Unit 3
Process Coordination and Deadlocks

Contents

Synchronization

1. Background
2. The Critical Section Problem
3. Peterson's Solution
4. Synchronization Hardware
5. Semaphores
6. Problems of Synchronization
7. Monitors

Deadlock

1. System Model
2. Deadlock Characterization
3. Methods for Handling Deadlocks
4. Deadlock Prevention
5. Deadlock Avoidance
6. Deadlock Detection
7. Recovery from Deadlock

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem:

  Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers. We can do so by having an integer **counter** that keeps track of the number of full buffers. Initially, **counter** 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.

Background: Producer – Consumer Problem

```
while (true) {
        /* produce an item in next produced */
        while (counter == BUFFER_SIZE) ;
                /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        /* consume the item in next consumed */
}
```

Background: Producer – Consumer Problem RACE CONDITION

- **counter++**

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **counter--**

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with "count = 5" initially:

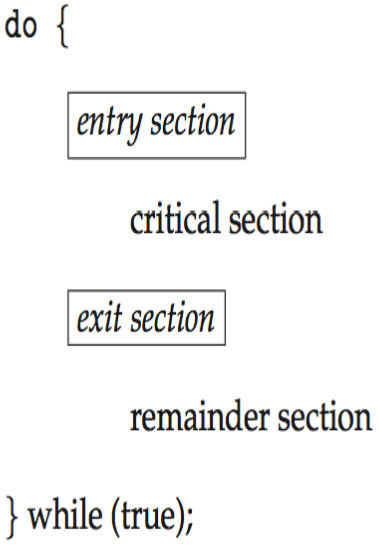
S0: producer execute	`register1 = counter`	{register1 = 5}
S1: producer execute	`register1 = register1 + 1`	{register1 = 6}
S2: consumer execute	`register2 = counter`	{register2 = 5}
S3: consumer execute	`register2 = register2 - 1`	{register2 = 4}
S4: producer execute	`counter = register1`	{counter =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

- **Race condition** – a situation when:
  - several processes access the same data concurrently
  - the outcome of the execution depends on the order of the accesses

The Critical Section Problem

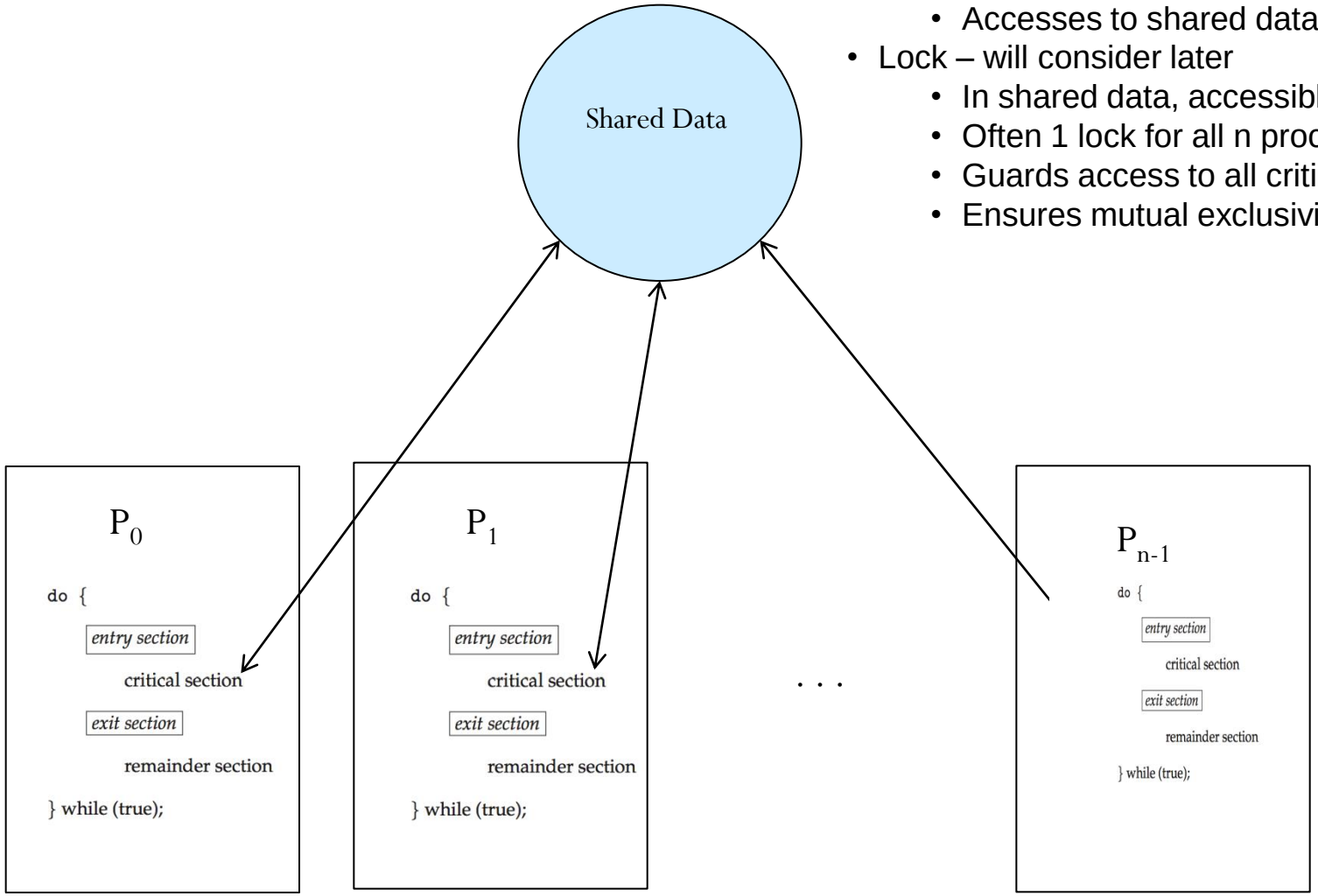
- Consider system of ***n*** processes $\{P_0, P_1, \ldots, P_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other is allowed to be in its critical section
- ***Critical section problem*** is to design protocol to solve this

