Scheduling

Contents

- Uniprocessor Scheduling: Types of Scheduling: Preemptive, Non-preemptive, Long-term, Medium-term, Short-term scheduling
- Scheduling Algorithms: FCFS, SJF, RR, Priority
- Multiprocessor Scheduling: Granularity
- Design Issues, Process Scheduling
- Deadlock: Principles of deadlock, Deadlock Avoidance
- Deadlock Detection, Deadlock Prevention
- Deadlock Recovery

Uniprocessor (CPU) Scheduling

- **The problem:** Scheduling the usage of a single processor among all the existing processes in the system
- **The goal is to achieve:**
  - High processor utilization
  - High throughput
    - number of processes completed per unit time
  - Low response time
    - time elapse from the submission of a request to the beginning of the response

Scheduling Objectives

The scheduling function should

- Share time fairly among processes
- Prevent starvation of a process
- Use the processor efficiently
- Have low overhead
- Prioritise processes when necessary (e.g. real time deadlines)

Processor Scheduling

- Aim is to assign processes to be executed by the processor in a way that meets system objectives, such as response time, throughput, and processor efficiency
- Broken down into three separate functions:

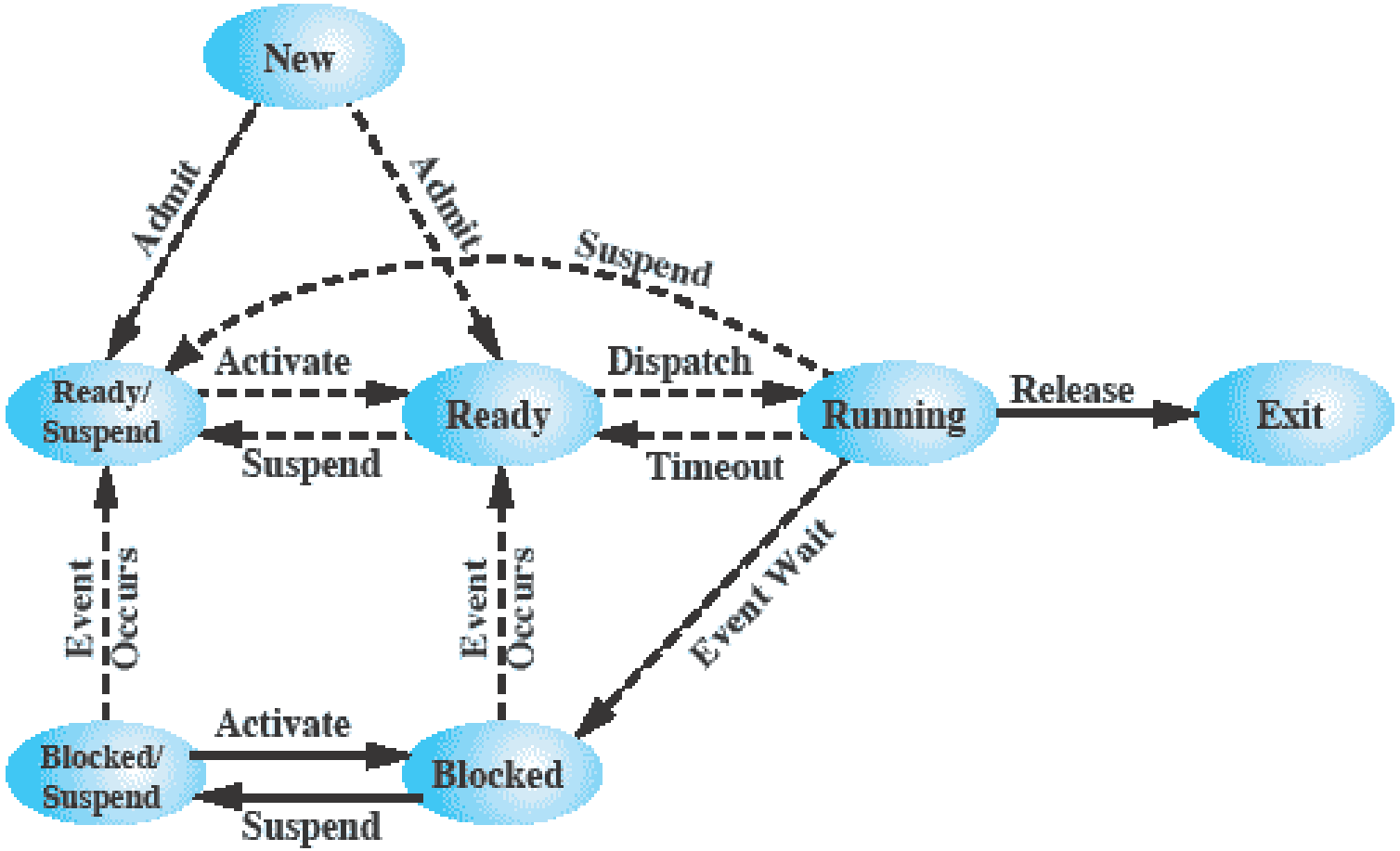
Types of scheduling

- In terms of in which phase is it applicable
- Long term scheduling
  - When a new process is created
  - Whether to add this to the set of processes who are currently active
- Medium term scheduling
  - Part of swapping function
  - Whether to add a process to those that are at least partially in main memory and therefore available for execution
- Short term scheduling
  - Actual decision as to which ready process to execute next

Types of Scheduling

Long-term scheduling	The decision to add to the pool of processes to be executed
Medium-term scheduling	The decision to add to the number of processes that are partially or fully in main memory
Short-term scheduling	The decision as to which available process will be executed by the processor
I/O scheduling	The decision as to which process's pending I/O request shall be handled by an available I/O device

Scheduling and Process State Transitions



(b) With Two Suspend States

Scheduling and Process State Transitions

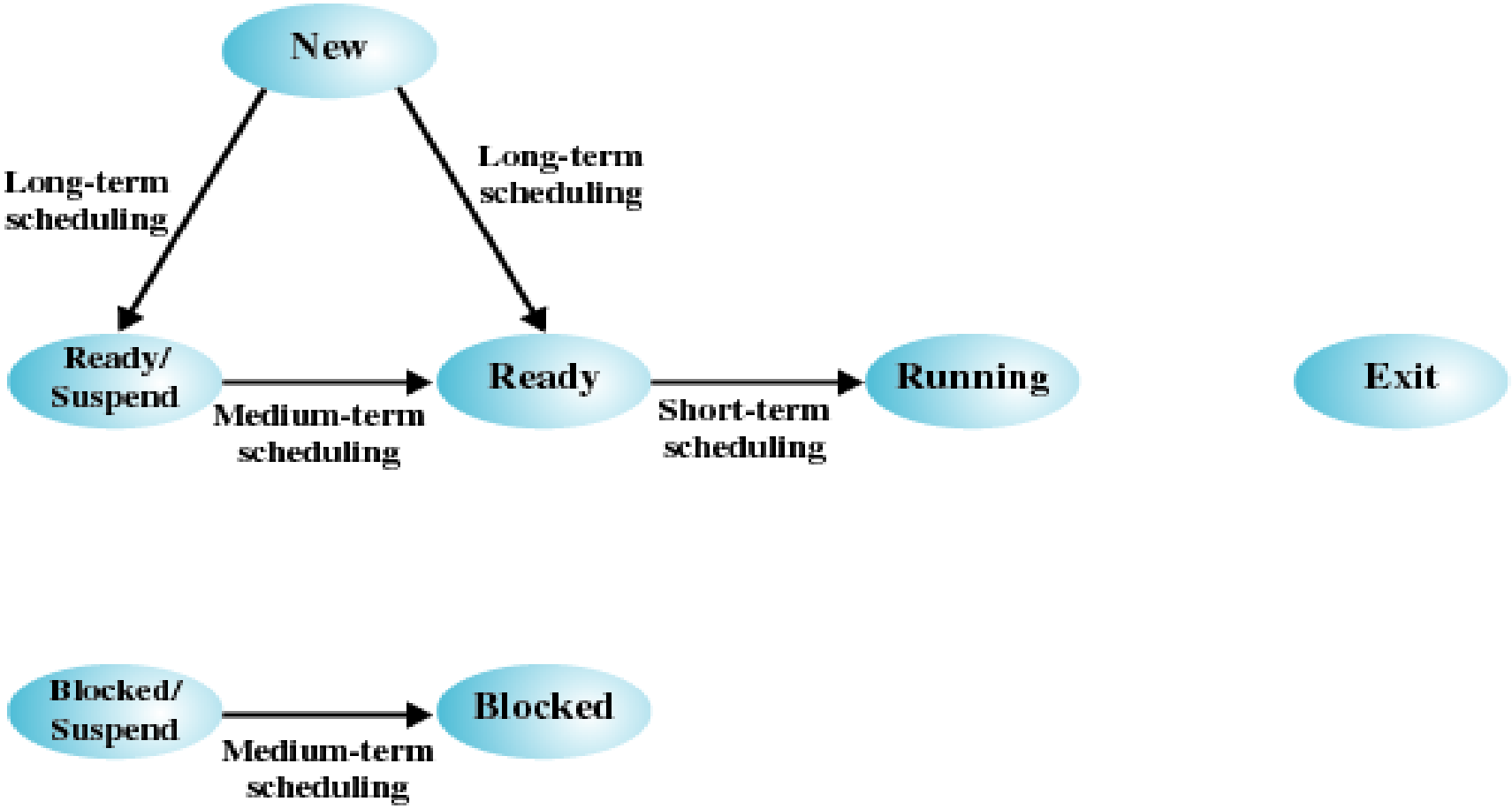
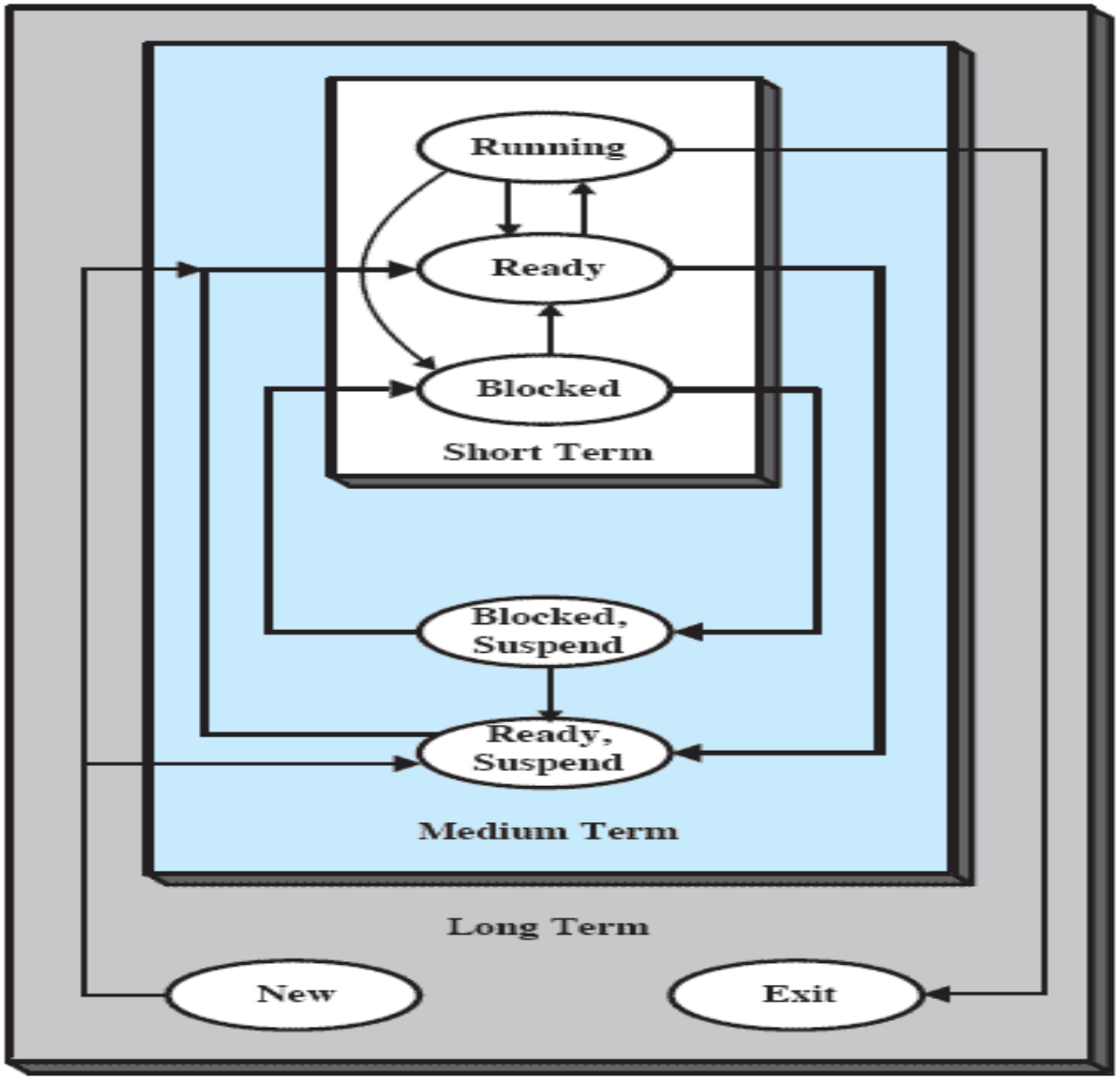


Figure 9.1 Scheduling and Process State Transitions

Nesting of Scheduling Functions



Another way to look at scheduling

- Scheduling determines which process will wait and which will progress
- Different queues are involved
- Scheduling is a matter of managing queues to minimize queuing delay and to optimize performance in a queuing environment

Queuing Diagram for Scheduling

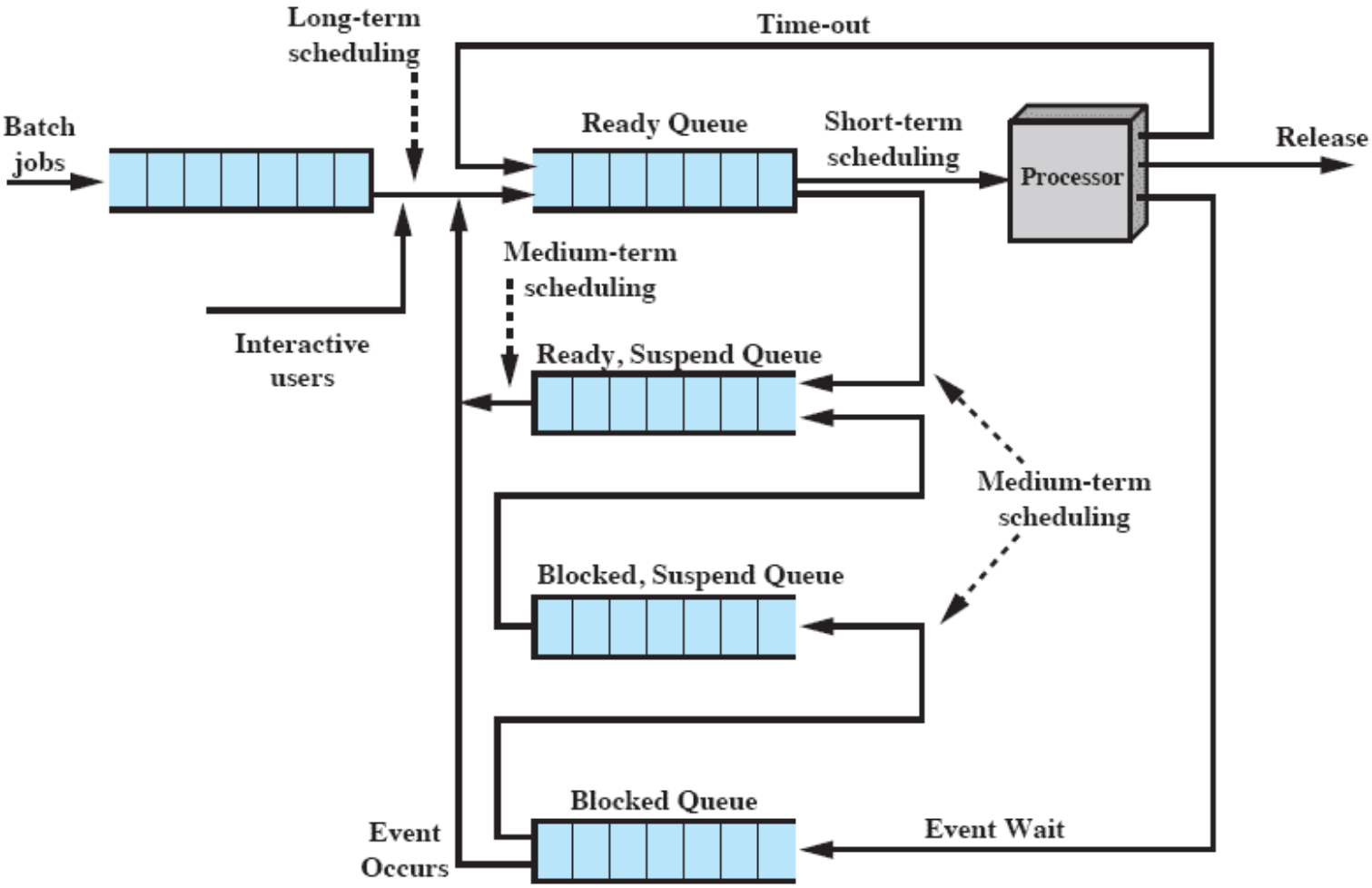


Figure 9.3 Queuing Diagram for Scheduling

Long-Term Scheduling: (Job Scheduler)

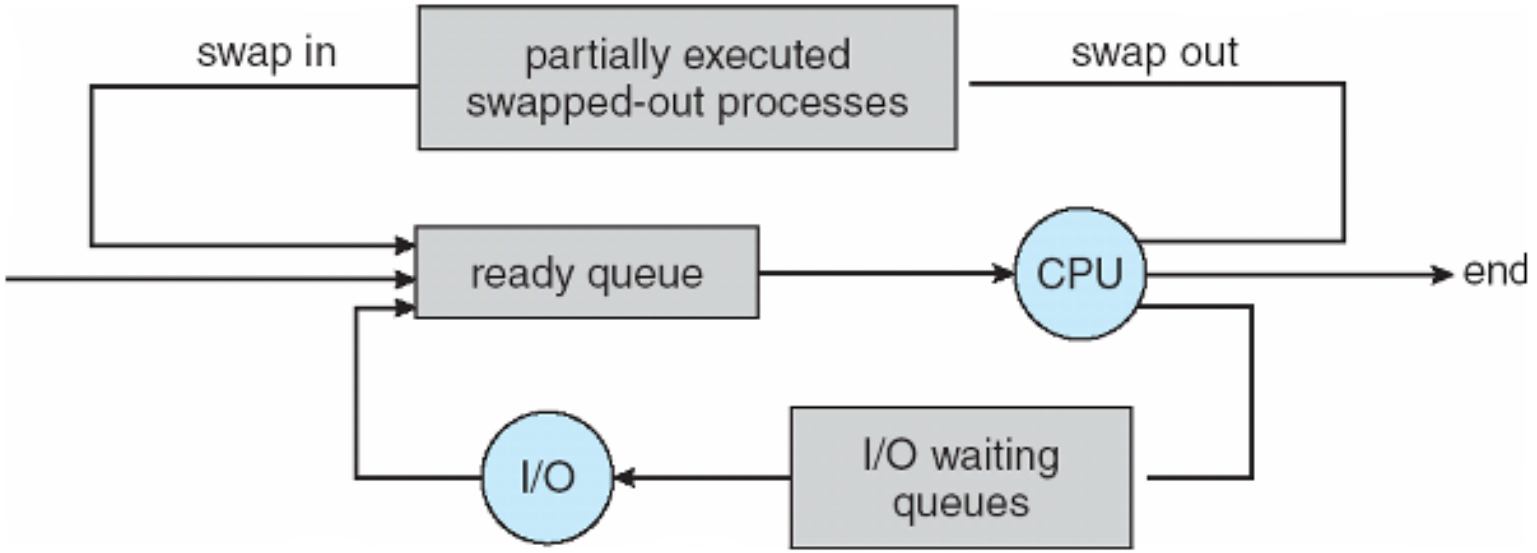
- Selects which processes should be brought into the ready queue
  - May be first-come-first-served
  - Or according to criteria such as priority, I/O requirements or expected execution time
- Controls the degree of multiprogramming
- If more processes are admitted
  - less likely that all processes will be blocked
  - better CPU usage
  - each process has less fraction of the CPU
- The long term scheduler will attempt to keep a mix of processor-bound and I/O-bound processes

Two kinds of processes

- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
    - Processes that are mostly waiting for the completion of input or output (I/O) are I/O Bound.
    - Interactive processes, such as office applications are mostly I/O bound the entire life of the process. Some processes may be I/O bound for only a few short periods of time.
    - The expected short run time of I/O bound processes means that they will not stay the running the process for very long.
    - They should be given high priority by the scheduler.
  - CPU-bound process – spends more time doing computations; few very long CPU bursts
    - CPU Bound processes are ones that are implementing algorithms with a large number of calculations.
    - They can be expected to hold the CPU for as long as the scheduler will allow.
    - Programs such as simulations may be CPU bound for most of the life of the process.
    - Users do not typically expect an immediate response from the computer when running CPU bound programs.
    - They should be given a lower priority by the scheduler.

