



Thread Synchronization & Deadlock

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating threads
- Suppose that we wanted to provide a solution to the producer-consumer problem that fills **all** the buffers. We can do so by having an integer **count** that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true)
```

```
    /* produce an item and put in nextProduced */
```

```
        while (count == BUFFER_SIZE)
```

```
            ;// do nothing
```

```
        buffer [in] = nextProduced;
```

```
        in = (in + 1) % BUFFER_SIZE;
```

```
        count++;
```

```
    }
```

Consumer

```
while (1)
{
    while (count == 0)
        ; // do nothing
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
    /* consume the item in nextConsumed */
}
```

Race Condition

- `count++` could be implemented as

```
register1 = count
register1 = register1 + 1
count = register1
```

- `count--` could be implemented as

```
register2 = count
register2 = register2 - 1
count = register2
```

- Consider this execution interleaving with “count = 5” initially:

```
S0: producer execute register1 = count {register1 = 5}
S1: producer execute register1 = register1 + 1 {register1 = 6}
S2: consumer execute register2 = count {register2 = 5}
S3: consumer execute register2 = register2 - 1 {register2 = 4}
S4: producer execute count = register1 {count = 6}
S5: consumer execute count = register2 {count = 4}
```

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If thread T_i is executing in its critical section ***with respect to a particular resource***, then no other threads can be executing in their critical sections
2. **Progress** - If no thread is executing in its critical section and there exist some threads that wish to enter their critical section, then the selection of the thread that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other threads are allowed to enter their critical sections after a thread has made a request to enter its critical section and before that request is granted
 - Assume that each thread executes at a nonzero speed
 - No assumption concerning relative speed of the N threads

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
 - Currently running code would execute without preemption
 - Generally too inefficient on multiprocessor systems
 - Operating systems that use this technique are not broadly scalable
- Modern machines provide special atomic hardware instructions
 - Atomic = non-interruptable
 - Two types of atomic hardware instructions:
 - test memory word and set value **in one atomic operation**
 - swap contents of two memory words **in one atomic operation**

TestAndSet Instruction

- Definition of the functionality:

```
boolean TestAndSet (boolean *target) {  
    boolean rv = *target;  
    *target = TRUE;  
    return rv;  
}
```


Solution using TestAndSet

- Shared boolean variable **lock**, initialized to false.
- Solution:

```
do {  
    while ( TestAndSet (&lock ))  
        ; /* do nothing */
```

```
    //    critical section
```

```
    lock = FALSE;
```

```
    //    remainder section
```

```
} while ( TRUE);
```

Swap Instruction

- Definition of the functionality:

```
void Swap (boolean *a, boolean *b) {  
    boolean temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

Solution using Swap

- Shared Boolean variable **lock** initialized to FALSE; Each thread has a local Boolean variable **key**.

- Solution:

```
do {  
    key = TRUE;  
    while ( key == TRUE)  
        Swap (&lock, &key );
```

```
    //    critical section
```

```
    lock = FALSE;
```

```
    //    remainder section
```

```
} while ( TRUE);
```

Semaphore

- We need a synchronization tool that does not require busy waiting
- Semaphore S – integer variable
- Two standard operations modify S : `wait()` and `signal()`
 - Originally called `P()` and `V()` [Proberen(test) and Verhogen(increment) in Dutch]
- Less complicated than using TestAndSet and Swap
- A semaphore can only be accessed via two indivisible (atomic) operations
- Definition of the functionality of `wait()` and `signal()`
 - `wait (S) {`
 - `while S <= 0`
 - `; // no-op`
 - `S--;`
 - `}`
 - `signal (S) {`
 - `S++;`
 - `}`

Semaphore as General Synchronization Tool

- **Counting** semaphore – integer value can range over an unrestricted domain
- **Binary** semaphore – integer value can range only between 0 and 1; can be simpler to implement
 - Also known as **mutex locks**
- Provides mutual exclusion
 - Semaphore S; // initialized to 1
 - wait (S);
 Critical Section
 signal (S);

Semaphore Implementation

- The implementation of semaphores in the kernel must guarantee that no two threads can execute `wait ()` and `signal ()` on the same semaphore at the same time – i.e. that they are atomic
- The previous definitions for `wait()` and `signal()` required busy waiting over the integer values
- Busy waiting is almost never a good idea unless the system is structured so that it happens infrequently
- Applications may spend lots of time in critical sections and therefore this is not a good solution.

