Chapter 6: Process Synchronizing Tools

Sheli Sinha Chaudhuri
Professor
ETCE Department

Critical Section Problem

```
do {
    entry section
        critical section
    exit section
        remainder section
  } while (TRUE);
```

Critical Section Problem

- n cooperating processes are there all competing to use shared resources
- Each process has a code segment, called critical segment, in which the shared data is accessed.
- When one process is accessing its critical section, no other process is allowed to execute in its critical section

Solution to Critical Section Problem

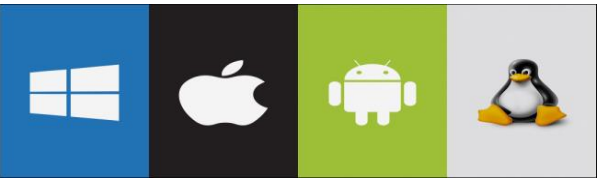
- The solution to critical section problem must satisfy the following three requirements:
- 1. Mutual Exclusion- Process P is executing in its critical section, then no other processes can be executing in their critical sections.
- 2. Progress- Process is executing in its critical section and some processes wish to enter their critical section, then only the processes not in their remainder sections can participate in deciding which will enter its critical section next and in definite time.
- 3. 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.

Solution to Critical Section Problem

- Two general approaches are used to handle critical sections in operating systems:
  - Preemptive kernels

    Allows a process to be preempted, while running in kernel mode
  - Nonpreemptive kernels

    Does not allow a process running in kernel mode to be preempted; the process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU



Peterson's Solution

- Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections

Peterson's solution

```
turn = 0;
flag[0] =  false;
flag[1] =  false;
```

P0

```
while(true){
   flag[0] = true;
   turn =1;
   while (flag[1] and turn ==1) do no-op;
      CS
   flag[0] = false;
   remainder section;
}
```

P1

```
while(true){
    flag[1] = true;
    turn = 0;
      while (flag[0] and turn ==0) do no-op;
      CS
    flag[1] = false;
    remainder section;
}
```

Solution to Critical Section Problem

- Synchronization Hardware

-Solutions are based on the premise of locking, protecting critical sections through use of locks.

-Critical sections problem could be solved simply in a single processor environment if interrupts can be prevented from occurring while a shared variable was being modified.

while(Test-and-Set(Lock));	Entry Section
Critical Section	
Lock = 0	Exit Section

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- We will look at three forms of hardware support:

1. Hardware instructions
2. Atomic variables

Synchronization Hardware

- Special hardware instructions that allow us to either *test-and-modify* the content of a word, or to *swap* the contents of two words atomically (uninterruptedly.)
  - **Test-and-Set** instruction
  - **Compare-and-Swap** instruction

Solution to Critical Section Problem

- Disabling interrupts on a multiprocessor environment can be time consuming, reducing system efficiency.
- Modern computers provide special hardware instructions allowing:
  - Test and modify the content of a or to swap the contents of two words **atomically**- as one uninterruptible unit.
  - These special instructions are useful to solve the critical-section problem in a simple way

Solution to Critical Section Problem

- Test and set () instruction is executed atomically
- If two test and set () instructions are executed simultaneously they will be executed sequentially in some arbitrary order.
- Using this instruction mutual exclusion can be implemented by declaring a boolean variable lock initialized to false.
- Compare_and_swap () instruction operates on three operands. Operand value is set to new_value only if the expression (*value==executed) is true. Instruction is executed atomically.

Solution to Critical Section Problem

- Definition

```
boolean test_and_set (boolean *target)
{
        boolean rv = *target;
        *target = true;
        return rv:
}
```

- Properties
  - Executed atomically
  - Returns the original value of passed parameter
  - Set the new value of passed parameter to **`true`**

Solution to Critical Section Problem

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

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

            /* critical section */

    lock = false;
            /* remainder section */
 } while (true);
```

- Does it solve the critical-section problem?

Solution to Critical Section Problem

- Definition

```
  int compare_and_swap(int *value, int expected,
int new_value)
    {
     int temp = *value;
      if (*value == expected)
          *value = new_value;
       return temp;
    }
```

- Properties
  - Executed atomically
  - Returns the original value of passed parameter **`value`**
  - Set the variable **`value`** the value of the passed parameter **`new_value`** but only if **`*value == expected`** is true. That is, the swap takes place only under this condition.

Solution to Critical Section Problem

- Shared integer **`lock`** initialized to 0;
- Solution:

```
 while (true){
while (compare_and_swap(&lock, 0, 1) != 0)
     ; /* do nothing */

/* critical section */

lock = 0;

/* remainder section */
}
```

- Does it solve the critical-section problem?

Solution to Critical Section Problem

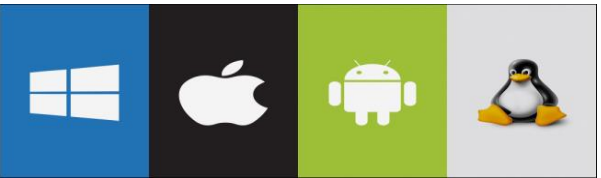
Bounded Waiting with compare_and _swap

```
while (true) {
   waiting[i] = true;
   key = 1;
   while (waiting[i] && key == 1)
      key = compare_and_swap(&lock,0,1);
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = 0;
   else
      waiting[j] = false;
   /* remainder section */
}
```

Solution to Critical Section Problem

Bounded-waiting Mutual Exclusion with compare-and-swap