- Each process
  - must ask permission to enter critical section in **entry section**,
  - may follow critical section with **exit section**,
  - then **remainder section** (anything else a process does besides using the **critical section**)



The Critical Section Problem

- Each process $\mathbf{P_i}$ has its own critical section
  - Accesses to shared data only in CS
- Lock – will consider later
  - In shared data, accessible by all
  - Often 1 lock for all n processes
  - Guards access to all critical sections
  - Ensures mutual exclusivity



The Critical Section Problem

- Solution for a Critical Section Problem must satisfy the following requirements
  - Mutual Exclusion – if process P_i is executing in its critical section, then no other processes can be executing in their critical sections
  - Progress – if no process in critical section and if some processes wish to enter into critical sections, then only those processes that are not executing in their remainder section can participate in deciding which will enter its critical section next and this selection cannot be postponed indefinitely.
  - Bounded Waiting – There exists a bound or limit on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Peterson's solution

- One of the good solution for solving critical section problem
- Two process solution
- The two processes share two variables:
  - `int turn;`
  - `Boolean flag[2]`
- The variable **`turn`** indicates whose turn it is to enter the critical section
- The **`flag`** array is used to indicate if a process is ready to enter the critical section. **`flag[i] = true`** implies that process P_i is ready!

Peterson's solution

```
do {
        flag[i] = true;
        turn = j;
        while (flag[j] && turn == j) do no-op;
                critical section
        flag[i] = false;
                remainder section
} while (true);
```

- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved

     $\mathbf{P_i}$ enters CS only if:

     either **`flag[j] = false`** or **`turn = i`**
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met
     - But we cannot avoid busy waiting

Synchronization Hardware

- Many systems provide **hardware support** for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Modern machines provide special **atomic hardware instructions**
  - **Atomic** = non-interruptible
  - **test_and_set** instruction
    - test memory word and set value
  - **compare_and_swap** instruction
    - swap contents of two memory words

Synchronization Hardware

- Solution to Critical-section Problem Using Locks

```
do {
    acquire lock
        critical section
    release lock
        remainder section
 }  while (true)
```

Synchronization Hardware

- test_and_set Instruction

```
boolean test_and_set(boolean &lock)
        {
            boolean temp = lock;
            lock= true;
            return temp;
          }
```

- Shared boolean variable lock, initialized to FALSE
- Solution:

```
do {
    while ( TestAndSet (&lock )) ;
            //    critical section

    lock = FALSE;

            //      remainder section

} while (TRUE);
```

Synchronization Hardware

- compare_and_swap Instruction

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

- Shared Boolean variable lock initialized to FALSE; Each process 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

- Semaphore is a counter that is manipulated atomically through two operations : **Signal and Wait.**
- **Wait: decrement if counter is zero** then block until the semaphore is signaled
  - **`P()`** (wait) – from Dutch proberen: "to test"
- **Signal: increment counter**, wakeup one waiter if any
  - **`V()`** (signal) – from Dutch verhogen: "to increment"
- sem_init(semaphore,counter) – set the first counter value
- Semaphore types: Counting Semaphore, Binary Semaphore
- Binary – integer value can be 0 or 1
- Counting – integer value can be a range of values

Semaphore

- Definition of the **`wait()`** operation

```
wait(S) {
    while (S <= 0)
        ; // Notice, busy waits, spending CPU cycles, not (always) good
    S=s-1; }
```

- Definition of the **`signal()`** operation

```
signal(S) {
    S=s+1;
}
```

- Code for CS using Semaphore

```
P(S) // Wait(S)
        CriticalSection
V(S) // Signal(S)
```

Semaphore - Implementation

- Operations **`wait()`** and **`signal()`** must be executed atomically
- Thus, the implementation becomes the solution to critical section problem
  - **`wait`** and **`signal`** code are placed in the critical section
- Hence, can now have **busy waiting** in critical section implementation
  - If critical section rarely occupied
    - Little busy waiting
  - But, applications may spend lots of time in critical sections
    - Hence, the busy-waiting approach is not a good solution

Classic Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Readers and Writers Problem
  - Bounded-Buffer Problem (Producer – Consumer Problem)
  - Dining-Philosophers Problem

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do **not** perform any updates
  - Writers – can both read and write

- Problem –
- allow multiple readers to read at the same time
- Only one single writer can access the shared data at the same time
- Initially mutex is assigned 1

- Solution 1: One writer and One reader
- This solution allows only one reader or one writer to access shared resources

Writer	Reader
Wait(mutex)	Wait(mutex)
Write here (CS)	Read here(CS)
Signal(mutex)	Signal(mutex)

- Solution 2: One writer and Multiple readers
- This solution can allow multiple readers or one writer to access the resource
- To allow multiple readers, check whether he is a first reader, then acquire the lock, else just read without locking.

