Chapter 8 – Processor/Process Scheduling

Outline

Objectives

- After reading this chapter, you should understand:
  - the goals of processor scheduling.
  - preemptive vs. nonpreemptive scheduling.
  - the role of priorities in scheduling.
  - scheduling criteria.
  - common scheduling algorithms.
  - the notions of deadline scheduling and real-time scheduling.
  - Java thread scheduling.

8.1 Introduction

- Processor scheduling policy
  - Decides which process runs at given time
  - Different schedulers will have different goals
    - Maximize throughput
    - Minimize latency
    - Prevent indefinite postponement
    - Complete process by given deadline
    - Maximize processor utilization

8.2 Scheduling Levels

- High-level scheduling
  - Determines which jobs can compete for resources
  - Controls number of processes in system at one time
- Intermediate-level scheduling
  - Determines which processes can compete for processors
  - Responds to fluctuations in system load
- Low-level scheduling
  - Assigns priorities
  - Assigns processors to processes

8.2 Scheduling Levels

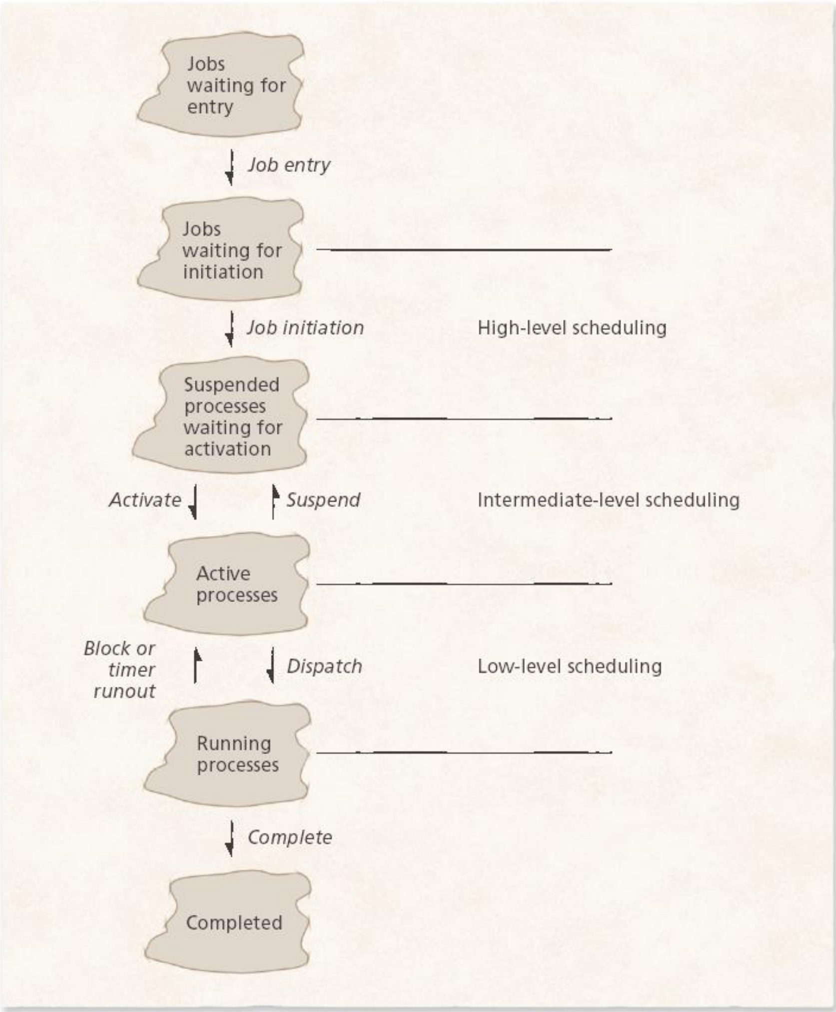


Figure 8.1 Scheduling levels.

8.3 Preemptive vs. Nonpreemptive Scheduling

- Preemptive processes
  - Can be removed from their current processor
  - Can lead to improved response times
  - Important for interactive environments
  - Preempted processes remain in memory
- Nonpreemptive processes
  - Run until completion or until they yield control of a processor
  - Unimportant processes can block important ones indefinitely

8.4 Priorities

- Static priorities
  - Priority assigned to a process does not change
  - Easy to implement
  - Low overhead
  - Not responsive to changes in environment
- Dynamic priorities
  - Responsive to change
  - Promote smooth interactivity
  - Incur more overhead than static priorities
    - Justified by increased responsiveness

8.5 Scheduling Objectives

- Different objectives depending on system
  - Maximize throughput
  - Maximize number of interactive processes receiving acceptable response times
  - Minimize resource utilization
  - Avoid indefinite postponement
  - Enforce priorities
  - Minimize overhead
  - Ensure predictability

8.5 Scheduling Objectives

- Several goals common to most schedulers
  - Fairness
  - Predictability
  - Scalability

8.5.1 CPU Scheduler

- Short-term scheduler
- Selects a process from among the processes in the ready queue
- Invokes the dispatcher to have the CPU allocated to the selected process

8.5.2 Dispatcher

- Dispatcher gives control of the CPU to the process selected by the short-term scheduler; this involves:
- switching context
- switching to user mode
- jumping to the proper location in the user program to start (or restart) it

8.5.2 Dispatcher

- Dispatch latency – time it takes for the dispatcher to stop one process and start another running.
- Typically, a few microseconds

8.5.3 CPU Scheduler

- CPU scheduling decisions may take place when a process:
- Switches from running to waiting state
- Switches from running to ready state
- Switches from waiting to ready
- Terminates

8.6 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

8.6 Scheduling Criteria

- 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)

8.6 Optimization Criteria

- Maximize CPU utilization
- Maximize throughput
- Minimize turnaround time
- Minimize waiting time
- Minimize response time

8.6 Scheduling Criteria

- Processor-bound processes
  - Use all available processor time
- I/O-bound
  - Generates an I/O request quickly and relinquishes processor
- Batch processes
  - Contains work to be performed with no user interaction
- Interactive processes
  - Requires frequent user input

8.7 Scheduling Algorithms

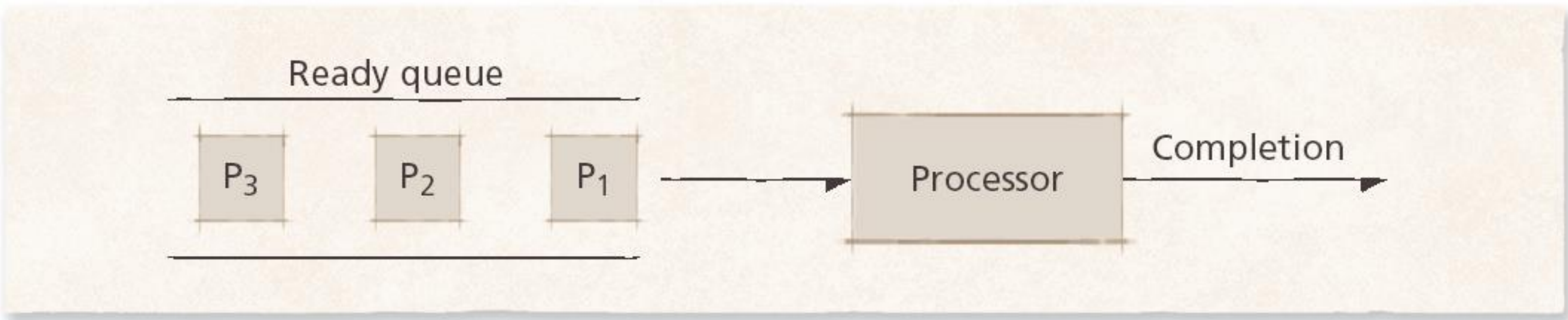
- Scheduling algorithms
  - Decide when and for how long each process runs
  - Make choices about
    - Preemptibility
    - Priority
    - Running time
    - Run-time-to-completion
    - fairness