```
while (true) {
   waiting[i] = true;  //process "i" is waiting
   key = 1;
   while (waiting[i] && key == 1) //process "i" is waiting and you haven't used the key because the lock is currently locked
      key = compare_and_swap(&lock,0,1); //key is used and lock is locked
   waiting[i] = false;  //process "i" is no longer waiting
   /* critical section */
   j = (i + 1) % n; //j = next process, n = total processes
   while ((j != i) && !waiting[j]) //Keep going to next process if it is different from the current process and not waiting.
      j = (j + 1) % n;
   if (j == i) //If the next process is the same as the current
      lock = 0;
   else  //If the next process is different from the current and waiting
      waiting[j] = false;
   /* remainder section */
}
```



Sheli Sinha Chaudhuri

Atomic Variables

- Typically, instructions such as compare-and-swap are used as building blocks for other synchronization tools.
- One tool is an **atomic variable** that provides *atomic* (uninterruptible) updates on basic data types such as integers and booleans.
- For example:
  - Let `sequence` be an atomic variable
  - Let `increment()` be operation on the atomic variable `sequence`
  - The Command:

```
increment(&sequence);
```

ensures `sequence` is incremented without interruption:

Atomic Variables

- The **`increment()`** function can be implemented as follows:

```
void increment(atomic_int *v)
{
    int temp;
    do {
          temp = *v;
    }
    while (temp !=
compare_and_swap(v,temp,temp+1));
}
```

Mutex Locks

Mutex locks are software tools

-used to protect critical regions and prevent race conditions

-process must acquire the lock before entering a critical section; releases when it leaves the critical section

-calls to either acquire () or release() must be performed atomically

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

Solution to the critical-section problem using mutex locks.

Mutex Locks

- Main disadvantage of mutex lock implementation is busy waiting
- This leads to processes waiting to enter critical section to loop continuously in the call to acquire ()
- This type of mutex lock is also called **spinlock**
- Spinlocks are not required when a process waits on a lock and a context switch takes time
- Spinlocks are useful when locks are expected to be held for short times

Semaphores

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`**
- Definition of the **`wait() operation`**

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

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

```
signal(S) {
    S++;
}
```

Semaphores

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can implement a counting semaphore ***S*** as a binary semaphore
- With semaphores we can solve various synchronization problems

Semaphores

- Solution to the CS Problem
  - Create a semaphore "**mutex**" initialized to 1

```
wait(mutex);
    CS
signal(mutex);
```

- Consider P_1 and P_2 that with two statements S_1 and S_2 and the requirement that S_1 to happen before S_2
  - Create a semaphore "**synch**" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

Semaphores

- Must guarantee that no two processes can execute the `wait()` and `signal()` on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the `wait` and `signal` code are placed in the critical section
- Could now have **busy waiting** in critical section implementation
  - But implementation code is short
  - Little busy waiting if critical section rarely occupied
- Note that 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:
  - Value (of type integer)
  - Pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

Semaphore Implementation with no busy waiting

- Waiting queue

```
typedef struct {
int value;
struct process *list;
} semaphore;
```

Semaphore Implementation with no busy waiting

```
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block();
   }
}

signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
   }
}
```

Problems with Semaphores

- Incorrect use of semaphore operations:
  - **`signal(mutex)`** …. **`wait(mutex)`**
  - **`wait(mutex)`** … **`wait(mutex)`**
  - Omitting of **`wait (mutex)`** and/or **`signal (mutex)`**
- These – and others – are examples of what can occur when semaphores and other synchronization tools are used incorrectly.

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- Pseudocode syntax of a monitor:

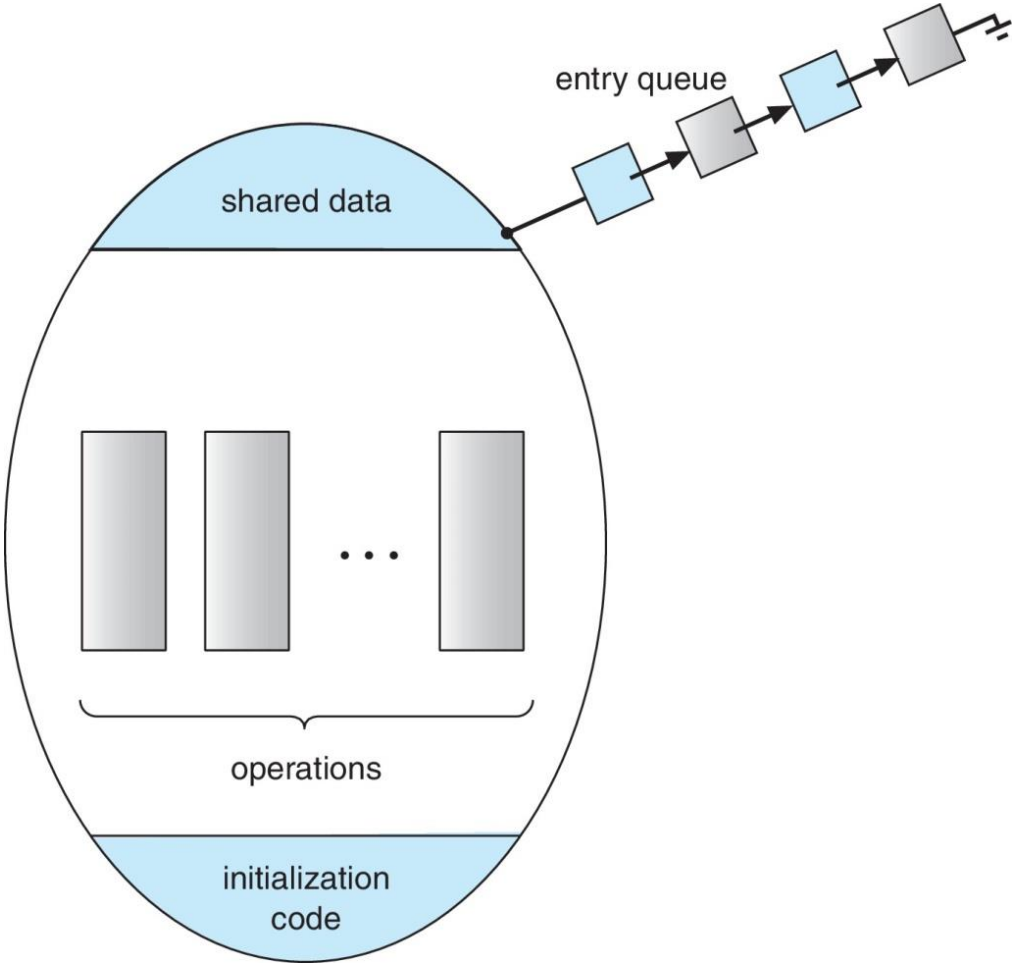
```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }

  procedure P2 (…) { …. }

  procedure Pn (…) {……}

  initialization code (…) { … }
}
```

Schematic view of a Monitor



Sheli Sinha Chaudhuri

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex
mutex = 1
```

- Each procedure ***P*** is replaced by

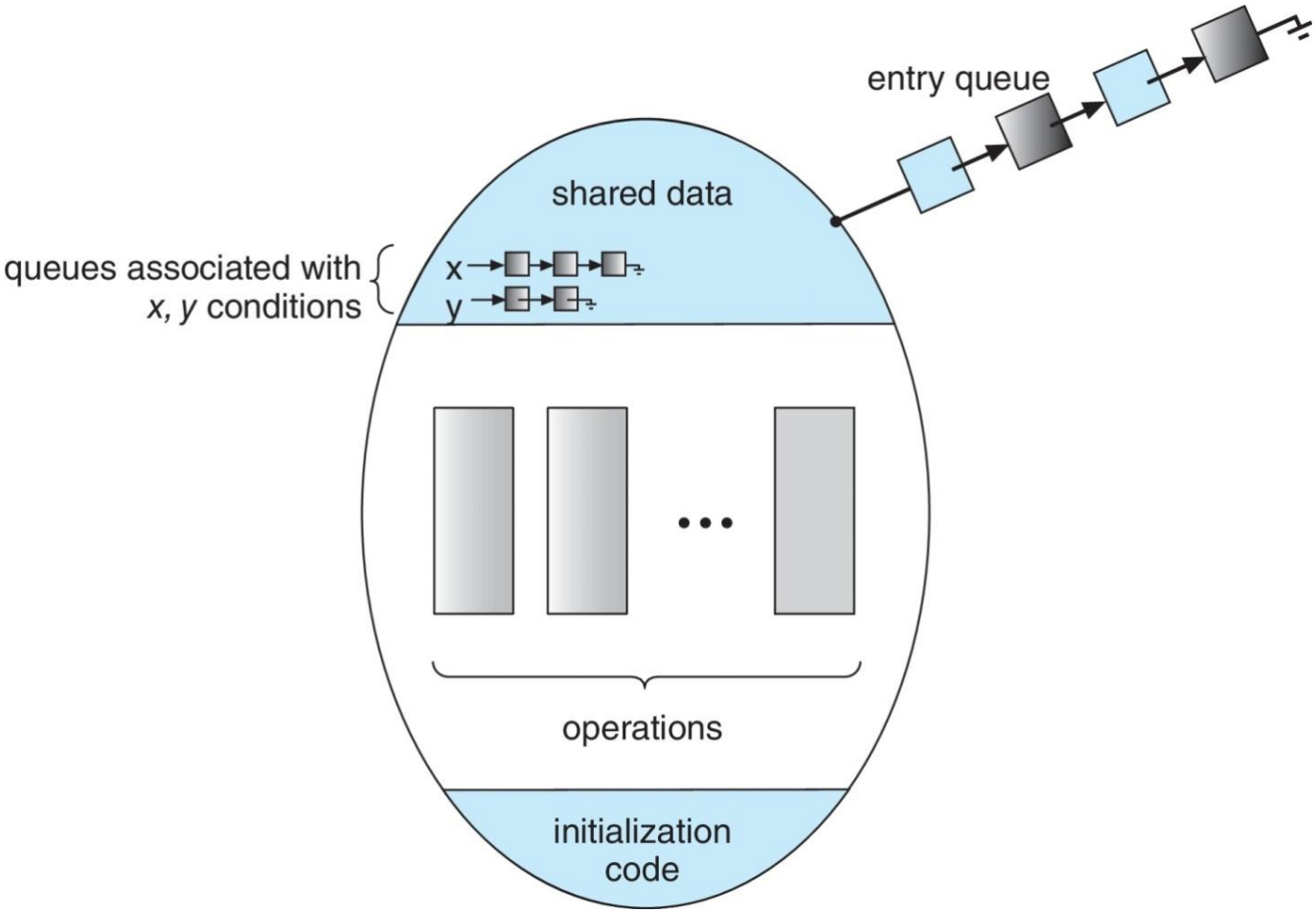
```
wait(mutex);
        …
            body of P;
        …
signal(mutex);
```

