Chapter-3
Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors

Background

- A cooperating process can affect or be affected by other processes.
- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

Cooperating Processes: Shared Memory

```
#define BUFFER_SIZE 10

typedef struct {

        . . .

} item;

item buffer[BUFFER_SIZE];

int in = 0;

int out = 0;
```

Producer and Consumer: Producer routine

```
while (true)

  {
```

```
        /* produce an item and put in nextProduced*/
                        while (count == BUFFER_SIZE)
                                ; // do nothing
                        buffer [in] = nextProduced;
                        in = (in + 1) % BUFFER_SIZE;
                        count++;
        }
```

Producer and Consumer: Consumer routine

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

Producer and Consumer

- Both producer and consumer routines are correct separately.
- They may not function correctly when run concurrently.
- If counter is 5 and both routines execute “counter++” and “counter--” concurrently, value of counter may be 4 or 5 or 6. (while correct value of counter is 5)
- count++ could be implemented as

  $register_1 = count$
  $register_1 = register_1 + 1$
  $count = register_1$

- count-- could be implemented as

  $register_2 = count$
  $register_2 = register_2 - 1$

count = $register_2$

- Consider this execution interleaving with "count = 5" initially (interleaving may be in possible in different ways):

T0: producer execute $register_1$ = count {$register_1$= 5}
T1: producer execute $register_1$ = $register_1$ + 1 {$register_1$= 6}
T2: consumer execute $register_2$ = count {$register_2$ = 5}
T3: consumer execute $register_2$ = $register_2$ - 1 {$register_2$ = 4}
T4: producer execute count = $register_1$ {count = 6 }
T5: consumer execute count = $register_2$ {count = 4}

- An incorrect state occurs when both the processes are allowed to manipulate the variable counter concurrently.
- A situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the execution takes place, is called race condition.
- Thus only one process should be allowed to manipulate the counter at a time, thus synchronization is required.

The Critical-Section Problem

- If *n* processes are competing to use some shared data.
- Each process has a code segment, called *critical section*, in which the shared data is accessed e.g. changing common variables, updating a table, writing a file etc.
- It is necessary to ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

Solution to CS using LOCKS

- Race conditions can be prevented by protecting the critical section by LOCK
- A process must acquire a lock before entering a critical section and releases the lock when it exits the critical section.

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

Solution to Critical-Section Problem

General structure of process P_i

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

The Critical-Section Problem

- Each process must request permission to enter its critical section which is implemented as entry section.
- Critical section is followed by exit section.
- The remaining code is remainder section.
- A solution to Critical-Section Problem must satisfy three requirements.

Three requirements for the solution to Critical-Section Problem

1. Mutual Exclusion:

If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

2. Progress:

If no process is executing in its critical section and there exist some processes that wish to enter their critical section,

then the selection of the processes that will enter the critical section next cannot be postponed indefinitely

(only those processes can enter which are not executing in their remainder section)

3. Bounded Waiting:

A bound must exist 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

Critical-Section Problem

- Many kernel mode processes may be involved in race condition.
- Two general approaches are used to handle critical sections in OS
  - Preemptive kernels
    - It allows a process to be preempted while it is running in kernel mode.
  - Nonpreemptive kernels
    - A process will run until it exits kernel mode.
- Nonpreemptive kernels are free from race condition on kernel data structures.
- Nonpreemptive kernels: Windows XP, 2000, prior to Linux 2.6
- Preemptive kernels must be carefully designed to ensure that shared kernel data are free from race conditions.
- Preemptive kernels is difficult to design for SMP.
- A preemptive kernels is suitable for real-time programming having short response time.
- Preemptive kernels: Linux 2.6 onwards, Solaris, IRIX

Peterson's Solution

- 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.

Algorithm for Process P_i

```
do {

    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

Solution to CS using LOCKS

- Race conditions can be prevented by protecting the critical section by LOCK
- A process must acquire a lock before entering a critical section and releases the lock when it exits the critical section.

```
do {

    acquire lock

        critical section

    release lock

        remainder section

} while (TRUE);
```

Synchronization Hardware

- Many systems provide hardware support for locking the critical section code
- Disabling the interrupts on Uniprocessor sytems
- Preventing interrupts to happen when a shared variable is being modified.

- Such code would execute without preemption which is the approach used for non-preemptive.
- This approach is inefficient for multiprocessor systems
- Operating systems using this is not broadly scalable
- Modern machines provide special atomic hardware instructions
    - TestAndSet()
    - Swap()

TestAndSet Instruction

- TestAndSet() instruction is executed atomically.

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

Solution using TestAndSet

- Shared boolean variable lock, initialized to false.

```
do {
    while (TestAndSetLock(&l
        ; // do nothing

        // critical section

    lock = FALSE;

        // remainder section
}while (TRUE);
```

Swap Instruction

- Swap() is executed atomically

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

*a = *b;

*b = temp:

}

Solution using Swap

- Shared Boolean variable lock initialized to FALSE
- Each process has a local Boolean variable key.

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

                // critical section
                lock = FALSE;
                // remainder section
}while (TRUE);
```

Bounded waiting algorithm using TestAndSet()

- TestAndSet() and Swap() algorithms support mutual exclusion but do not follow bounded waiting.
- Bounded waiting algorithm using TestAndSet() instruction satisfies all critical section problems.
- Following data structures are initialized as false

boolean waiting [n];

boolean lock;

```
do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
       key = TestAndSet(&lock);
    waiting[i] = FALSE;

       // critical section

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
       j = (j + 1) % n;

    if (j == i)
       lock = FALSE;
    else
       waiting[j] = FALSE;

       // remainder section
}while (TRUE);
```

