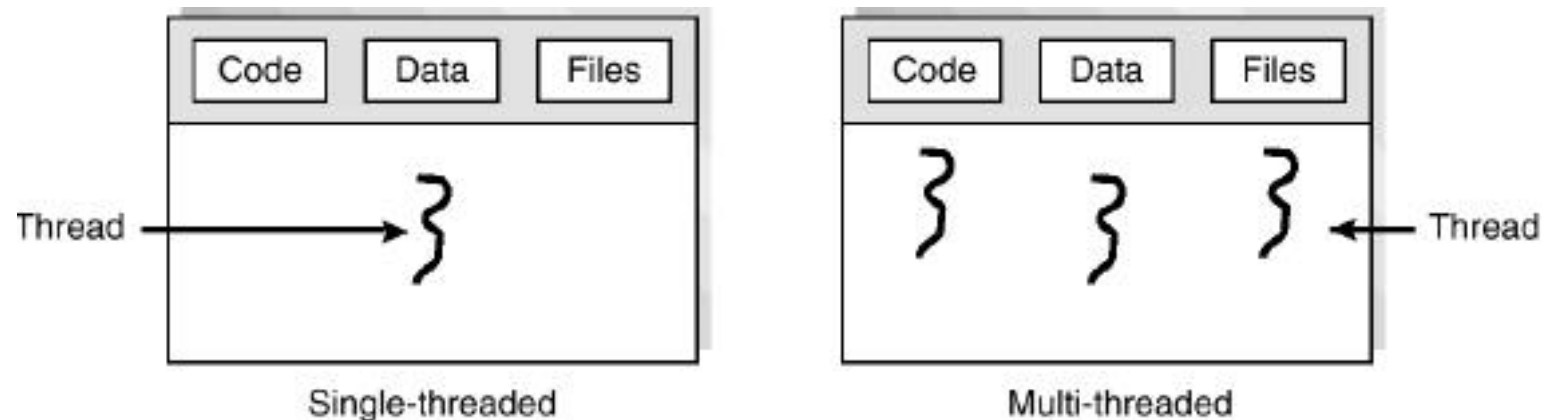


Review of Chap.s 6-8
Applied Operating System Concepts

-

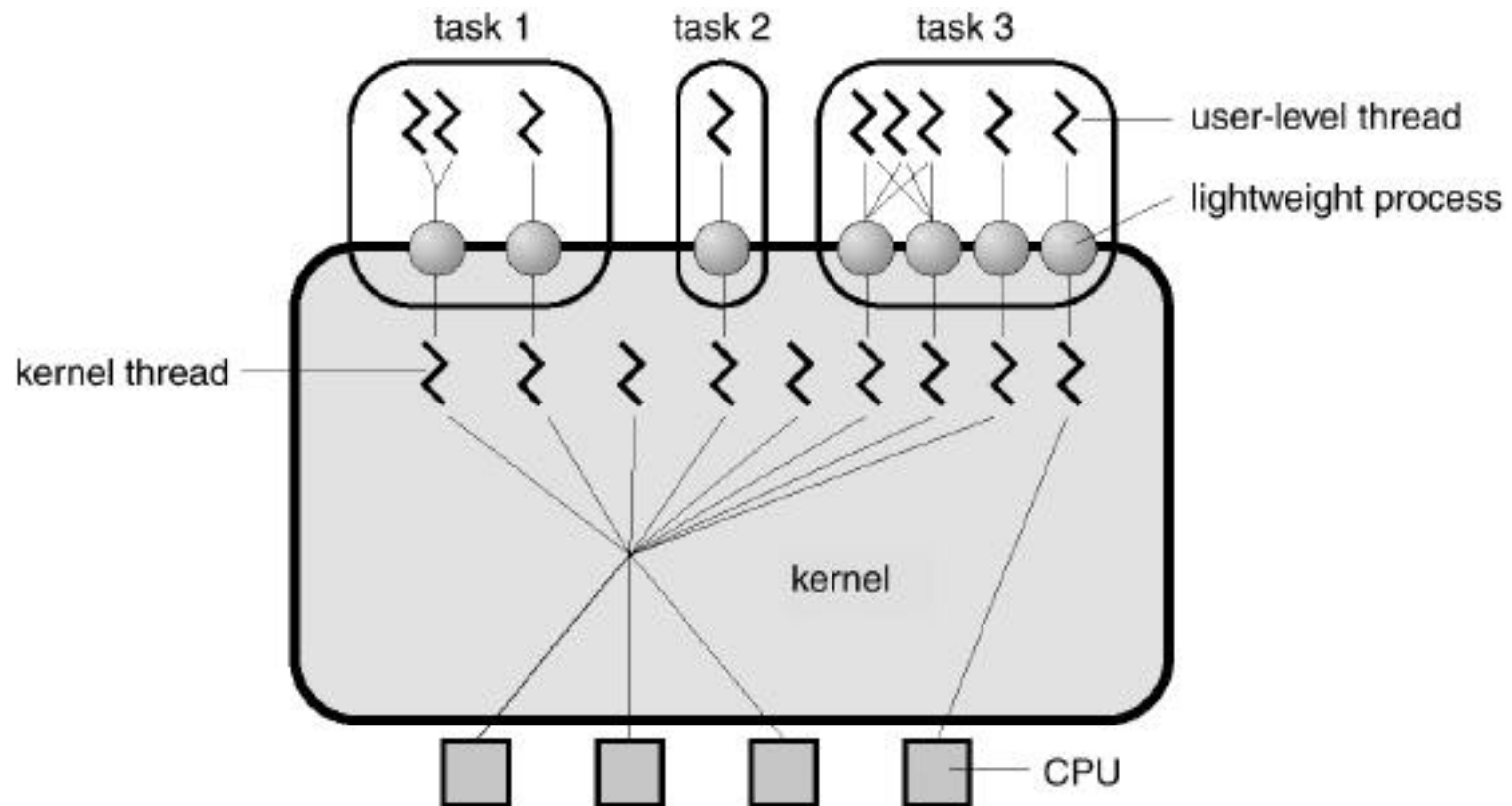
Scheduling, Synchronization, Deadlocks
ECE3055a, Spring 3055

Module 5: Threads