8.7.1 First-In-First-Out (FIFO) Scheduling

- FIFO scheduling
  - Simplest scheme
  - Processes dispatched according to arrival time
  - Nonpreemptible
  - Rarely used as primary scheduling algorithm

8.7.1 First-In-First-Out (FIFO) Scheduling

Figure 8.2 First-in-first-out scheduling.

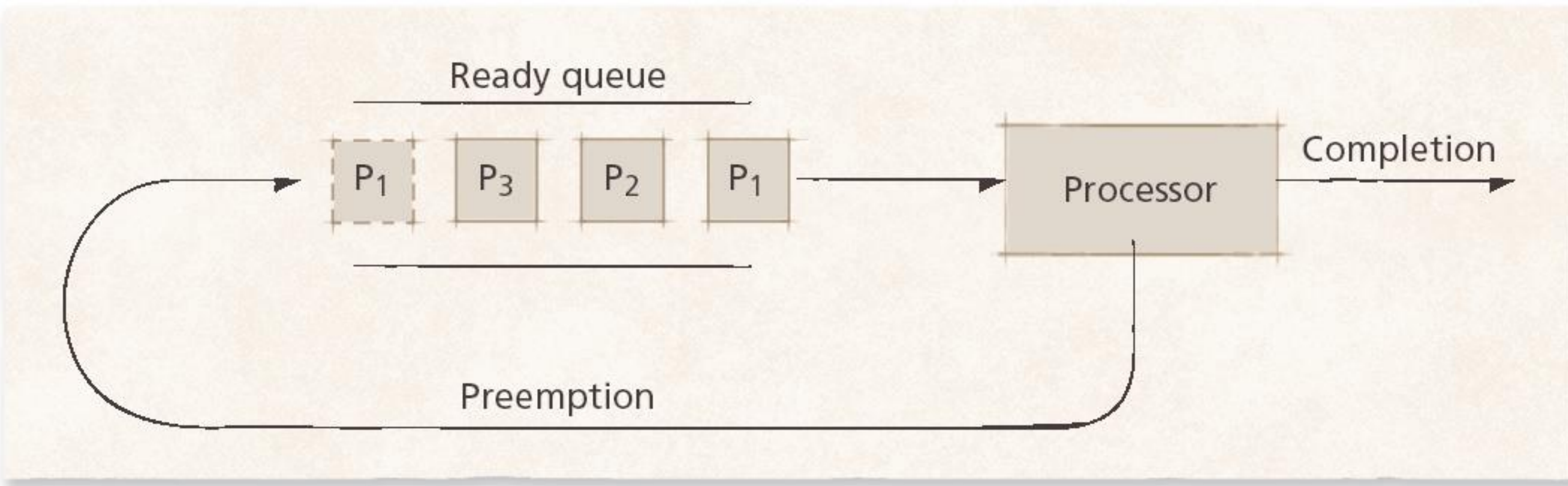


8.7.2 Round-Robin (RR) Scheduling

- Round-robin scheduling
  - Based on FIFO
  - Processes run only for a limited amount of time called a time slice or quantum
  - Preemptible
  - Requires the system to maintain several processes in memory to minimize overhead
  - Often used as part of more complex algorithms

8.7.2 Round-Robin (RR) Scheduling

Figure 8.3 Round-robin scheduling.



8.7.2 Round-Robin (RR) Scheduling

- Selfish round-robin scheduling
  - Increases priority as process ages
  - Two queues
    - Active
    - Holding
  - Favors older processes to avoids unreasonable delays

8.7.2 Round-Robin (RR) Scheduling

- Quantum size
  - Determines response time to interactive requests
  - Very large quantum size
    - Processes run for long periods
    - Degenerates to FIFO
  - Very small quantum size
    - System spends more time context switching than running processes
  - Middle-ground
    - Long enough for interactive processes to issue I/O request
    - Batch processes still get majority of processor time

8.7.3 Shortest-Process or Job-First (SPF/SJF) Scheduling

- Scheduler selects process with smallest time to finish
  - Lower average wait time than FIFO
    - Reduces the number of waiting processes
  - Potentially large variance in wait times
  - Nonpreemptive
    - Results in slow response times to arriving interactive requests
  - Relies on estimates of time-to-completion
    - Can be inaccurate or falsified
  - Unsuitable for use in modern interactive systems

8.7.4 Priority Scheduling

- Priority scheduling
  - Preemptive or Non-Preemptive version of SPF/SJF
  - Pick processes that has highest priority
  - Priorities are assigned in numeric form. e.g. 1 to 10
  - If preemptive, then a high priority job can remove a low priority job from the CPU and take over
  - In non-preemptive, once a process gets the CPU, it will finish its work and then release the CPU