Semaphore

- Semaphore *S*: integer variable
- Two standard indivisible (atomic) operations can access semaphore
- wait() and signal()

(Originally called P() and V())

- All the modifications to the semaphore in the wait() and signal() must be executed atomically.

wait()

```
wait(S) {
    while S <= 0
       ; // no-op
    S--;
}
```

signal()

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

Semaphore: usage

- Counting semaphore: integer value can range over an unrestricted domain
- Binary semaphore: integer value can range only between 0 and 1, it is simpler to implement
  - Also known as mutex locks

Semaphore: usage-1

- Binary semaphore can be used to deal with critical section problem for multiple processes.
- Semaphore mutex is initialized as 1.

```
do {
   waiting(mutex);

      // critical section

   signal(mutex);

      // remainder section
}while (TRUE);
```

Semaphore: usage-2

- Counting semaphore can be used to control access to a given resource consisting of a finite number of instances.
- Semaphore is initialized as number of instances of a resource type.
- Each process wishes to use the resource, will perform wait() operation and decrement semaphore by one.
- When a process release the resource, will perform signal() operation and increment semaphore by one.
- Zero value of semaphore means no instance of the resource is available.

Semaphore: usage-3

- Two concurrent processes, P1 executing statement S1 and P2 executing statement S2.
- S1 should be followed by S2.
- Semaphore synch is initialized as zero.

```
S1;                          wait(synch);
signal(synch);               S2;
```

Semaphore: Implementation

- Semaphore requires busy waiting.
- When a process is in its critical section, any other process tries to enter in its critical section must loop continuously to check condition $s \leq 0$.
- This type of semaphore is called a spinlock.
- Busy waiting wastes CPU cycle.

Modified definition of Semaphore

- When a process executes wait() and finds that value of semaphore is not positive, process blocks itself rather than busy waiting by block() operation.
- The block operation places a process into a waiting queue associated with semaphore, and the state of the process is switched to the waiting state.

Modified definition of Semaphore

- A blocked process on a semaphore will be restarted when some other process executes a signal() operation.
- Blocked process is restarted by a wakeup() operation, which changes the process from waiting state to ready state.
- Block() and wakeup() are provided as basic system calls by operating system.

Modified definition of Semaphore

- Each semaphore has an integer value and a list of processes.
- When a process must wait on a semaphore, it is added to the list of processes.
- A signal() removes one process from the list of waiting processes and awakens that process.

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

```
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);
        }
}
```

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

- Implementation must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- Thus, it becomes the critical section problem where the wait and signal code are placed in the critical section.
- In uniprocessor system, interrupts can be disabled during wait() and signal() operations.
- In SMP, wait() and signal() should be performed atomically.

- If critical section code is smaller, spinlock is a good solution
- There will be little busy waiting if critical section rarely occupied
- If applications spend lots of time in critical sections then block() and wakeup() solution is good although it has context switching.

Deadlock and Starvation

- Deadlock
  - Two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Starvation
  - Indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended (if implementation is LIFO).

Semaphore: Deadlocks

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
.	.
.	.
.	.
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

- Two processes share a common fixed size buffer. One of them, the producer, puts information in buffer. Other one, the consumer, takes it out.

Problems:

- Producer wants to put a new item but buffer is already full.
- Consumer wants to take a new item but buffer is empty.

Solutions:

- When buffer is full, producer is sent to sleep and gets awakened when consumer takes an item from buffer.
- When buffer is empty, consumer is sent to sleep and gets awakened when producer puts an item in buffer.
- Shared data
  - define N 10

    /*buffer size*/
  - semaphore full = 0

    /*counts filled buffer slots*/
  - semaphore empty = N

    /*counts empty buffer slots*/
  - semaphore mutex = 1

    /*controls access to critical regions*/

Producer Process	**Consumer Process**
do { … //produce an item in nextp	do { wait (full);

…
wait (empty);
wait (mutex);
…
//add nextp to buffer
…
signal (mutex);
signal (full);
} while (1);
	wait (mutex);
	…
	//remove an item from buffer to nextc
	…
	signal (mutex);
	signal (empty);
	…
	//consume the item in nextc
	…
	} while (1);

Readers-Writers Problem

- A data object can be shared among several concurrent processes. Some processes (readers) only read the content of the object but some processes (writers) write the content in object.

Problems:

- When two readers access the object, there is no adverse effects will result.
- When one writer and other reader; or two writer access the object simultaneously, inconsistency may occur.

Solutions:

- Writers should have exclusive access of the object.

Shared data

```
semaphore mutex, wrt;
int readcount;
```

- semaphore mutex = 1

  /*to ensure mutual exclusion when readcount is being updated*/

- semaphore wrt = 1

  /*controls access to the shared object*/

- int readcount = 0

/*number of processes reading the shared object*/

Readers-Writers Problem (Writer Process)

```
do {
   wait(wrt);
      . . .
   // writing is performed
      . . .
   signal(wrt);
}while (TRUE);
```