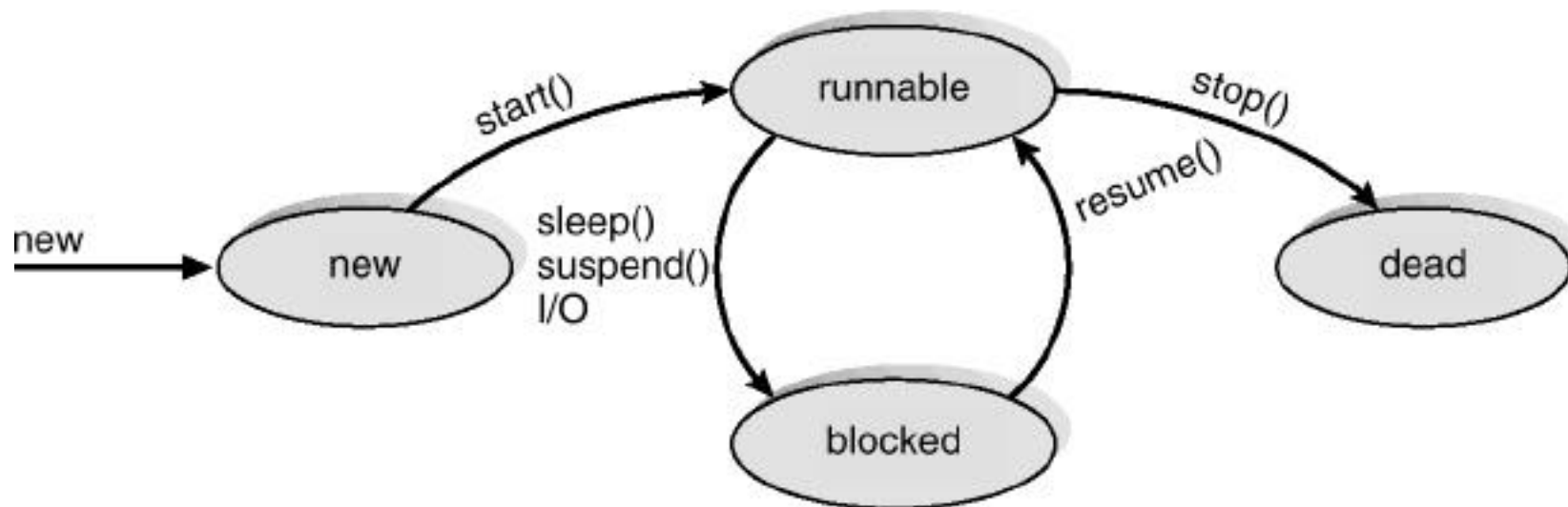
- Thread Management Done by User-Level Threads Library
- Examples
 - POSIX *Pthreads*
 - Mach *C-threads*
 - Solaris *threads*
- Supported by the Kernel
- Examples
 - Windows 95/98/NT
 - Solaris
 - Digital UNIX