Writer	Reader	Reader2
Wait(mutex)	Reader++	Reader++
Write here	If (reader==1)	If (reader==1)
(CS)	Wait(mutex)	Wait(mutex)
Signal(mutex)	Read here(CS)	Read here(CS)
	Reader –	Reader –
	If(reader==0)	If(reader==0)
	Signal(mutex)	Signal(mutex)

Producer-Consumer Problem

- Producer
  - adds items into the buffer
  - It has to stop when the buffer is full
  - It should not allow the consumer to consume while producing
- Consumer
  - Consumes an item from the buffer
  - It cannot consume from an empty buffer
  - It should not allow the producer to produce while consumer is consuming

- Solution
- One semaphore to make these two process mutual exclusive
- One semaphore to maintain number of item in buffer
- One semaphore to maintain number of empty places in buffer

Bounded Buffer Problem (Cont.)

- The structure of the producer process

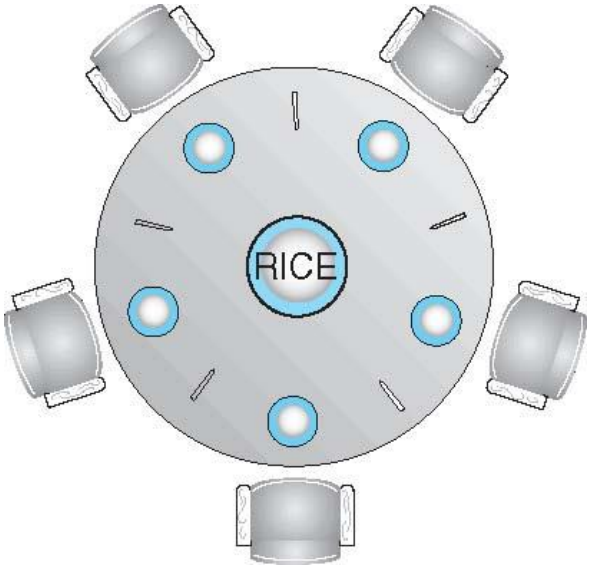
```
do  {

        //   produce an item in nextp
    wait (mutex);
    wait (empty);

        //  add the item to the  buffer

     signal (full);
    signal (mutex);
} while (TRUE);
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
do {
        wait (mutex);
       wait (full);

              //  remove an item from  buffer to nextc

        signal (empty);
       signal (mutex);

              //  consume the item in nextc

} while (TRUE);
```

Dining-Philosophers Problem



- Philosophers spend their lives thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

- Solution
- Take the left fork and then acquire the right fork
- If you don't get the left fork then wait. After acquiring left, check whether you get right, else wait for right fork

Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do {
        wait ( chopstick[i] );
        wait ( chopStick[ (i + 1) % 5] );

            //  eat

        signal ( chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );

            //  think

} while (TRUE);
```

Monitors

- A monitor – is an ADT that provides a convenient and effective mechanism for process synchronization
- A Programming language construct that supports controlled access to shared data
- Monitor is a software module that encapsulates:
  - Shared data structures
  - Procedure that operate on the shared data
  - Synchronization between concurrent process that include those procedures
- Monitor protects the data from unstructured access
  - Guarantees only access data through procedures

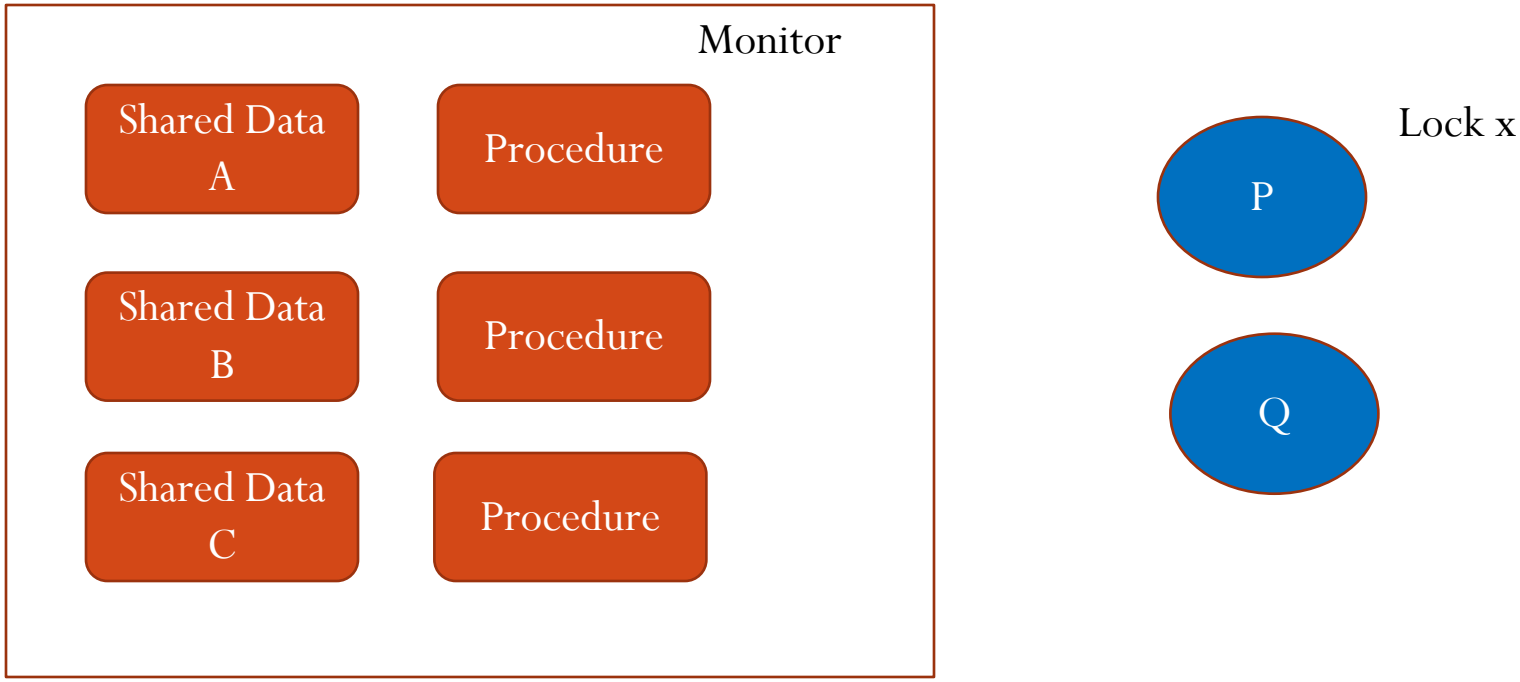
Monitors

- Only one process at a time may be active within the monitor
  - A lock guards shared data
  - Hence, the programmer does not need to code this constraint

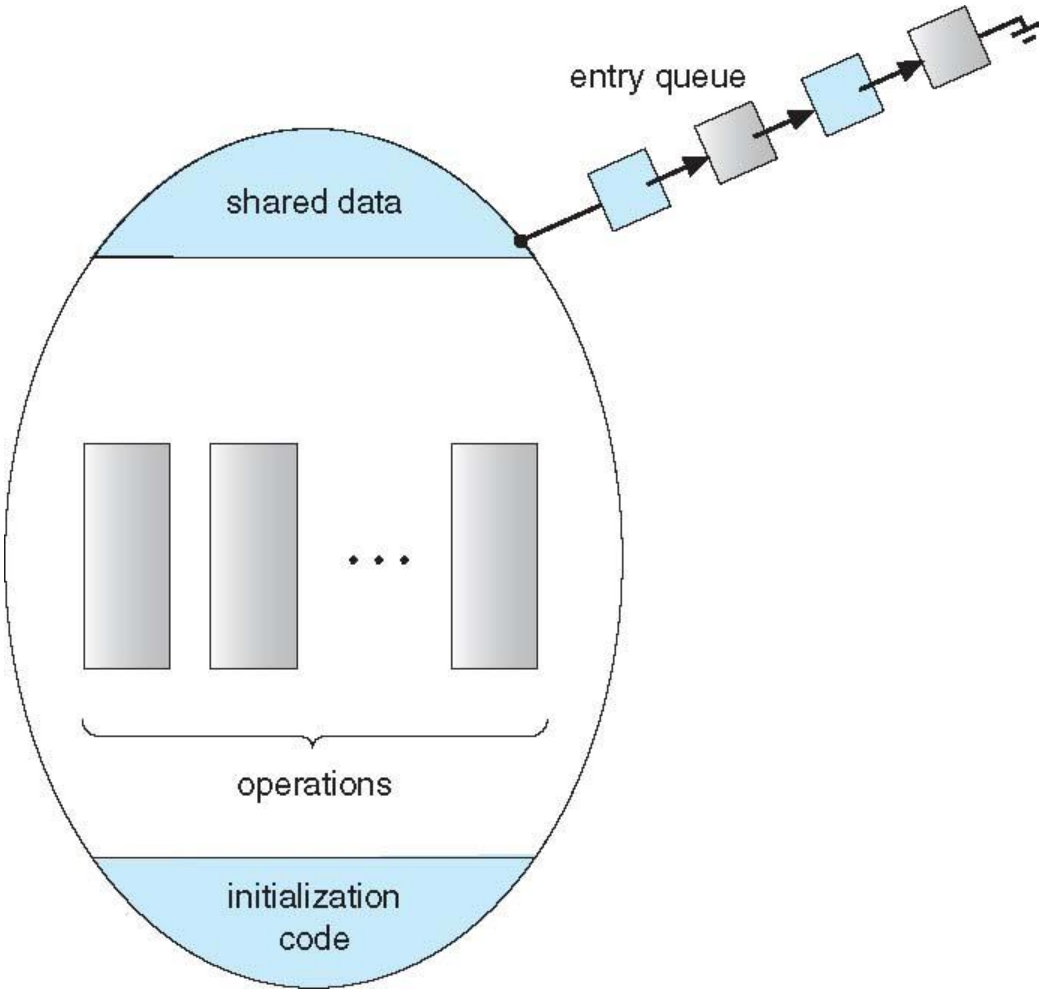
```
monitor monitor-name
{
  // shared variable declarations
  function P1 (…) { … }
  …
  function Pn (…) {……}
   initialization_code (…) { … }
}
```

Monitors

Schematic view of a Monitor



Schematic view of a Monitor



Monitors

- Queues can be associated with **conditions**
- Two operations are allowed on a condition variable **x .** Now suppose x.signal() operation is invoked by process P, there is a suspended process Q associated with condition x
- A compromise between the two choices can be adopted in the programming construct
  - Signal and wait: P either waits until Q leaves the monitor
  - Signal and continue: Q either waits until P leaves the monitor

Deadlocks

System Model

- System consists of finite number of resources which has to be shared among the competing process.
- Resource types $R_1, R_2, \ldots, R_m$