8.7.5 Multilevel Feedback Queues

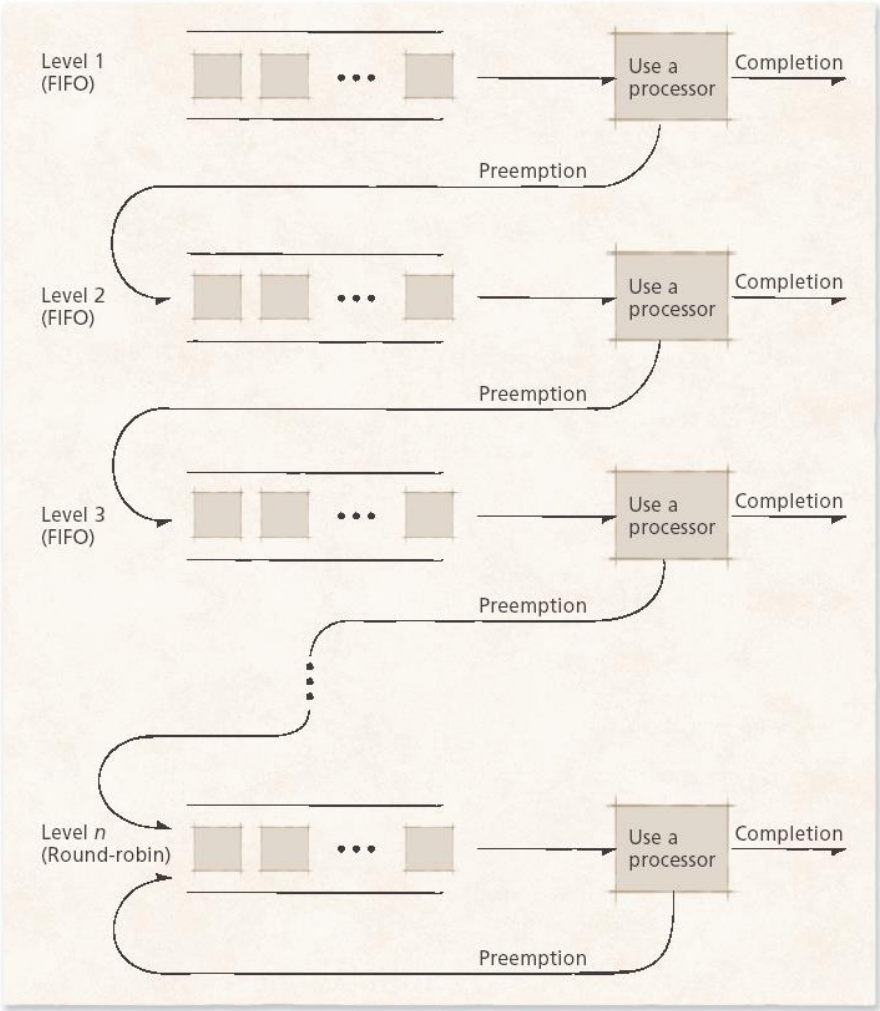
- Different processes have different needs
  - Short I/O-bound interactive processes should generally run before processor-bound batch processes
  - Behavior patterns not immediately obvious to the scheduler
- Multilevel feedback queues
  - Arriving processes enter the highest-level queue and execute with higher priority than processes in lower queues
  - Long processes repeatedly descend into lower levels
    - Gives short processes and I/O-bound processes higher priority
    - Long processes will run when short and I/O-bound processes terminate
  - Processes in each queue are serviced using round-robin
    - Process entering a higher-level queue preempt running processes

8.7.5 Multilevel Feedback Queues

- Algorithm must respond to changes in environment
  - Move processes to different queues as they alternate between interactive and batch behavior
- Example of an adaptive mechanism
  - Adaptive mechanisms incur overhead that often is balance by increased sensitivity to process behavior

8.7.5 Multilevel Feedback Queues

Figure 8.4 Multilevel feedback queues.

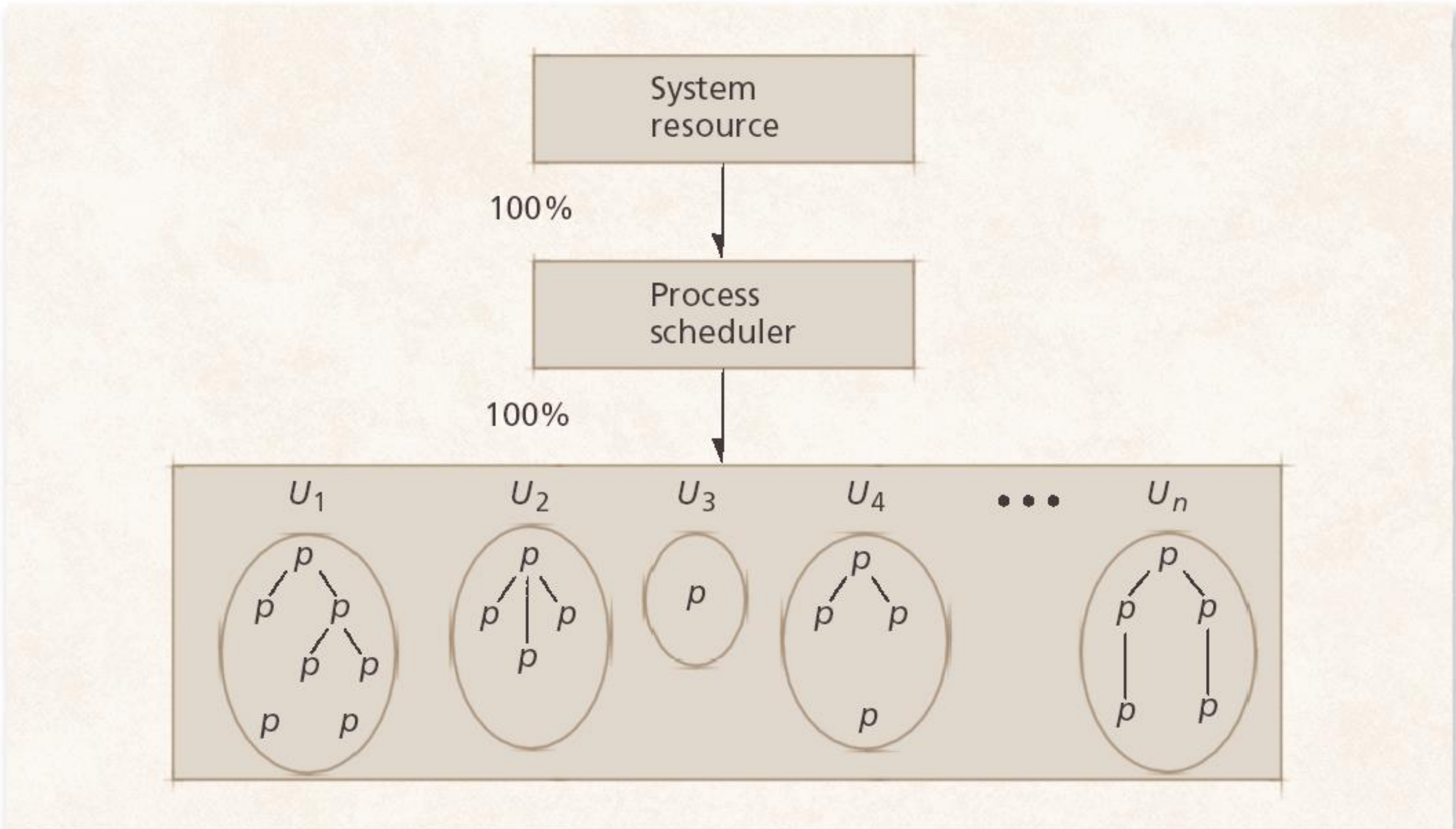


8.7.6 Fair Share Scheduling

- FSS controls users' access to system resources
  - Some user groups more important than others
  - Ensures that less important groups cannot exploit resources
  - Unused resources distributed according to the proportion of resources each group has been allocated
  - Groups not meeting resource-utilization goals get higher priority