Medium-Term Scheduling (Swapper)

- Part of the swapping function
- Swapping decisions based on the need to manage multiprogramming
- Done by memory management software



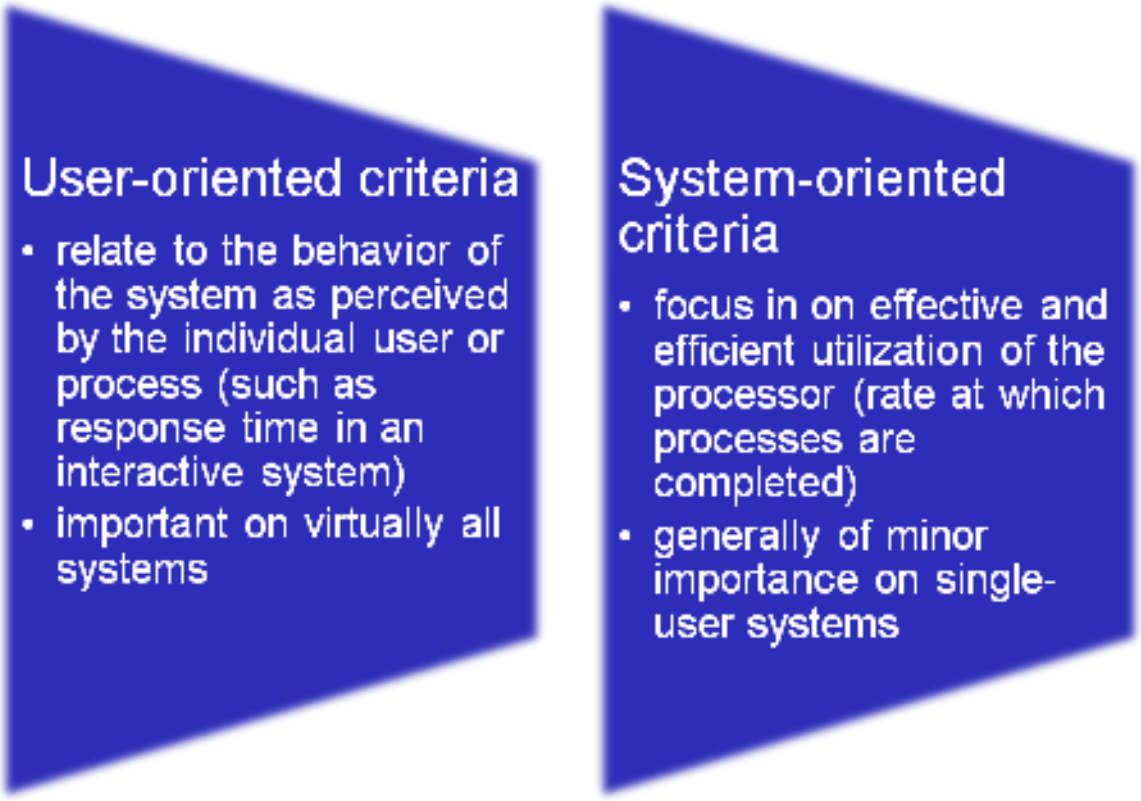
Short-Term Scheduling: (CPU Scheduler)

- Selects which process should be executed next and allocates CPU
- The short term scheduler is known as the dispatcher
- Executes most frequently
- Is invoked on a event that may lead to choose another process for execution:

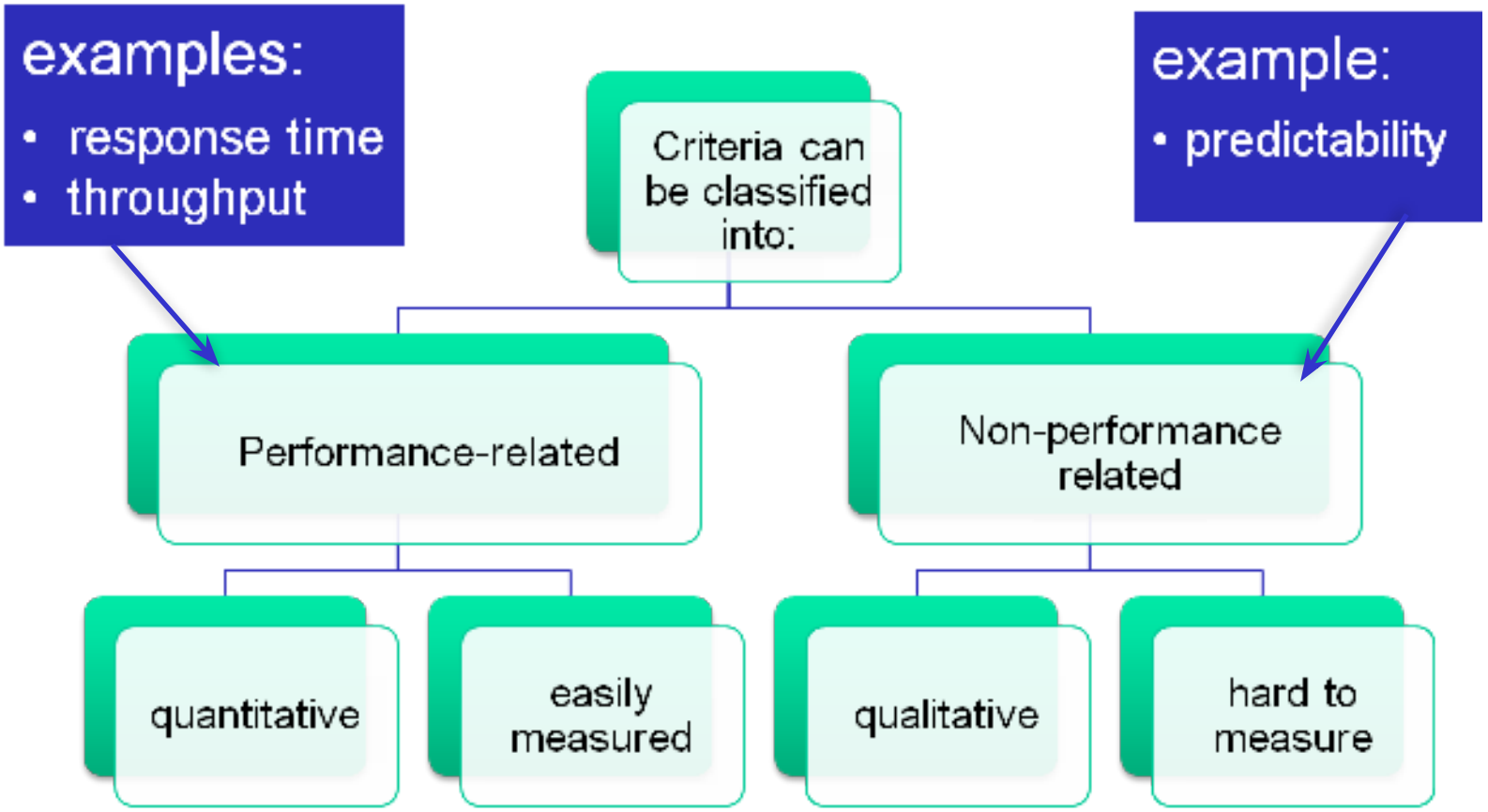
Examples:
• Clock interrupts • I/O interrupts • Operating system calls • Signals (e.g., semaphores)

Short Term Scheduling Criteria

- Main objective is to allocate processor time to optimize certain aspects of system behaviour
- A set of criteria is needed to evaluate the scheduling policy



Short-Term Scheduling Criteria: Performance



Short-Tem Scheduling Criteria

User-oriented: relate to the behaviour of the system as perceived by the individual user or process

- **Measure of user satisfaction**
- **Response time in case of interactive systems**
- **Turnaround Time**
- **Important on almost all systems**

System-oriented: focused on effective and efficient utilization of the processor.

- Measure of system performance
- Processor utilization: keep the CPU as busy as possible
- Fairness
- Throughput: number of process completed per unit time
- Waiting time – amount of time a process has been waiting in the ready queue
- Not that important on single user systems

Interdependent Scheduling Criteria

User Oriented, Performance Related

Turnaround time This is the interval of time between the submission of a process and its completion. Includes actual execution time plus time spent waiting for resources, including the processor. This is an appropriate measure for a batch job.

Response time For an interactive process, this is the time from the submission of a request until the response begins to be received. Often a process can begin producing some output to the user while continuing to process the request. Thus, this is a better measure than turnaround time from the user's point of view. The scheduling discipline should attempt to achieve low response time and to maximize the number of interactive users receiving acceptable response time.

Deadlines When process completion deadlines can be specified, the scheduling discipline should subordinate other goals to that of maximizing the percentage of deadlines met.

User Oriented, Other

Predictability A given job should run in about the same amount of time and at about the same cost regardless of the load on the system. A wide variation in response time or turnaround time is distracting to users. It may signal a wide swing in system workloads or the need for system tuning to cure instabilities.

Interdependent Scheduling Criteria

System Oriented, Performance Related

Throughput The scheduling policy should attempt to maximize the number of processes completed per unit of time. This is a measure of how much work is being performed. This clearly depends on the average length of a process but is also influenced by the scheduling policy, which may affect utilization.

Processor utilization This is the percentage of time that the processor is busy. For an expensive shared system, this is a significant criterion. In single-user systems and in some other systems, such as real-time systems, this criterion is less important than some of the others.

System Oriented, Other

Fairness In the absence of guidance from the user or other system-supplied guidance, processes should be treated the same, and no process should suffer starvation.

Enforcing priorities When processes are assigned priorities, the scheduling policy should favor higher-priority processes.

Balancing resources The scheduling policy should keep the resources of the system busy. Processes that will underutilize stressed resources should be favored. This criterion also involves medium-term and long-term scheduling.

Scheduling Algorithm Optimization Criteria

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

Selection Function

- Determines which Ready process is dispatched next
- May be based on priority, resource requirements, or the execution characteristics of the process
- If based on execution characteristics, some factors to consider are
  - ***w*** = time spent in system so far, waiting
  - ***e*** = time spent in execution so far
  - ***s*** = total service time required by the process, including *e*; (estimated by system or user)

For example, the selection function max[*w] indicates an FCFS discipline.*

Decision Mode

- When/under what circumstances is the selection function is exercised?
- Two categories:
  - Nonpreemptive
  - Preemptive

Nonpreemptive vs Preemptive

Nonpreemptive

- once a process is in the running state, it will continue until it terminates or blocks itself for I/O

Preemptive

- currently running process may be interrupted and moved to ready state by the OS
- preemption may occur when a new process arrives, on an interrupt, or periodically

Priorities

- Scheduler will always choose a process of higher priority over one of lower priority
- Have multiple ready queues to represent each level of priority
- Lower-priority may suffer starvation
  - Allow a process to change its priority based on its age or execution history

Processes have priorities

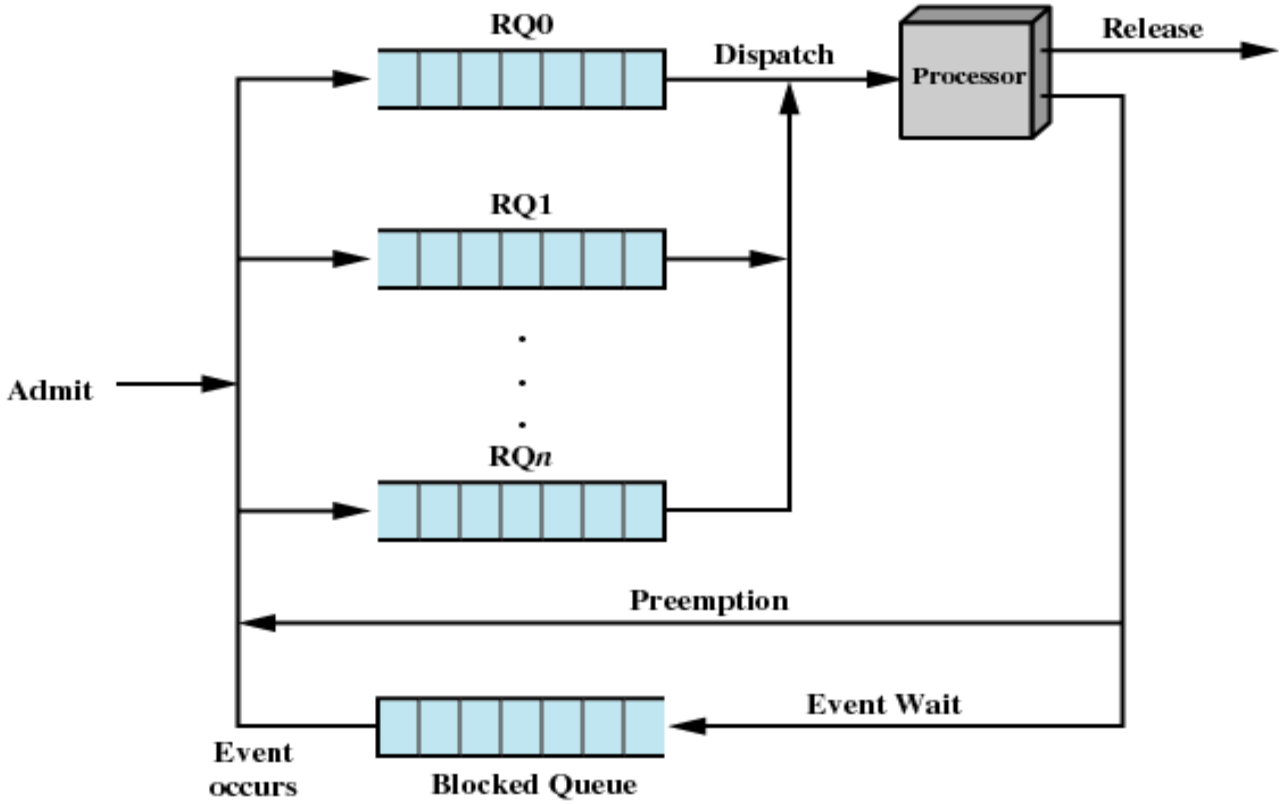


Figure 9.4 Priority Queuing

Contents

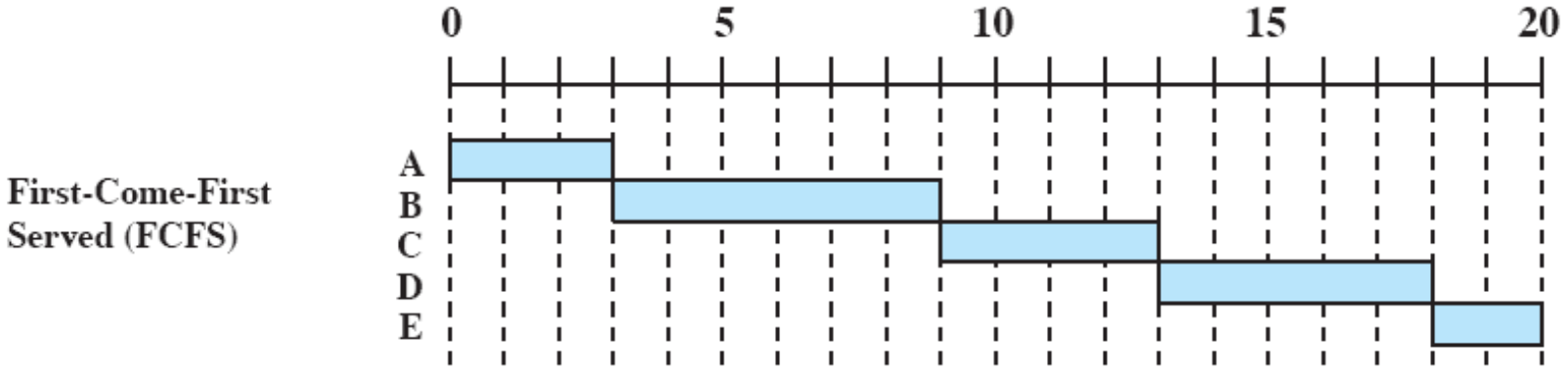
- Uniprocessor Scheduling: Types of Scheduling: Preemptive, Non-preemptive, Long-term, Medium-term, Short-term scheduling
- Scheduling Algorithms: FCFS, SJF, RR, Priority
- Multiprocessor Scheduling: Granularity
- Design Issues, Process Scheduling
- Deadlock: Principles of deadlock, Deadlock Avoidance
- Deadlock Detection, Deadlock Prevention
- Deadlock Recovery

Running example to discuss various scheduling policies

Table 9.4 Process Scheduling Example

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

First Come First Served (FCFS)



- The simplest scheduling policy is first-come-first-served (FCFS),first-in-first-out (FIFO) or a strict queuing scheme.
- As each process becomes ready, it joins the ready queue.
- When the currently running process ceases to execute, the process that has been in the ready queue the longest is selected for running.
- Selection function: the process that has been waiting the longest in the ready queue (hence, FCFS)
- Decision mode: nonpreemptive, a process runs until it blocks itself

FCFS Drawbacks

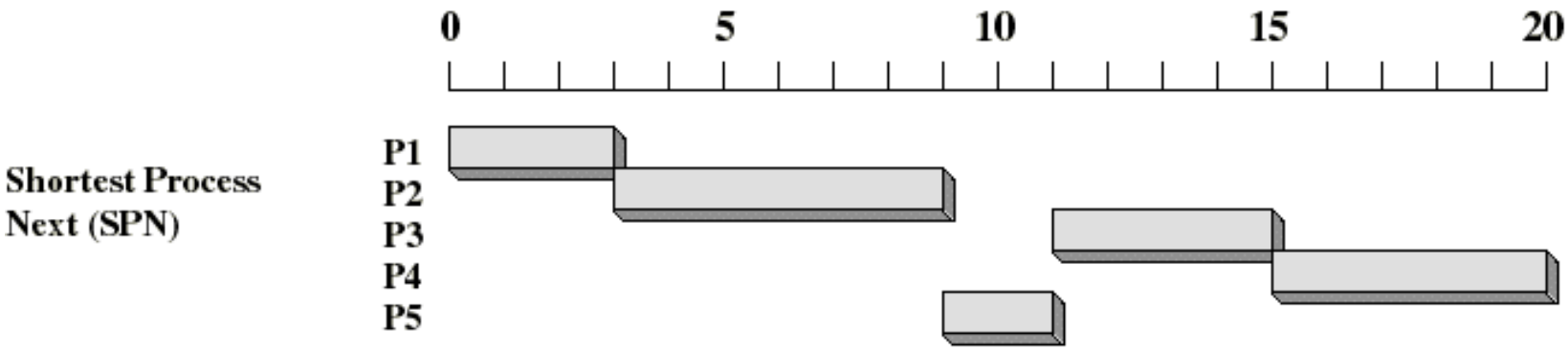
- A short process may have to wait a very long time before it can execute.