  - *CPU cycles, files, memory space, I/O devices, etc*
- Each resource type R_i has W_i instances
  - Type: CPU
    - 2 instances - CPU1, CPU2
  - Type: Printer
    - 3 instances - printer1, printer2, printer3
- Each process utilizes a resource as follows:
  - **Request** resource
  - **Use** resource
  - **Release** resource

What is Deadlock?

- A set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set.
- Resources may be physical (cpu cycles, printer, tape drive..) / logical (semaphores, monitors, files…)
- Deadlock may also involve same/ different resource type.

Deadlock Characterization

Necessary Conditions that causes deadlock

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

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

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

Circular wait: there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by $P_2, \ldots, P_{n-1}$ is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

Deadlock may arise if 4 conditions hold simultaneously in a system

Deadlock Characterization

Resource-Allocation Graph

- Useful tool to describe and analyze deadlocks
- Set of vertices V
- Set of edges E

V is partitioned into two types:

- $P = \{P_1, P_2, \ldots, P_n\}$, the set of all the processes in the system
- $R = \{R_1, R_2, \ldots, R_m\}$, the set of all resource types in the system

Request edge – directed edge $P_i \rightarrow R_j$

- Means P_i has requested (an instance of) R_j and now is waiting for it

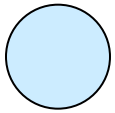
Assignment edge – directed edge $R_j \rightarrow P_i$

- Means an instance of R_j has been assigned to P_i

