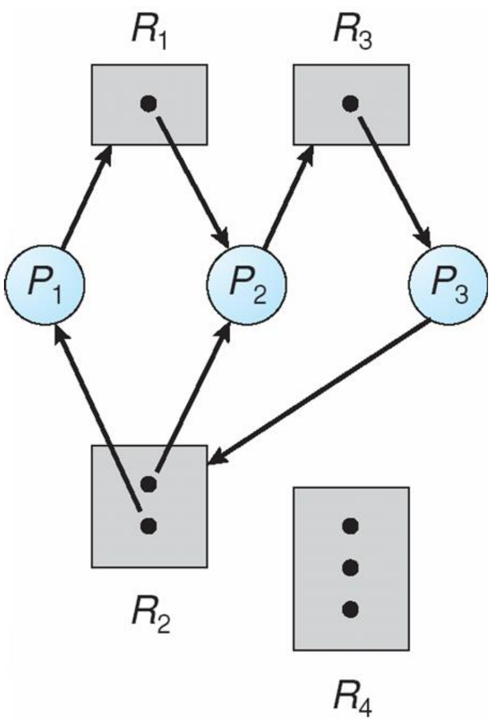
  - Process 

    Resource Type with 4 instances 

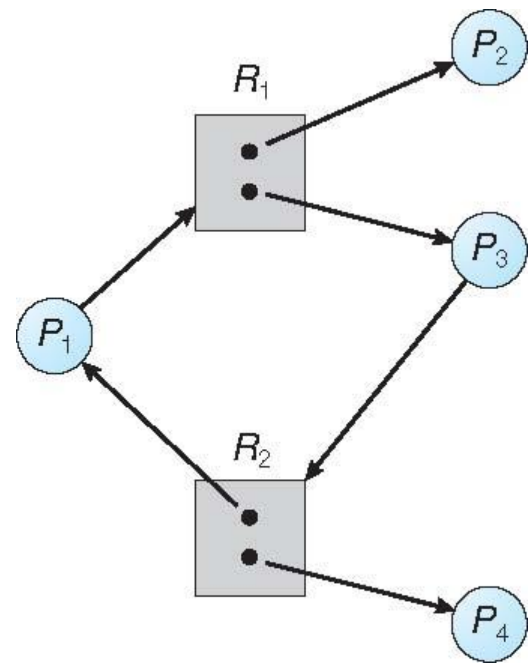
  - P_i requests instance of R_j 

  - P_i is holding an instance of R_j 

Resource-Allocation Graph



Graph With A Deadlock



Graph With A Cycle But No Deadlock

At this point, two minimal cycles exist in the system

$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow \boxed{R_2} \rightarrow P_1$

$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow \boxed{R_2} \rightarrow P_2$

Processes *P*1, *P*2, and *P*3 are deadlocked. Process *P*2 is waiting for the resource *R*3, which is held by process *P*3. Process *P*3 is waiting for either process *P*1 or process *P*2 to release resource *R*2. In addition, process *P*1 is waiting for process *P*2 to release resource *R*1

Methods for Handling Deadlocks

- Ensure that the system will ***never*** enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidance
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

Deadlock Avoidance

Requires that the system has some additional ***a priori*** information available

- Simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

Safe State

- System is in **safe state** if there exists a **safe sequence** $<P_1, P_2, \ldots, P_n>$ of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

Basic Facts

- If a system is in safe state $\Rightarrow$ no deadlocks
- If a system is in unsafe state $\Rightarrow$ possibility of deadlock
- Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.

Avoidance Algorithms

- Single instance of a resource type Use a resource-allocation graph
- Multiple instances of a resource type Use the banker's algorithm

Banker's Algorithm

- Multiple instances
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

The deadlock avoidance algorithm that we describe next is applicable to such a system but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the **banker's algorithm.** The name was chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.

Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resources types.

- **Available**: Vector of length m. If available $[j] = k$, there are k instances of resource type R_j available
- **Max**: $n \times m$ matrix. If $Max[i,j] = k$, then process P_i may request at most k instances of resource type R_j
- **Allocation**: $n \times m$ matrix. If Allocation$[i,j] = k$ then P_i is currently allocated k instances of R_j
- **Need**: $n \times m$ matrix. If $Need[i,j] = k$, then P_i may need k more instances of R_j to complete its task

$$Need[i,j] = Max[i,j] - Allocation[i,j]$$

These data structures vary over time in both size and value.

To simplify the presentation in the algorithm, we establish some notation.