- A process that does not perform any I/O will monopolize the processor.
- Favours CPU-bound processes
  - I/O-bound processes have to wait until CPU-bound process completes
  - They may have to wait even when their I/O are completed (poor device utilization)
  - we could have kept the I/O devices busy by giving more priority to I/O Bound processes.

FCFS Drawbacks

- FCFS is not an attractive alternative on its own for a Uniprocessor system.
- However, it is often combined with a priority scheme to provide an effective scheduler.
- Thus, the scheduler may maintain a number of queues, one for each priority level, and dispatch within each queue on a first-come-first-served basis.

Shortest Process Next (SPN) (SJF-Nonpreemptive)



- Selection function: the process with the shortest expected CPU burst time
- Decision mode: **nonpreemptive**
- I/O bound processes will be picked first
- We need to estimate the required processing time (CPU burst time) for each process

- Average waiting time will be minimum.

Shortest Process Next: Critique

- Average waiting time will be minimum.
- Although it is optimal ,can not be implemented at the level of short term CPU scheduling
- As there is no way to know the length of the next cpu burst time.

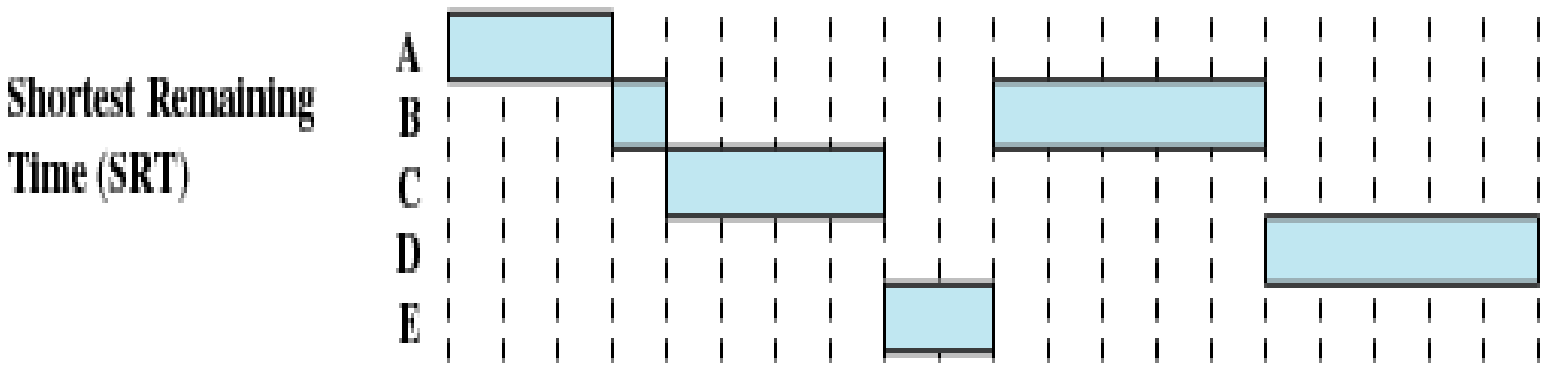
Shortest Process Next: Critique

- Possibility of starvation for longer processes as long as there is a steady supply of shorter processes
- Lack of pre-emption is not suited in a time sharing environment
  - CPU bound process gets lower priority (as it should) but a process doing no I/O could still monopolize the CPU if it is the first one to enter the system
- SPN implicitly incorporates priorities: shortest jobs are given preferences
- The next (pre-emptive) algorithm penalizes directly longer jobs

Shortest Process Next: Critique

- Average waiting time will be minimum
- FCFS is used to break the tie if two processes have same burst time
- SJF is a special case of general priority scheduling
- Priority is the inverse of the next cpu burst
- Larger the cpu burst, lower the priority & vice versa.

Shortest Remaining Time(SJF-Pre-emptive)



- Pre-emptive version of shortest process next policy
- Must estimate processing time

SRT / SJF P

- Estimate of remaining time required here as well
- Risk of starvation of longer processes
- No bias in favor of long processes as with FCFS
- No additional interrupts are generated unlike RR thereby reducing overhead
- But elapsed service times must be recorded, increasing the overhead
- Short job is given preference over running longer job, thus better turn around time

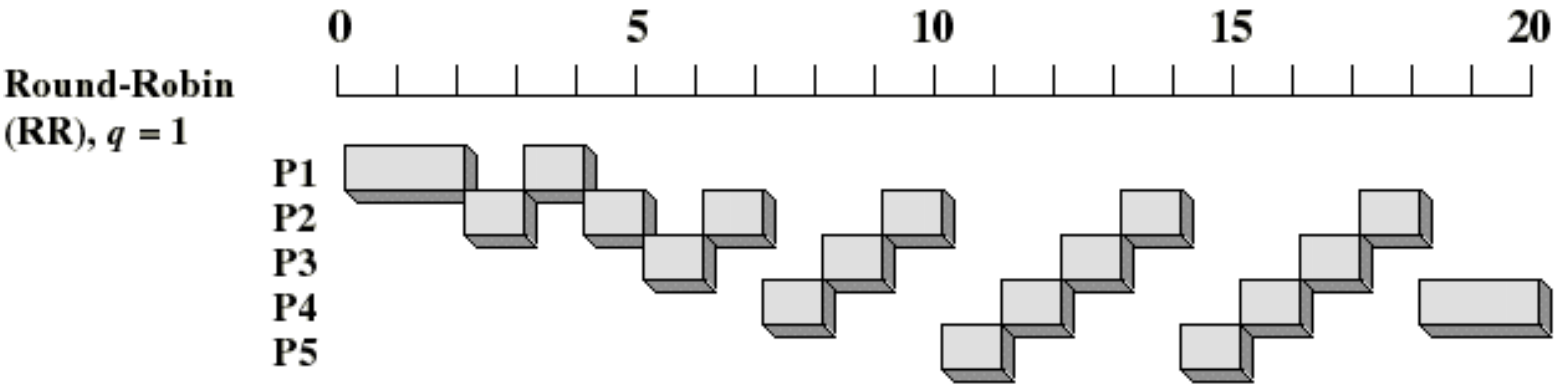
Priority Scheduling

- A priority number (integer) is associated with each process.
- The CPU is allocated to the process with the highest priority
- Some systems have a high number represent high priority.
- Other systems have a low number represent high priority.
- Text uses a low number to represent high priority.
- Priority scheduling may be preemptive or nonpreemptive.

Assigning Priorities

- SJF is a priority scheduling where priority is the predicted next CPU burst time.
- Other bases for assigning priority:
    - Memory requirements
    - Number of open files
    - Avg I/O burst / Avg CPU burst
    - External requirements (amount of money paid, political factors, etc).
- Range of priority 0 to 7 or 0 to 4095
- Priorities can be defined either internally or externally
- Internal priority: time limit, no. of open files
- External priority: set by criteria outside the os e.g. imp of processes
- Problem: Starvation -- low priority processes may never execute.
- Solution: Aging -- as time progresses increase the priority of the process.

Round-Robin



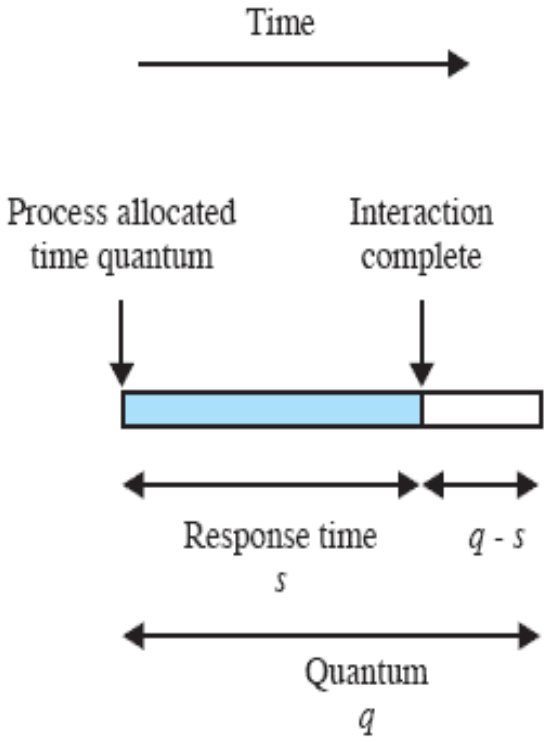
- Selection function: same as FCFS
- Decision mode: preemptive
  - a process is allowed to run until the time slice period (quantum, typically from 10 to 100 ms) has expired
  - then a clock interrupt occurs and the running process is put on the ready queue

Round-Robin

- Clock interrupt is generated at periodic intervals
- When an interrupt occurs, the currently running process is placed in the ready queue
- Next ready job is selected.
- The principal design issue is the length of the time quantum, or slice, to be used.
- If the quantum is very short, then short processes will move through the system relatively quickly.
    - BUT there is processing over-head involved in handling the clock interrupt and performing the scheduling and dispatching function.
    - Thus, very short time quantum should be avoided.

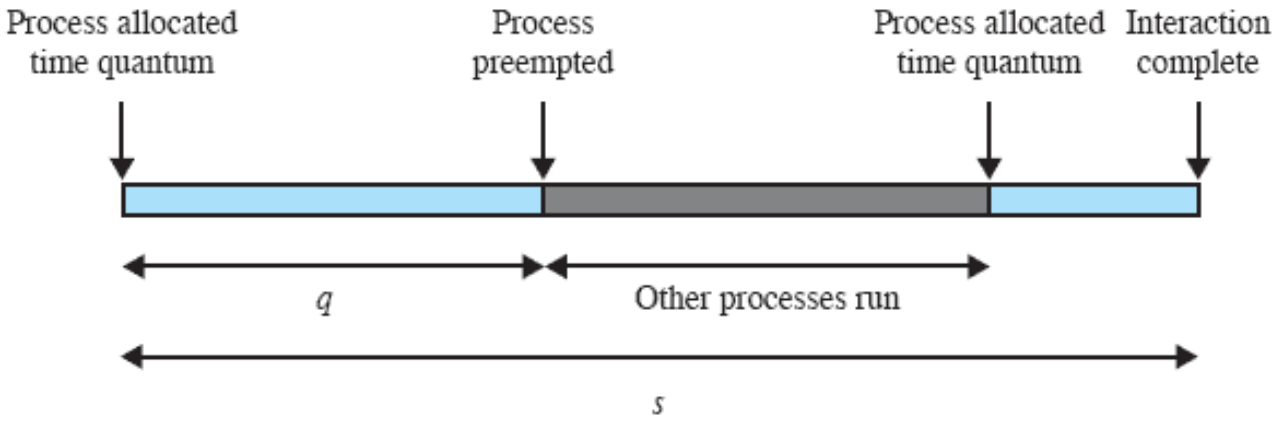
Effect of Size of Preemption Time Quantum

One useful guide is that the time quantum should be slightly greater than the time required for a typical interaction or process function. If it is less, then most processes will require at least two time quanta.



(a) Time quantum greater than typical interaction

Effect of Size of Preemption Time Quantum



(b) Time quantum less than typical interaction

Figure 9.6 Effect of Size of Preemption Time Quantum

Time Quantum for Round Robin

- Must be substantially larger than the time required to handle the clock interrupt and dispatching
- Should be larger than the typical interaction (but not much more to avoid penalizing I/O bound processes)

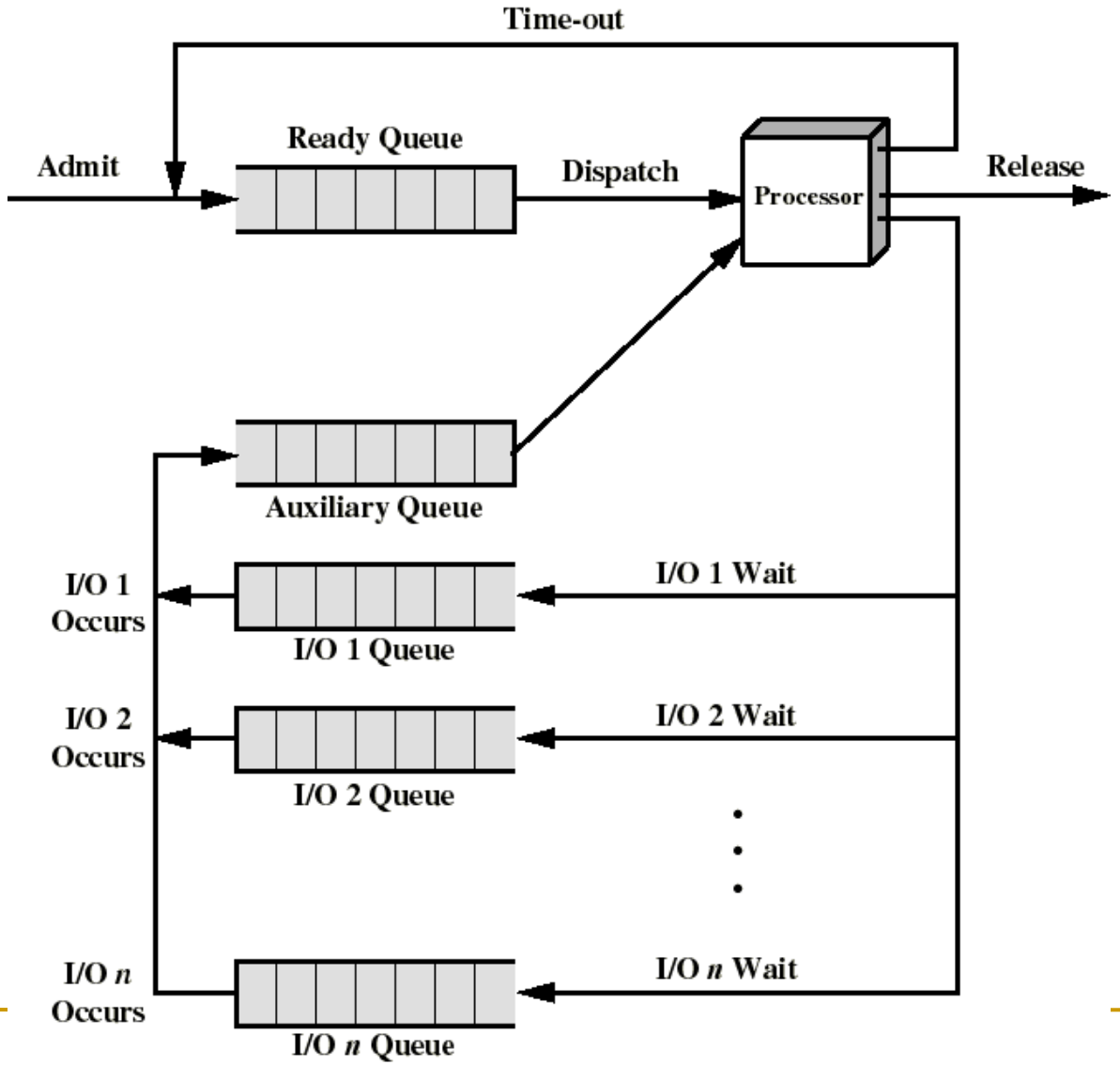
Round Robin: Critique

- Still favours CPU-bound processes
  - A I/O bound process uses the CPU for a time less than the time quantum, then is blocked waiting for I/O
  - A CPU-bound process runs for all its time slice and is put back into the ready queue (thus, getting in front of blocked processes)

Virtual Round Robin

- A solution: virtual round robin
  - When a I/O has completed, the blocked process is moved to an auxiliary queue, which gets preference over the main ready queue
  - A process dispatched from the auxiliary queue runs no longer than the basic time quantum minus the time spent running, since it was selected from the ready queue
  - Performance studies indicate that this performs better than regular RR

Queuing for Virtual Round Robin



Selection Function

- Determines which Ready process is dispatched next
- May be based on priority, resource requirements, or the execution characteristics of the process
- If based on execution characteristics, some factors to consider are
  - ***w*** = time spent in system so far, waiting
  - ***e*** = time spent in execution so far
  - ***s*** = total service time required by the process, including *e*; (estimated by system or user)

Table 9.3 Characteristics of Various Scheduling Policies

	FCFS	Round Robin	SPN	SRT
Selection Function	max[w]	constant	min[s]	min[$s - e$]
Decision Mode	Non-preemptive	Preemptive (at time quantum)	Non-preemptive	Preemptive (at arrival)
Throughput	Not emphasized	May be low if quantum is too small	High	High
Response Time	May be high, especially if there is a large variance in process execution times	Provides good response time for short processes	Provides good response time for short processes	Provides good response time
Overhead	Minimum	Minimum	Can be high	Can be high
Effect on Processes	Penalizes short processes; penalizes I/O bound processes	Fair treatment	Penalizes long processes	Penalizes long processes
Starvation	No	No	Possible	Possible

First-Come, First-Served (FCFS) Scheduling

Process that requests the CPU first is allocated the CPU first.