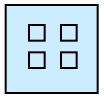
Deadlock Characterization

Resource-Allocation Graph – Pictorial Representation

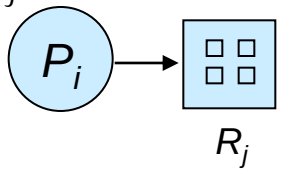
- Process



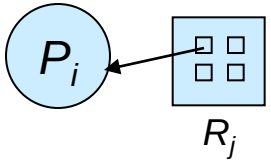
- Resource Type (with 4 dots for 4 instances in this specific example)



- P_i requests instance of R_j

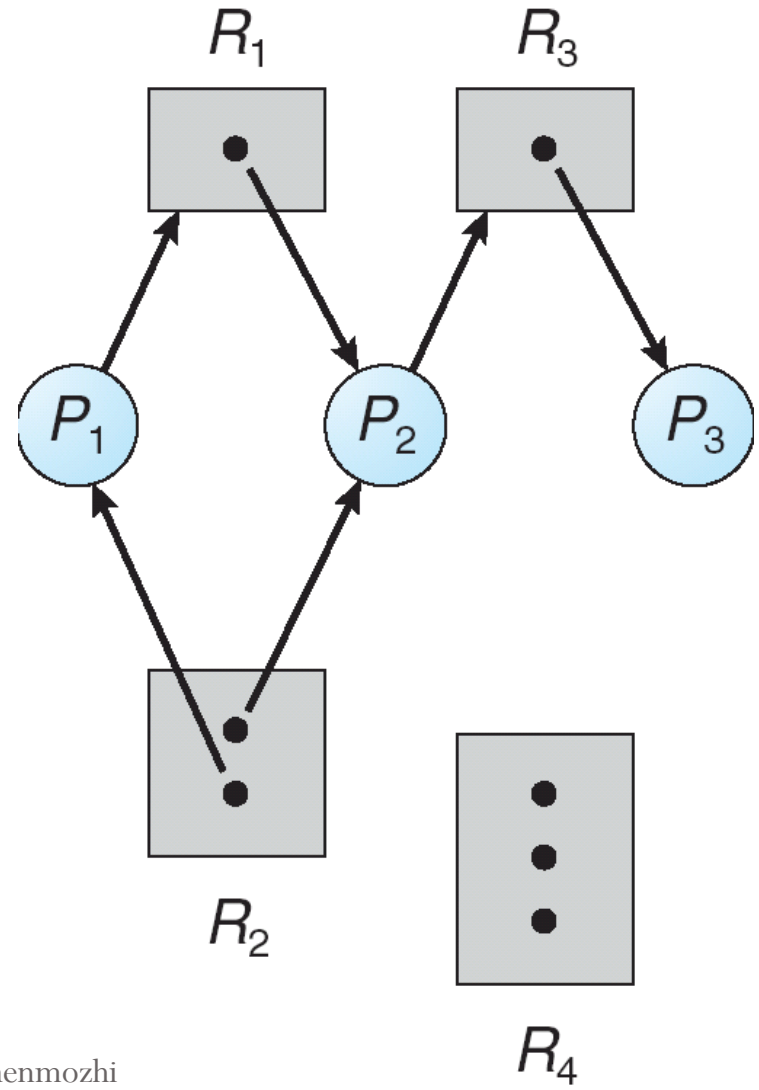


- P_i is holding an instance of R_j



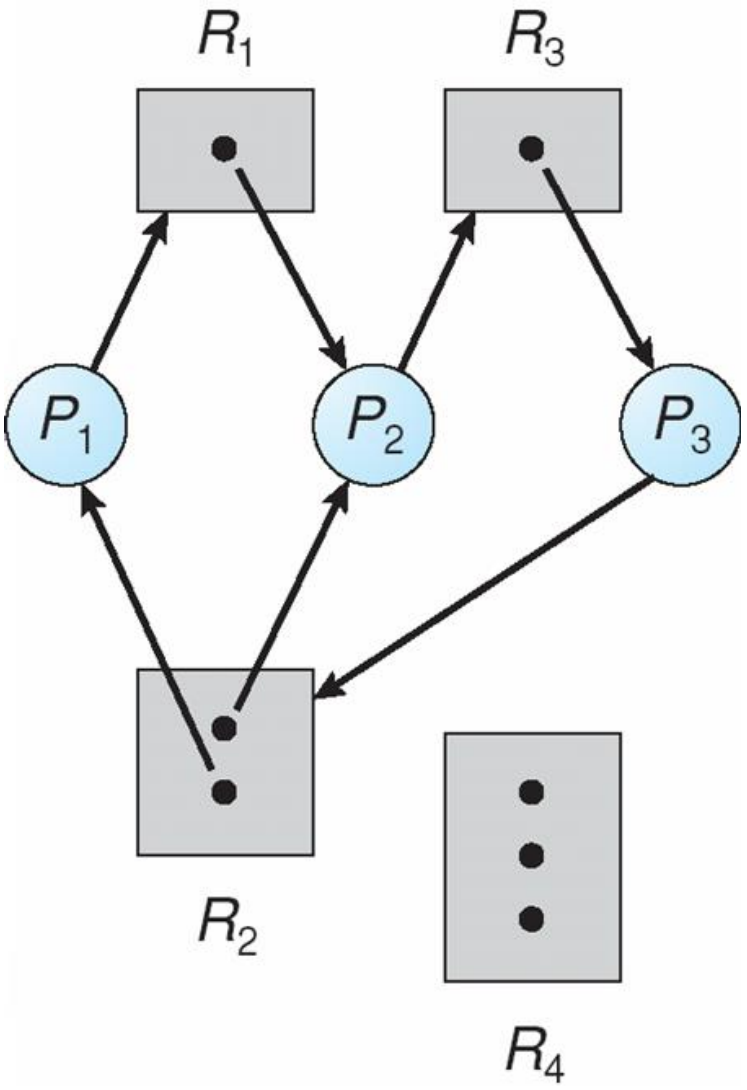
Deadlock Characterization

Resource-Allocation Graph – Example



Deadlock Characterization

Resource-Allocation Graph – Deadlocked State



Basic Fact

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

Methods for Handling Deadlocks

- Three ways to handle deadlocks:

1. Ensure that the system will ***never*** enter a deadlock state:
   - Deadlock prevention
   - Deadlock avoidance
2. Allow the system to enter a deadlock state and then recover
3. Ignore the problem and pretend that deadlocks never occur in the system
   - Used by most operating systems, including UNIX
   - One reason: Handling deadlocks can be computationally expensive

Deadlock Prevention

- **Key idea:** Restrain the ways request for resources can be made
- Recall, all 4 necessary conditions must occur for a deadlock to happen
  1. Mutual Exclusion
  2. Hold and Wait
  3. No preemption
  4. Circular Wait
- If we ensure at least one condition is not met, we prevent a deadlock

Prevention – contd..

- **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

Prevention – contd..

- **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

Idea: Require that the system has some additional ***a priori*** information about how resources will be used

