UNIT II

Process Synchronization

Background

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Shared-memory solution to bounded-butter problem (Chapter 4) allows at most $n - 1$ items in buffer at the same time. A solution, where all N buffers are used is not simple.
  - Suppose that we modify the producer-consumer code by adding a variable *counter*, initialized to 0 and incremented each time a new item is added to the buffer

Bounded-Buffer

- Shared data

```
#define BUFFER_SIZE 10
        typedef struct {
          . . .
        } item;
        item buffer[BUFFER_SIZE];
        int in = 0;
        int out = 0;
        int counter = 0;
```

Bounded-Buffer

- Producer process

```
item nextProduced;

while (1) {
    while (counter == BUFFER_SIZE)
        ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

Bounded-Buffer

- Consumer process

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

Bounded Buffer

- The statements
  counter++;
  counter--;
  must be performed *atomically*.
- Atomic operation means an operation that completes in its entirety without interruption.
- The statement "**count++**" may be implemented in machine language as:
  register1 = counter
  register1 = register1 + 1
  counter = register1
  The statement "**count—**" may be implemented as:
  register2 = counter
  register2 = register2 – 1
  counter = register2
- If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved.
- Interleaving depends upon how the producer and consumer processes are scheduled.
- Assume **counter** is initially 5. One interleaving of statements is:
  producer: **register1 = counter** (*register1 = 5*)
  producer: **register1 = register1 + 1** (*register1 = 6*)
  consumer: **register2 = counter** (*register2 = 5*)
  consumer: **register2 = register2 – 1** (*register2 = 4*)
  producer: **counter = register1** (*counter = 6*)
  consumer: **counter = register2** (*counter = 4*)

- The value of **count** may be either 4 or 6, where the correct result should be 5.

Race Condition

- **Race condition**: The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.
- To prevent race conditions, concurrent processes must be synchronized.

The Critical-Section Problem

- *n* processes all competing to use some shared data
- Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

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.
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.

- Assume that each process executes at a nonzero speed
- No assumption concerning relative speed of the *n* processes.

Initial Attempts to Solve Problem

- Only 2 processes, P_0 and P_1
- General structure of process P_i (other process P_j)

```
do {
        entry section
                critical section
        exit section
                reminder section
} while (1);
```

- Processes may share some common variables to synchronize their actions.

Algorithm 1

- Shared variables:
  - **int turn**;

    initially **turn = 0**
  - **turn - i** $\Rightarrow$ P_i can enter its critical section
- Process P_i

```
do {
        while (turn != i) ;
                critical section
        turn = j;
                reminder section
} while (1);
```

- Satisfies mutual exclusion, but not progress

Algorithm 2

Shared variables

- **boolean flag[2]**;
  initially **flag [0] = flag [1] = false.**
- **flag [i] = true** $\Rightarrow$ P_i ready to enter its critical section
- Process P_i

```
do {
        flag[i] := true;
        while (flag[j]) ;
                critical section
        flag [i] = false;
                remainder section
} while (1);
```

- Satisfies mutual exclusion, but not progress requirement.

Algorithm 3

- Combined shared variables of algorithms 1 and 2.
- Process P_i

```
do {
        flag [i]:= true;
        turn = j;
        while (flag [j] and turn = j) ;
                critical section
        flag [i] = false;
                remainder section
} while (1);
```

- Meets all three requirements; solves the critical-section problem for two processes.

Bakery Algorithm

- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.
- If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first.
- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...
- Notation <≡ lexicographical order (ticket #, process id #)
  - $(a,b) < c,d)$ if $a < c$ or if $a = c$ and $b < d$
  - max $(a_0,\ldots, a_{n-1})$ is a number, k, such that $k \geq a_i$ for i - 0, …, $n - 1$
- Shared data

```
boolean choosing[n];
int number[n];
```

Data structures are initialized to **false** and **0** respectively