- Mutual exclusion within a monitor is ensured

Condition Variables

- **`condition x, y;`**
- Two operations are allowed on a condition variable:
  - **`x.wait()`** – a process that invokes the operation is suspended until **`x.signal()`**
  - **`x.signal()`** – resumes one of processes (if any) that invoked **`x.wait()`**
    - If no **`x.wait()`** on the variable, then it has no effect on the variable

Monitor with Condition Variables



Sheli Sinha Chaudhuri

Usage of Condition Variable Example

- Consider P_1 and P_2 that that need to execute two statements S_1 and S_2 and the requirement that S_1 to happen before S_2
  - Create a monitor with two procedures F_1 and F_2 that are invoked by P_1 and P_2 respectively
  - One condition variable "x" initialized to 0
  - One Boolean variable "done"
  - **F1:**

```
S1;
done = true;
x.signal();
```

  - **F2:**

```
if done = false
   x.wait()
S2;
```

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next_count = 0; // number of
processes waiting  inside the monitor
```

- Each function ***P*** will be replaced by

```
wait(mutex);
      …
       body of P;
      …
if (next_count > 0)
 signal(next)
else
 signal(mutex);
```

- Mutual exclusion within a monitor is ensured

Implementation – Condition Variables

- For each condition variable ***x***, we have:

```
semaphore x_sem; // (initially  = 0)
int x_count = 0;
```

- The operation **`x.wait()`** can be implemented as:

```
x_count++;
if (next_count > 0)
    signal(next);
else
   signal(mutex);
wait(x_sem);
x_count--;
```

Implementation (Cont.)

- The operation **`x.signal()`** can be implemented as:

```
if (x_count > 0)
{
   next_count++;
   signal(x_sem);
   wait(next);
   next_count--;
}
```

Resuming Processes within a Monitor

- If several processes queued on condition variable `x`, and `x.signal()` is executed, which process should be resumed?
- FCFS frequently not adequate
- Use the **conditional-wait** construct of the form

  `x.wait(c)`

  where:
  - `c` is an integer (called the priority number)
  - The process with lowest number (highest priority) is scheduled next

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

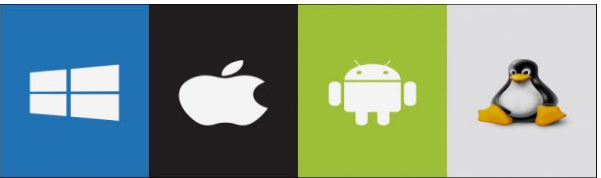
- **Mutual exclusion**: only one process at a time can use a resource
- **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption**: a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait**: there exists a set $\{P_0, P_1, \ldots, P_n\}$ 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, …, 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 with Mutex Locks

- Deadlocks can occur via system calls, locking, etc.
- See example box in text page 318 for mutex deadlock
- Deadlock can be occurred in a multithreaded Pthread program using mutex locks. Let's see how it can be occurred. An unlocked mutex is initialized by the pthread_mutex_init() function.
- Using pthread_mutex_lock() and pthread_mutex_unlock() Mutex locks are acquired and released. If a thread try to acquire a locked mutex, the call to pthread_mutex_lock() blocks the thread until the owner of the mutex lock invokes pthread_mutex_unlock().
- Let's take an example, two Mutex locks are created in the following Code –
- /* Create and initialize the mutex locks */
- pthread mutex t mutex1;
- pthread mutex t mutex2;
- pthread mutex init(&mutex1,NULL);
- pthread mutex init(&mutex2,NULL);

Deadlock with Mutex Locks

- Next, two threads - thread1 and thread2 - are created, and both these threads have access to both mutex locks.
- Thread1 and thread2 run in the functions dosomework_1 and dos
- /* thread1 runs in this function */
- void *dosomework_1(void *param)
- { pthread mutex lock(&mutex1);
- pthread mutex lock(&mutex2); /**
- * Do some work */