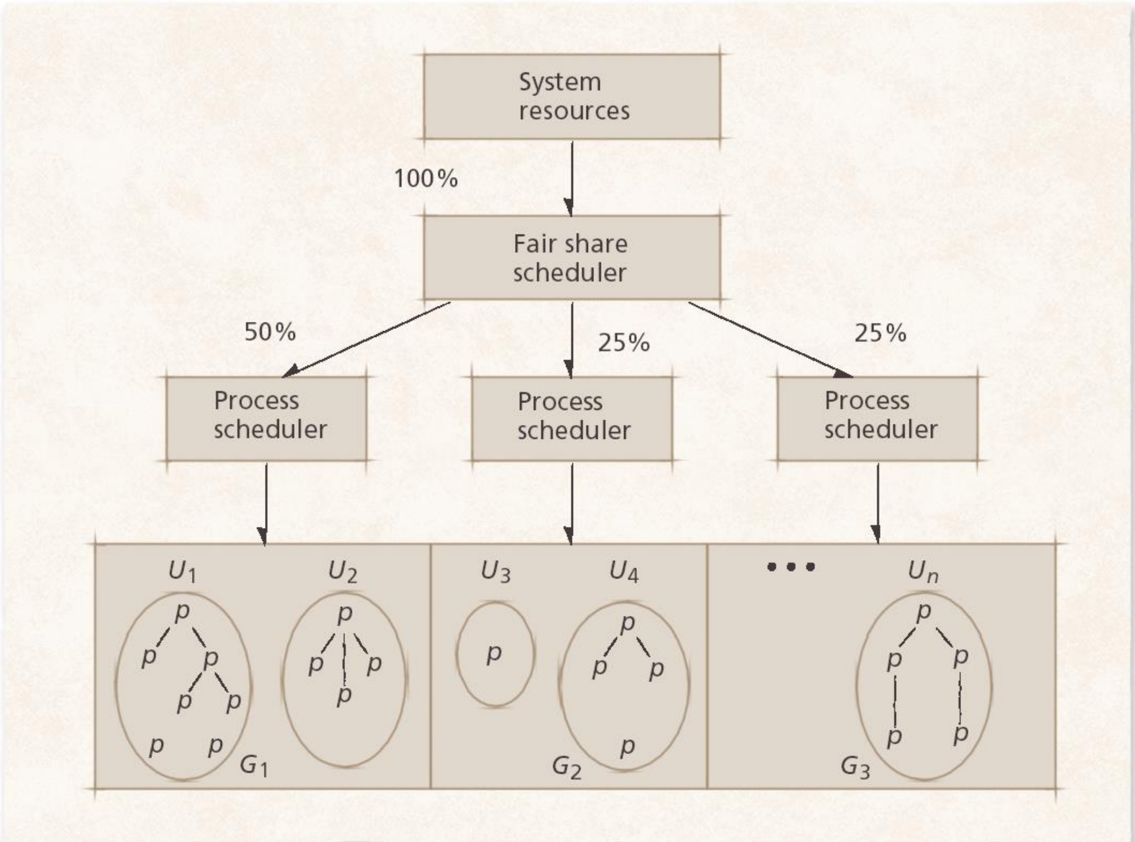
8.7.6 Fair Share Scheduling

Figure 8.5 Standard UNIX process scheduler. The scheduler grants the processor to users, each of whom may have many processes. (Property of AT&T Archives. Reprinted with permission of AT&T.)



8.7.6 Fair Share Scheduling

Figure 8.6 Fair share scheduler. The fair share scheduler divides system resource capacity into portions, which are then allocated by process schedulers assigned to various fair share groups. (Property of AT&T Archives. Reprinted with permission of AT&T.)



8.7 Deadline Scheduling

- Deadline scheduling
  - Process must complete by specific time
  - Used when results would be useless if not delivered on-time
  - Difficult to implement
    - Must plan resource requirements in advance
    - Incurs significant overhead
    - Service provided to other processes can degrade

8.8 Real-Time Scheduling

- Real-time scheduling
  - Related to deadline scheduling
  - Processes have timing constraints
  - Also encompasses tasks that execute periodically
- Two categories
  - Soft real-time scheduling
    - Does not guarantee that timing constraints will be met
    - For example, multimedia playback
  - Hard real-time scheduling
    - Timing constraints will always be met
    - Failure to meet deadline might have catastrophic results
    - For example, air traffic control

8.8 Real-Time Scheduling

- Static real-time scheduling
  - Does not adjust priorities over time
  - Low overhead
  - Suitable for systems where conditions rarely change
    - Hard real-time schedulers
  - Rate-monotonic (RM) scheduling
    - Process priority increases monotonically with the frequency with which it must execute
  - Deadline RM scheduling
    - Useful for a process that has a deadline that is not equal to its period

8.8 Real-Time Scheduling

- Dynamic real-time scheduling
  - Adjusts priorities in response to changing conditions
  - Can incur significant overhead, but must ensure that the overhead does not result in increased missed deadlines
  - Priorities are usually based on processes' deadlines
    - Earliest-deadline-first (EDF)
      - Preemptive, always dispatch the process with the earliest deadline
    - Minimum-laxity-first
      - Similar to EDF, but bases priority on laxity, which is based on the process's deadline and its remaining run-time-to-completion

8.9 Java Thread Scheduling

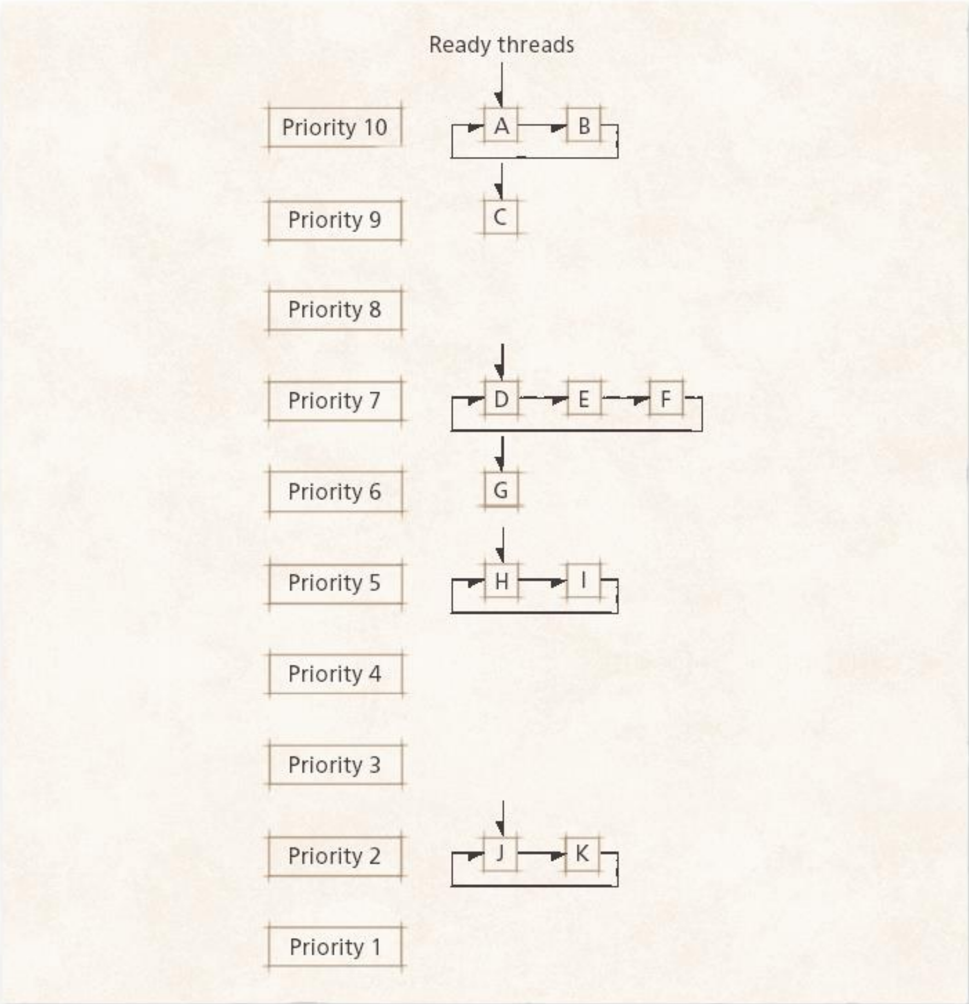
- Operating systems provide varying thread scheduling support
  - User-level threads
    - Implemented by each program independently
    - Operating system unaware of threads
  - Kernel-level threads
    - Implemented at kernel level
    - Scheduler must consider how to allocate processor time to a process's threads