- 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:
  1. number of available resources
  2. number of allocated resources
  3. maximum resource 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

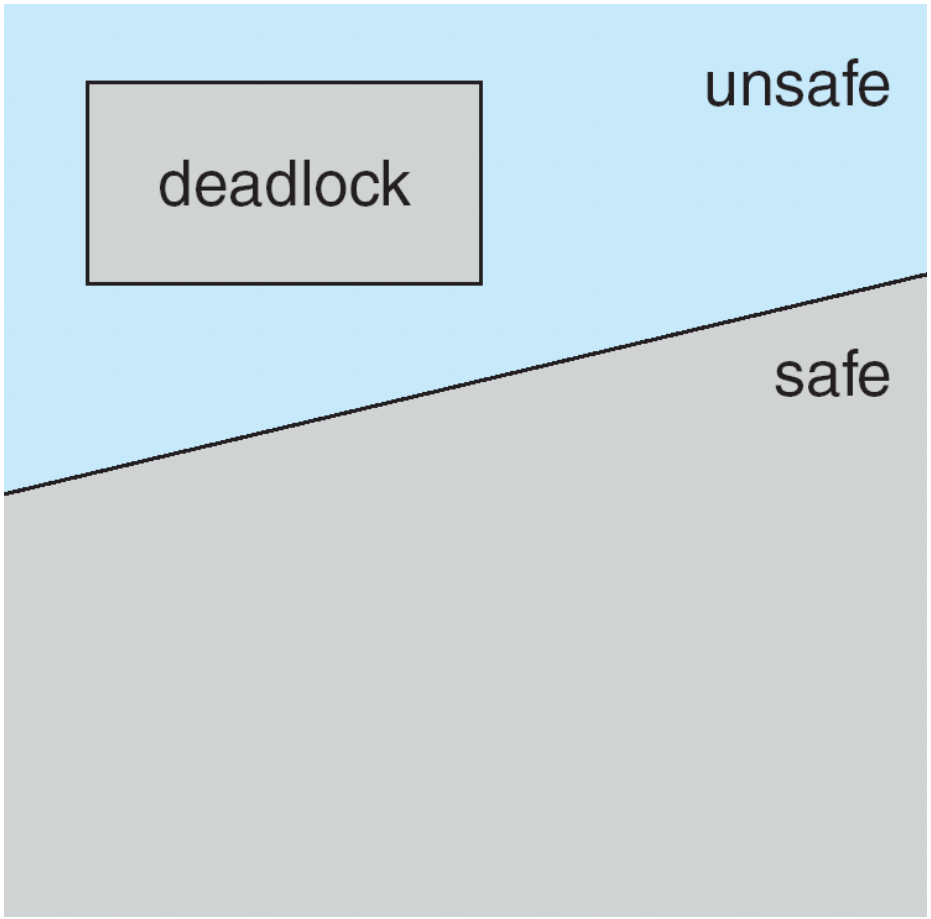
- System is in **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < I$

- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

Basic Facts

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

Safe, Unsafe, Deadlock State



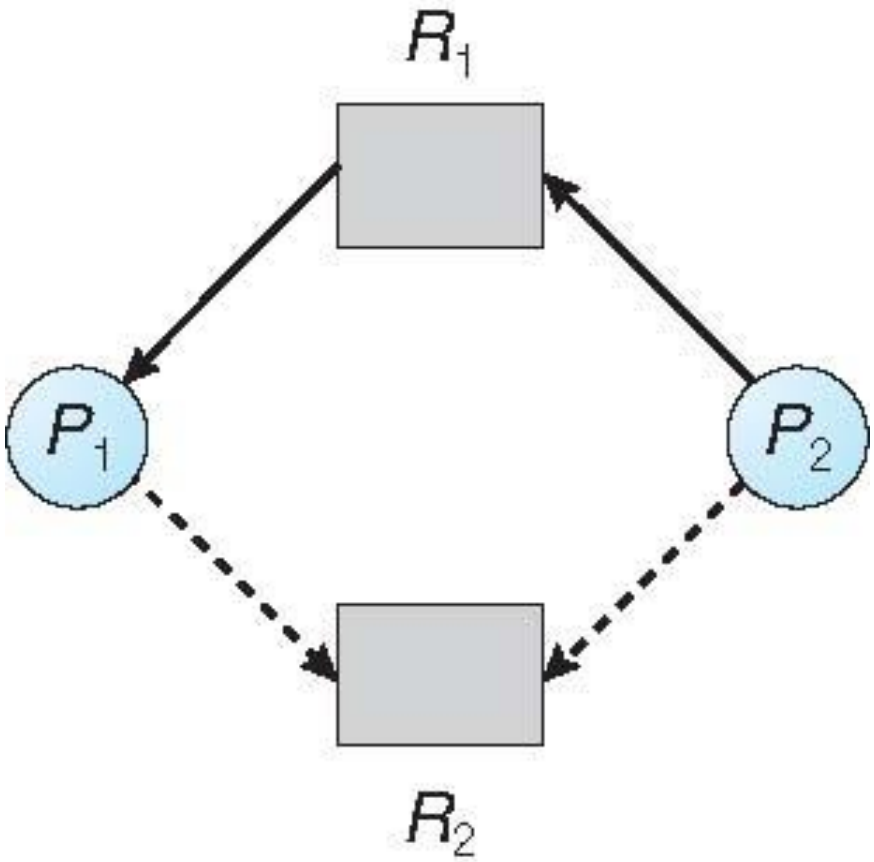
Avoidance algorithms

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

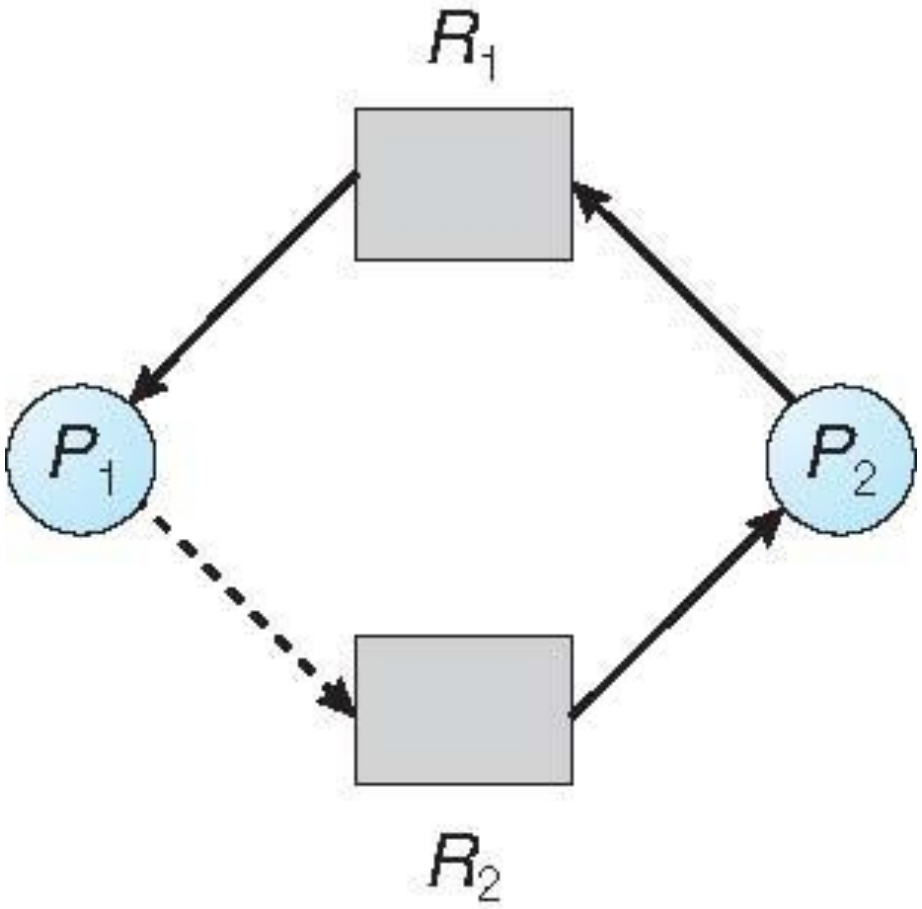
Resource-Allocation Graph Scheme

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

Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph



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

Banker's Algorithm

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

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

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

Safety Algorithm

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

   $Work = Available$

   $Finish\ [i] = false$ for $i = 0, 1, \ldots, n- 1$

2. Find an *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

Resource-Request Algorithm for Process P_i

$Request$ = request vector for process P_i. If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j

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

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

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

Example of Banker's Algorithm

- 5 processes P_0 through P_4;