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
 - thread control block for the waiting thread
 - pointer to next entry in the queue
- Two operations:
 - block – place the thread invoking the operation on the appropriate waiting queue.
 - wakeup – remove one of threads in the waiting queue and place it in the ready queue.

Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of wait:

```
wait (S){  
    value--;  
    if (value <= 0) {  
        add this thread to waiting queue  
        block(); }  
}
```

- Implementation of signal:

```
Signal (S){  
    value++;  
    if (value <= 0) {  
        remove a thread T from the waiting queue  
        wakeup(T); }  
}
```


Deadlock and Starvation

- **Deadlock** – two or more threads are waiting indefinitely for an event that can only be caused by one of the waiting threads
- Let **S** and **Q** be two semaphores initialized to 1

T_0

```
wait (S);  
wait (Q);  
.  
.  
.  
signal (S);  
signal (Q);
```

T_1

```
wait (Q);  
wait (S);  
.  
.  
.  
signal (Q);  
signal (S);
```

- **Starvation** – indefinite blocking. A thread may never be removed from the semaphore queue in which it is suspended.

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem (in textbook)
- Dining-Philosophers Problem (in textbook)

Bounded-Buffer Problem

- N buffers, each can hold one item
- Semaphore **mutex** initialized to the value 1
- Semaphore **full** initialized to the value 0
- Semaphore **empty** initialized to the value N .

Bounded Buffer Problem (Cont.)

- The structure of the producer thread

```
do {  
  
    // produce an item  
  
    wait (empty);  
    wait (mutex);  
  
    // add the item to the buffer  
  
    signal (mutex);  
    signal (full);  
} while (true);
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer thread

```
do {  
    wait (full);  
    wait (mutex);  
  
    // remove an item from buffer  
  
    signal (mutex);  
    signal (empty);  
  
    // consume the removed item  
  
} while (true);
```

Problems with Semaphores

- Incorrect use of semaphore operations:
 - `signal (mutex) wait (mutex)`
 - `wait (mutex) ... wait (mutex)`
 - Omitting `wait (mutex)` or `signal (mutex)` (or both)

Monitors

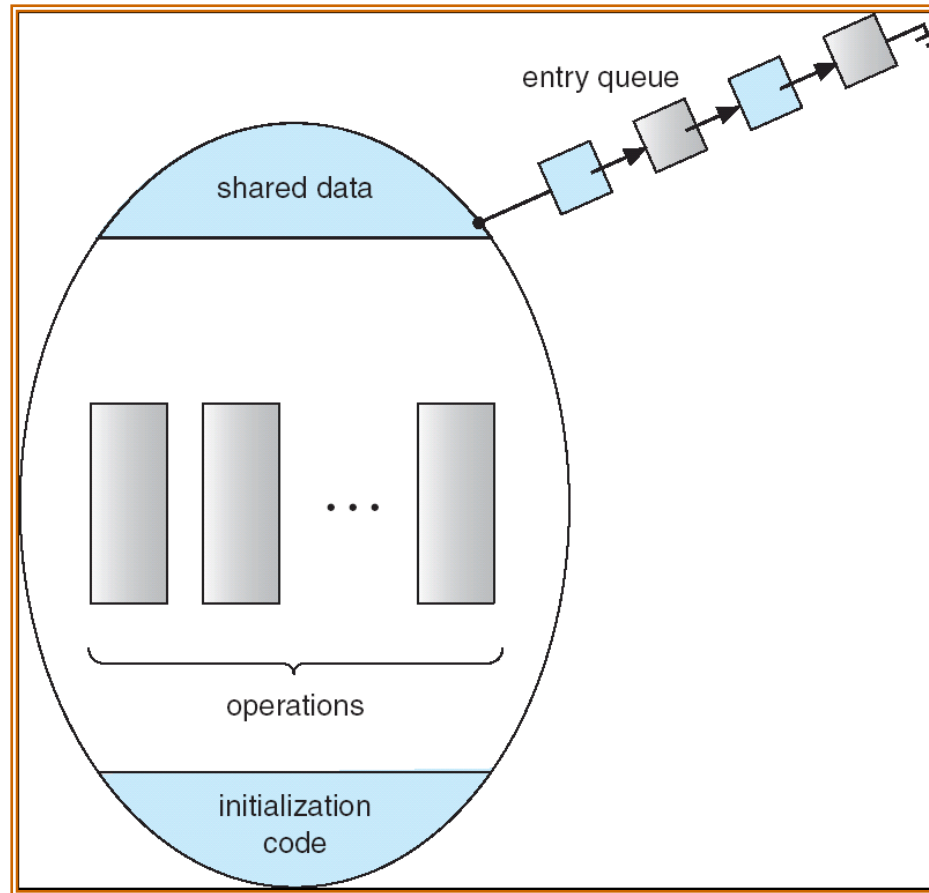
- A high-level abstraction that provides a convenient and effective mechanism for thread synchronization
- Only one thread may be active within the monitor at a time