Solaris 2 Threads

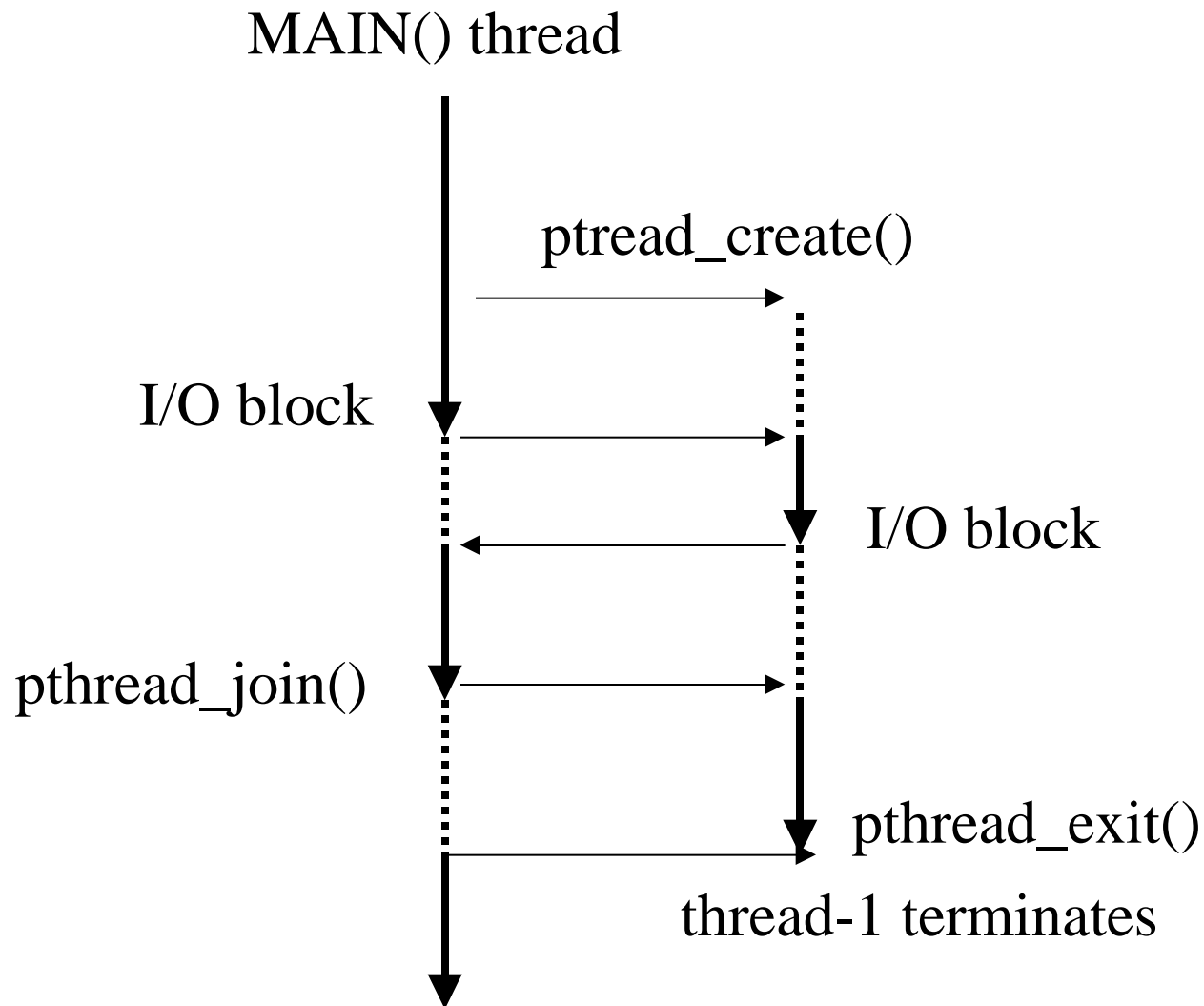


Java Thread Management

- **suspend()** – suspends execution of the currently running thread.
- **sleep()** – puts the currently running thread to sleep for a specified amount of time.
- **resume()** – resumes execution of a suspended thread.
- **stop()** – stops execution of a thread.



UNIX (POSIX) THREAD MANAGEMENT



Classical Problems

Producer-Consumer (Bounded-Buffer)