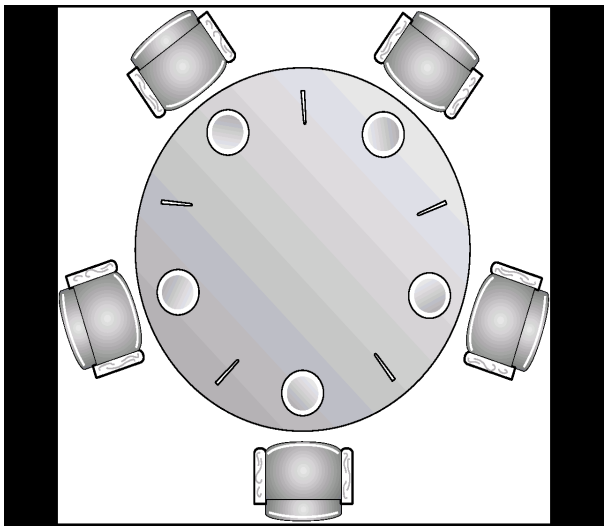
Readers-Writers Problem (Reader Process)

```
do {
   wait(mutex);
   readcount++;
   if (readcount == 1)
      wait(wrt);
   signal(mutex);
      . . .
   // reading is performed
      . . .
   wait(mutex);
   readcount--;
   if (readcount == 0)
      signal(wrt);
   signal(mutex);
}while (TRUE);
```

Dining-Philosophers Problem

- Five philosophers spend their lives thinking and eating.
- Philosophers share a common circular table surrounded by five chairs, each belonging to one philosopher.
- In center of the table is a bowl of rice (or spaghetti), and the table is laid with five single chopsticks.

- From time to time, philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors).



- A philosopher may pick up only one chopstick at a time.
- She can not pick up a chopstick that is already in hand of a neighbor.
- When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks.
- When she finishes eating, she puts down both of her chopsticks and start thinking again.
- The dinning philosopher problem is considered a classic problem because it is an example of a large class of concurrency-control problems.
- It is a representation of the need to allocate several resources among several processes in a deadlock and starvation free manner.

Dining-Philosophers Problem: Solution

- Shared data
  - semaphore chopstick[5];
  - Initially all values are 1
  - A philosopher tries to grab the chopstick by executing wait operation and releases the chopstick by executing signal operation on the appropriate semaphores.

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   // eat
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   // think
      . . .
}while (TRUE);
```

- This solution guarantees that no two neighbors are eating simultaneously but it has a possibility of creating a deadlock and starvation.
- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks if both chopsticks are available.
- An odd philosopher picks up her left chopstick first and an even philosopher picks up her right chopstick first.
- Finally no philosopher should starve.

Problems with Semaphores

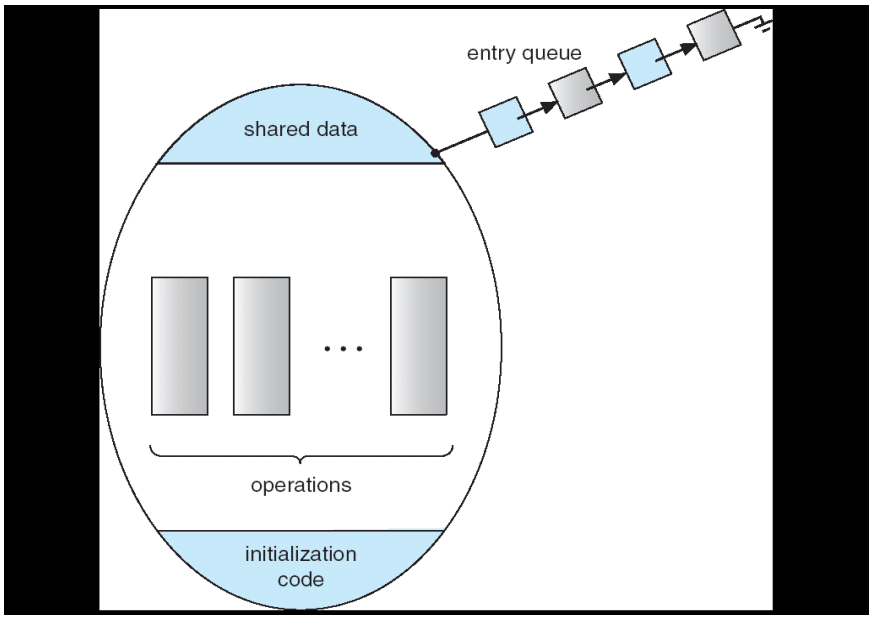
- Correct use of semaphore operations is necessary.
- Incorrect usage:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Monitor encapsulates private data with public methods to operate on that data.
- The representation of monitor type can not be used directly by the various processes.
- A procedure defined within the monitor can access only those variables declared locally within the monitor and its formal parameters.
- The local variables of a monitor can be accessed by only the local procedures.

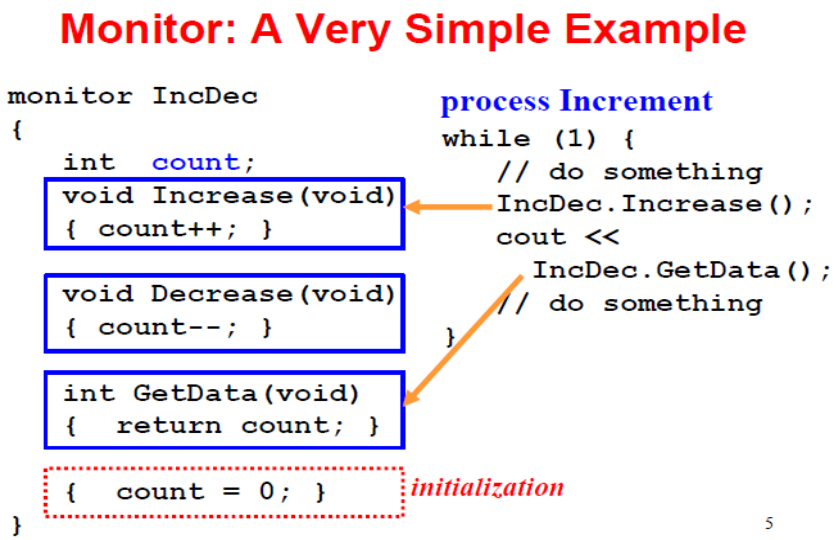
- No more than one process can be executing within a monitor. Thus, mutual exclusion is guaranteed within a monitor.
- When a process calls a monitor procedure and enters the monitor successfully, it is the only process executing in the monitor.
- When a process calls a monitor procedure and the monitor has a process running, the caller will be blocked outside of the monitor.

Schematic view of a Monitor



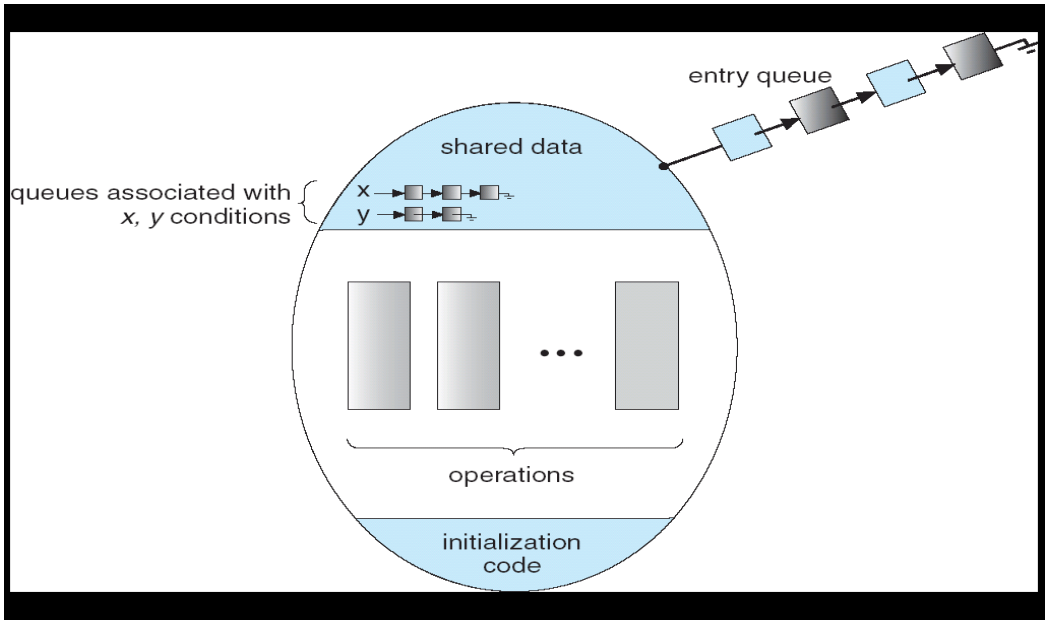