- pthread mutex unlock(& mutex2);
- pthread mutex unlock(& mutex2);
- pthread exit(0); }
- /* thread2 runs in this function */
- void *dosomework_2(void *param)
- { pthread mutex lock(&mutex2);
- pthread mutex lock(&mutex1); /**
- * Do some work */
- pthread mutex unlock(&mutex1);
- pthread mutex unlock(&mutex2);
- pthread exit(0); }



Deadlock with Mutex Locks

- In this example, thread1 tries to acquire the mutex locks in the order
- mutex1,
- mutex2,
- while thread 2 tries to acquire the mutex locks in the order
- mutex2,
- mutex1,
- Deadlock is possible if thread1 acquires mutex1 while thread2 acquires mutex2. Even though deadlock is possible, it will not occur if thread1 can acquire and release the mutex locks for mutex1 and mutex2 before thread2 tries to acquire the locks.
- CPU scheduler scheduled the order in which the threads run. The above example illustrates a problem with handling deadlocks: it is difficult to identify and test for deadlocks that may occur only under certain scheduling circumstances

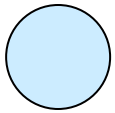
Resource-Allocation Graph

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

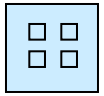
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed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

Resource-Allocation Graph (Cont.)

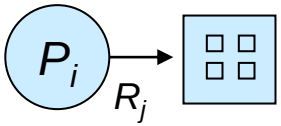
- Process



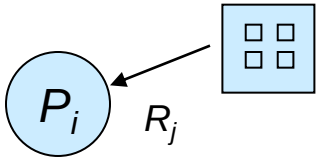
- Resource Type with 4 instances



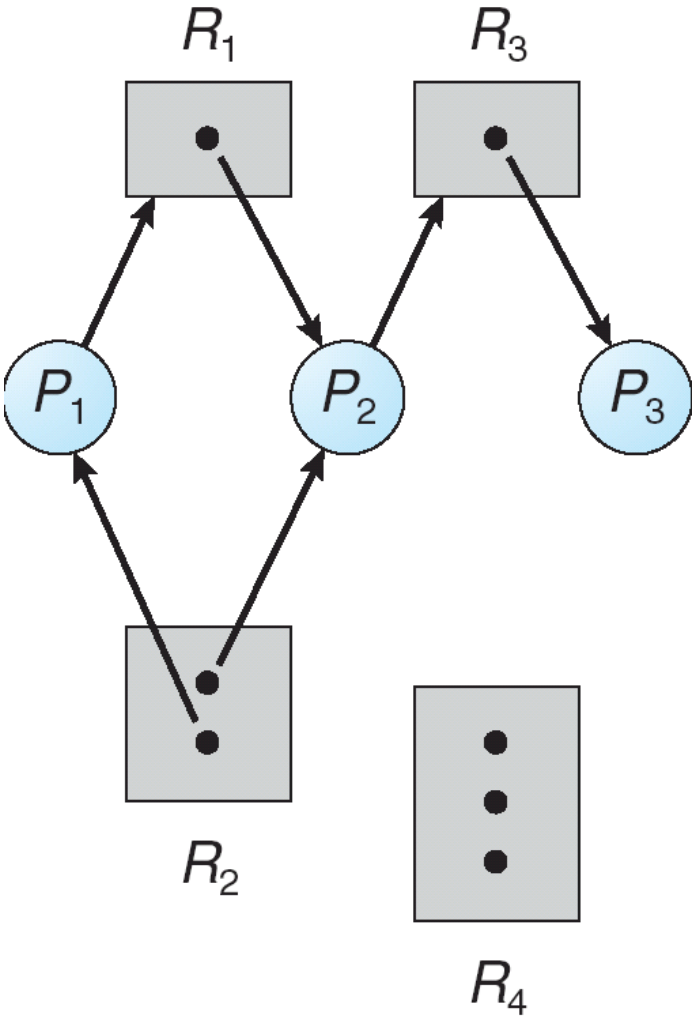
- P_i requests instance of R_j



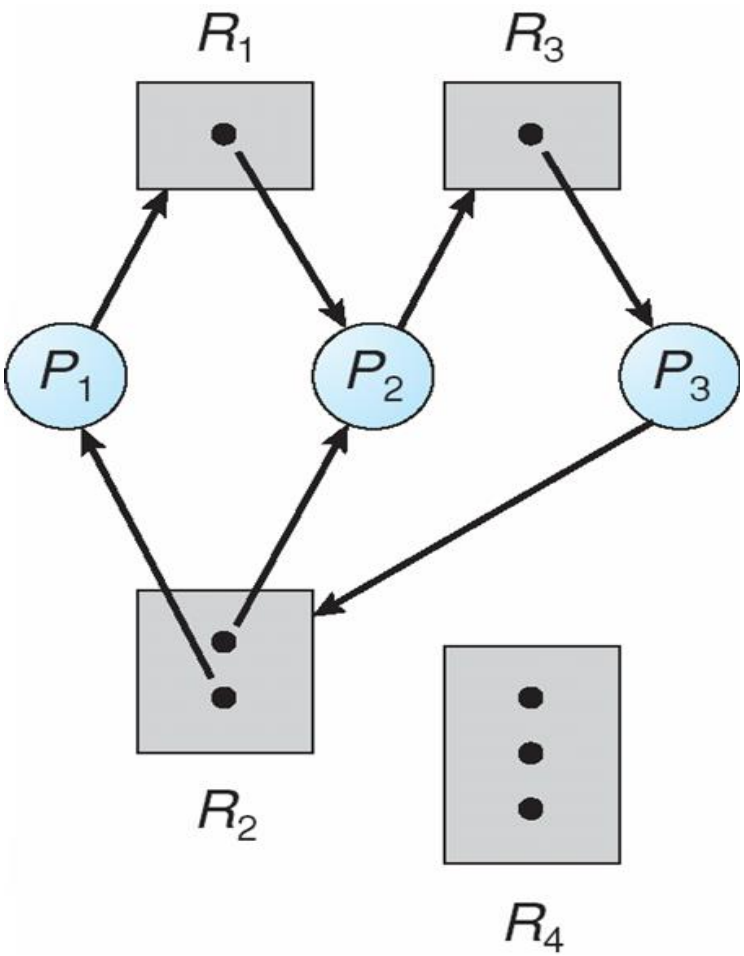
- P_i is holding an instance of R_j



Example of a Resource Allocation Graph

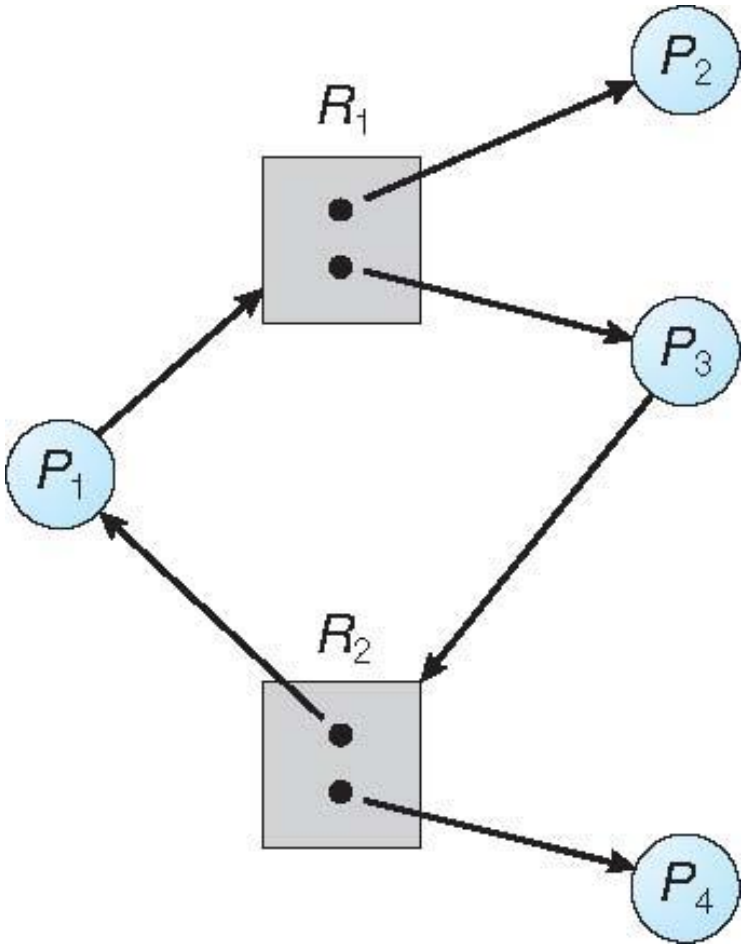


Resource Allocation Graph With A Deadlock



Sheli Sinha Chaudhuri

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:
  - Deadlock prevention
  - Deadlock avoidence
- 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 (e.g., read-only files); must hold for non-sharable 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 allocated to it.
  - 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 Example