8.9 Java Thread Scheduling

- Java threading scheduler
  - Uses kernel-level threads if available
  - User-mode threads implement timeslicing
    - Each thread is allowed to execute for at most one quantum before preemption
  - Threads can `yield` to others of equal priority
    - Only necessary on nontimesliced systems
    - Threads waiting to run are called waiting, sleeping or blocked

8.9 Java Thread Scheduling

Figure 8.7 Java thread priority scheduling.



Chapter 7 – Deadlock and Indefinite Postponement

Outline

Chapter 7 – Deadlock and Indefinite Postponement

Objectives

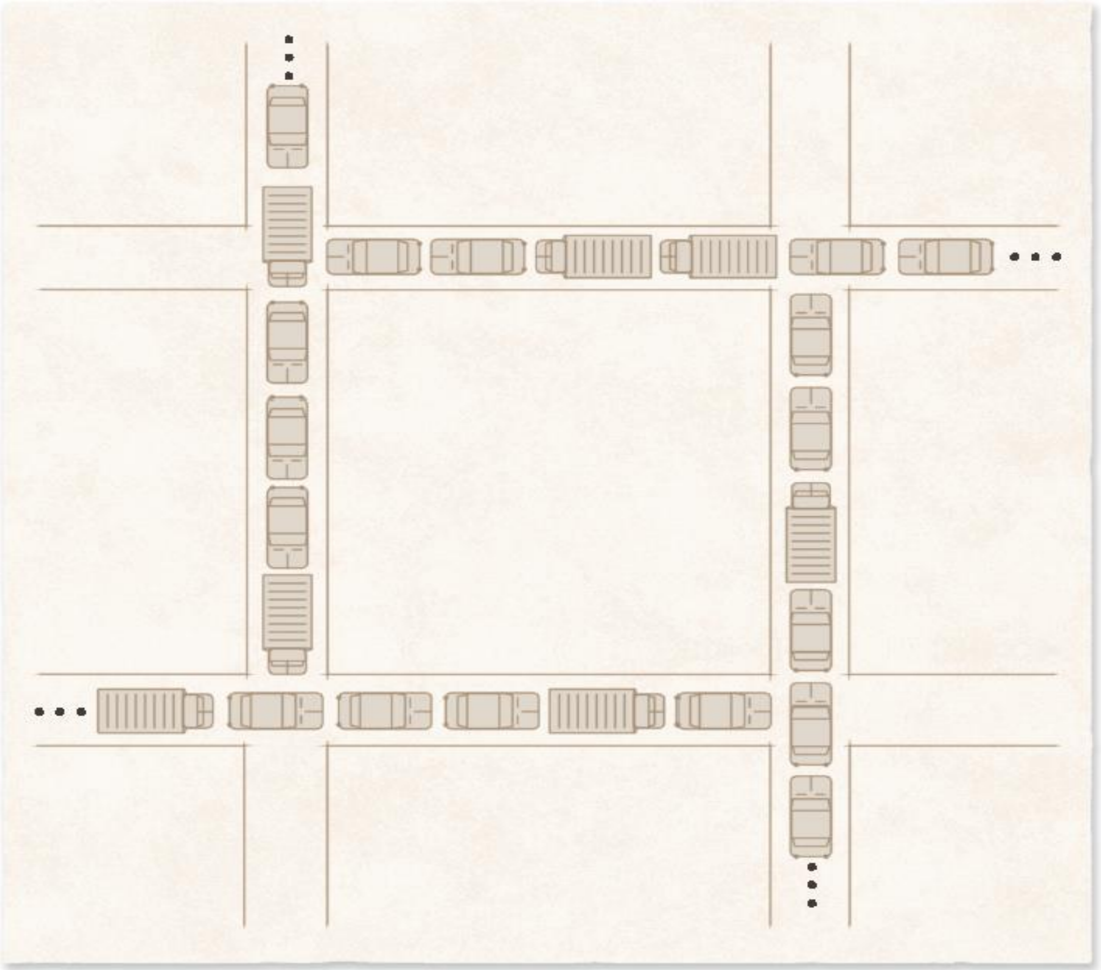
- After reading this chapter, you should understand:
  - the problem of deadlock.
  - the four necessary conditions for deadlock to exist.
  - the problem of indefinite postponement.
  - the notions of deadlock prevention, avoidance, detection and recovery.
  - algorithms for deadlock avoidance and detection.
  - how systems can recover from deadlocks.

7.1 Introduction

- Deadlock
  - A process or thread is waiting for a particular event that will not occur
- System deadlock
  - One or more processes are deadlocked

7.2.1 Traffic Deadlock

Figure 7.1 Traffic deadlock example.



7.2.2 Simple Resource Deadlock

- Most deadlocks develop because of the normal contention for dedicated resources
- Circular wait is characteristic of deadlocked systems

Deadlock Characterization

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 {P0, P1, …, P0} of waiting processes such that P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, …, Pn–1 is waiting for a resource that is held by Pn, and P0 is waiting for a resource that is held by P0.

Deadlock Prevention

Restrain the ways request can be made.

- **Mutual Exclusion –** not required for sharable resources; must hold for nonsharable resources.

- **Hold and Wait –** must guarantee that whenever a process requests a resource, it does not hold any other resources.
- Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none.
- Low resource utilization; starvation possible.

Deadlock Prevention (Cont.)

- **No Preemption –** If a process that is holding some resources 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.