```
monitor monitor name
{
   // shared variable declarations

   procedure P1 ( . . . ) {
      . . .
   }

   procedure P2 ( . . . ) {
      . . .
   }

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

   initialization code ( . . . ) {
      . . .
   }
}
```

Monitor Example



Monitors with Condition Variables

- While the process is executing within a monitor, a programmer may want to block this process and force it to wait until an event occurs.
- Thus, each programmer-defined event is associated with a condition variable.
- The only operations that can be invoked on a condition variable are wait() and signal().
- A process that executes a condition wait blocks immediately and is put into the waiting list of that condition variable.
- Thus this process is waiting for the indicated event to occur.
- Condition signal is used to indicate an event has occurred.
- If there are processes waiting on the signaled condition variable, one of them will be released.
- If there is no waiting process waiting on the signaled condition variable, this signal is lost as if it never occurs.



- condition x, y;

Two operations on a condition variable:

- x.wait() – a process that invokes the operation is suspended.
- x.signal() – resumes one of processes (if any) that invoked x.wait()

- A process invokes signal() operation: P
- A process (that was suspended) is resumed: Q

Problem is who will continue P or Q. two possibilities:

- Signal and Wait:
  - P waits until Q leaves the monitor or wait for some condition
- Signal and Continue:
  - Q waits until P leaves the monitor or wait for some condition

Solution to Dining Philosophers

- Solution to Dining Philosophers through monitor provides deadlock free solution (no surety for starvation).
- Instead of picking up chopsticks one by one, restriction is that a philosopher can eat only if both the chopsticks are available.

- Data structure:

```
enum {thinking, hungry, eating} state[5];

condition self[5];
```

- Philosopher i can set the variable state[i] = eating only if her two neighbors are not eating i.e.

  state [(i+4)%5] !=eating

  state [(i+1)%5] !=eating

- Philosopher i can delay herself when she is hungry but unable to obtain the chopsticks she needs.

```
dp.pickup(i);
    ...
    eat
    ...
dp.putdown(i);
```

```
monitor dp
{
   enum {THINKING, HUNGRY, EATING}state[5];
   condition self[5];

   void pickup(int i) {
      state[i] = HUNGRY;
      test(i);
      if (state[i] != EATING)
         self[i].wait();
   }

   void putdown(int i) {
      state[i] = THINKING;
      test((i + 4) % 5);
      test((i + 1) % 5);
   }

   void test(int i) {
      if ((state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING)) {
           state[i] = EATING;
           self[i].signal();
      }
   }

   initialization_code() {
      for (int i = 0; i < 5; i++)
         state[i] = THINKING;
    }
}
```

```
monitor dp
{
   enum {THINKING, HUNGRY, EATING}state[5];
   condition self[5];
```