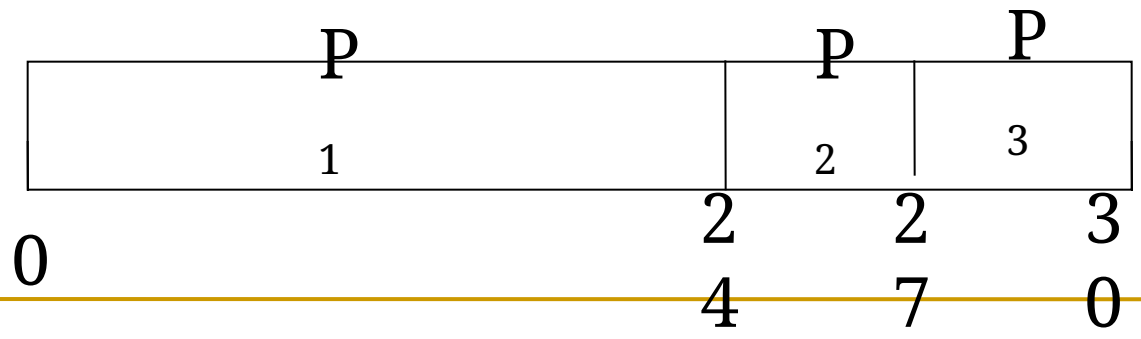
Easily managed with a FIFO queue.

Often the average waiting time is long.

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

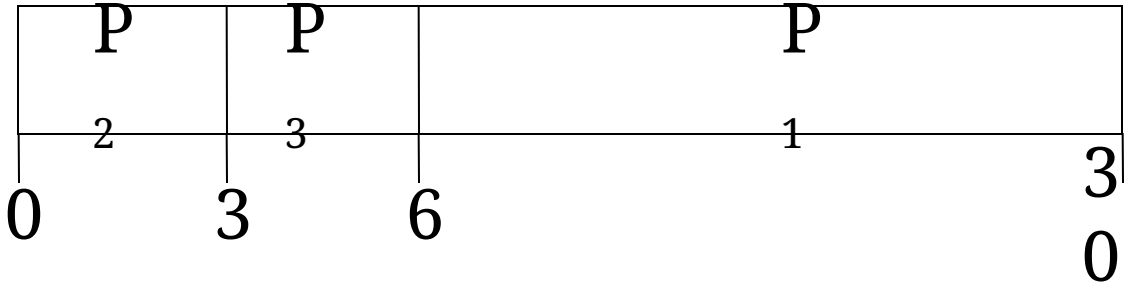
The Gantt Chart for the schedule is:



Suppose that the processes arrive in the order

P_2 , P_3 , P_1 .

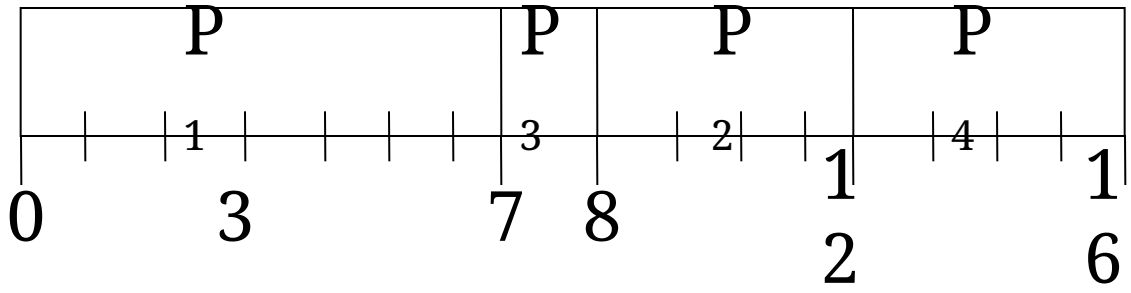
The Gantt chart for the schedule is:



Example of Non-Preemptive SJF

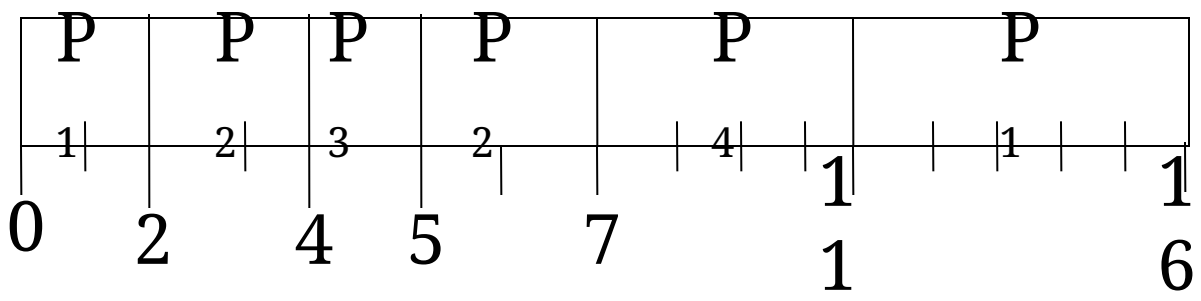
Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

SJF (non-preemptive)



Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4



Practice

Process	Arrival Time	Burst Time	Priority
A	0	1	3
B	1	9	3
C	3	1	2
D	3	9	1

Draw Gantt Chart to find finish times and also calculate TT and WT in case of each of the following strategies:

- FCFS
- SJF Non Preemptive
- SJF Preemptive
- Round Robin
- Priority Non preemptive

Contents

- Uniprocessor Scheduling: Types of Scheduling: Preemptive, Non-preemptive, Long-term, Medium-term,
  Short-term scheduling
- Scheduling Algorithms: FCFS, SJF, RR, Priority
- Multiprocessor Scheduling: Granularity
- Design Issues, Process Scheduling
- Deadlock: Principles of deadlock, Deadlock Avoidance
- Deadlock Detection, Deadlock Prevention
- Deadlock Recovery
-

Multiprocessor Scheduling

Introduction

- When a computer system contains more than a single processor, several new issues are introduced into the design of scheduling functions.
- We will examine these issues and the details of scheduling algorithms for tightly coupled multi-processor systems.

Classifications of Multiprocessor Systems

1. Loosely coupled, distributed multiprocessors, or clusters

- Fairly autonomous systems.
- Each processor has its own memory and I/O channels

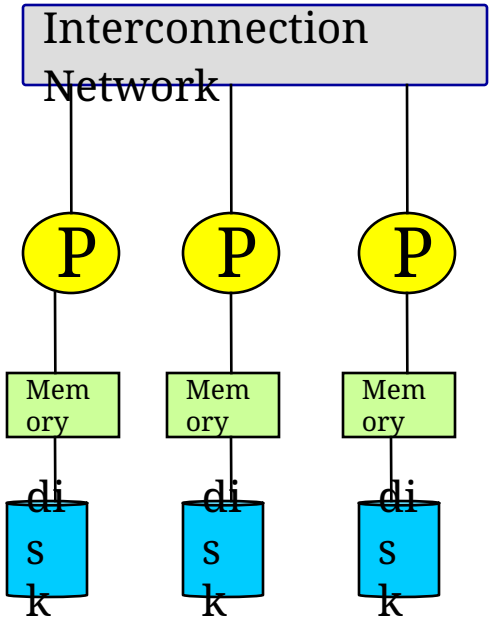
2. Functionally specialized processors

- Typically, specialized processors are controlled by a master(general-purpose processor) and provide services to it. An example would be an I/O processor, graphics processor

3. Tightly coupled multiprocessing

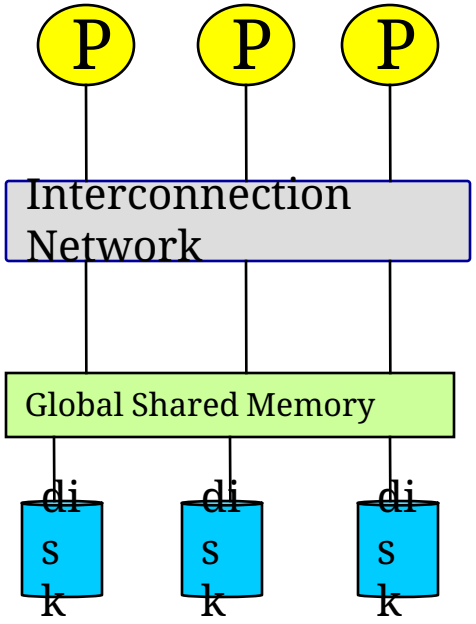
- Consists of a set of processors that share a common main memory and are under the integrated control of an operating system.
- We'll be most concerned with this group.

Multiprocessor Systems



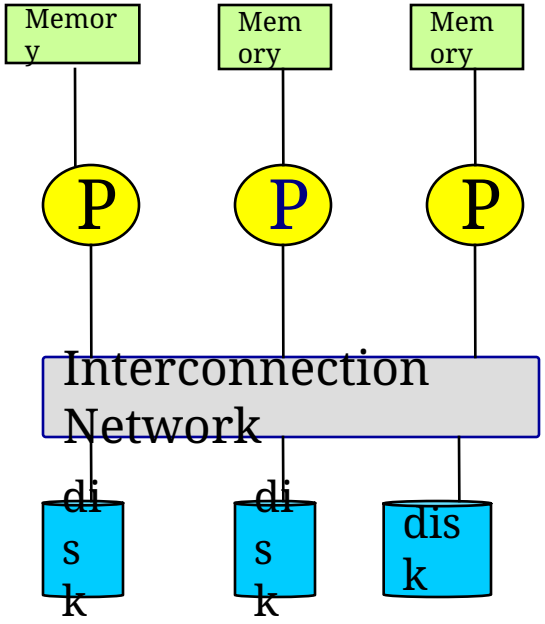
Shared Nothing

(Loosely Coupled)



Shared Memory

(Tightly Coupled)



Shared Disk

Parallel computing

- **Parallel computing** is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently ("in parallel").
- There are several different forms of parallel computing: bit-level, instruction level, data, and task parallelism.
- Task parallelisms is the characteristic of a parallel program that "entirely different calculations can be performed on either the same or different sets of data“.
- Task parallelism involves the decomposition of a task into sub-tasks and then allocating each sub-task to a processor for execution. The processors would then execute these sub-tasks simultaneously and often cooperatively.

Granularity

- A good metric for characterizing multiprocessors and placing them in context with other architectures is to consider the synchronization granularity,
- Or frequency of synchronization, between processes in a system.
- Five categories, differing in granularity:

1. Independent Parallelism
2. Coarse Parallelism
3. Very Coarse-Grained Parallelism
4. Medium-Grained Parallelism
5. Fine-Grained Parallelism

Independent Parallelism

- No explicit synchronization among processes
- Each represents a separate, independent application or job.
  - Multiple unrelated processes
- For e.g. in time sharing system
  - Each user is performing a particular application such as word processor or spreadsheet
- The multiprocessor provides the same service as a multiprogrammed uniprocessor.
  - Only difference is that since more processors are there, avg response time to the user will be less
- As if each user is on a separate machine, as good as a distributed system – still multi processor is more cost effective

Synchronization interval (instruction) :N/A

Coarse and Very Coarse-Grained Parallelism

- Synchronization among processes is there but at a very gross level.
- It is a set of concurrent processes running on a multi-programmed uni-processor and can be supported on a multiprocessor with little or no change to user software.
- In general, any collection of concurrent processes that need to communicate or synchronize can benefit from the use of a multiprocessor architecture.
- In the case of very infrequent interaction among the processes, a distributed system can provide good support.
- However, if the interaction is somewhat more frequent, then the overhead of communication across the network may negate some of the potential speedup. In that case, the multiprocessor organization provides the most effective support.

Medium-Grained Parallelism

- A single application can be effectively implemented as a collection of threads within a single process.
- In this case, the programmer must explicitly specify the potential parallelism of an application.
- Threads usually interact frequently, affecting the performance of the entire application

Synchronization interval (instruction) :20 -200

Fine Grained Parallelism

Highly parallel applications:

Fine-grained parallelism represents a much more complex use of parallelism than is found in the use of threads.

Synchronization interval (instruction)< 20

Parallelism

- Applications are often classified according to how often their subtasks need to synchronize or communicate with each other.
- An application exhibits fine-grained parallelism if its subtasks must communicate many times per second;
- It exhibits coarse-grained parallelism if they do not communicate many times per second,
- And it is embarrassingly parallel if they rarely or never have to communicate. Embarrassingly parallel applications are considered the easiest to parallelize.

Synchronization Granularity and Processes

Table 10.1 Synchronization Granularity and Processes

Grain Size	Description	Synchronization Interval (Instructions)
Fine	Parallelism inherent in a single instruction stream.	<20
Medium	Parallel processing or multitasking within a single application	20-200
Coarse	Multiprocessing of concurrent processes in a multiprogramming environment	200-2000
Very Coarse	Distributed processing across network nodes to form a single computing environment	2000-1M
Independent	Multiple unrelated processes	not applicable

Design Issues

Scheduling on a multiprocessor involves three interrelated issues:

- Assignment of processes to processors.
- Use of multiprogramming on individual processors.
- Actual dispatching of a process.

Assignment of processes to processors

- If we assume that the architecture of the multiprocessor is uniform, in the sense that no processor has a particular physical advantage with respect to access to main memory or I/O devices, then the simplest scheduling protocol is to treat processors as a pooled resource and assign processes to processors on demand.

- The question then arises as to whether the assignment should be static or dynamic?

Assignment of processes to processors

- If a process is permanently assigned (static assignment) to a processor from activation until completion, then a dedicated short-term queue is maintained for each processor.
- Allows for group or gang scheduling (details later).
- Advantage: less overhead – processor assignment occurs only once.
- Disadvantage: One processor could be idle (has an empty queue) while another processor has a backlog. To prevent this situation from arising, a common queue can be utilized. In this case, all processes go into one global queue and are scheduled to any available processor. Thus, over the life of a process, it may be executed on several different processors at different times.
- Dynamic load balancing in linux,thread moves from queue of one processor to queue of another processor.

Assignment of processes to processors

- Regardless of whether processes are dedicated to processors, some mechanism is needed to assign processes to processors.
- Two approaches have been used: master/slave and peer.

Master/slave architecture

- Key kernel functions always run on a particular processor. The other processors can only execute user programs.
- Master is responsible for scheduling jobs.
- Slave sends service request to the master.
- Advantages
  - Simple, requires little enhancement to a uniprocessor multiprogramming OS.
  - Conflict resolution is simple since one processor has control of all memory and I/O resources.
- Disadvantages
  - Failure of the master brings down whole system
  - Master can become a performance bottleneck

Assignment of processes to processors

2. Peer architecture
   - The OS kernel can execute on any processor.
   - Each processor does self-scheduling from the pool of available processes.
   - Advantages:
     - All processors are equivalent.
     - No one processor should become a bottleneck in the system.
   - Disadvantage:
     - Complicates the operating system
     - The OS must make sure that two processors do not choose the same process and that the processes are not somehow lost from the queue.
     - Techniques must be employed to resolve and synchronize competing claims for resources.

Multiprogramming at each processor

- After static allocation of process to the processor, should that processor be multiprogrammed?
- Completion time and other application-related performance metrics are much more important than processor utilization in multi-processor environment.
- For example, a multi-threaded application may require all its threads be assigned to different processors for good performance.
- Static or dynamic allocation of processes.

Process dispatching

- After assignment, deciding who is selected from among the pool of waiting processes --- process dispatching.
- Single processor multiprogramming strategies may be counter-productive here.
- Priorities and process history may not be sufficient.

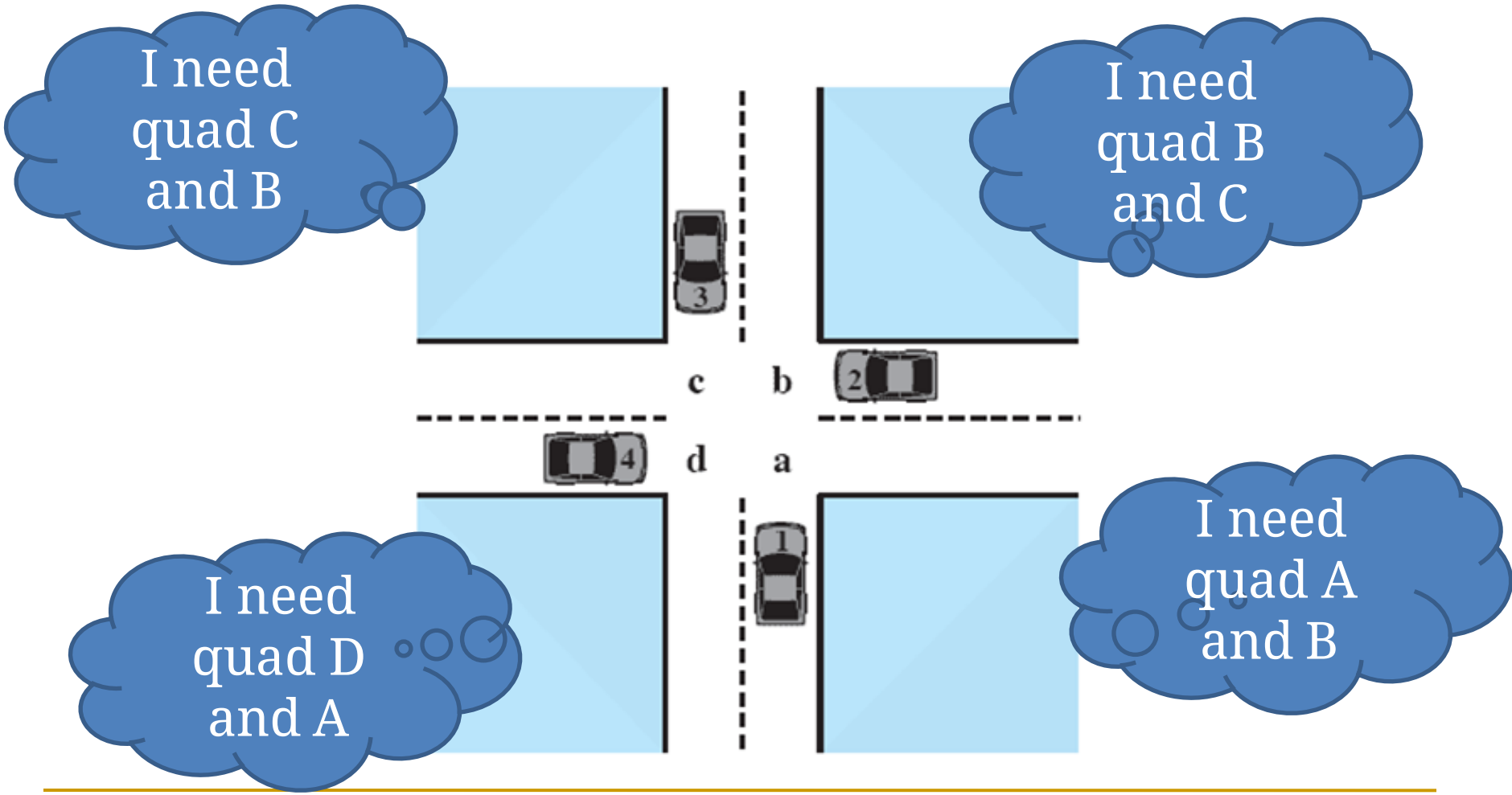
Process scheduling

- Single queue of processes or if multiple priority is used, multiple priority queues, all feeding into a common pool of processors.
- Specific scheduling policy does not have much effect as the processor number increases.
- Conclusion: Use FCFS with priority levels.