```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (...) { .... }
    ...

    procedure Pn (...) { ..... }

    Initialization code ( .... ) { ... }
    ...
}
}
```

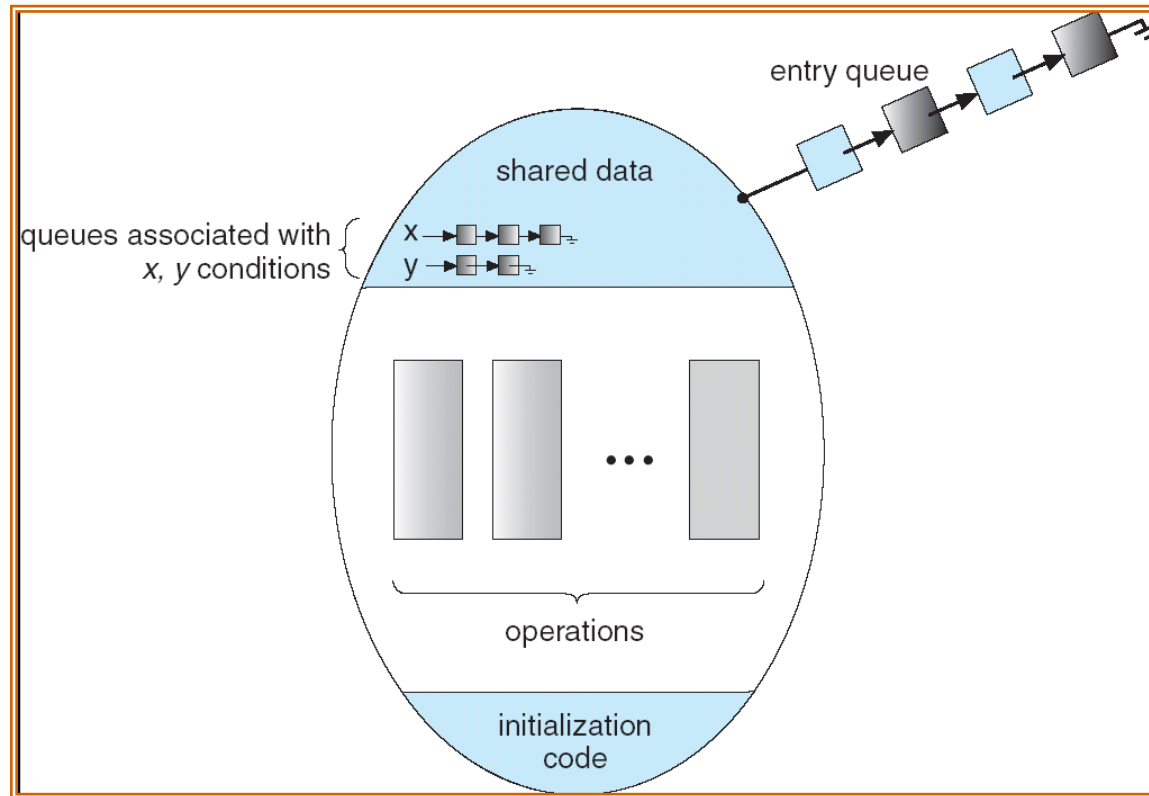
Schematic view of a Monitor



Condition Variables

- condition `x, y`;
- Two operations on a condition variable:
 - `x.wait ()` – the thread that invokes the operation is suspended.
 - `x.signal ()` – resumes one of the threads (if any) that invoked `x.wait ()`
 - `x.broadcast ()` – resumes all of the threads (if any) that invoked `x.wait ()`

Monitor with Condition Variables



Linux Synchronization

- Linux:
 - disables interrupts to implement short critical sections
- Linux provides:
 - semaphores
 - spin locks

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
 - mutex locks
 - condition variables
- Non-portable extensions include:
 - read-write locks
 - spin locks

Java Synchronization