```
do {
   choosing[i] = true;
   number[i] = max(number[0], number[1], …, number [n – 1])+1;
   choosing[i] = false;
   for (j = 0; j < n; j++) {
                while (choosing[j]) ;
                while ((number[j] != 0) && (number[j,j] < number[i,i])) ;
   }
        critical section
   number[i] = 0;
        remainder section
} while (1);
```

Synchronization Hardware

- Test and modify the content of a word atomically

```
. boolean TestAndSet(boolean &target) {
        boolean rv = target;
        tqrget = true;
        return rv;
    }
```

Mutual Exclusion with Test-and-Set

- Shared data:

```
boolean lock = false;
```

- Process P_i

```
do {
        while (TestAndSet(lock)) ;
                critical section
        lock = false;
                remainder section
    }
```

- Atomically swap two variables.

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

Mutual Exclusion with Swap

- Shared data (initialized to **false**):

```
boolean lock;
boolean waiting[n];
```

Process P_i

```
do {
        key = true;
        while (key == true)
                Swap(lock,key);
                critical section
        lock = false;
                remainder section
    }
```

Semaphores

- Synchronization tool that does not require busy waiting.
- Semaphore *S* – integer variable
- can only be accessed via two indivisible (atomic) operations

wait (*S*):

```
while S≤ 0 do no-op;
```

```
        S--;
    signal (S):
    S++;
```

Critical Section of *n* Processes

- Shared data:
  semaphore mutex; //initially *mutex* = 1
- Process *Pi:*

```
do {
   wait(mutex);
      critical section
   signal(mutex);
      remainder section
} while (1);
```

Semaphore Implementation

- Define a semaphore as a record

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

- Assume two simple operations:
  - **block** suspends the process that invokes it.
  - **wakeup(*P*)** resumes the execution of a blocked process **P**.
- Semaphore operations now defined as

```
wait(S):
        S.value--;
        if (S.value < 0) {
                add this process to S.L;
                block;
        }
signal(S):
        S.value++;
        if (S.value <= 0) {
                remove a process P from S.L;
                wakeup(P);          }
```

Semaphore as a General Synchronization Tool

- Execute *B* in P_j only after *A* executed in P_i
- Use semaphore *flag* initialized to 0

- Code:

P_i	P_j
⋮	⋮
A	*wait*(*flag*)
signal(*flag*)	*B*

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.
- Let *S* and *Q* be two semaphores initialized to 1

P_0	P_1
wait(*S*);	*wait*(*Q*);
wait(*Q*);	*wait*(*S*);
⋮	⋮
signal(*S*);	*signal*(*Q*);
signal(*Q*)	*signal*(*S*);

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

Two Types of Semaphores

- *Counting* semaphore – integer value can range over an unrestricted domain.
- *Binary* semaphore – integer value can range only between 0 and 1; can be simpler to implement.
- Can implement a counting semaphore *S* as a binary semaphore.

Implementing *S* as a Binary Semaphore

- Data structures:

```
binary-semaphore S1, S2;
int C:
```

- Initialization:

S1 = 1 & S2 = 0
C = initial value of semaphore **S**

Implementing *S*

- *wait* operation

```
wait(S1);
C--;
if (C < 0) {
        signal(S1);
        wait(S2);
}
signal(S1);
```

- *signal* operation

```
wait(S1);
C ++;
if (C <= 0)
        signal(S2);
else
        signal(S1);
```

Classical Problems of Synchronization

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

Bounded-Buffer Problem

- Shared data

  semaphore full, empty, mutex;

  Initially:

  full = 0, empty = n, mutex = 1

Bounded-Buffer Problem Producer Process

```
do {
        …
    produce an item in nextp
        …
    wait(empty);
    wait(mutex);
        …
    add nextp to buffer
        …
    signal(mutex);
    signal(full);
} while (1);
```

Bounded-Buffer Problem Consumer Process

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