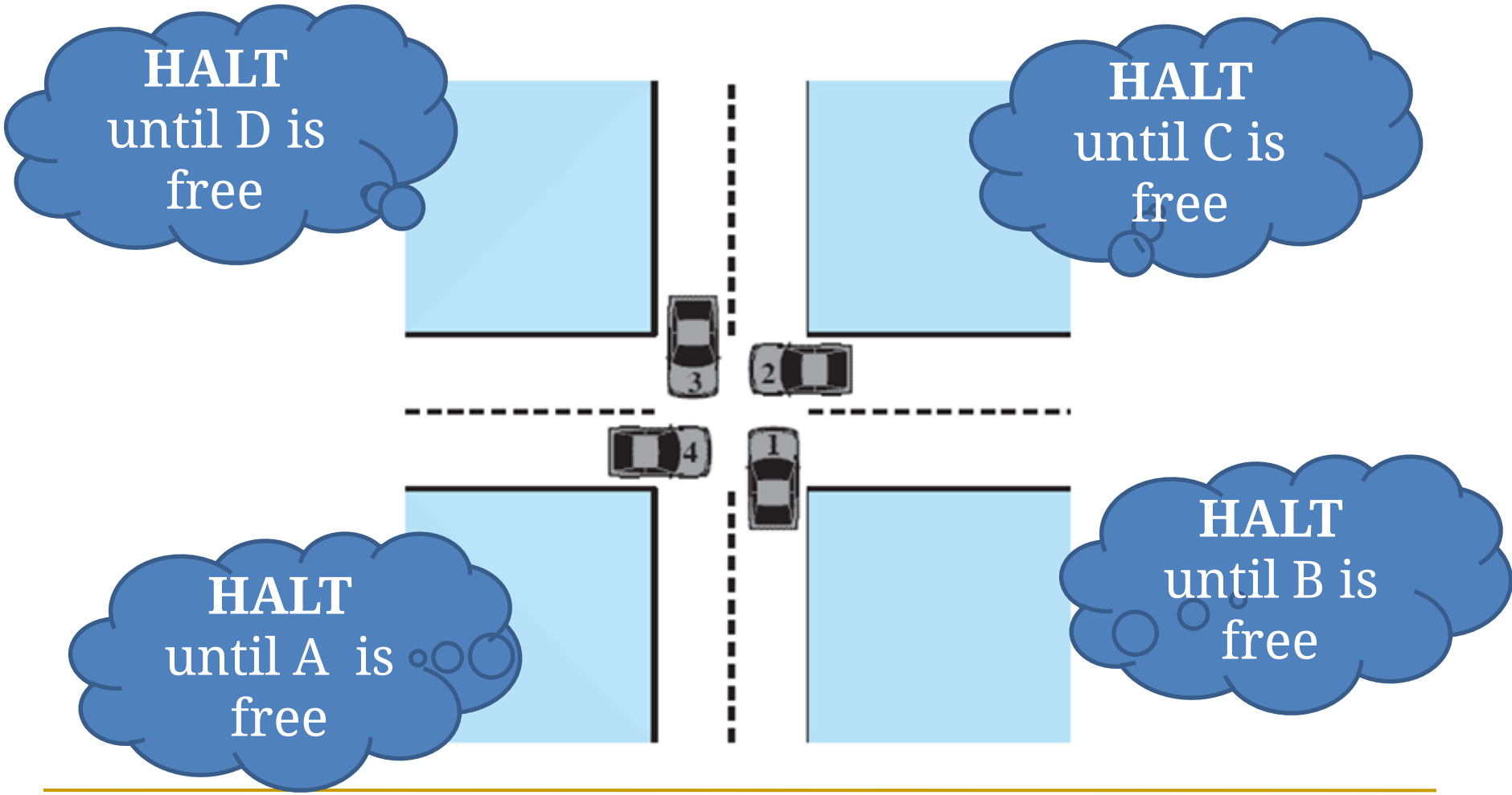
Contents

- Uniprocessor Scheduling: Types of Scheduling: Preemptive, Non-preemptive, Long-term, Medium-term, Short-term scheduling
- Scheduling Algorithms: FCFS, SJF, RR, Priority
- Multiprocessor Scheduling: Granularity
- Design Issues, Process Scheduling
- Deadlock: Principles of deadlock, Deadlock Prevention
- Deadlock Avoidance (Operating system concepts by Galvin ,Gagne)
- Deadlock Detection, Deadlock Recovery

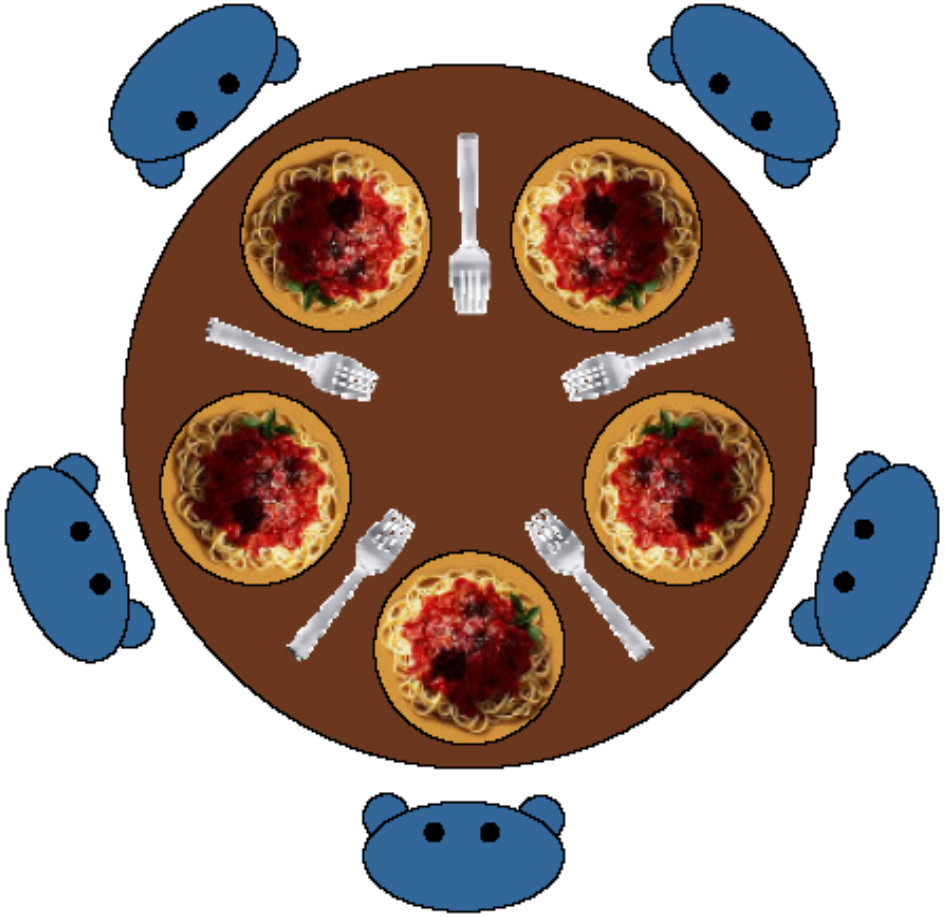
Potential Deadlock



Actual Deadlock



Example of deadlock



Dining Philosophers Problem

Example of deadlock

Boss said to secretary: For a week we will go abroad, so make arrangement.
Secretary make call to Husband: For a week my boss and I will be going abroad, you look after yourself.
Husband make call to secret lover: My wife is going abroad for a week, so lets spend the week together.
Secret lover make call to small boy whom she is giving private tution: I have work for a week, so you need not come for class.
Small boy make call to his grandfather: Grandpa, for a week I don't have class 'coz my teacher is busy. Lets spend the week together.
Grandpa make call to his secretary: This week I am spending my time with my grandson. We cannot attend that meeting.
Secretary make call to her husband: This week my boss has some work, we cancelled our trip.
Husband make call to secret lover: We cannot spend this week together, my wife has cancelled her trip.
Secret lover make call to small boy whom she is giving private tution: This week we will have class as usual.
Small boy make call to his grandfather: Grandpa, my teacher said this week I have to attend class.
Sorry I can't give you company.
Grandpa make call to his secretary: Don't worry this week we will attend that meeting, so make arrangement .

DEADLOCKS

- Permanent blocking of a single or set of processes, competing for system resources or may want to cooperate for communication.
- Formal definition :
  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.
- Usually the event is release of a currently held resource.
- Generally it is because of the conflicting needs of different processes.
- There is no general solution to solve it completely.

Resource Categories

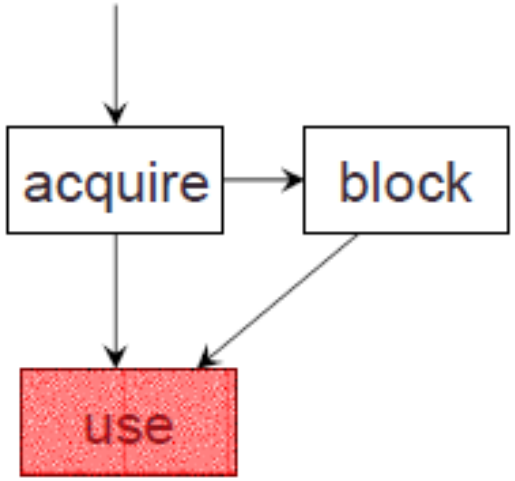
- Reusable resources vs. Consumable resources
- Physical vs. logical resources
- Preemptable resources vs. Non preemptable resources

Reusable Resources

- Can be safely used by only one process at a time and is not depleted by that use
- Processors, I/O channels, memory, devices and data structure such as database, files,semaphores etc.
- Examples of deadlock with reusable resources
  - If each process holds on resource and requests for the other
  - Dining Philosophers
- General access pattern:
  - Request
  - Lock
  - Use
  - Release

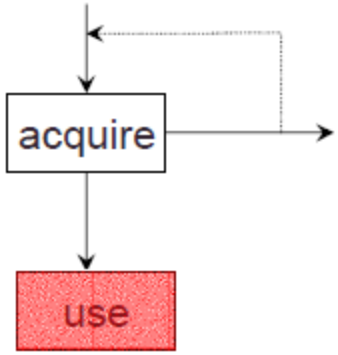
Resources

- Process must wait if request is denied
  - ❑ Requesting process may be blocked
  - ❑ May fail with error code



- Deadlocks

❑ Occur only when processes are granted exclusive access to resources



Example of accessing Disk file and Tape drive

Process P

Step	Action
p_0	Request (D)
p_1	Lock (D)
p_2	Request (T)
p_3	Lock (T)
p_4	Perform function
p_5	Unlock (D)
p_6	Unlock (T)

Process Q

Step	Action
q_0	Request (T)
q_1	Lock (T)
q_2	Request (D)
q_3	Lock (D)
q_4	Perform function
q_5	Unlock (T)
q_6	Unlock (D)

Consider sequence p0 p1 q0 q1 p2 q2

Another Example

- Space is available for allocation of 200Kbytes, and the following sequence of events occur

P1

. . .

Request 80 Kbytes;

. . .

Request 60 Kbytes;

P2

. . .

Request 70 Kbytes;

. . .

Request 80 Kbytes;

- Deadlock occurs if both processes progress to their second request

Consumable Resources

- One that can be created/produced and destroyed/ consumed
- Typically no limit on the number of consumable resources of a particular type
- Resource ceases to exist after consumption
- Examples
  - Interrupts, signals, messages, contents of I/O buffers
- Can there be deadlock involving consumable resources?

Example

- Each process attempts to receive a message from the other and then sends a message to it

```
P1
. . .
Receive(P2);
. . .
Send(P2, M1);
```

```
P2
. . .
Receive(P1);
. . .
Send(P1, M2);
```

- Deadlock occurs if the “receive” is blocking
- Takes a rare combination of events!
  - Ever heard that a s/w is never bug free?

Other Kinds

- Physical
  - Printer, tape drive
- Logical
  - File, semaphore, data structure
- Preemptable
  - Can be taken away from process for some time with no ill effects
  - CPU,memory
- Non Preemptable
  - Will cause process to fail if taken away
  - Printer
- Resource type (e.g. Printer) vs resource instances (e.g. 2)

Deadlock Example

- utility program

❑ Copies a file from a tape to disk

❑ Prints the file to a printer

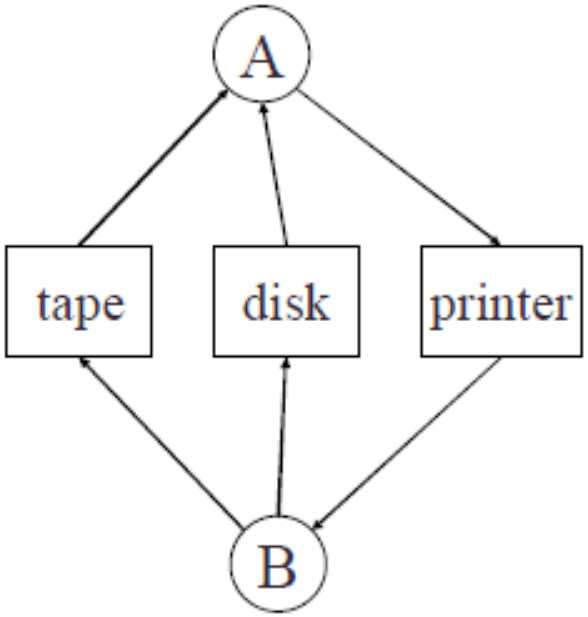
- Resources

❑ Tape

❑ Disk

❑ Printer

A deadlock



System Model

- Resource types R_1, R_2, . . ., 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**

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

Mutual exclusion :Only single process is allowed to use the resource.

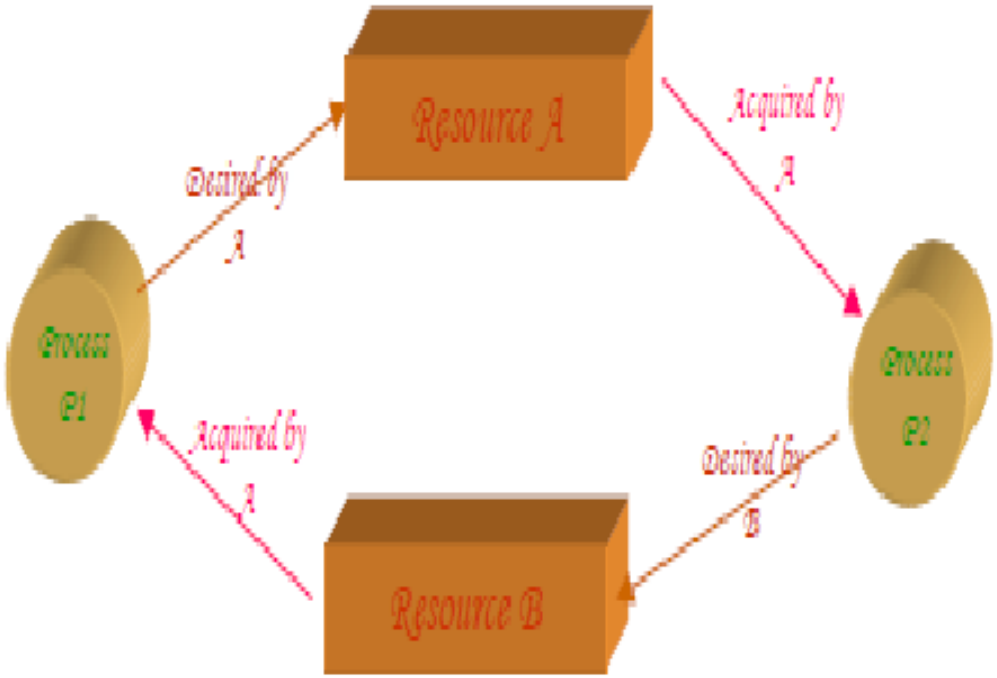
Hold and wait :Process holding at least one resource and waiting to acquire additional resources currently held by other processes.

No preemption :No resource can be removed forcibly from a process.