Let X and Y be vectors of length n. We say that $X \leqslant Y$ if and only if $X[i] \leqslant Y[i]$ for all $i = 1, 2, \ldots, n$. For example, if $X = (1,7,3,2)$ and $Y = (0,3,2,1)$, then $Y < X$.

Safety Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively. Initialize:

   Work = Available

   ***Finish* [*i*] = *false* for *i* = 0, 1, …, *n*- 1**

2. Find an ***i*** such that both:

   (a) ***Finish* [*i*] = *false***

   (b) $\boldsymbol{Need_i \leq Work}$

   If no such ***i*** exists, go to step 4

3. ***Work* = *Work* + $\boldsymbol{Allocation_i}$**
   Finish*[*i*] = *true
   go to step 2

4. If ***Finish* [*i*] == *true*** for all ***i***, then the system is in a safe state

Resource-Request Algorithm for Process P_i

Request**$_i$* = request vector for process ***P**$_i$*. If ***Request**$_i$* **[*j*] = *k then process ***P**$_i$* wants ***k*** instances of resource type ***R**$_j$*

1. If ***Request**$_i$* $\leq$ ***Need**$_i$* go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If ***Request**$_i$* $\leq$ ***Available***, go to step 3. Otherwise ***P**$_i$* must wait, since resources are not available
3. Pretend to allocate requested resources to ***P**$_i$* by modifying the state as follows:

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- If safe $\Rightarrow$ the resources are allocated to ***P**$_i$*
- If unsafe $\Rightarrow$ ***P**$_i$* must wait, and the old resource-allocation state is restored

Example of Banker's Algorithm

- 5 processes P_0 through P_4;

  3 resource types:

  A (10 instances), B (5instances), and C (7 instances)

- Snapshot at time T_0:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Example (Cont.)

- The content of the matrix ***Need*** is defined to be ***Max – Allocation***

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

- The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria

Example: P_1 Request (1,0,2)

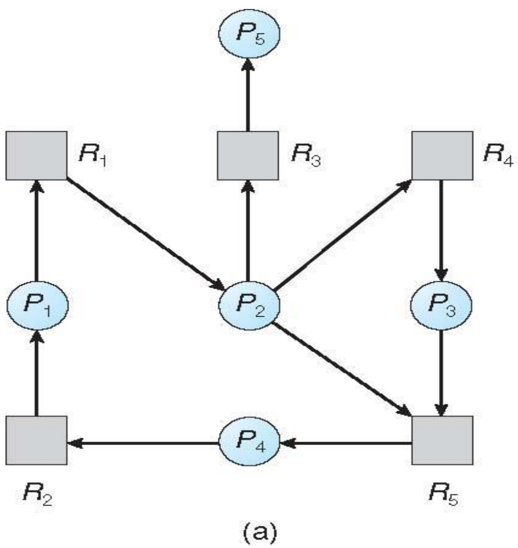
- Check that Request $\leq$ Available (that is, $(1,0,2) \leq (3,3,2) \Rightarrow$ true

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

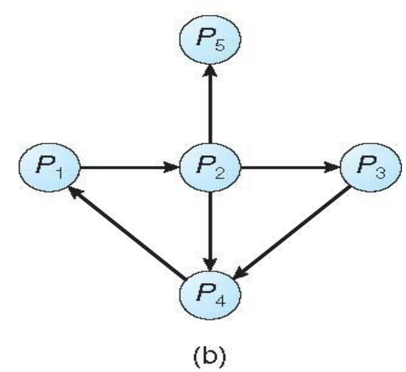
- Executing safety algorithm shows that sequence < $\boldsymbol{P_1, P_3, P_4, P_0, P_2}$> satisfies safety requirement
- Can request for (3,3,0) by $\boldsymbol{P_4}$ be granted?
- Can request for (0,2,0) by $\boldsymbol{P_0}$ be granted?

Single Instance of Each Resource Type

- Maintain **wait-for** graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph