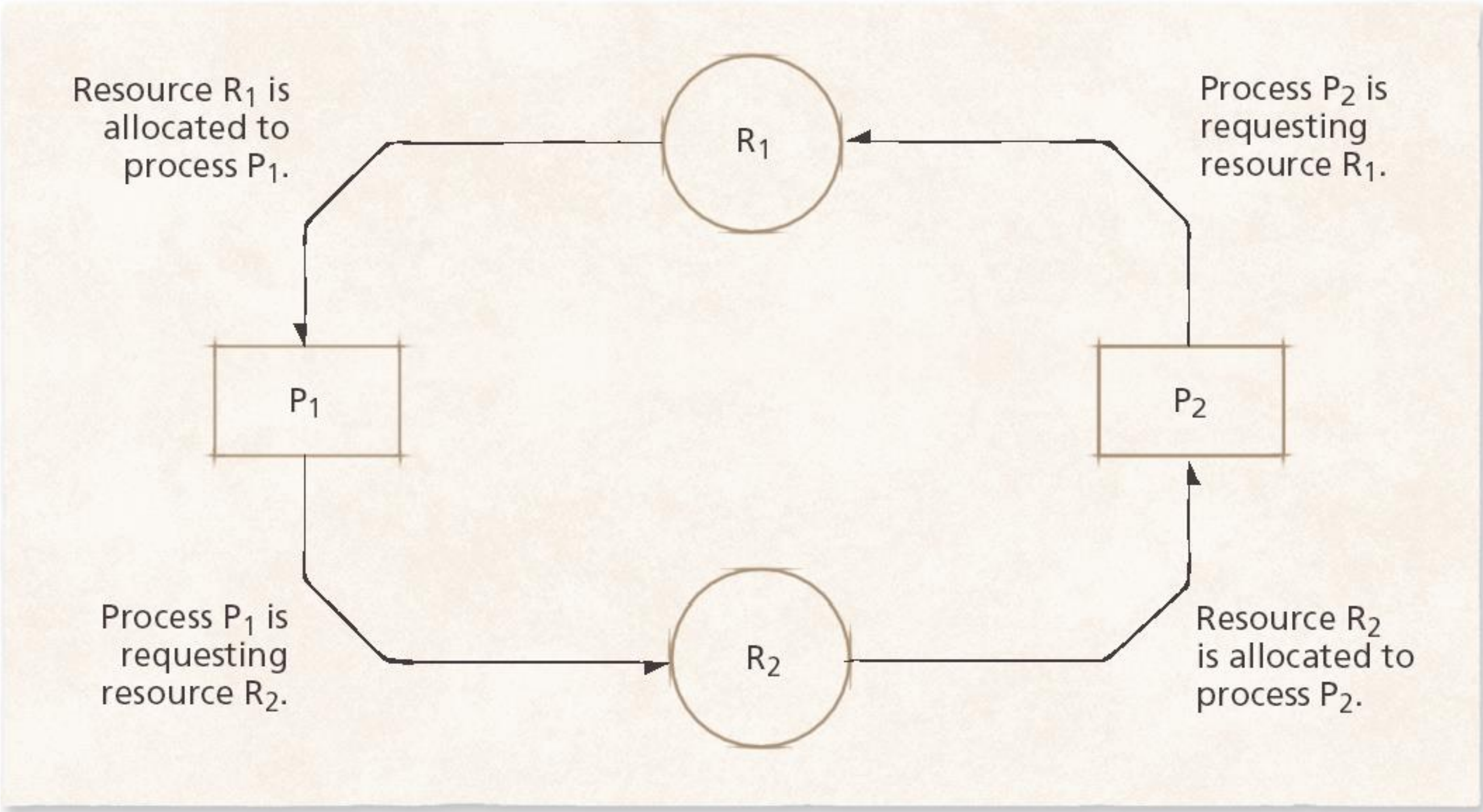
- **Circular Wait –** impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

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.

7.2.2 Simple Resource Deadlock

Figure 7.2 Resource deadlock example. This system is deadlocked because each process holds a resource being requested by the other process and neither process is willing to release the resource it holds.



7.2.3 Deadlock in Spooling Systems

- Spooling systems are prone to deadlock
- Common solution
  - Restrain input spoolers so that when the spooling file begins to reach some saturation threshold, the spoolers do not read in more print jobs
- Today's systems
  - Printing begins before the job is completed so that a full spooling file can be emptied even while a job is still executing
  - Same concept has been applied to streaming audio and video

7.2.4 Example: Dining Philosophers

- Problem statement:

  Five philosophers sit around a circular table. Each leads a simple life alternating between thinking and eating spaghetti. In front of each philosopher is a dish of spaghetti that is constantly replenished (refilled) by a dedicated wait staff. There are exactly five forks on the table, one between each adjacent pair of philosophers. Eating spaghetti (in the most proper manner) requires that a philosopher use both adjacent forks (simultaneously). Develop a concurrent program free of deadlock and indefinite postponement that models the activities of the philosophers.

7.2.4 Example: Dining Philosophers

Figure 7.3 Dining philosopher behavior.

```
void typicalPhilosopher()
{
   while ( true )
   {
      think();
      eat();
   } // end while

} // end typicalPhilospher
```

7.2.4 Example: Dining Philosophers

- Constraints:
  - To prevent philosophers from starving:
    - Free of deadlock
    - Free of indefinite postponement
  - Enforce mutual exclusion
    - Two philosophers cannot use the same fork at once
- The problems of mutual exclusion, deadlock and indefinite postponement lie in the implementation of method `eat`.

7.2.4 Example: Dining Philosophers

Figure 7.4 Implementation of method `eat`.

```
void eat()
{
   pickUpLeftFork();
   pickUpRightFork();
   eatForSomeTime();
   putDownRightFork();
   putDownLeftFork();
} // eat
```

7.3 Related Problem: Indefinite Postponement

- Indefinite postponement
  - Also called indefinite blocking or starvation
  - Occurs due to biases in a system’s resource scheduling policies
- Aging
  - Technique that prevents indefinite postponement by increasing process’s priority as it waits for resource

7.4 Resource Concepts

- Preemptible resources (e.g. processors and main memory)
  - Can be removed from a process without loss of work
- Nonpreemptible resources (e.g. tape drives and optical scanners)
  - Cannot be removed from the processes to which they are assigned without loss of work
- Reentrant code
  - Cannot be changed while in use
  - May be shared by several processes simultaneously
- Serially reusable code
  - May be changed but is reinitialized each time it is used
  - May be used by only one process at a time

7.5 Four Necessary Conditions for Deadlock

- Mutual exclusion condition
  - Resource may be acquired exclusively by only one process at a time
- Wait-for condition (hold-and-wait condition)
  - Process that has acquired an exclusive resource may hold that resource while the process waits to obtain other resources
- No-preemption condition
  - Once a process has obtained a resource, the system cannot remove it from the process's control until the process has finished using the resource
- Circular-wait condition
  - Two or more processes are locked in a "circular chain" in which each process is waiting for one or more resources that the next process in the chain is holding

7.6 Deadlock Solutions

- Four major areas of interest in deadlock research
  - Deadlock prevention
  - Deadlock avoidance
  - Deadlock detection
  - Deadlock recovery

7.7 Deadlock Prevention

- Deadlock prevention
  - Condition a system to remove any possibility of deadlocks occurring
  - Deadlock cannot occur if any one of the four necessary conditions is denied
  - First condition (mutual exclusion) cannot be broken

7.7.1 Denying the "Wait-For" Condition

- When denying the "wait-for condition"
  - All of the resources a process needs to complete its task must be requested at once
  - This leads to inefficient resource allocation