4. Circular wait: A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain

- First three are necessary but not sufficient conditions for a deadlock to exist

Resource-Allocation Graph

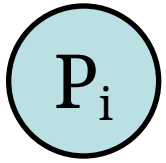


Resource-Allocation Graph

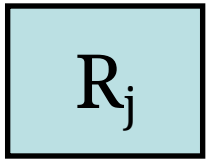
- Characterizes allocation of resources to processes.
- Directed graph to describe deadlocks
- A set of vertices V and a set of edges E.
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- request edge – directed edge $P_i \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Symbols)

- Process node



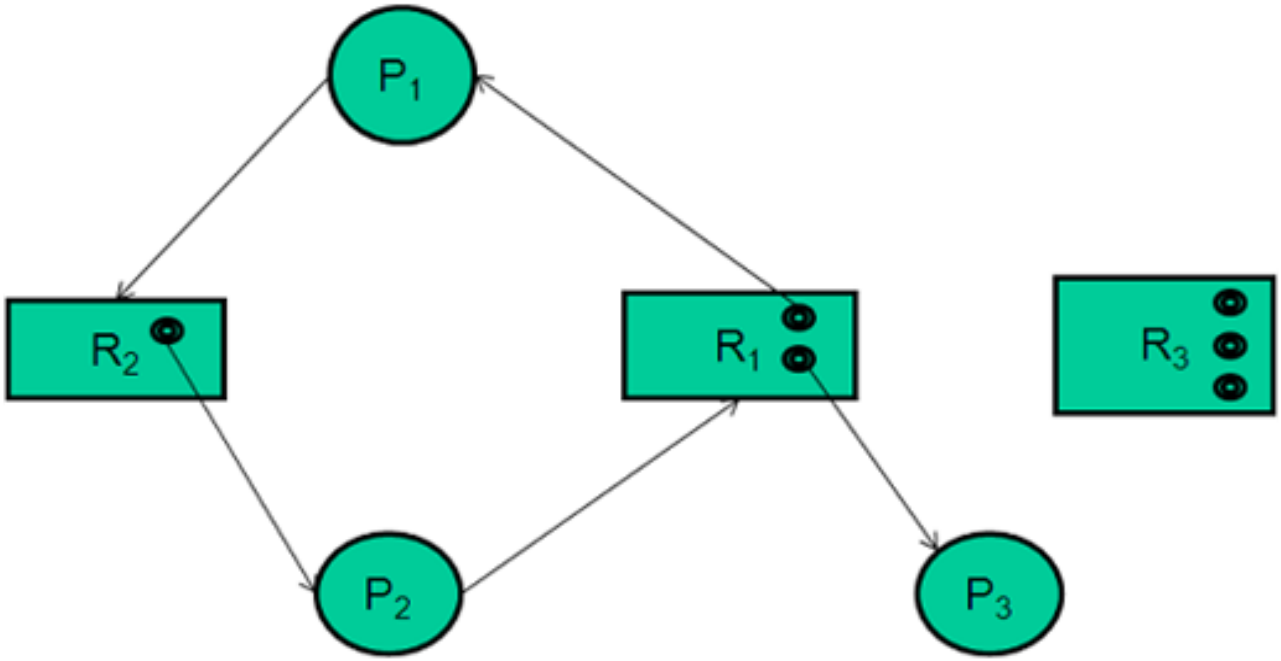
- Resource node



- Assignment edge

- Request edge

Example of a Resource allocation graph



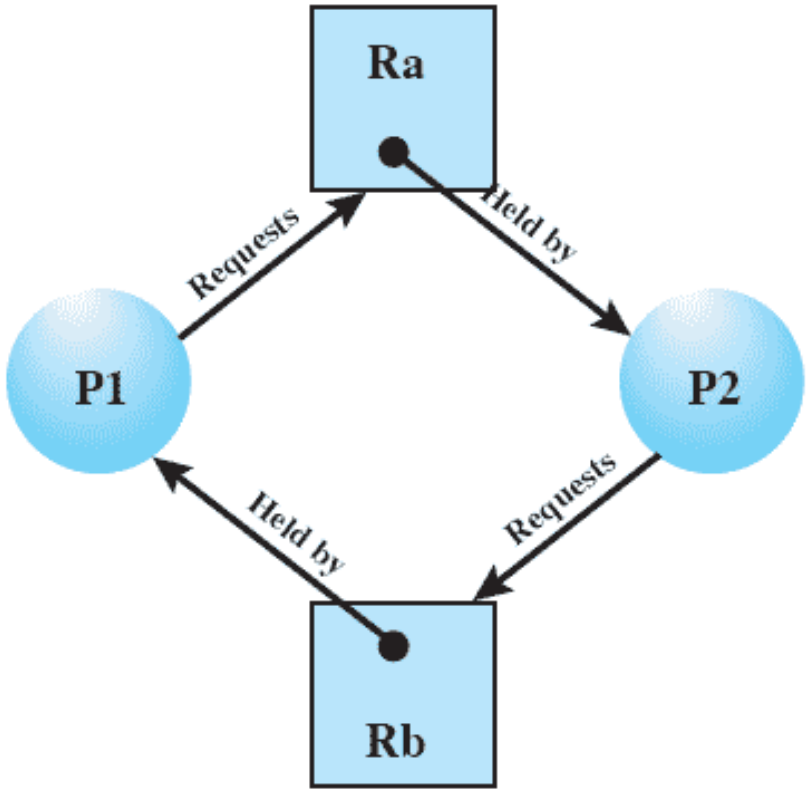
Process node

Assignment edge

Resource node

Request edge

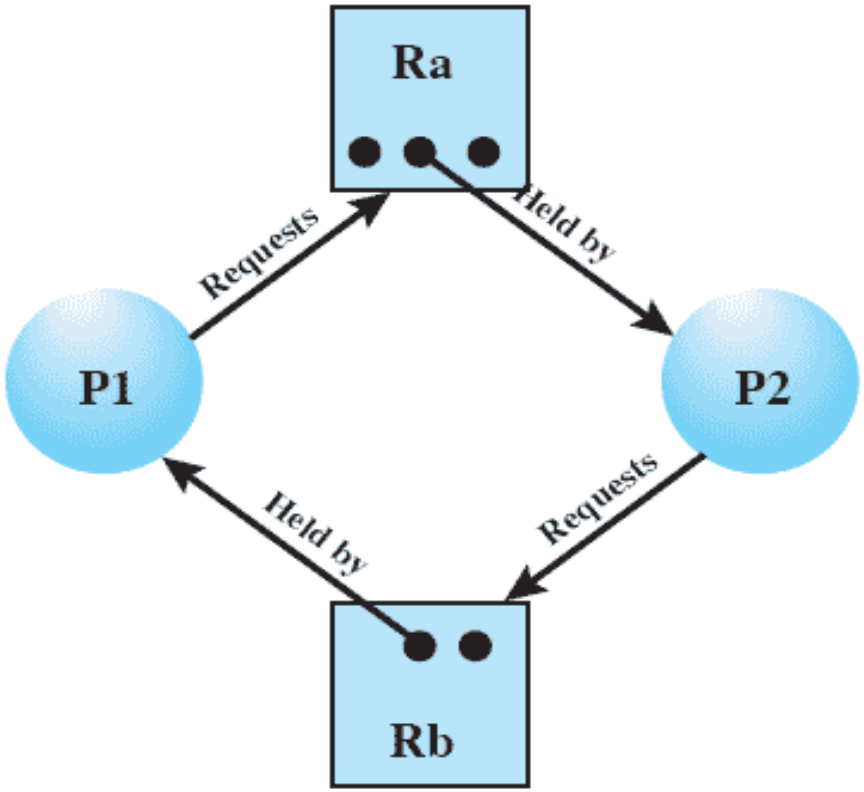
Case Study



Is there a deadlock here?

yes

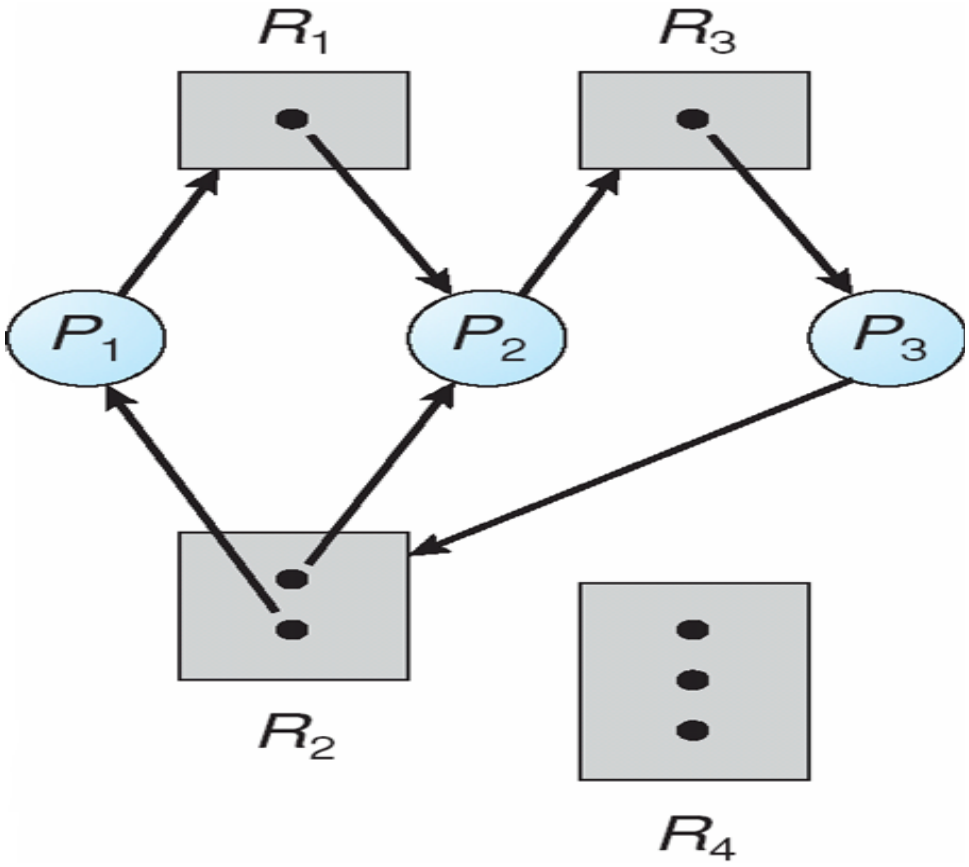
Case Study



What about this?

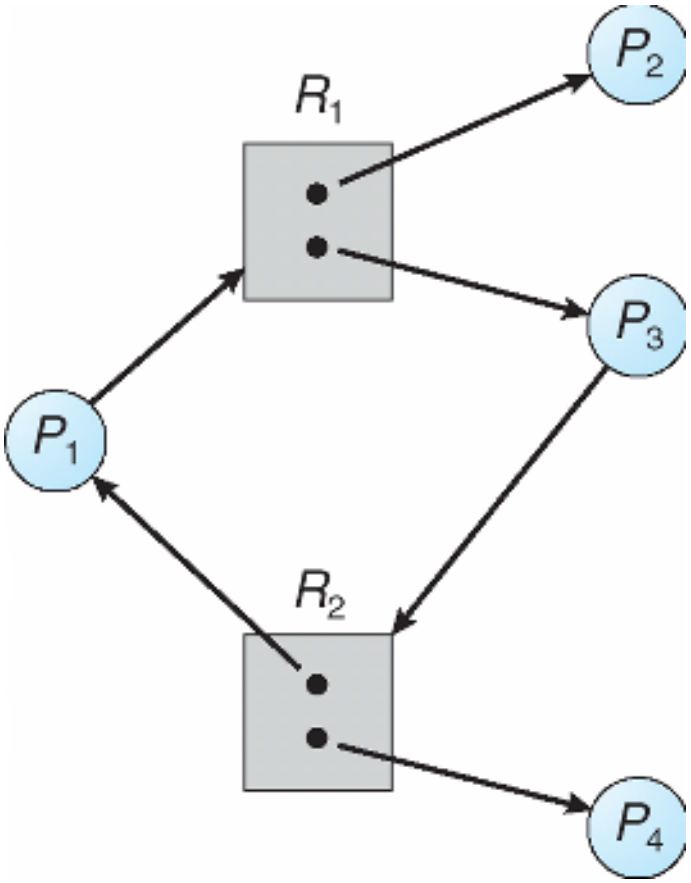
no

And this?



Yes deadlock

… and this?



Cycle but no deadlock

Basic Facts

- Deadlock cycle
- No cycle no deadlock
- Cycle deadlock
  - if only one instance per resource type
  - if several instances per resource type then there is a possibility of a deadlock
- If there is single instance of each resource type then cycle in the RAG is necessary and sufficient condition for the existence of a deadlock
- If each resource type has multiple instances ,then a cycle is a necessary but not sufficient condition for the existence of a deadlock

Contents

- Uniprocessor Scheduling: Types of Scheduling: Preemptive, Non-preemptive, Long-term, Medium-term,
  Short-term scheduling
- Scheduling Algorithms: FCFS, SJF, RR, Priority
- Multiprocessor Scheduling: Granularity
- Design Issues, Process Scheduling
- Deadlock: Principles of deadlock,
- Deadlock Prevention
- Deadlock Avoidance, Deadlock Detection,
- Deadlock Recovery

Dealing with deadlocks

- Four general approaches
  - Deadlock **prevention** – by adopting a policy that eliminates one of the four conditions
  - Deadlock **avoidance** – by making the appropriate dynamic choices based on the current state of resource allocation
  - Deadlock **detection** and **recovery** – attempt to detect presence of deadlock and take actions to recover
  - **Ignore** the problem and pretend that deadlocks never occur in the system
    - Used by most operating systems, including UNIX & windows
    - Remains a programmer's responsibility to write deadlock free code

Deadlock prevention

- Design a system in such a way that the possibility of deadlock is excluded by ensuring that at least one of the necessary conditions cannot hold
- Two main methods

1. Indirect – prevent all three of the necessary conditions occurring at once (Mutual exclusion, Hold-wait,No-preemption)
2. Direct – prevent Circular wait

Deadlock Prevention: Deny Mutual Exclusion

- **Mutual Exclusion** – not required for sharable resources;
- must hold for non-sharable resources.
- A process never needs to wait for a sharable resource
- But in reality, some resources are intrinsically non-sharable and hence we cannot prevent deadlock by denying mutual exclusion
- If access to a resource requires mutual exclusion, then mutual exclusion must be supported by the OS.
- Some resources, such as files, may allow multiple accesses for reads but only exclusive access for writes.

Deadlock Prevention:Disable hold & wait

- Must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Requires each process to request and be allocated all its resources before it begins execution
  - Allow process to request resources only when the process has none

  Disadvantages:
  - Low resource utilization
  - Starvation is possible
  - Process may not know in advance all of the resources that it will require.

Example

- Process that copies data from DVD drive to a file on disk, sorts the file and prints the results to a printer
- First protocol
  - First request DVD drive, disk file and printer
  - Do the job
  - Obvious demerit?
- Second protocol
  - Request DVD drive and disk file
  - Copy, release
  - Request disk file and printer
  - Print, release
  - Problem: data may have changed by now
- Starvation common to both
  - A process that needs several popular resources may have to wait indefinitely because at least one of the resources it needs is always allocated to some other process

Deadlock Prevention: Enable Pre-emption

- If a process that is holding some resources and requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
- Preempted resources are added to the list of resources for which the process is waiting.
- Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

Enable preemption

- Alternatively, if a process request some resources
  - Check if they are available.
    - If yes, grant
    - If not, check if they are allocated to some other process that is waiting for additional resources
      - If yes, preempt desired resources from waiting process and allocate to the requesting process
      - If no, requesting process must wait
        - » Some of its resources may be preempted, but only if another process requests them
        - » Process resumes only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting
  - Often applied to resources whose state can be easily saved and restored – CPU registers, memory space

Deadlock Prevention: Disable Circular Wait

- Impose a total ordering of all resource types,.
- Each process requests resources in an increasing order of enumeration.
- Lets say tape drive is 1, disk drive is 5 and printer is 12
- A process can initially request any number of instances of a resource type Ri
- After that process can request instances of resource type Rj only if f(j)>f (i)
- Alternatively, before requesting Rj, release all Ri such that f(i) >= f(j)
- When several instances of same type are needed, a single request for all of them must be issued
- Re-ordering of resources requires re-programming
- Ordering should be as per usage pattern of resources.

Deadlock Avoidance

- Deadlock prevention leads to inefficient use of resources & execution of processes.
- Requires that OS be given in advance additional information concerning which resources a process will request and use during its lifetime
- Based on this info, OS decides whether to grant the request or delay it
- System must consider resources currently available, resources currently allocated, future requests and releases

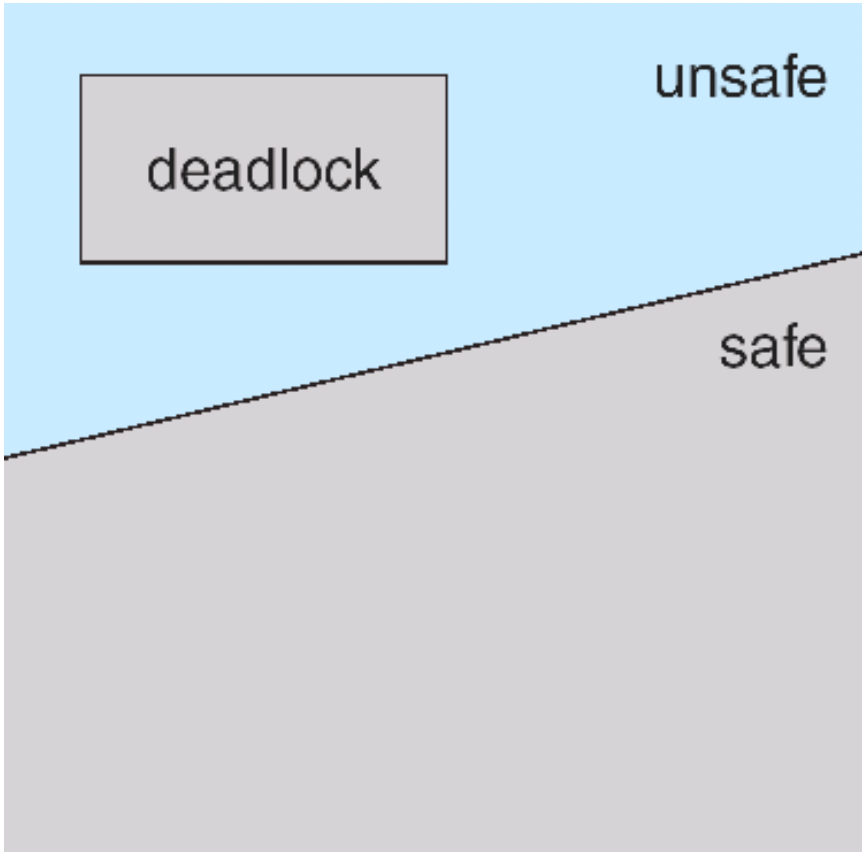
Deadlock Avoidance

- With this knowledge of complete sequence of requests and releases for each process, system can decide for each request whether or not the process should wait in order to avoid a future deadlock
- Variations in algorithms differ in amount and type of information required
- Simplest: each process declares the max number of resources of each type that it may need
- Dynamically examines the resource allocation state to ensure that a circular wait condition can never exist
- State: number of available and allocated resources, maximum demands of processes

Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.
- System is in safe state if there exists a safe sequence of all processes.
- Sequence $<P_1, P_2, \ldots, P_n>$ is safe if 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 $j<i$.
  - 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.