- Monitors using the ***synchronized*** keyword
- Condition variables using wait(), notify() and notifyAll()



Deadlocks

Chapter Objectives

- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
- To present a number of different methods for preventing or avoiding deadlocks in a computer system.

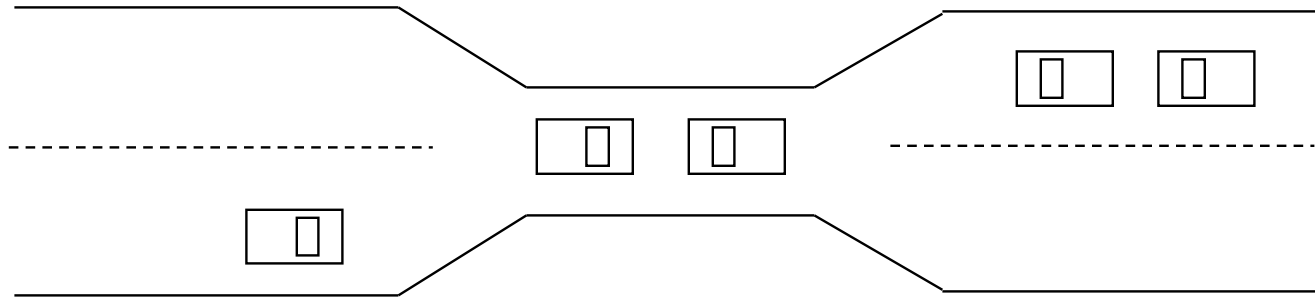
The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Example
 - System has 2 tape drives.
 - P_1 and P_2 each hold one tape drive and each needs another one.
- Example
 - semaphores A and B , initialized to 1

P_0
wait (A);
wait (B);

P_1
wait (B)
wait (A)

Bridge Crossing Example



- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.

System Model

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

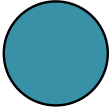

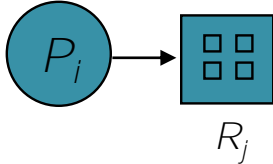
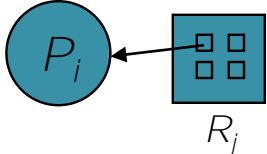
- **Mutual exclusion:** only one process at a time can use a resource.