Readers-Writers

Dining Philosophers

Resource Allocation

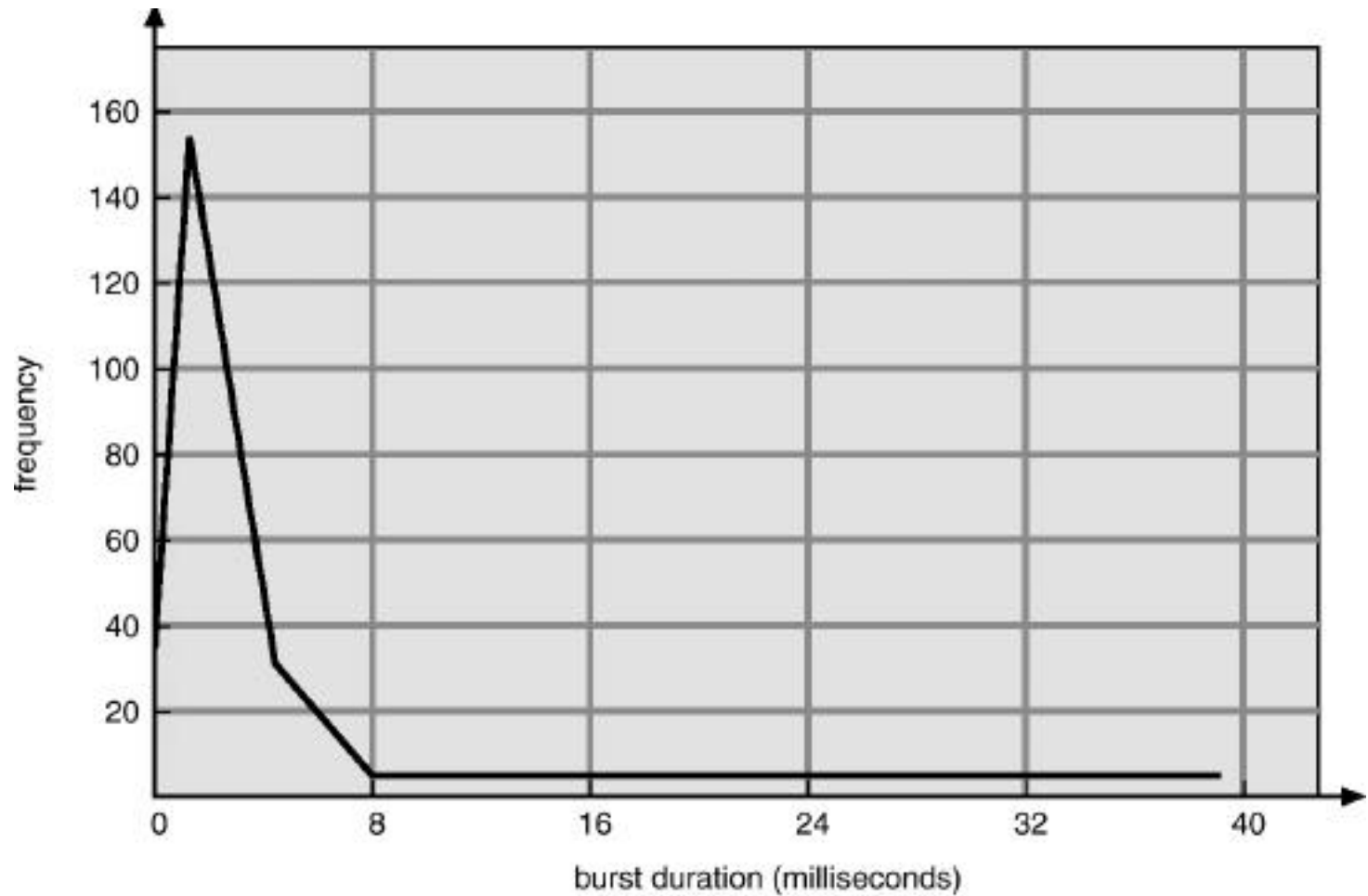
Mutual Exclusion

Critical Sections

Module 6: CPU Scheduling

- Basic Concepts
 - Maximum CPU utilization obtained with multiprogramming
 - CPU-I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait.
 - CPU burst distribution
- Scheduling Criteria
- Scheduling Algorithms
- Multiple-Processor Scheduling
- Real-Time Scheduling
- Algorithm Evaluation

Histogram of CPU-burst Times



CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.
- CPU scheduling decisions may take place when a process:
 1. Switches from running to waiting state.
 2. Switches from running to ready state.
 3. Switches from waiting to ready.
 4. Terminates.
- Scheduling under 1 and 4 is *nonpreemptive*.
- All other scheduling is *preemptive*.

Find the order of processing and the run times for
P1 (3 ticks), P2 (5 ticks), P3 (4 ticks), and P4 (1 tick)
using (delta = 2 ticks, *where applicable)

First-Come, First-Served (FCFS) Scheduling

Shortest-Job-First (SJR) Scheduling

Preemptive*

Non-preemptive

Round Robin*

=====

Find the exponential average T of the last 5 burst lengths
(67, 89, 13, 56, 45) using a factor $a = 0.8$ (67 is most recent)

$$\begin{aligned} T &= a \cdot 67 + a^2 \cdot 89 + a^3 \cdot 13 + a^4 \cdot 56 + a^5 \cdot 45 \\ &= a * (67 + a * (89 + a * (13 + a * (56 + a * (45 + \dots)))))) \end{aligned}$$