```
    void pickup(int i) {
       state[i] = HUNGRY;
       test(i);
       if (state[i] != EATING)
          self[i].wait();
    }

    void putdown(int i) {
       state[i] = THINKING;
       test((i + 4) % 5);
       test((i + 1) % 5);
    }

    void test(int i) {
       if ((state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING)) {
          state[i] = EATING;
          self[i].signal();
       }
    }

    initialization_code() {
       for (int i = 0; i < 5; i++)
          state[i] = THINKING;
     }
}
```

Implementing a Monitor using Semaphore

- For each monitor, a semaphore mutex (initialized as 1) is used.
- A process must execute wait(mutex) before entering the monitor and must execute signal(mutex) after leaving the monitor.

Semaphore

- mutex = 1 for monitor
- next = 0 for signaling process suspending itself
- x_sem = 0 for suspending processes on condition x

Integer variable

- next_count = 0 count of processes suspended on next.
- x_count =0 count of processes suspended on x_sem

```
wait(mutex);
    ...
  body of F
    ...
if (next_count > 0)
  signal(next);
else
  signal(mutex);
```

x.wait()

x.signal()

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

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

Resuming processes within a Monitor

- *Conditional-wait* construct: *x.wait*(*c*)
  - *c* – integer expression evaluated when the wait operation is executed.