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes.
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that 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
 - ...
 - P_{n-1} is waiting for a resource that is held by P_n
 - P_n is waiting for a resource that is held by P_0 .

Resource-Allocation Graph

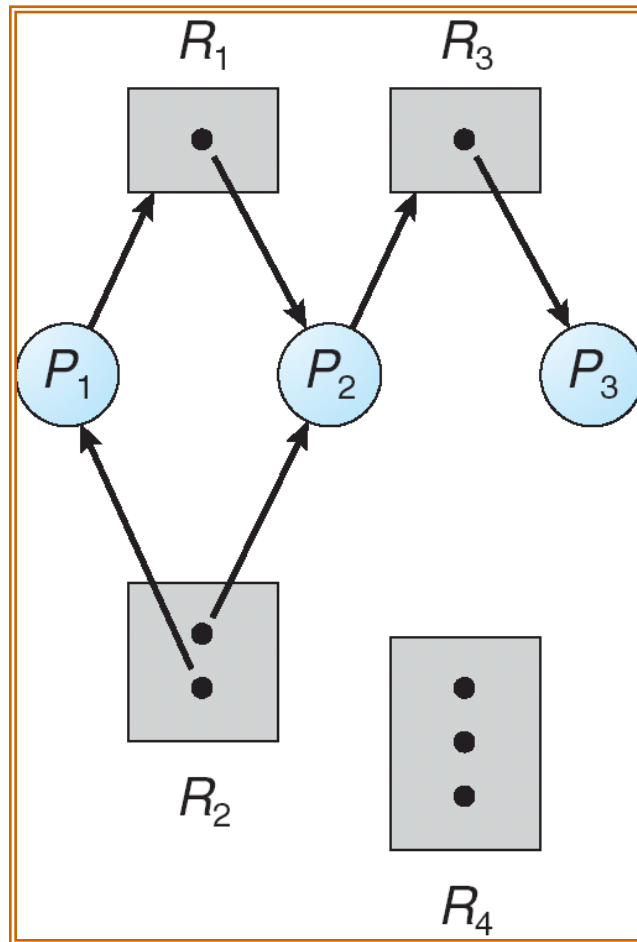
A set of vertices V and a set of edges E .

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

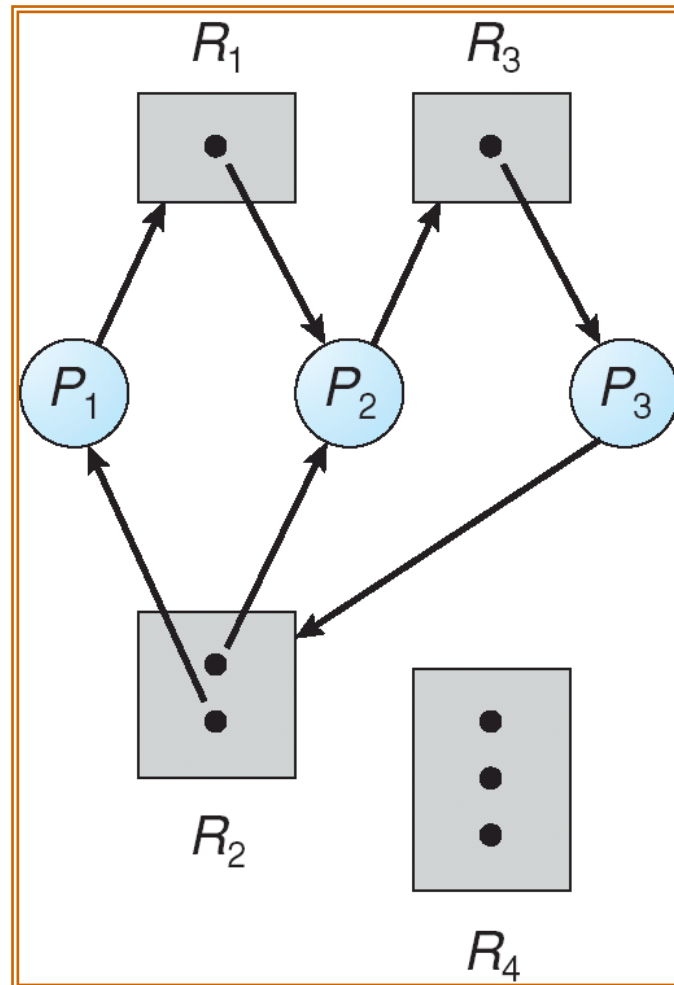
Resource-Allocation Graph (Cont.)

- Process 
- Resource Type with 4 instances 
- P_i requests instance of R_j 
- P_i is holding an instance of R_j 

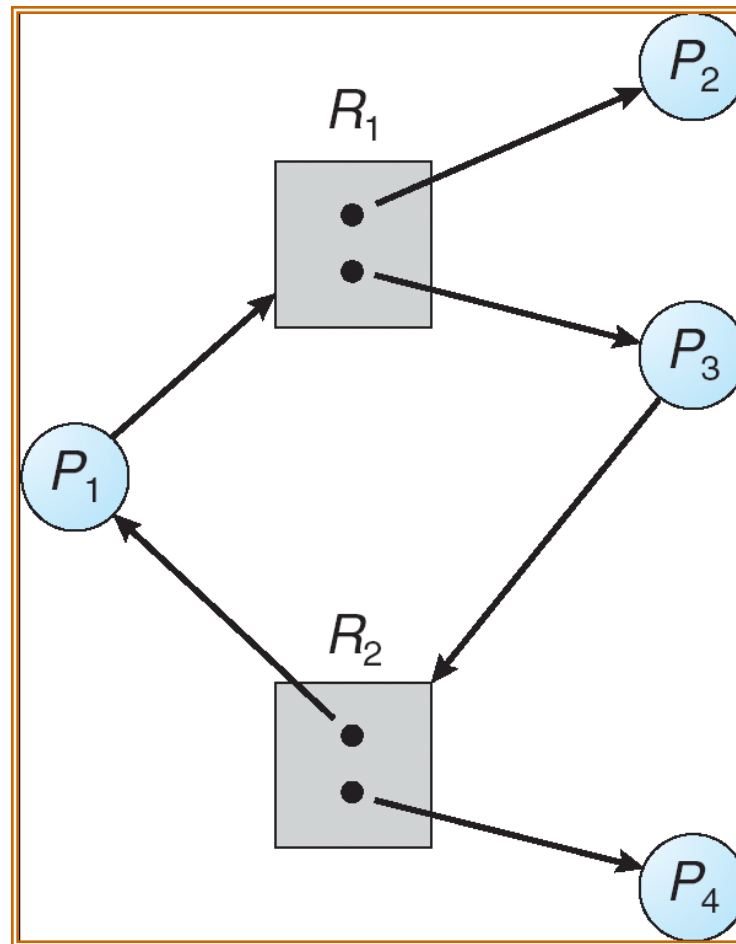
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Resource Allocation Graph With A Cycle But No Deadlock



Basic Facts

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