Find the next value if $t=76$ using one * and one + operation.

$$T = a * (76 + \text{<old value>})$$

Thread Scheduling

- Local Scheduling – How the threads library decides which thread to put onto an available LWP.
- Global Scheduling – How the kernel decides which kernel thread to run next.
- JAVA
 - JVM Uses a Preemptive, Priority-Based Scheduling Algorithm
 - FIFO Queue is Used if There Are Multiple Threads With the Same Priority.

JVM Schedules a Thread to Run When:

- The Currently Running Thread Exits the Runnable State.
 - A Higher Priority Thread Enters the Runnable State
- JVM Does Not Specify Whether Threads are Time-Sliced or Not.

Module 8: Deadlocks

System Model

Deadlock Characterization

Methods for Handling Deadlocks

Deadlock Prevention

Deadlock Avoidance

Deadlock Detection

Recovery from Deadlock

Combined Approach to Deadlock Handling

Deadlock can arise if four conditions hold simultaneously.

Mutual exclusion: only one process at a time can use a resource.

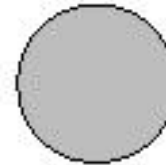
Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes.

No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task.

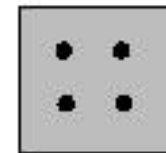
Circular wait: there exists a set $\{P_0, P_1, \dots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1 , P_1 is waiting for a resource that is held by P_2 , ...