  - value of *c* (*priority number*) stored with the name of the process that is suspended.
  - when *x.signal* is executed, process with smallest associated priority number is resumed next.
- Check two conditions to establish correctness of system:
  - User processes must always make their calls on the monitor in a correct sequence.
  - Must ensure that an uncooperative process does not ignore the mutual-exclusion gateway provided by the monitor, and try to access the shared resource directly, without using the access protocols.

Questions

- **Define critical section problem and explain the requirements to be met by a solution to the critical section problem.**
- **Discuss Peterson's solution to critical section problem providing pseudocode.**
- **Give a solution to critical section problem using "Test-and-set" and show its correctness.**
- **Give a solution to critical section problem using "swap" and show its correctness.**
- **Discuss solution for bounded waiting mutual exclusion with TestAndSet().**
- **What is a semaphore? Define WAIT and SIGNAL operations.**
- **What is a semaphore? Explain the role of semaphores in solving mutual exclusion problem.**
- **Discuss the usage of semaphores and possibility of deadlock and starvation.**
- **Discuss implementation of semaphores to avoid spinlock.**
- **Discuss any two classic problems of synchronization.**
- **Write pseudocode to solve the Reader and Writer problem for both reader & writer processes?**
- **Explain Dining-Philosophers problem and its solution.**
- **Define the bounded buffer problem and its solution.**
- **What is monitor? Discuss the usage and syntax of monitor.**
- **Discuss the solution for Dining-Philosophers problem using monitors.**

Chapter 7: Deadlocks

- The Deadlock Problem
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

The Deadlock Problem

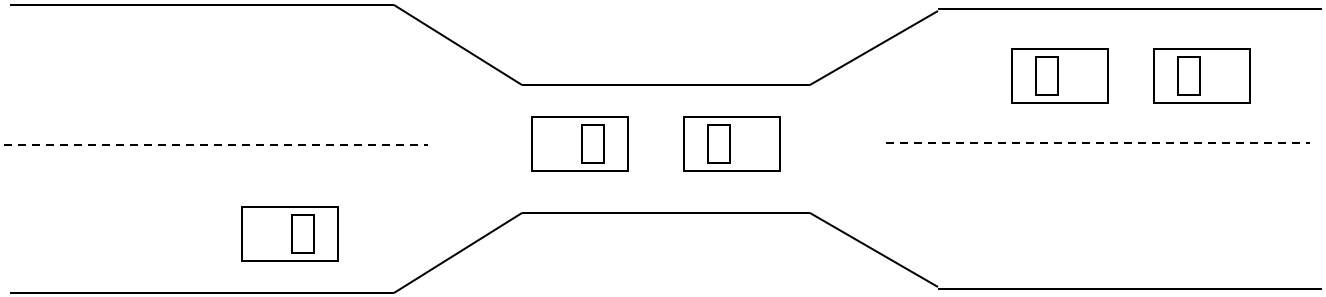
- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set

- Example
  - System has 2 disk drives
  - P_1 and P_2 each hold one disk drive and each needs another one

- Example
  - semaphores *A* and *B*, initialized to 1 P_0 P_1

    wait (A); wait(B) 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
- Note – Most OSes do not prevent or deal with deadlocks

System Model

- Resource types $R_1, R_2, \ldots, R_m$

  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 $\{P_0, P_1, \ldots, P_n\}$ 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.

Resource-Allocation Graph

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

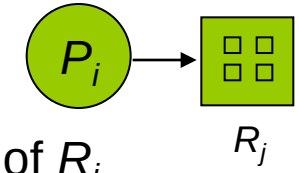
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed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

Resource-Allocation Graph (Cont.)