```
/* thread one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /** * Do some work */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);
   pthread_exit(0);
}
/* thread two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /** * Do some work */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);
   pthread_exit(0);
}
```

Deadlock Example with Lock Ordering

```
void transaction(Account from, Account to, double amount)
{
   mutex lock1, lock2;
   lock1 = get_lock(from);
   lock2 = get_lock(to);
   acquire(lock1);
      acquire(lock2);
         withdraw(from, amount);
         deposit(to, amount);
      release(lock2);
   release(lock1);
}
```

Transactions 1 and 2 execute concurrently. Transaction 1 transfers $25 from account A to account B, and Transaction 2 transfers $50 from account B to account A

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

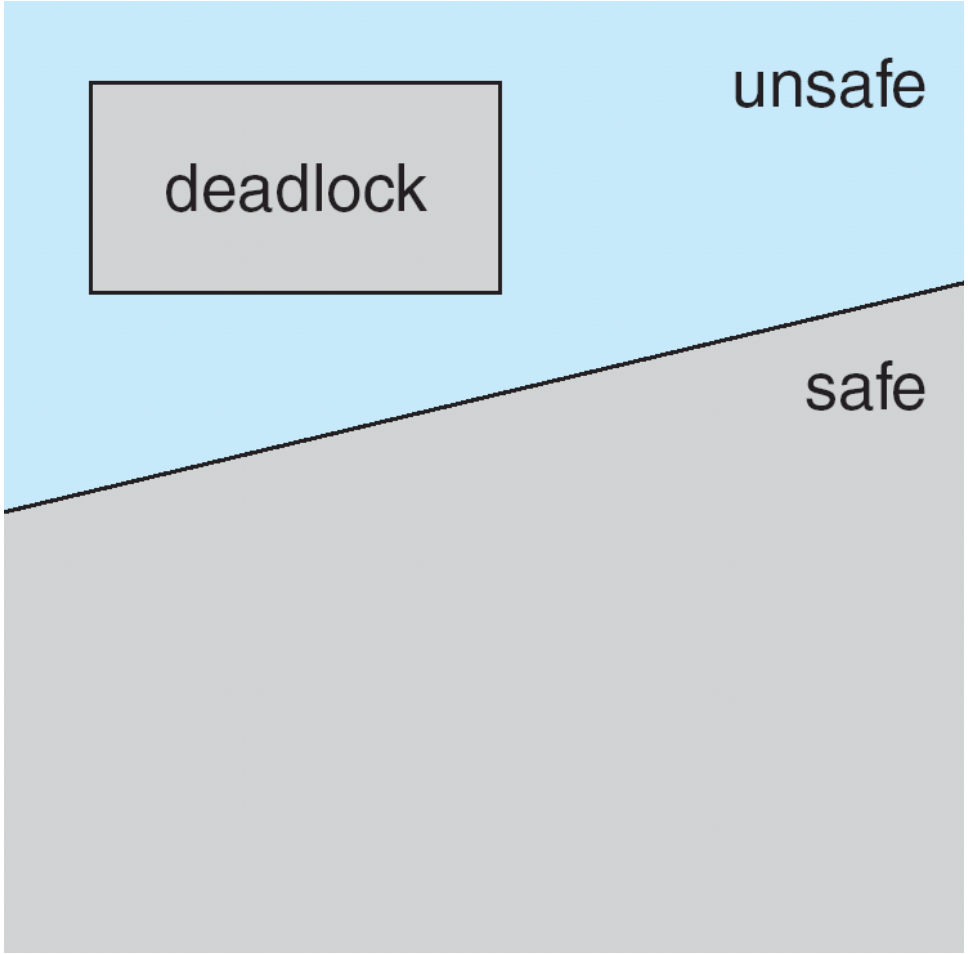
Safe State

- 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 sequence <P_1, P_2, …, 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 of System Deadlock

- 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



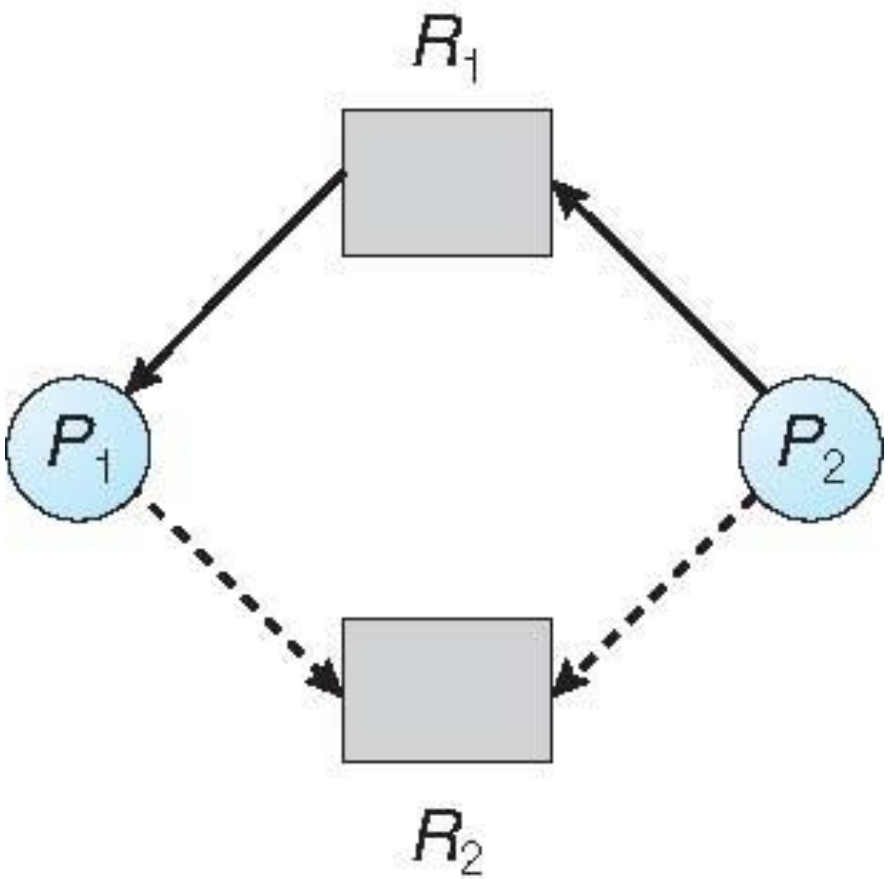
Avoidance Algorithms

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

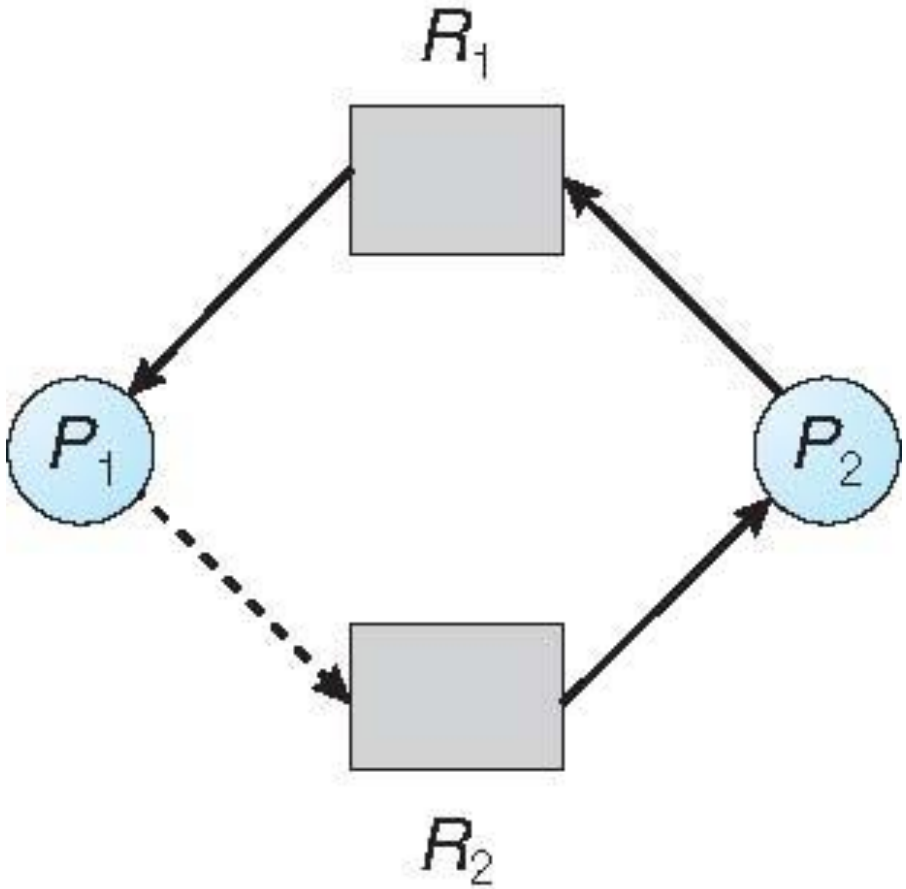
Resource-Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_i 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***:* $n \times m$ matrix. If $Max\,[i,j] = k$, then process P_i may request at most k instances of resource type R_j
- **Allocation***:* $n \times m$ matrix. If Allocation$[i,j] = k$ then P_i is currently allocated k instances of R_j
- **Need***:* $n \times m$ 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, …, *n*- 1**

2. Find an ***i*** such that both:

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

   (b) $\boldsymbol{Need_i \leq Work}$

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

3. $\boldsymbol{Work = Work + Allocation_i}$