7.7.2 Denying the “No-Preemption” Condition

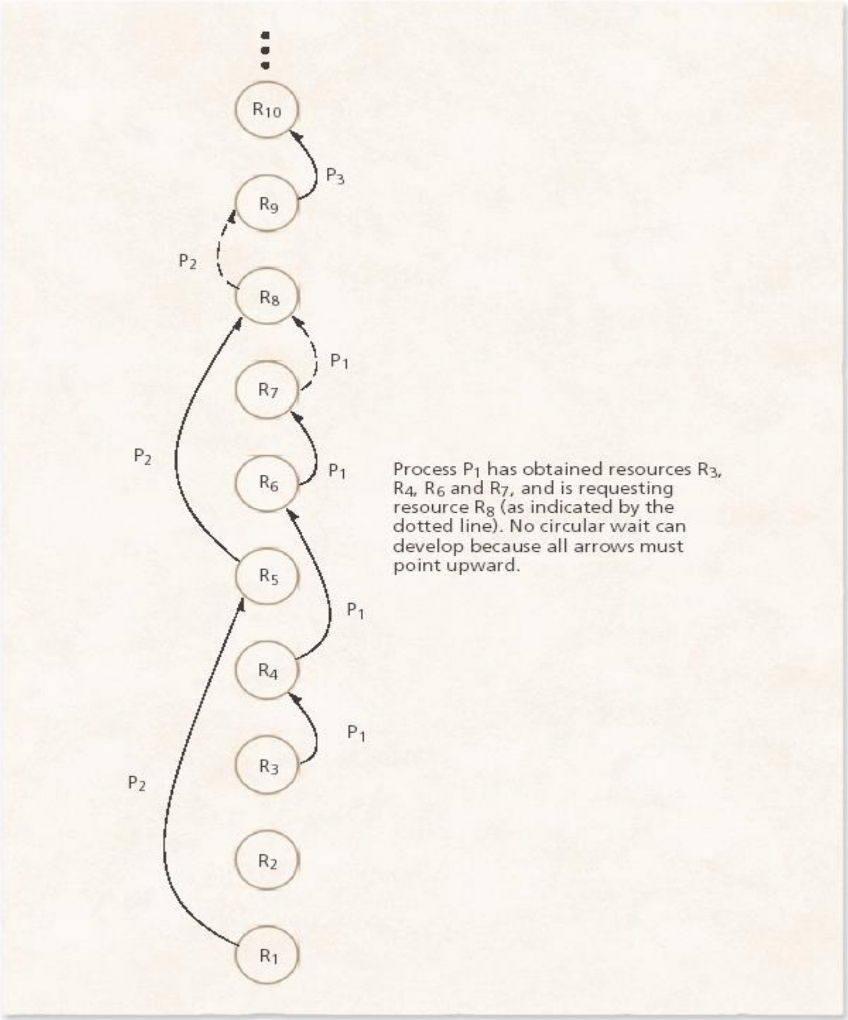
- When denying the “no-preemption” condition
  - Processes may lose work when resources are preempted
  - This can lead to substantial overhead as processes must be restarted

7.7.3 Denying the “Circular-Wait” Condition

- Denying the “circular-wait” condition:
  - Uses a linear ordering of resources to prevent deadlock
  - More efficient resource utilization than the other strategies
- Drawbacks
  - Not as flexible or dynamic as desired
  - Requires the programmer to determine the ordering or resources for each system

7.7.3 Denying the "Circular-Wait" Condition

Figure 7.5 Havender's linear ordering of resources for preventing deadlock.



7.8 Deadlock Avoidance with Dijkstra's Banker's Algorithm

- Banker's Algorithm
  - Impose less stringent conditions than in deadlock prevention in an attempt to get better resource utilization
  - Safe state
    - Operating system can guarantee that all current processes can complete their work within a finite time
  - Unsafe state
    - Does not imply that the system is deadlocked, but that the OS cannot guarantee that all current processes can complete their work within a finite time

7.8 Deadlock Avoidance with Dijkstra's Banker's Algorithm

- Banker's Algorithm (cont.)
  - Requires that resources be allocated to processes only when the allocations result in safe states.
  - It has a number of weaknesses (such as requiring a fixed number of processes and resources) that prevent it from being implemented in real systems

7.8.2 Example of an Unsafe State

Figure 7.6 Safe state.

Process	$max(P_i)$ (maximum need)	$loan(P_i)$ (current loan)	$claim(P_i)$ (current claim)
P_1	4	1	3
P_2	6	4	2
P_3	8	5	3
Total resources, t, = 12		*Available resources, a, = 2*	

7.8.2 Example of an Unsafe State

Figure 7.7 Unsafe state.

Process	$max(P_i)$ (maximum need)	$loan(P_i)$ (current loan)	$claim(P_i)$ (current claim)
P_1	10	8	2
P_2	5	2	3
P_3	3	1	2
Total resources, t, = 12		*Available resources, a, = 1*	

7.8.3 Example of Safe-State-to-Unsafe-State Transition

- Safe-state-to-unsafe-state transition:
  - Suppose the current state of a system is safe, as shown in Fig. 7.6.
  - The current value of *a* is 2.
  - Now suppose that process P_3 requests an additional resource

7.8.3 Example of Safe-State-to-Unsafe-State Transition

Figure 7.8 Safe-state-to-unsafe-state transition.

Process	max(P_i) (maximum need)	loan(P_i) (current loan)	claim(P_i) (current claim)
P_1	4	1	3
P_2	6	4	2
P_3	8	6	2
Total resources, t, = 12		*Available resources, a, = 1*	

7.8.4 Banker's Algorithm Resource Allocation

- Is the state in the next slide safe?

7.8.4 Banker's Algorithm Resource Allocation

Figure 7.9 State description of three processes.

Process	$max(P_i)$	$loan(P_i)$	$claim(P_i)$
P_1	5	1	4
P_2	3	1	2
P_3	10	5	5
		$a = 2$	

7.8.4 Banker's Algorithm Resource Allocation

- Answer:
  - There is no guarantee that all of these processes will finish
    - P_2 will be able to finish by using up the two remaining resources
    - Once P_2 is done, there are only three available resources left
    - This is not enough to satisfy either P_1's claim of 4 or P_3's claim of five

7.8.5 Weaknesses in the Banker's Algorithm

- Weaknesses
  - Requires there be a fixed number of resource to allocate
  - Requires the population of processes to be fixed
  - Requires the banker to grant all requests within "finite time"
  - Requires that clients repay all loans within "finite time"
  - Requires processes to state maximum needs in advance