Resource Allocation Graph

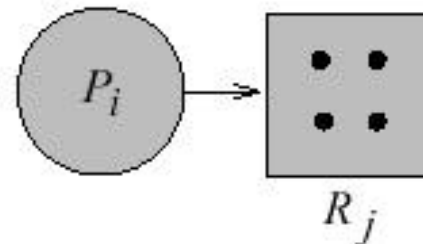
- Process



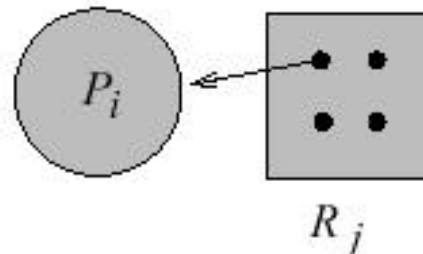
- Resource type with 4 instances



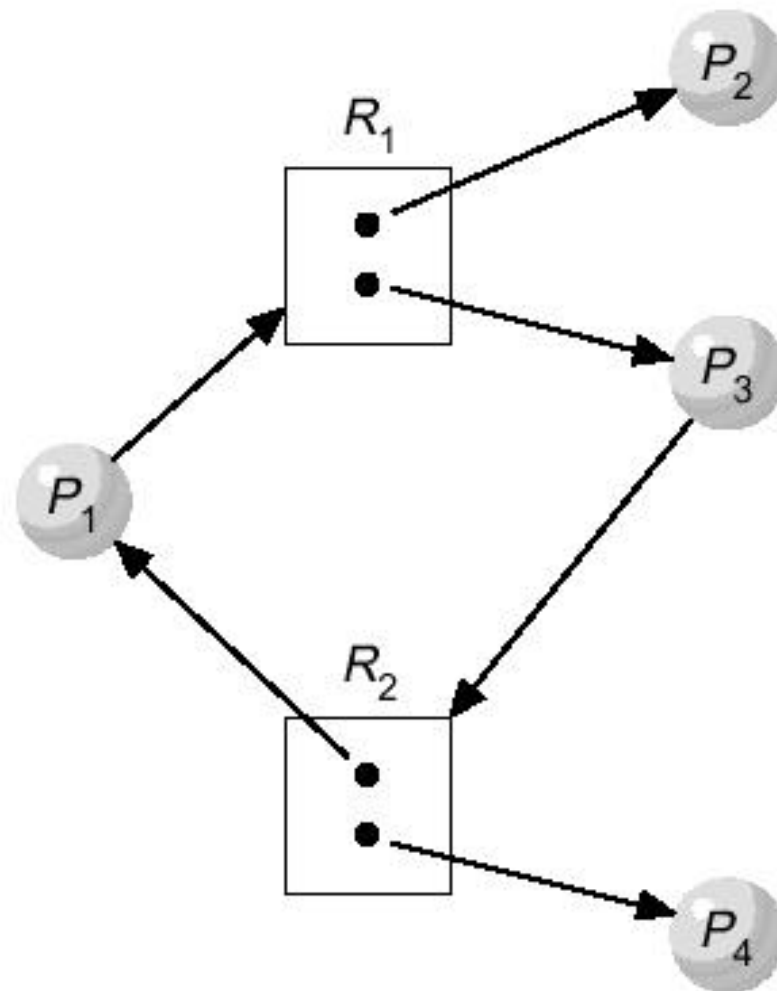
- P_i requests instance of R_j



- P_i is holding an instance of R_j



Example of a Graph With Cycle



Methods for Handling Deadlocks

Ensure that the system will never enter a deadlock state.

Allow the system to enter a deadlock state and then recover.

Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

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.

Example of Banker's Algorithm

- 5 processes P_0 through P_4 ; 3 resource types A (10 instances), B (5 instances), and C (7 instances).
- Snapshot at time T_0 :

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>		<u>Need</u>
	$A \ B \ C$	$A \ B \ C$	$A \ B \ C$		$A \ B \ C$
P_0	0 1 0	7 5 3	3 3 2	P_0	7 4 3
P_1	2 0 0	3 2 2		P_1	1 2 2
P_2	3 0 2	9 0 2		P_2	6 0 0
P_3	2 1 1	2 2 2		P_3	0 1 1
P_4	0 0 2	4 3 3		P_4	4 3 1

Which Order can P's Run? (P1, P3, P4, P2, P0)

What resources are available after P3 runs? (7 4 3)

Deadlock Detection

Allow system to enter deadlock state

Detection algorithm

Recovery scheme

Security

Must be considered in:

- Computer Hardware design
- Operating System Design
- Application Software Design
- All of the Above