If there is a cycle, there exists a deadlock

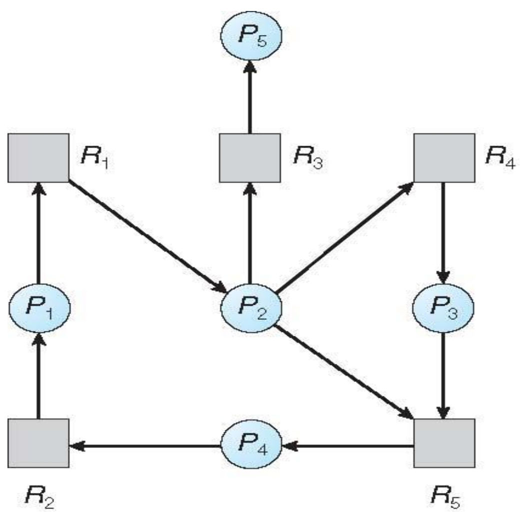


Resource-Allocation Graph

Corresponding wait-for graph

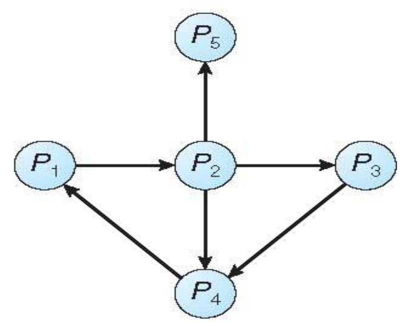
Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme



(a)

If there is a cycle, there exists a deadlock



(b)

Resource-Allocation Graph   Corresponding wait-for graph

Several Instances of a Resource Type

Let n = number of processes, and m = number of resources types.

- **Available***:* A vector of length ***m*** indicates the number of available resources of each type. If *Available*[*j*] equals *k*, then *k* instances of resource type ***Rj*** are available.
- **Allocation***:* An ***n* x *m*** matrix defines the number of resources of each type currently allocated to each process. If ***Allocation***[*i*][*j*] equals *k,* then process *Pi* is currently allocated ***k*** instances of resource type *Rj* .
- **Request***:* An ***n* x *m*** matrix indicates the current request of each process. If *Request* **[*i*][*j*] = *k***, then process P_i is requesting ***k*** instances of resource type R_j.

Repeat:

These data structures vary over time in both size and value.

To simplify the presentation in the algorithm, we establish some notation.

Let X and Y be vectors of length n. We say that $X \leqslant Y$ if and only if $X[i] \leqslant Y[i]$ for all $i = 1, 2, \ldots, n$. For example, if $X = (1,7,3,2)$ and $Y = (0,3,2,1)$, then $Y < X$.

Detection Algorithm

Let n = number of processes, and m = number of resources types.

Let ***Work*** and ***Finish*** be vectors of length ***m*** and ***n***, respectively Initialize:

(a) ***Work = Available*** /* indicates the number of available resources of each type*/

(b) For ***i* = 1,2, …, *n***, if $\boldsymbol{Allocation_i \neq 0}$, /* then process *Pi* is currently allocated instances of resource*/

then

Finish*[i] = *false; /* process *Pi is holding resources*/* otherwise, ***Finish*[i] = *true***

2. Find an index ***i*** such that both:

(a) *Finish*[*i*] == *false* /* *process Pi is currently is holding resources */*

(b) $\boldsymbol{Request_i \leq Work}$ /* available resources of each type is greater than or equal to the *requests */*

If no such ***i*** exists, go to step 4

3. $\boldsymbol{Work = Work + Allocation_i}$ /* ***Available +*** *process Pi's current allocated instances of resource*/*

Finish*[*i*] = *true

go to step 2

4. If ***Finish[i] == false***, for some ***i***, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if ***Finish*[*i*] == *false***, then P_i is deadlocked

Algorithm requires an order of O(m x n^2) operations to detect whether the system is in deadlocked state

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)

- Snapshot at time T_0:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in ***Finish[i] = true*** for all ***i***

① Availabe ⇒ Work[0,0,0]

All $\text{Allocation}_{0-4} \neq 0$. ⇒ $\underbrace{F[0], F[1], F[2], F[3], F[4]}$ = False

② Finish[0] = False

Req_0 [0,0,0] ≤ Work [0,0,0]

↓

work = [0,0,0] + [0,1,0] = [0,1,0]

Finish[0] = True

③ Finish[2] = False

Req_2 = [0,0,0] ≤ [0,1,0]

↓

work = [0,1,0] + [3,0,3] = [3,1,3]

Finish[2] = True

④ Finish[1] = False

Req_1 = [2,0,2] ≤ [3,1,3]

↓

work = [3,1,3] + [2,0,0] = [5,1,3]

Finish[1] = True

⑤ Finish[3] = False

Req_3 = [1,0,0] ≤ [5,1,3]

↓

work = [5,1,3] + [2,1,1] = [7,2,4]

Finish[3] = True

⑥ Finish[4] = False

Req = [0,0,2] ≤ [7,2,4]

↓

work = [7,2,4] + [0,0,2] = [7,2,6]

Finish[4] = True.

Example (Cont.)

- P_2 requests an additional instance of type ***C***

	Request
	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_{418}

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle

- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes "caused" the deadlock.