  3 resource types:

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

Snapshot at time T_0:

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

Example (Cont.)

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

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

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

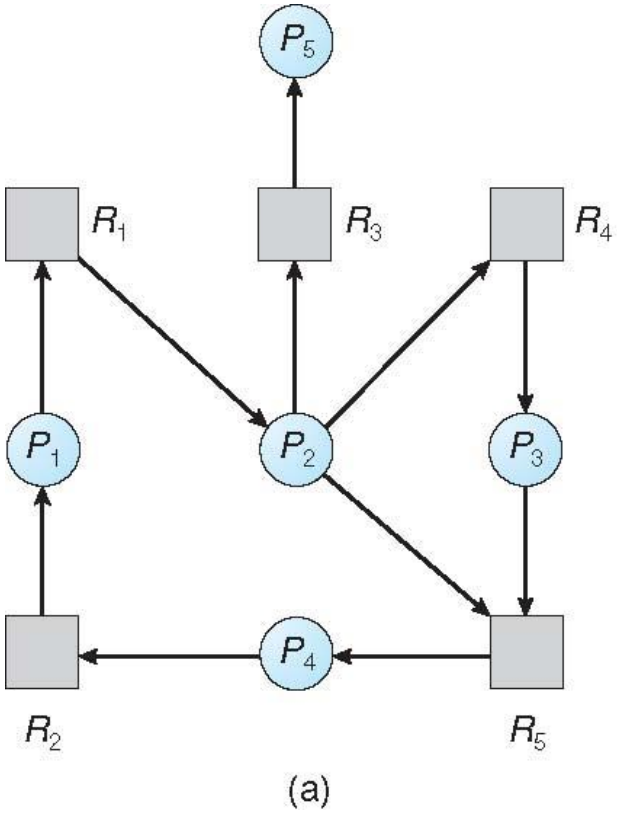
Deadlock Detection

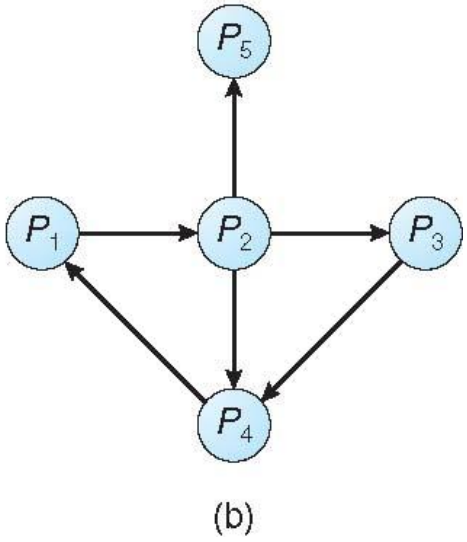
- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

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. If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

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* x *m* matrix defines the number of resources of each type currently allocated to each process.
- **Request**: An *n* x *m* matrix indicates the current request of each process. If *Request* [*i*][*j*] = *k*, then process P_i is requesting *k* 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 $i = 1,2, \ldots, n$, 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 x $n^{2)}$ operations to detect whether the system is in deadlocked state

Example of Detection Algorithm

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

- Snapshot at time T_0:

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

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

Example (Cont.)

- P_2 requests an additional instance of type C

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

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

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 required for the process 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