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 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.

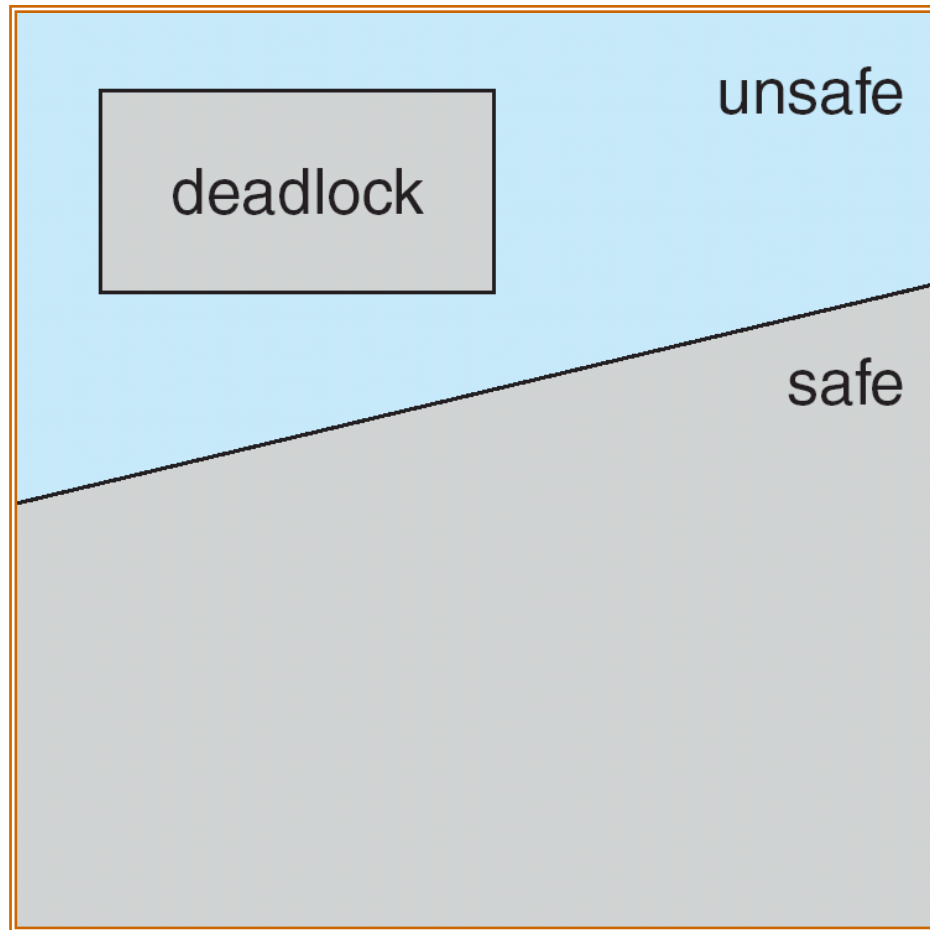
Safe State

- 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 $\langle P_1, P_2, \dots, P_n \rangle$ 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_j resource needs are not immediately available, then P_i can wait until all P_j have finished.
 - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.
 - When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

Basic Facts

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

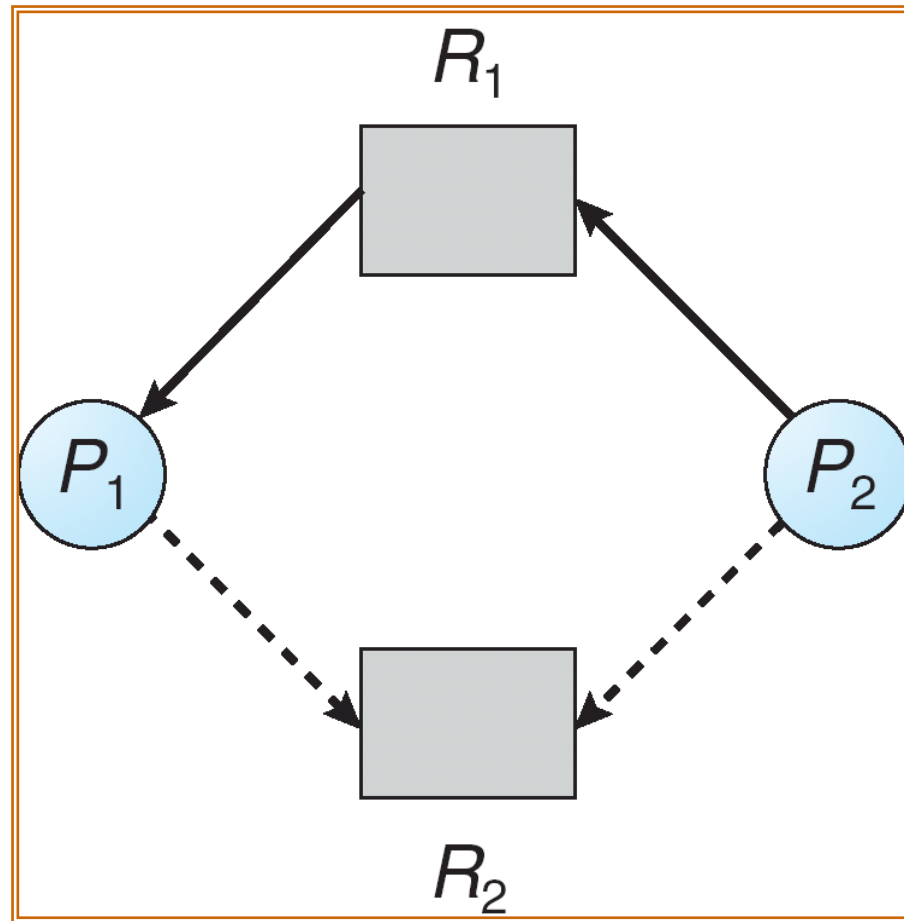
Safe, Unsafe , Deadlock State



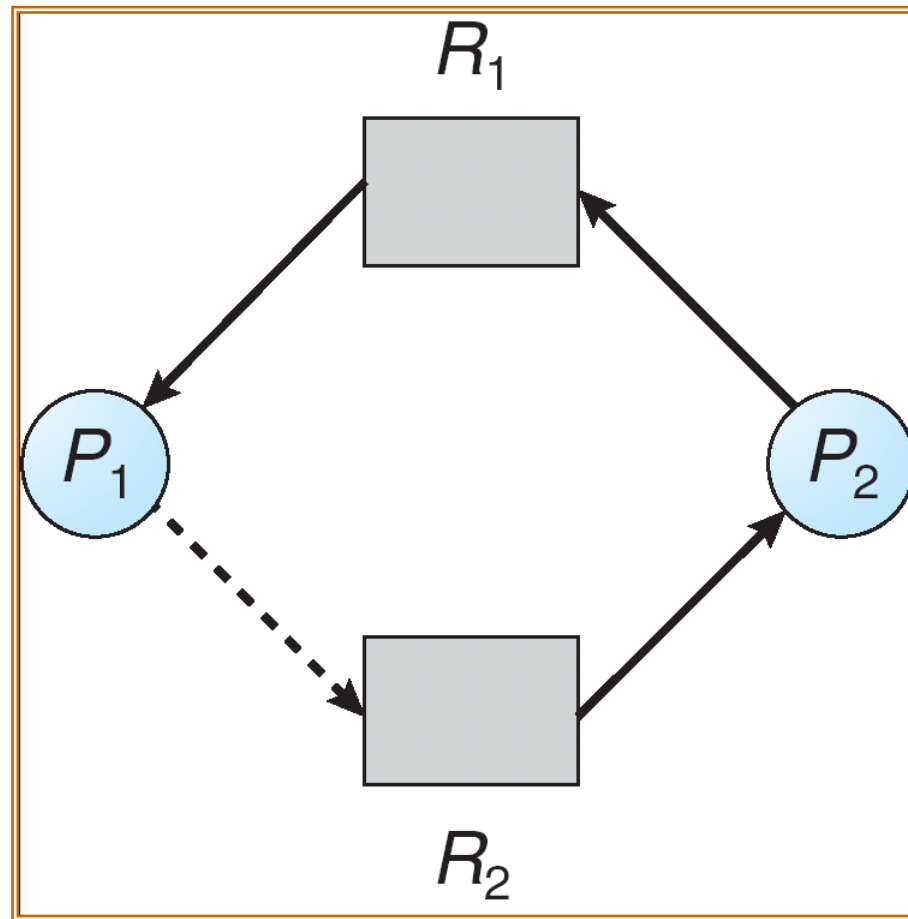
Resource-Allocation Graph Algorithm

- Claim edge $P_i \rightarrow R_j$ indicated that process P_i may request resource R_j ; represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed *a priori* in the system.