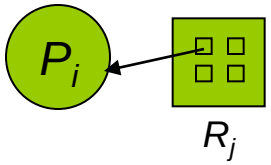
- Process

- Resource Type with 4 instances

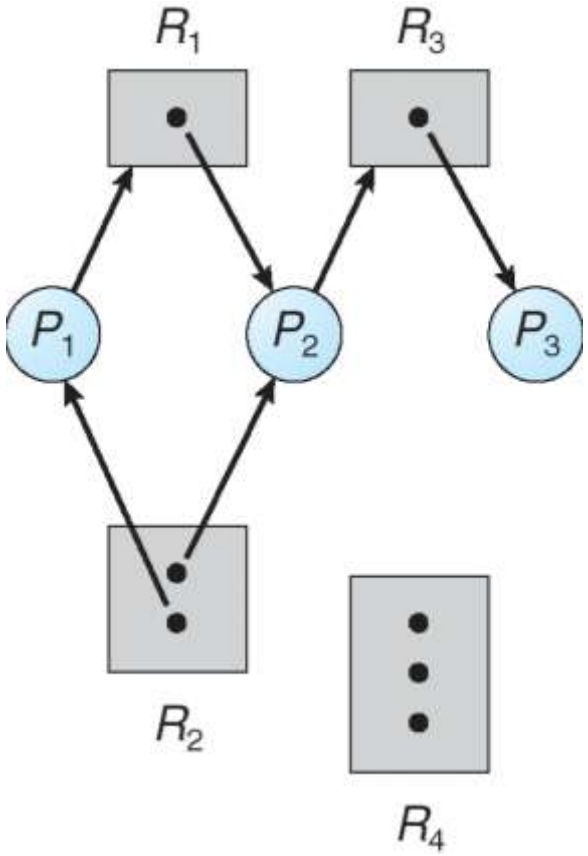
- P_i requests instance of R_j



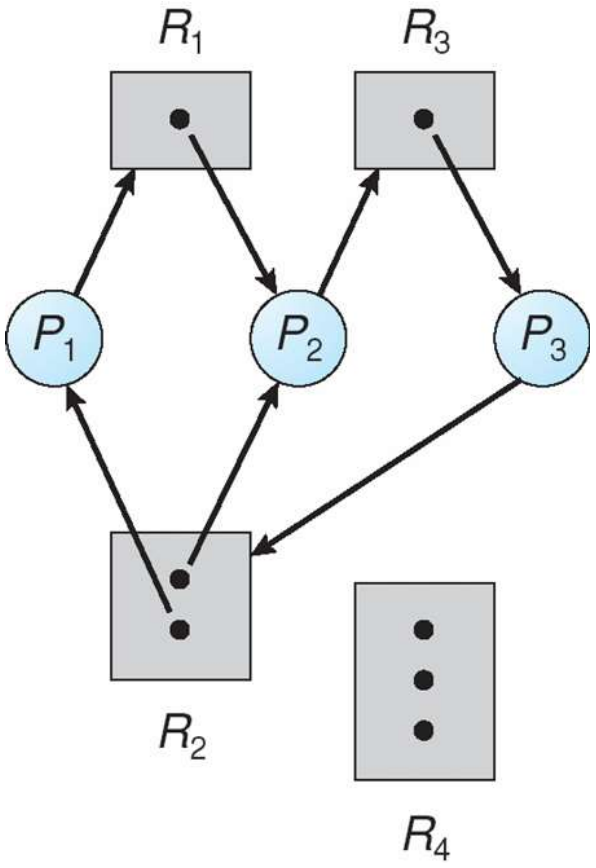
- P_i is holding an instance of R_j



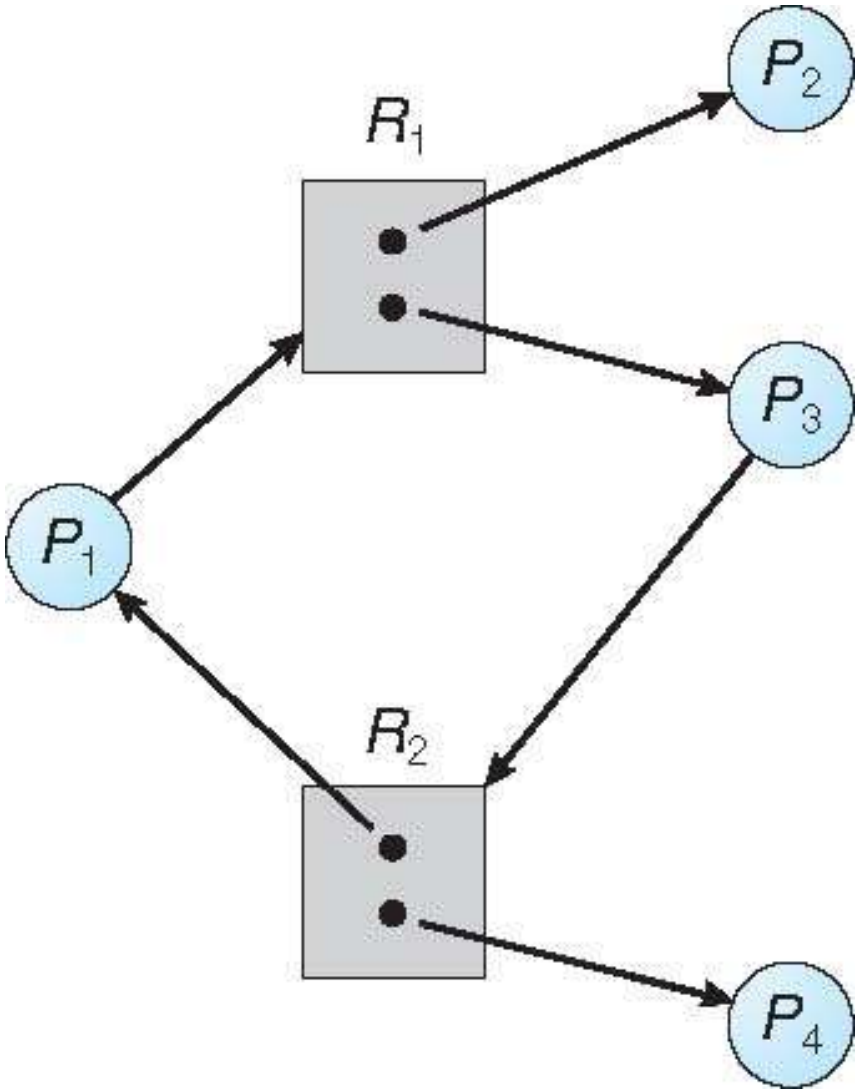
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



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

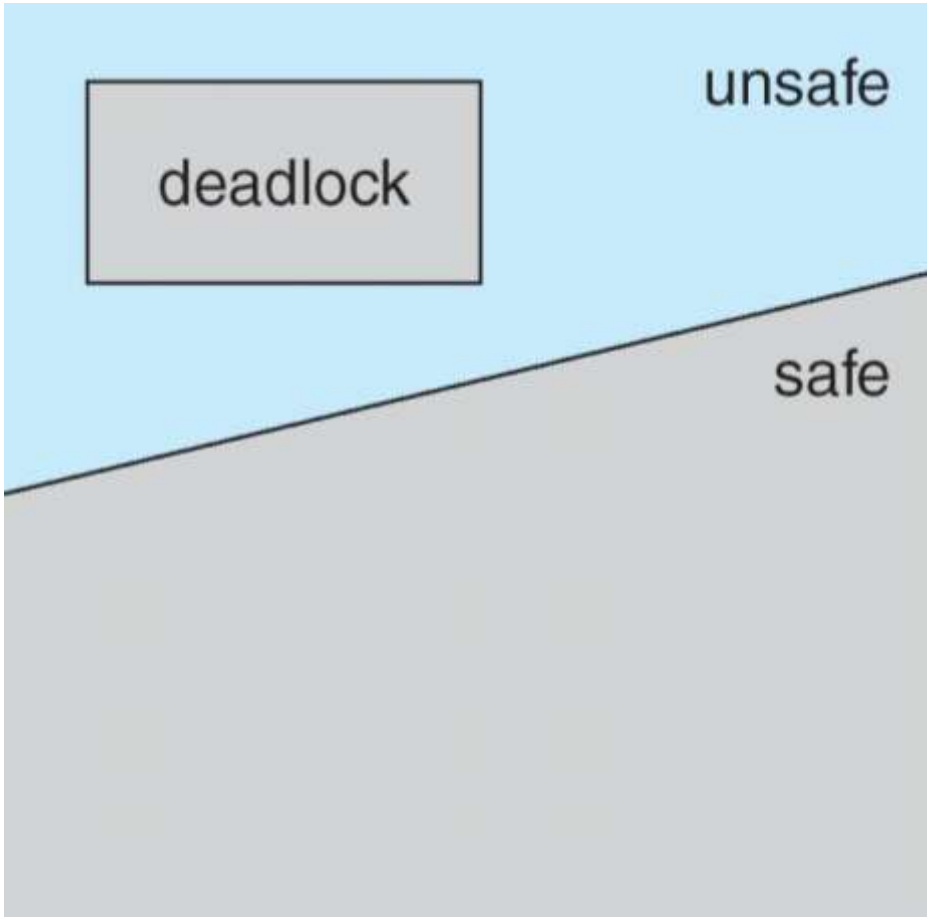
- System is in **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < I$

- 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



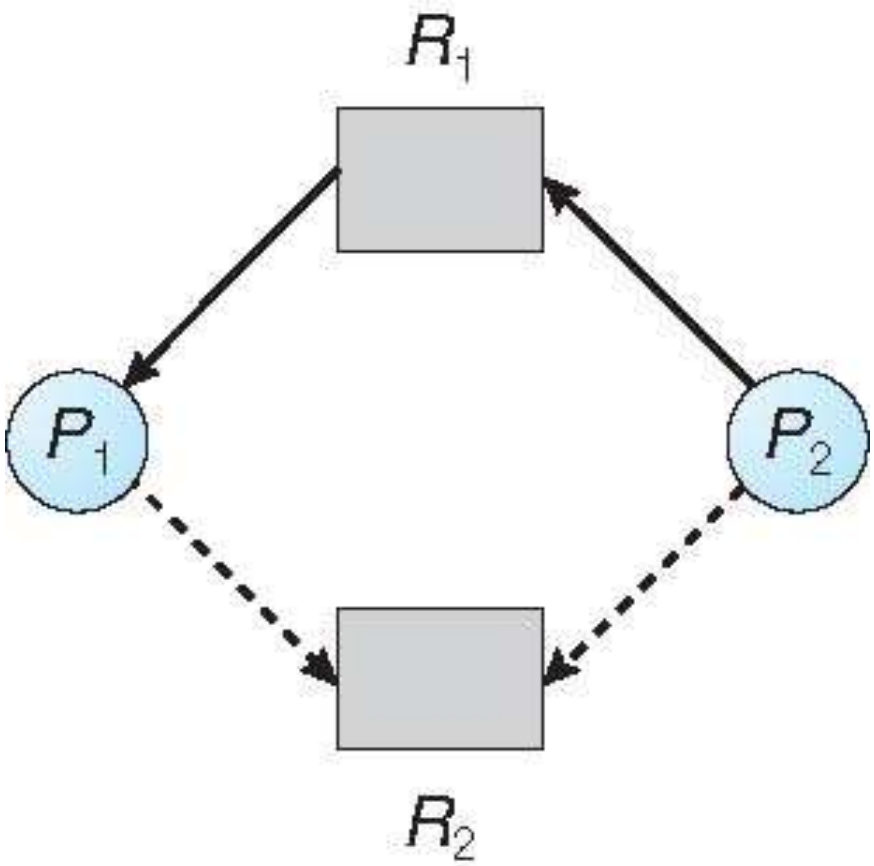
Avoidance algorithms

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

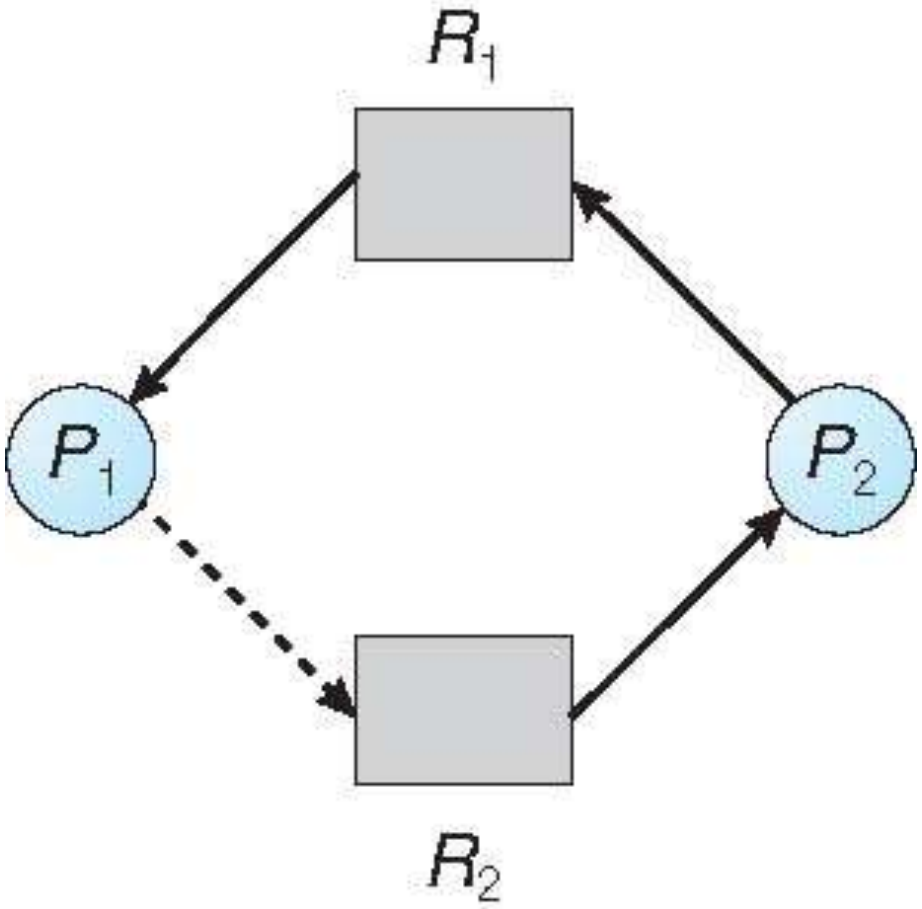
Resource-Allocation Graph Scheme

- **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, …, *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

Example: P_1 Request (1,0,2)

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ 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 < P_1, P_3, P_4, P_0, P_2> satisfies safety requirement
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

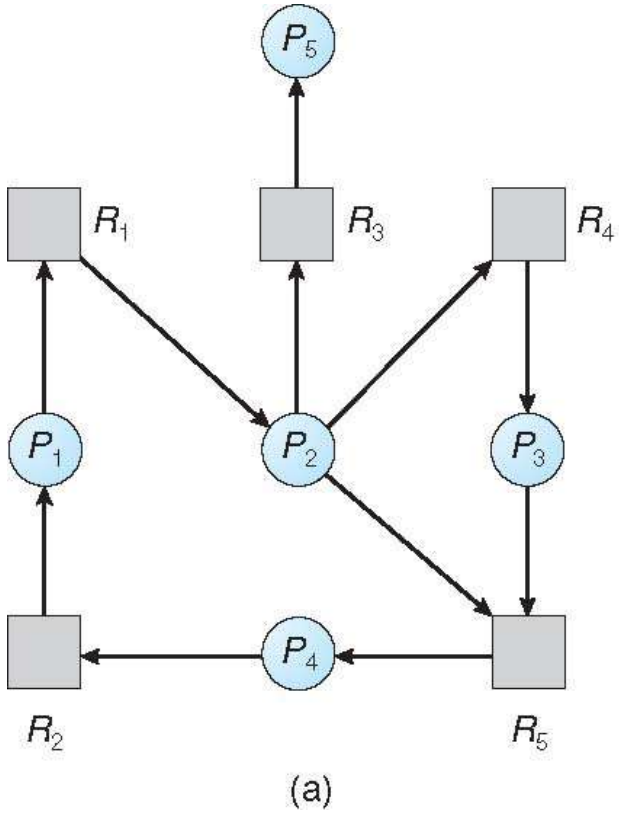
Deadlock Detection

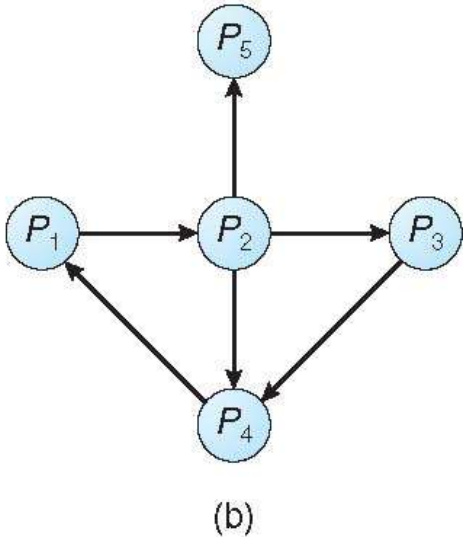
- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

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





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.