Safe, Unsafe , Deadlock State



- Safe state is not a deadlocked state
- Deadlocked state is an unsafe state
- Not all unsafe states are deadlocks
- Unsafe state may lead to a deadlock
- Better to always be in safe state

Example

- 12 magnetic tape drives, 3 processes
- Snapshot at t0

Process	Maximum needs	Currently holding
P0	10	5
P1	4	2
P2	9	2

- 3 mo
- System is in safe state with sequence P1, P0, P2

From safe to unsafe state

- Suppose at t1, P2 requests and is allocated 1 more tape drive

Process	Maximum needs	Currently holding
P0	10	5
P1	4	2
P2	9	3

- System is no longer in safe state
  - Only P1 can be allocated now. When done, both P0 and P2 will have to wait forever!
- If only we had made P2 wait until either of the other processes had finished, we could have avoided the deadlock

bottom line of avoidance algorithms

- Grant request only if the system remains in the safe state
  - Even if the resources it is asking for is currently available

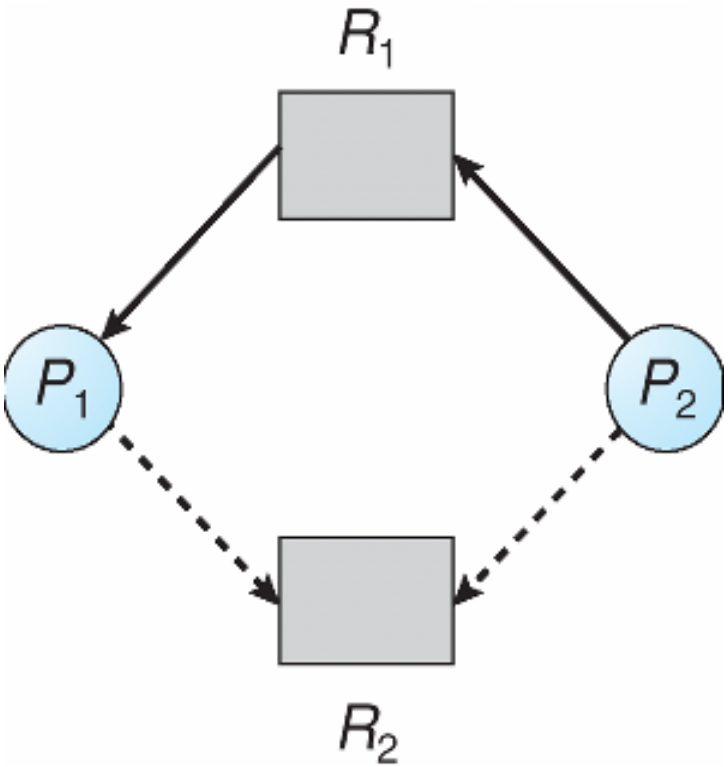
Avoidance algorithms

- When single instance of all resource types use RAG algorithm
  - Uses a variant of the RAG we saw earlier
- When multiple instances of resource types,
  - Use Banker's algorithm

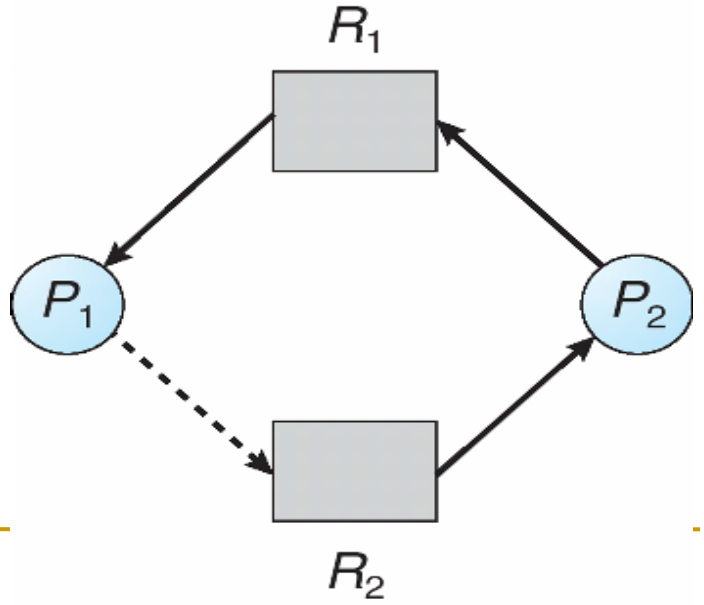
Resource-Allocation Graph Scheme

- Claim edge $P_i \rightarrow R_j$ indicated that process P_i may request resource R_j at some time in future
- Similar to request edge in direction but is represented by a dashed line
- when a process requests a resource, Claim edge converts to request edge
- Request edge converted to an assignment edge when the resource is allocated to the process.
- When a resource is released by a process, assignment edge reconverted to a claim edge
- Resources must be claimed *a prior* in the system

Resource-Allocation Graph



- Snapshot at time t
- Suppose P2 requests R2
- Although R2 is currently free it cannot be allocated to P2
  - Cycle!



Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph.
- Time complexity: For detecting a cycle in a graph it requires an order of $\mathbf{n^2}$ operations, Where n in number of processes in the system.

Banker's Algorithm

- Multiple instances of each resource type.
- Each process must a prior claim maximum use.
- Every process declares it maximum need/requirement.
- Maximum requirement should not exceed the total number of resources in the system.
- When a process requests a resource, system determines whether allocation will keep the system in safe state or not.
- If it is, resources get allocated. Otherwise need to wait until resources get available.
- When a process gets all its resources it must return them in a finite amount of time.

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 x m matrix. If *Max* [i,j] = k, then process P_i may request at most k instances of resource type R_j
- *Allocation*: n x m matrix. If Allocation[i,j] = k then P_i is currently allocated k instances of R_j
- *Need*: n x 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]

Resource = $\mathbf{R} = (R_1, R_2, \ldots, R_m)$	Total amount of each resource in the system
Available = $\mathbf{V} = (V_1, V_2, \ldots, V_m)$	Total amount of each resource not allocated to any process
Claim = $\mathbf{C} = \begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1m} \\ C_{21} & C_{22} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ C_{n1} & C_{n2} & \cdots & C_{nm} \end{pmatrix}$	C_{ij} = requirement of process i for resource j
Allocation = $\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix}$	A_{ij} = current allocation to process i of resource j

Following relationship holds

1. $R_j = V_j + \sum_{i=1}^{n} A_{ij}$, for all j — All resources are either available or allocated.

2. $C_{ij} \leq R_j$, for all i,j — No process can claim more than the total amount of resources in the system.

3. $A_{ij} \leq C_{ij}$, for all i,j — No process is allocated more resources of any type than the process originally claimed to need.

Safety Algorithm

Requires m * n^2 operation to decide whether a state is

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

   Work = *Available*

   Finish [i] = *false* for *i* - 1,3, …, *n*.

2. Find process *i* such that both:

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

   (b) $Need_i \leq Work$

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

3. $Work = Work + Allocation_i$

   Finish [i] = *true*

   go to step 2.

4. If *Finish* [i] == true for all *i*, then the system is in a safe state; otherwise process whose index is false *may potentially be in deadlock* in future

Example

- 5 processes P0 through P4
- 3 resource types: A (10 instances), B (5 instances), and C (7 instances)
- Snapshot of system at t_0

Process	Allocation			Max			Need		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3			
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			

Example

- 5 processes P0 through P4
- 3 resource types: A (10 instances), B (5 instances), and C (7 instances)
- Snapshot of system at t_0 : is it safe?

Process	Allocation			Max			Need		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	7	4	3
P_1	2	0	0	3	2	2	1	2	2
P_2	3	0	2	9	0	2	6	0	0
P_3	2	1	1	2	2	2	0	1	1
P_4	0	0	2	4	3	3	4	3	1

Safe State?

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

Work		
3	3	2

Safe State?

Process	Allocation			Need			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	0	1	0	7	4	3	F	F			
P_1	**2**	**0**	**0**	**1**	**2**	**2**	**F**	**T**			
P_2	3	0	2	6	0	0	F	F			
P_3	2	1	1	0	1	1	F	F			
P_4	0	0	2	4	3	1	F	F			

Work		
3	3	2
5	**3**	**2**

Safe State?

Process	Allocation			Need			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	0	1	0	7	4	3	F	F	F		
P_1	2	0	0	1	2	2	F	T	T		
P_2	3	0	2	6	0	0	F	F	F		
P_3	**2**	**1**	**1**	**0**	**1**	**1**	**F**	**F**	**T**		
P_4	0	0	2	4	3	1	F	F	F		

Work		
3	3	2
5	3	2
7	**4**	**3**

Safe State?

Process	Allocation			Need			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	**0**	**1**	**0**	**7**	**4**	**3**	**F**	**F**	**F**	**T**	
P_1	2	0	0	1	2	2	F	T	T	T	
P_2	3	0	2	6	0	0	F	F	F	F	
P_3	2	1	1	0	1	1	F	F	T	T	
P_4	0	0	2	4	3	1	F	F	F	F	

Work		
3	3	2
5	3	2
7	4	3
7	**5**	**3**

Safe State?

Process	Allocation			Need			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	0	1	0	7	4	3	F	F	F	T	T
P_1	2	0	0	1	2	2	F	T	T	T	T
P_2	**3**	**0**	**2**	**6**	**0**	**0**	**F**	**F**	**F**	**F**	**T**
P_3	2	1	1	0	1	1	F	F	T	T	T
P_4	0	0	2	4	3	1	F	F	F	F	F

Work		
3	3	2
5	3	2
7	4	3
7	5	3
10	**5**	**5**

Safe State?

Process	Allocation			Need			Finish	Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C						
P_0	0	1	0	7	4	3	F	F	F	T	T	T
P_1	2	0	0	1	2	2	F	T	T	T	T	T
P_2	3	0	2	6	0	0	F	F	F	F	T	T
P_3	2	1	1	0	1	1	F	F	T	T	T	T
P_4	**0**	**0**	**2**	**4**	**3**	**1**	**F**	**F**	**F**	**F**	**F**	**T**

Work		
3	3	2
5	3	2
7	4	3
7	5	3
10	5	5
10	**5**	**7**

Safe Sequence: <P1, P3, P0, P2, P4>

Resource-Request Algorithm

$Request_i \rightarrow$ request vector (P_i); e.g. $Request_i[j] = k$

1. If $Request_i \leq Need_i$ go to step 2; Else *raise error condition* $\rightarrow$ process exceeds its maximum claim
2. If $Request_i \leq Available$, go to step 3; Else P_i *must wait*, since resources are not available
3. Tentatively allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request_i;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

Check the safety of state -

- *If safe* $\Rightarrow$ *the resources are allocated to* P_i
- *If unsafe* $\Rightarrow$ P_i *must wait, and the tentative resource allocation is cancelled*

Example: P_1 Request (1,0,2)

- Check that Request ≤ need that is, (1,0,2) ≤ (1,2,2) ⇒ true
- Check that Request ≤ Available that is, (1,0,2) ≤ (3,3,2) ⇒ 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	

- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

$Request_1 = (1, 0, 2)$

- Request <need that is, (1,0,2) < (1,2,2) □->true
- $Request_1$ <= Available
  - (1, 0, 2) <= (3, 3, 2)
- Pretend to grant
  - $Allocation_1$, $Need_1$, Available will change (2, 3, 0)

Process	Allocation			Max			Need		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	7	4	3
P_1	**3**	**0**	**2**	**3**	**2**	**2**	**0**	**2**	**0**
P_2	3	0	2	9	0	2	6	0	0
P_3	2	1	1	2	2	2	0	1	1

Is the state safe?

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

Work		
2	3	0

Is the state safe?

Process	Allocation			Need			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	0	1	0	7	4	3	F	F			
P_1	**3**	**0**	**2**	**0**	**2**	**0**	**F**	**T**			
P_2	3	0	2	6	0	0	F	F			
P_3	2	1	1	0	1	1	F	F			
P_4	0	0	2	4	3	1	F	F			

Work		
2	3	0
5	**3**	**2**

Is the state safe?

Process	Allocation			Need			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	0	1	0	7	4	3	F	F	F		
P_1	3	0	2	0	2	0	F	T	T		
P_2	3	0	2	6	0	0	F	F	F		
P_3	**2**	**1**	**1**	**0**	**1**	**1**	**F**	**F**	**T**		
P_4	0	0	2	4	3	1	F	F	F		

Work		
2	3	0
5	3	2
7	**4**	**3**

Is the state safe?

Process	Allocation			Need			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	**0**	**1**	**0**	**7**	**4**	**3**	**F**	**F**	**F**	**T**	
P_1	3	0	2	0	2	0	F	T	T	T	
P_2	3	0	2	6	0	0	F	F	F	F	
P_3	2	1	1	0	1	1	F	F	T	T	
P_4	0	0	2	4	3	1	F	F	F	F	

Work		
2	3	0
5	3	2
7	4	3
7	**5**	**3**

Is the state safe?

Process	Allocation			Need			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	0	1	0	7	4	3	F	F	F	T	T
P_1	3	0	2	0	2	0	F	T	T	T	T
P_2	**3**	**0**	**2**	**6**	**0**	**0**	**F**	**F**	**F**	**F**	**T**
P_3	2	1	1	0	1	1	F	F	T	T	T
P_4	0	0	2	4	3	1	F	F	F	F	F

Work		
2	3	0
5	3	2
7	4	3
7	5	3
10	**5**	**5**

Is the state safe?

Process	Allocation			Need			Finish	Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C						
P_0	0	1	0	7	4	3	F	F	F	T	T	T
P_1	3	0	2	0	2	0	F	T	T	T	T	T
P_2	3	0	2	6	0	0	F	F	F	F	T	T
P_3	2	1	1	0	1	1	F	F	T	T	T	T
P_4	**0**	**0**	**2**	**4**	**3**	**1**	**F**	**F**	**F**	**F**	**F**	**T**

Work		
2	3	0
5	3	2
7	4	3
7	5	3
10	5	5
10	**5**	**7**

Safe Sequence: <P1, P3, P0, P2, P4>

$Request_4 = (3, 3, 0)$

- $Request_4$ <= Available?
  - (3, 3, 0) <= (2, 3, 0)?
  - No. Thus request from P4 cannot be granted

$Request_0 = (0, 2, 0)$

- $Request_0$ <= Available
  - (0, 2, 0) <= (2, 3, 0)
- Pretend to grant
  - $Allocation_0$, $Need_0$, Available will change (2, 1, 0)

Process	Allocation			Max			Need		
	A	B	C	A	B	C	A	B	C
P_0	**0**	**3**	**0**	**7**	**5**	**3**	**7**	**2**	**3**
P_1	3	0	2	3	2	2	0	2	0
P_2	3	0	2	9	0	2	6	0	0
P_3	2	1	1	2	2	2	0	1	1

Is the state safe?

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

Work		
2	1	0

Is the state safe?

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

Work		
2	1	0

Example : Is the below system in safe state?

Process	Allocation				Max			
	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2
P_1	1	0	0	0	1	7	5	0
P_2	1	3	5	4	2	3	5	6
P_3	0	6	3	2	0	6	5	2
P_4	0	0	1	4	0	6	5	6

Available			
1	5	2	0

Deadlock Detection

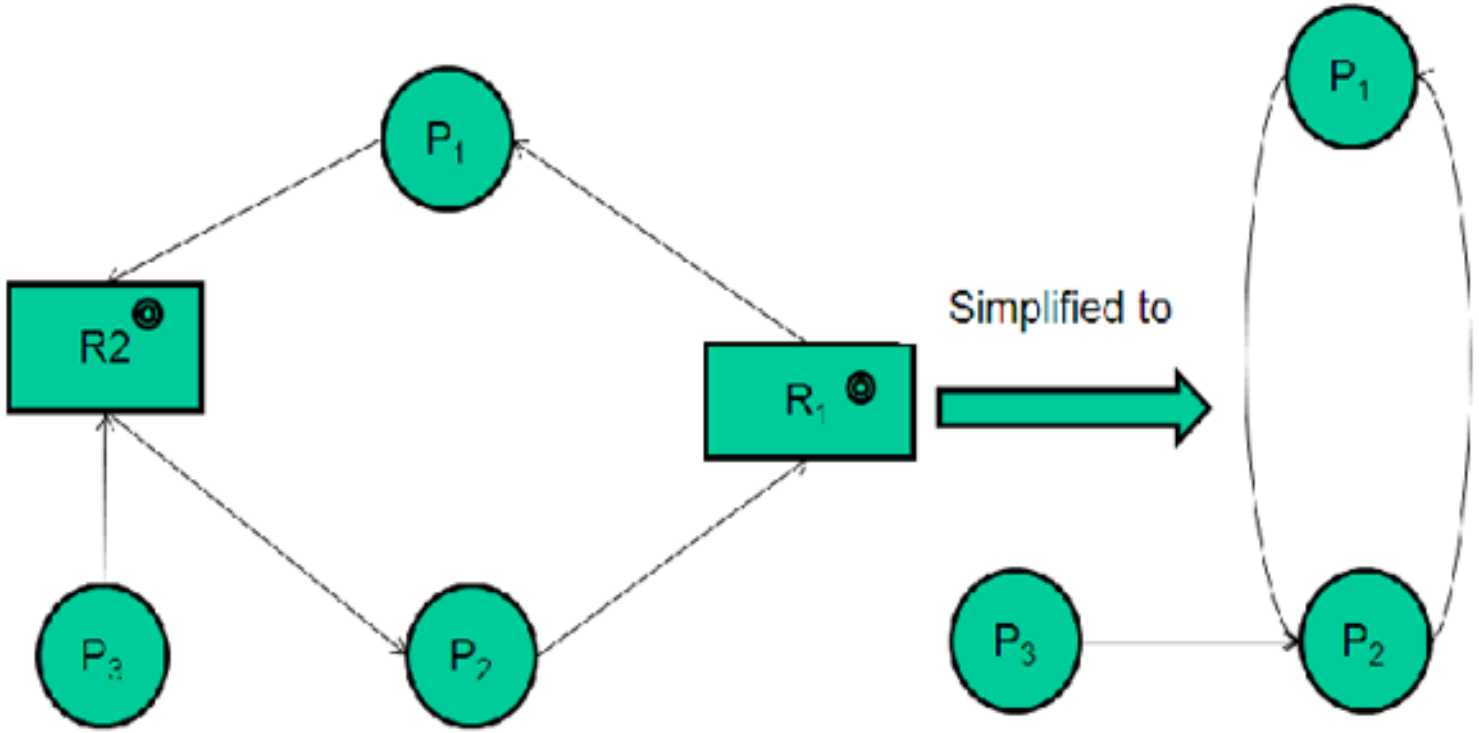
- When neither prevention nor avoidance is employed, a deadlock situation may arise
- System can provide an algorithm to detect an algorithm and an algorithm to recover from it if it has occurred
- Requires overhead that includes not only the run-time costs of maintaining the necessary information and executing the detection algorithm but also the potential losses inherent in recovering from a deadlock