   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

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

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

$$\boldsymbol{Available = Available - Request_i;}$$
$$\boldsymbol{Allocation_i = Allocation_i + Request_i;}$$
$$\boldsymbol{Need_i = Need_i - Request_i;}$$

- If safe $\Rightarrow$ the resources are allocated to $\boldsymbol{P_i}$
- If unsafe $\Rightarrow$ $\boldsymbol{P_i}$ 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

Example: P_1 Request (1,0,2)

- Check that Request $\leq$ Available (that is, $(1,0,2) \leq (3,3,2)$ $\Rightarrow$ 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	

- Executing safety algorithm shows that sequence < $\boldsymbol{P_1}$, $\boldsymbol{P_3}$, $\boldsymbol{P_4}$, $\boldsymbol{P_0}$, $\boldsymbol{P_2}$> satisfies safety requirement
- Can request for (3,3,0) by $\boldsymbol{P_4}$ be granted?
- Can request for (0,2,0) by $\boldsymbol{P_0}$ be granted?

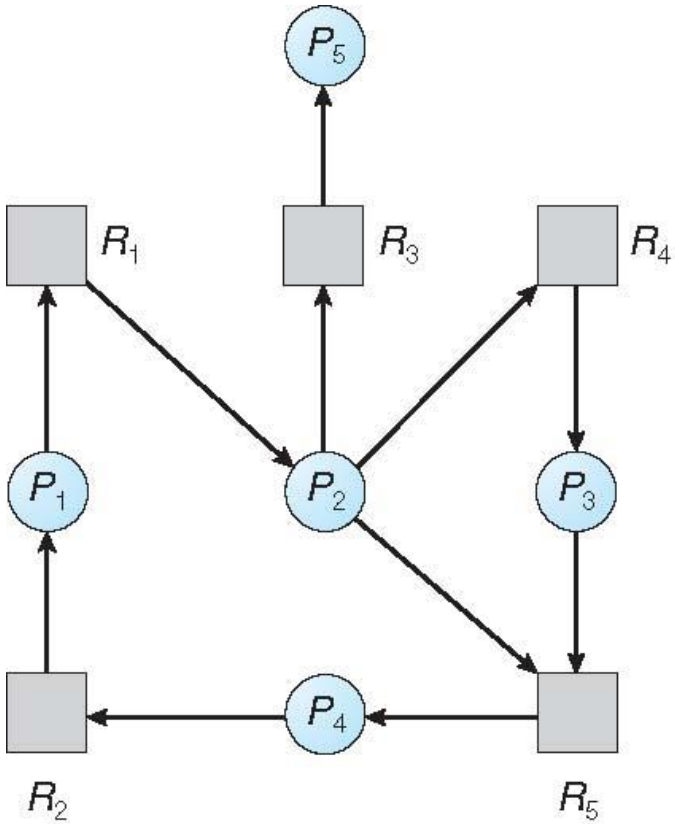
Deadlock Detection

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

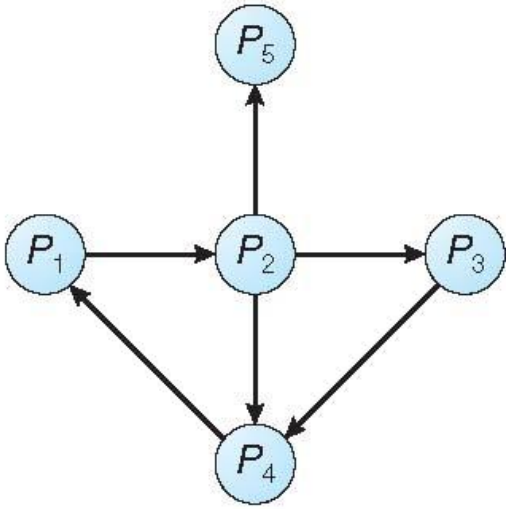
Single Instance of Each Resource Type

- 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



(a)



(b)

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, …, *n***, if $\boldsymbol{Allocation_i \neq 0}$, then ***Finish*[i] = *false***; otherwise, ***Finish*[i] = *true***

2. Find an index ***i*** such that both:
   (a) ***Finish*[*i*] == *false***
   (b) $\boldsymbol{Request_i \leq Work}$

   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.

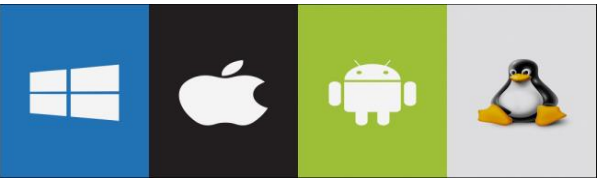
Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  1. Priority of the process
  2. How long process has computed, and how much longer to completion
  3. Resources the process has used
  4. Resources process needs to complete
  5. How many processes will need to be terminated
  6. 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

End of Chapter 6



Sheli Sinha Chaudhuri