Resource-Allocation Graph For Deadlock Avoidance



Unsafe State In Resource-Allocation Graph



Banker's Algorithm

- Covered in the text book

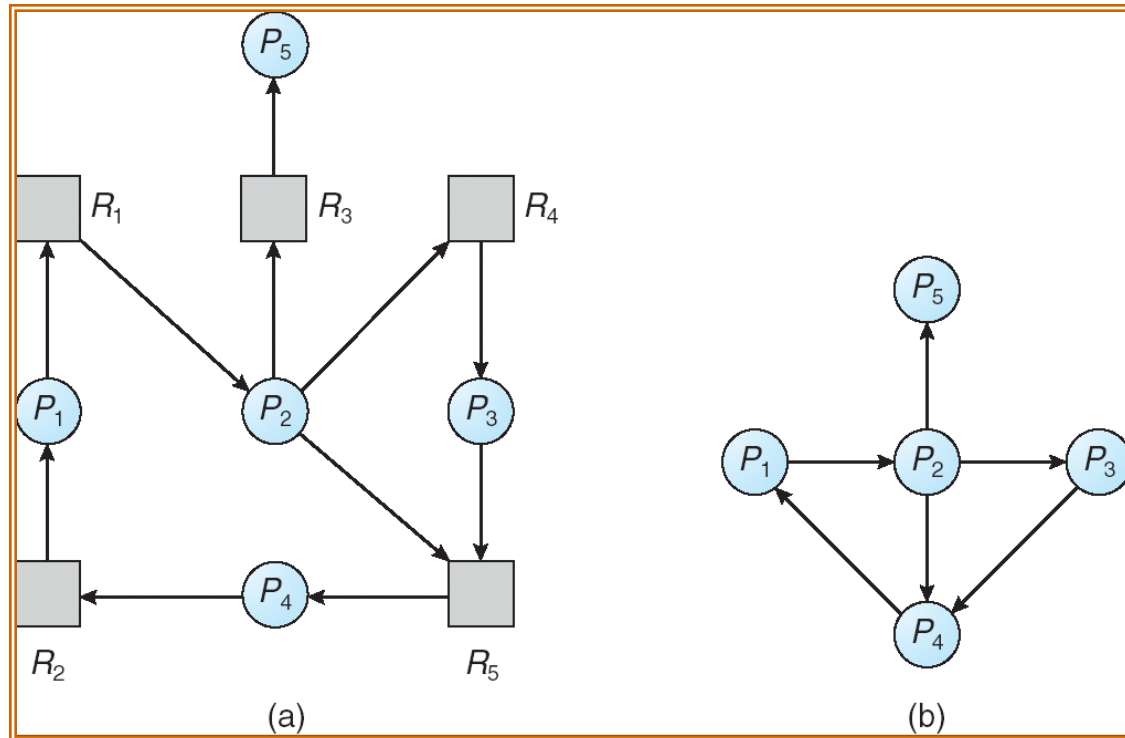
Deadlock Detection

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

Single Instance of Each Resource Type

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

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph

Corresponding wait-for graph

Several Instances of a Resource Type

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

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length m and n , respectively Initialize:
 - (a) *Work* = *Available*
 - (b) For $i = 1, 2, \dots, n$, 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.

Detection Algorithm (Cont.)

3. $Work = Work + Allocation_i$
 $Finish[i] = true$
go to step 2.
4. If $Finish[i] == false$, for some i , $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if $Finish[i] == false$, then P_i is deadlocked.

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

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
 - How often a deadlock is likely to occur?
 - How many processes will need to be rolled back?
 - one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes.
- Abort one process at a time until the deadlock cycle is eliminated.
- In which order should we choose to abort?
 - Priority of the process.
 - How long process has computed, and how much longer to completion.
 - Resources the process has used.
 - Resources process needs to complete.
 - How many processes will need to be terminated.
 - Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- Selecting a victim – minimize cost.
- Rollback – return to some safe state, restart process for that state.
- Starvation – same process may always be picked as victim, include number of rollback in cost factor.