Case 1: Single instance of each resource type

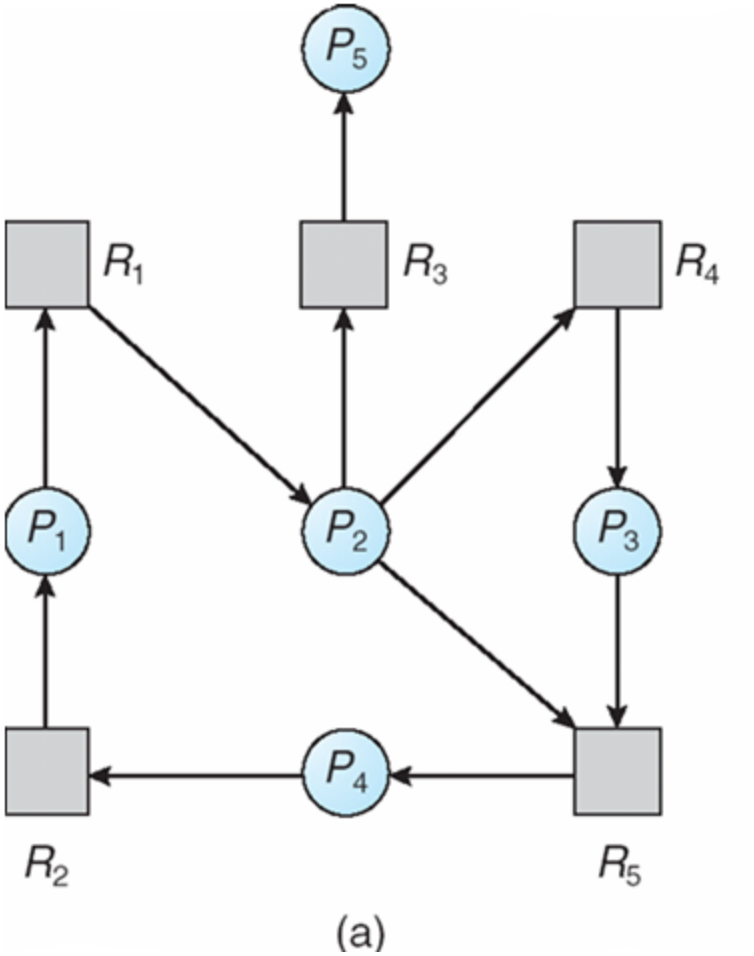
- Use a variant of RAG called WFG
  - by removing the resource nodes and collapsing the appropriate edges
- An edge from P_i to P_j in a wait-for graph implies that
- process P_i is waiting for process P_j to release a resource that P_i needs
  - if the corresponding RAG contains two edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$ for some R_q
- Periodically invoke an algorithm that searches for a cycle in the WFG
  - If there is a cycle, there exists a deadlock.
  - O(n^2) operations

Resource-Allocation Graph and Wait-for Graph

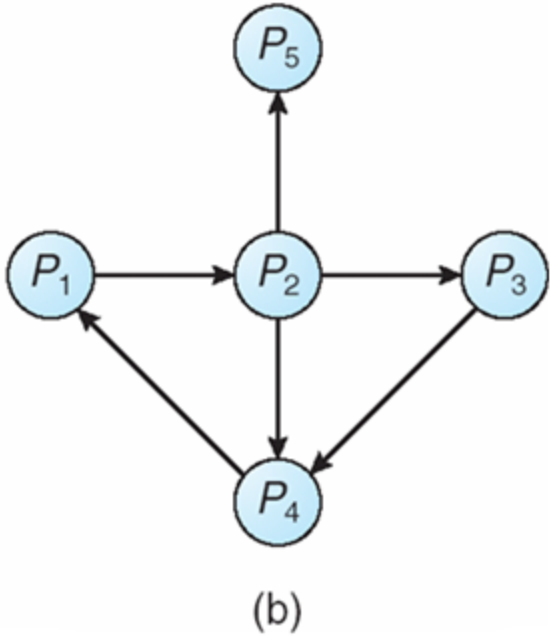
Def: Remove the resource edges from RAG and collapse the appropriate edges. WFG is constructed only when cycle is both necessary & sufficient condition for deadlock



Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph



Corresponding wait-for graph

Case 2: Several instances of each resource type

- WFG is no longer applicable for this case
  - Why?
- Detection algorithm
  - Uses several time varying data structures similar to those of Banker's
  - Available (m), Allocation (nXm), Request(nXm)
  - Same definition of <= notation
  - Same definition of $Allocation_i$ and $Request_i$
- Crux of the algorithm
  - Simply investigates every possible allocation sequence for the processes that remain to be completed

Case 2: Several instances of each resource type

Similar to the *Banker's algorithm* safety test with the following difference in semantics;

- Replacing $Need_i \rightarrow Request_i$; where $Request_i$ is the actual vector of resources, process *i* is currently waiting to acquire
- May be slightly optimized by initializing *Finish* [i] to *true* for every process *i* where $Allocation_i$ is zero
- Optimistic and *only care if there is a deadlock now*. If process will need more resources in future → deadlock, discovered in future
- Processes *in the end* remaining *with false entry* are the ones *involved in deadlock* at this time

Deadlock Detection Algorithm

Requires $m * n^2$ operation to detect a deadlock

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

   Work = *Available*

   If $Allocation_i \neq 0$ for $i = 1,2, \ldots, n$ then

   Finish [i] = **false**, else *Finish* [i] = **true**

2. Find process *i* such that both:

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

   (b) $Request_i \leq Work$

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

3. $Work = Work + Allocation_i$

   Finish [i] = *true*

   go to step 2

4. If *Finish* [i] == **false**, for some $1 \leq i \leq n$, $\rightarrow$ deadlocked;

   If *Finish* [i] == **false** then process P_i is deadlocked

Contrast with Banker's Algorithm

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize *Work* = *Available*. For $i = 0, 1, ..., n-1$, if $Allocation_i \neq 0$, then $Finish[i] = false$; otherwise, $Finish[i] = true$.

2. Find an index i such that both

   a. $Finish[i] == false$

   b. $Request_i \leq Work$

   If no such i exists, go to step 4.

3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   Go to step 2.

4. If $Finish[i] == false$, for some i, $0 < i < n$, then the system is in a deadlocked state. Moreover, if $Finish[i] == false$, then process P_i is deadlocked.

Example

- 5 processes P0 through P4
- 3 resource types: A (7 instances), B (2 instances), and C (6 instances)
- Snapshot of system at t_0 : is there a deadlock?

Process	Allocation			Request		
	A	B	C	A	B	C
P_0	0	1	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

Available		
0	0	0

Is there a deadlock?

Process	Allocation			Request			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	0	1	0	0	0	0	F				
P_1	2	0	0	2	0	2	F				
P_2	3	0	3	0	0	0	F				
P_3	2	1	1	1	0	0	F				
P_4	0	0	2	0	0	2	F				

Work		
0	0	0

Please note that all are initially 'F' here by coincidence

Is there a deadlock?

Process	Allocation			Request			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	**0**	**1**	**0**	**0**	**0**	**0**	**F**	**T**			
P_1	2	0	0	2	0	2	F	F			
P_2	3	0	3	0	0	0	F	F			
P_3	2	1	1	1	0	0	F	F			
P_4	0	0	2	0	0	2	F	F			

Work		
0	0	0
0	**1**	**0**

Is there a deadlock?

Process	Allocation			Request			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	0	1	0	0	0	0	F	T	T		
P_1	2	0	0	2	0	2	F	F	F		
P_2	**3**	**0**	**3**	**0**	**0**	**0**	**F**	**F**	**T**		
P_3	2	1	1	1	0	0	F	F	F		
P_4	0	0	2	0	0	2	F	F	F		

Work		
0	0	0
0	1	0
3	**1**	**3**

Is there a deadlock?

Process	Allocation			Request			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	0	1	0	0	0	0	F	T	T	T	
P_1	**2**	**0**	**0**	**2**	**0**	**2**	**F**	**F**	**F**	**T**	
P_2	3	0	3	0	0	0	F	F	T	T	
P_3	2	1	1	1	0	0	F	F	F	F	
P_4	0	0	2	0	0	2	F	F	F	F	

Work		
0	0	0
0	1	0
3	1	3
5	**1**	**3**

Is there a deadlock?

Process	Allocation			Request			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	0	1	0	0	0	0	F	T	T	T	T
P_1	2	0	0	2	0	2	F	F	F	T	T
P_2	3	0	3	0	0	0	F	F	T	T	T
P_3	**2**	**1**	**1**	**1**	**0**	**0**	**F**	**F**	**F**	**F**	**T**
P_4	0	0	2	0	0	2	F	F	F	F	F

Work		
0	0	0
0	1	0
3	1	3
5	1	3
7	**2**	**4**

Is there a deadlock?

Process	Allocation			Request			Finish	Finish	Finish	Finish	Finish	
	A	B	C	A	B	C						
P_0	0	1	0	0	0	0	F	T	T	T	T	T
P_1	2	0	0	2	0	2	F	F	F	T	T	T
P_2	3	0	3	0	0	0	F	F	T	T	T	T
P_3	2	1	1	1	0	0	F	F	F	F	T	T
P_4	**0**	**0**	**2**	**0**	**0**	**2**	**F**	**F**	**F**	**F**	**F**	**T**

Work		
0	0	0
0	1	0
3	1	3
5	1	3
7	2	4
7	2	6

Thus, no deadlock: Sequence $<P_0, P_2, P_3, P_1, P_4>$

P2 now makes an additional request for 1 C

- Request matrix is modified. Is there deadlock now?

Process	Allocation			Request			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	0	1	0	0	0	0	F				
P_1	2	0	0	2	0	2	F				
P_2	3	0	3	0	0	**1**	F				
P_3	2	1	1	1	0	0	F				

Work		
0	0	0

Is there a deadlock?

Process	Allocation			Request			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	**0**	**1**	**0**	**0**	**0**	**0**	**F**	**T**			
P_1	2	0	0	2	0	2	F	F			
P_2	3	0	3	0	0	**1**	F	F			
P_3	2	1	1	1	0	0	F	F			
P_4	0	0	2	0	0	2	F	F			

Work		
0	0	0
0	**1**	**0**

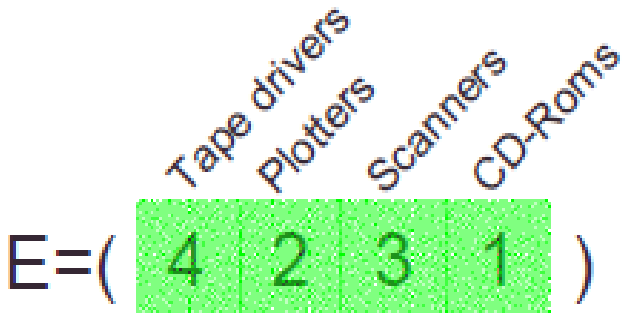
Is there a deadlock?

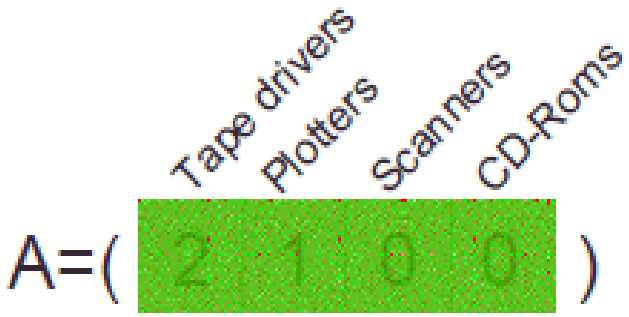
Process	Allocation			Request			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	0	1	0	0	0	0	F	T			
P_1	2	0	0	2	0	2	F	F			
P_2	3	0	3	0	0	**1**	F	F			
P_3	2	1	1	1	0	0	F	F			
P_4	0	0	2	0	0	2	F	F			

Work		
0	0	0
0	1	0

P1, P2, P3 and P4 are in a deadlock now!

Example: Multiple resources of each type





Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Example: Multiple resources of each type

Tape drivers Plotters Scanners CD-Roms

$E = (\ 4\ \ 2\ \ 3\ \ 1\ )$

Tape drivers Plotters Scanners CD-Roms

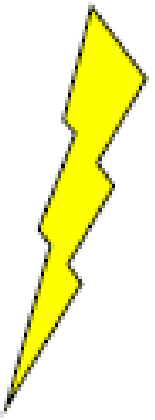
$A = (\ 2\ \ 0\ \ 0\ \ 0\ )$

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$



Detection-Algorithm Usage

- When, and how often, to invoke algorithm depends on:
  - How often a deadlock is likely to occur?
  - How many processes will be affected by deadlock when it happens?
- If deadlock occurs frequently, then the detection algorithm should be invoked frequently.
- We could invoke the deadlock detection algorithm every time a request for allocation cannot be granted immediately.
- By this we can identify deadlock causing process & processes involved in deadlock also.
- But this incurs overhead in computation time.
- So can call algorithms after 1 hour / CPU utilization drops below 40%.
- 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.

Contents

- Uniprocessor Scheduling: Types of Scheduling: Preemptive, Non-preemptive, Long-term, Medium-term, Short-term scheduling
- Scheduling Algorithms: FCFS, SJF, RR, Priority
- Multiprocessor Scheduling: Granularity
- Design Issues, Process Scheduling
- Deadlock: Principles of deadlock, Deadlock Avoidance
- Deadlock Detection, Deadlock Prevention
- Deadlock Recovery

Deadlock Recovery

- Once detected, several alternatives are available for recovery inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually
- Abort one or more processes to break the circular wait
- Preempt some resources from one or more of the deadlocked processes

Process Termination / Abort

- Two alternatives
- In both, system reclaims all resources held by the terminated process
- **Abort all deadlocked processes**
  - Everything the processes had done so far has gone down the drain!
- **Abort one process at a time until the deadlock cycle is eliminated**
  - Whose turn is next?
    - Policy decision similar to scheduling decisions
  - Considerable overhead of detecting deadlock after each termination!
- Aborting comes with several issues
  - What if in the middle of updating a file or printing to a printer

Whom to abort next?

- Abort those processes whose termination will incur the minimum *cost*
- *Cost* depends upon
  - What is the priority of the process?
  - How long the process has computed and how much longer the process will compute before completing its designated task?
  - How many and what type of resources the process has used (e. g., preemptable, non-preemptable)?
  - How many more resources the process needs in order to complete?
  - How many processes will need to be terminated?
  - Whether the process is interactive or batch?

Resource Preemption

- We successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken
- Three issues need to be addressed:
  - Selecting a victim
  - Rollback
  - Starvation

Selecting a victim

- Which resources and of which processes next in order to minimize cost

Rollback

- What to do with the process from whom resource has been preempted? It cannot continue normal execution
- Rollback to some safe state and restart it from that state
  - What is safe state, also system will require to keep states of all processes
  - Thus some systems prefer total rollback

Starvation

- How can we guarantee that resources will not always be preempted from the same process
- If decision to pick is primarily based on cost factors, same unfortunate fellow may get picked up every time!
- We thus need an upper bound (small and finite) on how many times you can be chosen as a victim
  - Include the number of rollbacks in the cost factor

Summary of Deadlock prevention, Avoidance,Detection

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	• Works well for processes that perform a single burst of activity • No preemption necessary	• Inefficient • Delays process initiation • Future resource requirements must be known by processes
		Preemption	• Convenient when applied to resources whose state can be saved and restored easily	• Preempts more often than necessary
		Resource ordering	• Feasible to enforce via compile-time checks • Needs no run-time computation since problem is solved in system design	• Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	• No preemption necessary	• Future resource requirements must be known by OS • Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	• Never delays process initiation • Facilitates online handling	• Inherent preemption losses

Self Study

- Thread Scheduling
- Real-time scheduling

Thread scheduling

- An application can be implemented as a set of threads that cooperate and execute concurrently in the same address space.

  Criteria: When related threads run in parallel performance improves.

- **Load sharing**: processes are not assigned to a particular processor.

  A global queue of ready threads is maintained and idle processor selects thread from the queue.

- **Gang scheduling**: Bunch of related threads scheduled together to run on a set of processors at the same time, on one –to one basis.

Thread scheduling

- **Dedicated processor assignment**: Each program gets as many processors as there are parallel threads.
- **Dynamic scheduling**: Scheduling done at run time.

Load sharing:

- Advantages:
- The load is distributed evenly across the processors, assuring that no processor is idle.
- No centralizer scheduler is required
- Three versions of Load sharing:
- FCFS
- Smallest number of threads first
- Pre-emptive smallest number of threads first

Real-time systems

- Real-time computing is an important emerging discipline in CS and CE.
- Control of lab experiments, robotics, process control, telecommunication etc.
- It is a type of computing where **correctness of the computation depends not only on the logical results but also on the time at which the results are produced.**
- Hard real-time systems: Must meet deadline. Ex: Space shuttle rendezvous with other space station.
- Soft real-time system: Deadlines are there but not mandatory. Results are discarded if the deadline is not met.

Characteristics of Real-Time (RT) systems

- Determinism
- Responsiveness
- User control
- Reliability
- Fail-soft operation

Deterministic Response

- External event and timings dictate the request of service.
- OS's response depends on the speed at which it can respond to interrupts and on whether the system has sufficient capacity to handle requests.
- **Determinism** is concerned with how long the OS delays before acknowledging an interrupt.
- In non-RT this delay may be in the order of 10's and 100's of millisecs, whereas in an RT it may have few microsec to 1millisec.

RT .. Responsiveness

- **Responsiveness** is the time for servicing the interrupt once it has been acknowledged.
- Comprises:
  - Time to transfer control, (and context switch) and execute the ISR
  - Time to handle nested interrupts, the interrupts that should be serviced when executing this ISR. Higher priority Interrupts.
- response time = F(responsiveness, determinism)

RT .. User Control

- **User control** : User has a much broader control in RT-OS than in regular OS.
- Priority
- Hard or soft deadlines
- Deadlines

RT .. Reliability

- **Reliability**: A processor failure in a non-RT may result in reduced level of service. But in an RT it may be catastrophic : life and death, financial loss, equipment damage.

Fail-soft operation:

- Fail-soft operation: Ability of the system to fail in such a way preserve as much capability and data as possible.
- In the event of a failure, immediate detection and correction is important.
- Notify user processes to rollback.

Requirements of RT

- Fast context switch
- Minimal functionality (small size)
- Ability to respond to interrupts quickly (Special interrupts handlers)
- Multitasking with signals and alarms
- Special storage to accumulate data fast
- **Pre-emptive scheduling**

Requirements of RT (contd.)

- Priority levels
- Minimizing interrupt disables
- Short-term scheduler (“omni-potent”)
- Time monitor
- Goal: Complete all hard real-time tasks by dead-line. Complete as many soft real-time tasks as possible by their deadline.

RT scheduling

- Static table-driven approach
- Static priority-driven preemptive scheduling
- Dynamic planning-based scheduling
- Dynamic best-effort scheduling:

RT scheduling

- Static table-driven approach
  - For periodic tasks.
  - Input for analysis consists of : periodic arrival time, execution time, ending time, priorities.
  - Inflexible to dynamic changes.
  - General policy: earliest deadline first.

RT scheduling

- Static priority-driven preemptive scheduling
  - For use with non-RT systems: Priority based preemptive scheduling.
  - Priority assignment based on real-time constraints.
  - Example: Rate monotonic algorithm

RT scheduling

- Dynamic planning-based scheduling
  - After the task arrives before execution begins, a schedule is prepared that includes the new as well as the existing tasks.
  - If the new one can go without affecting the existing schedules than nothing is revised.
  - Else schedules are revised to accommodate the new task.
  - Remember that sometimes new tasks may be rejected if deadlines cannot be met.

RT scheduling

- Dynamic best-effort scheduling:
  - used in most commercial RTs of today
  - tasks are aperiodic, no static scheduling is possible
  - some short-term scheduling such as shortest deadline first is used.
  - Until the task completes we do not know whether it has met the deadline.

End of Chapter