7.9 Deadlock Detection

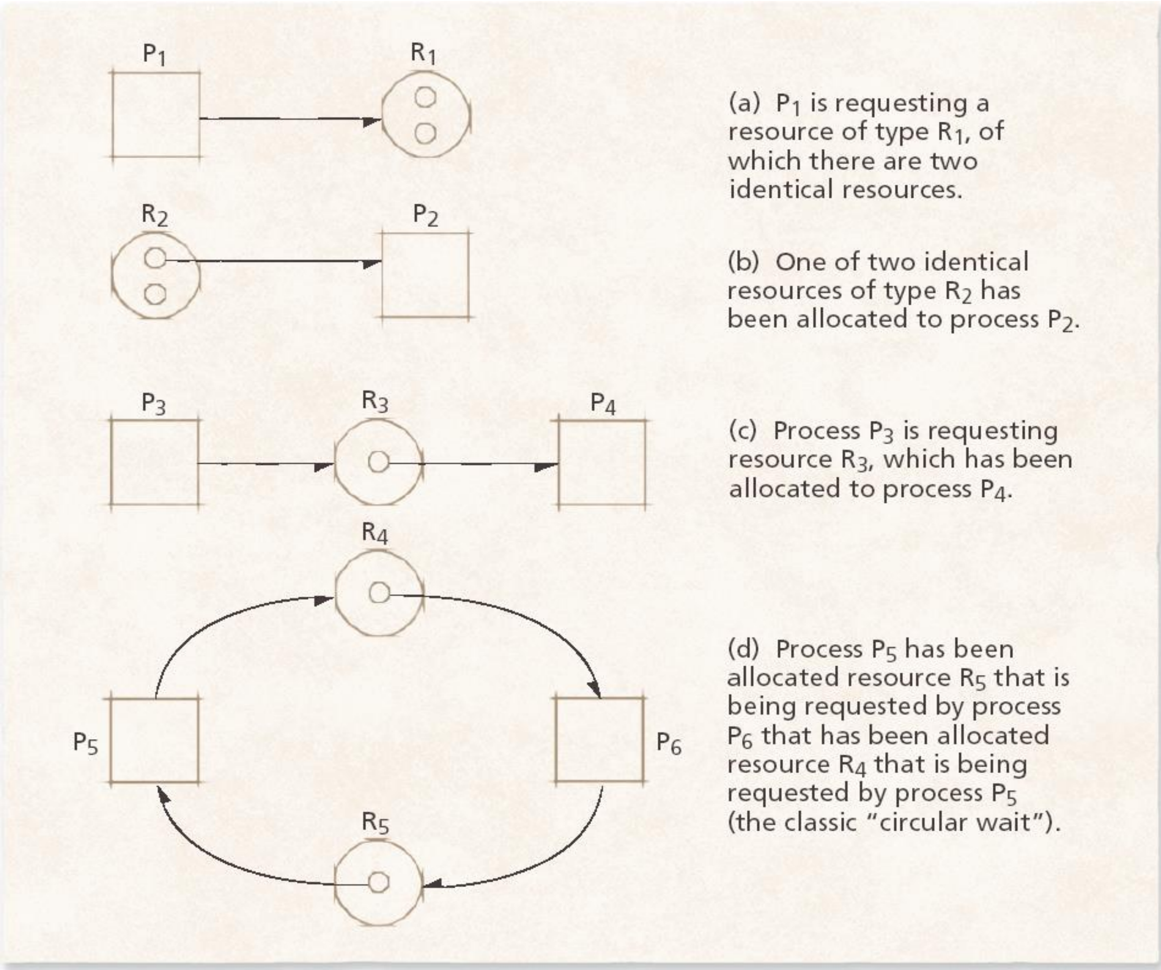
- Deadlock detection
  - Used in systems in which deadlocks can occur
  - Determines if deadlock has occurred
  - Identifies those processes and resources involved in the deadlock
  - Deadlock detection algorithms can incur significant runtime overhead

7.9.1 Resource-Allocation Graphs

- Resource-allocation graphs
  - Squares
    - Represent processes
  - Large circles
    - Represent classes of identical resources
  - Small circles drawn inside large circles
    - Indicate separate identical resources of each class

7.9.1 Resource-Allocation Graphs

Figure 7.10 Resource-allocation and request graphs.

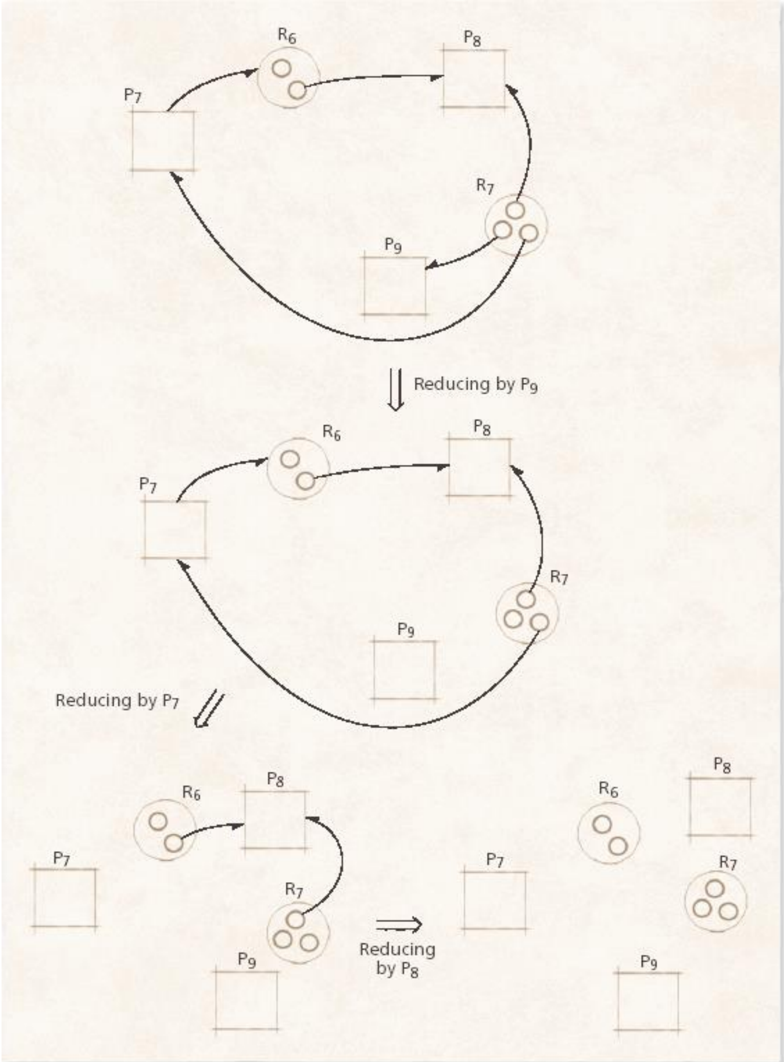


7.9.2 Reduction of Resource-Allocation Graphs

- Graph reductions
  - If a process's resource requests may be granted, the graph may be reduced by that process
  - If a graph can be reduced by all its processes, there is no deadlock
  - If a graph cannot be reduced by all its processes, the irreducible processes (complex processes) constitute the set of deadlocked processes in the graph

7.9.2 Reduction of Resource-Allocation Graphs

Figure 7.11 Graph reductions determining that no deadlock exists.



7.10 Deadlock Recovery

- Deadlock recovery
  - Clears deadlocks from system so that deadlocked processes may complete their execution and free their resources
- Suspend/resume mechanism
  - Allows system to put a temporary hold on a process
  - Suspended processes can be resumed without loss of work
- Checkpoint/rollback
  - Facilitates suspend/resume capabilities
  - Limits the loss of work to the time the last checkpoint was made

7.11 Deadlock Strategies in Current and Future Systems

- Deadlock is viewed as limited annoyance in personal computer systems
  - Some systems implement basic prevention methods suggested by Havender
  - Some others ignore the problem, because checking deadlocks would reduce systems' performance
- Deadlock continues to be an important research area