```
        remove an item from buffer to nextc
                …
        signal(mutex);
        signal(empty);
                …
        consume the item in nextc
                …
} while (1);
```

Readers-Writers Problem

- Shared data
  semaphore mutex, wrt;
  Initially
  mutex = 1, wrt = 1, readcount = 0

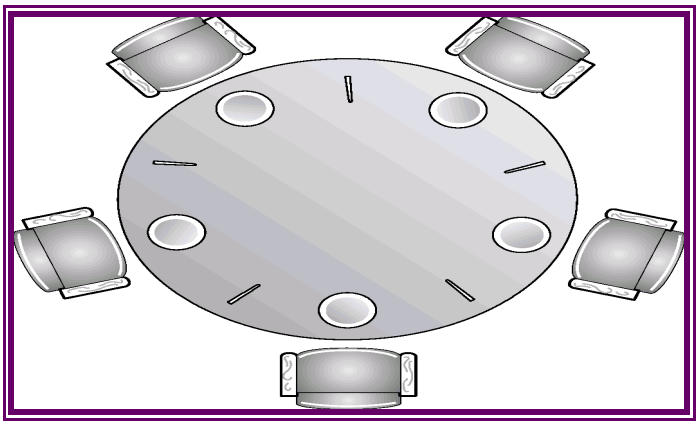
Readers-Writers Problem Writer Process

```
wait(wrt);
                …
        writing is performed
                …
signal(wrt);
```

Readers-Writers Problem Reader Process

```
wait(mutex);
readcount++;
if (readcount == 1)
                wait(rt);
signal(mutex);
                …
        reading is performed
                …
wait(mutex);
readcount--;
if (readcount == 0)
        signal(wrt);
signal(mutex):
```

Dining-Philosophers Problem



- Shared data

  semaphore chopstick[5];

Initially all values are 1

Dining-Philosophers Problem

- Philosopher *i*:

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

Critical Regions

- High-level synchronization construct
- A shared variable ***v*** of type ***T***, is declared as:

  v: shared T

- Variable ***v*** accessed only inside statement

  region v when B do S

  where ***B*** is a boolean expression.

- While statement ***S*** is being executed, no other process can access variable ***v***.
- Regions referring to the same shared variable exclude each other in time.
- When a process tries to execute the region statement, the Boolean expression *B* is evaluated. If *B* is true, statement *S* is executed. If it is false, the process is delayed until *B* becomes true and no other process is in the region associated with *v*.

Example – Bounded Buffer

- Shared data:

```
struct buffer {
        int pool[n];
        int count, in, out;
}
```

Bounded Buffer Producer Process

- Producer process inserts **nextp** into the shared buffer

```
        region buffer when( count < n) {
        pool[in] = nextp;
        in:= (in+1) % n;
        count++;
}
```

Bounded Buffer Consumer Process

- Consumer process removes an item from the shared buffer and puts it in **nextc**

```
        region buffer when (count > 0)
{       nextc = pool[out];
        out = (out+1) % n;
        count--;
}
```

Implementation region *x* when *B* do *S*

- Associate with the shared variable *x*, the following variables:

```
    semaphore mutex, first-delay, second-delay;
  int first-count, second-count;
```

- Mutually exclusive access to the critical section is provided by **mutex**.
- If a process cannot enter the critical section because the Boolean expression **B** is false, it initially waits on the **first-delay** semaphore; moved to the **second-delay** semaphore before it is allowed to reevaluate *B*.

Implementation

- Keep track of the number of processes waiting on **first-delay** and **second-delay**, with **first-count** and **second-count** respectively. The algorithm assumes a FIFO ordering in the queuing of processes for a semaphore.
  For an arbitrary queuing discipline, a more complicated implementation is required.

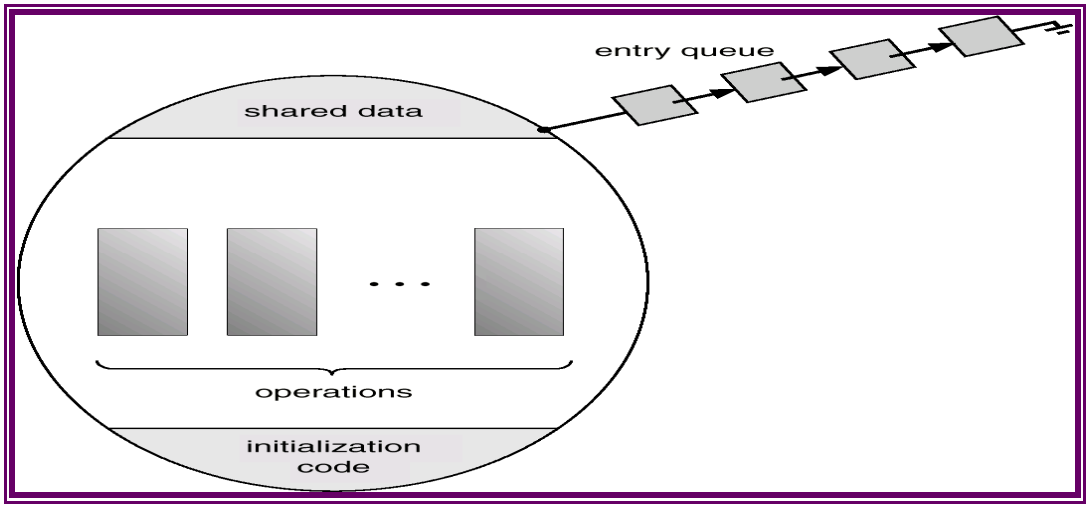
Monitors

- High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.

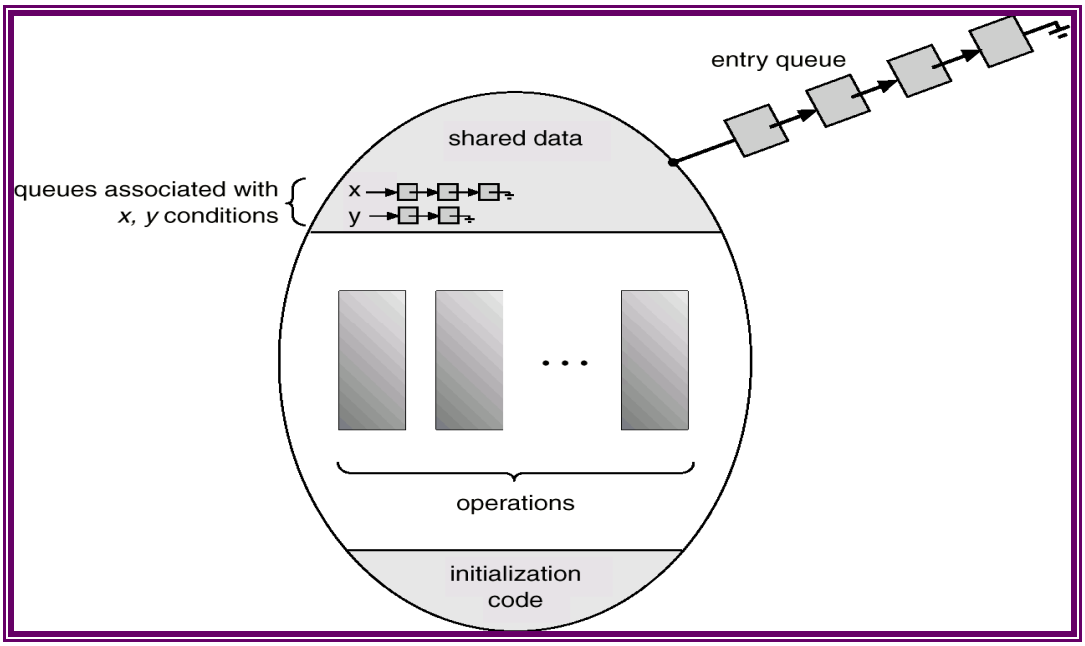
```
        monitor monitor-name
{
        shared variable declarations
        procedure body P1 (…) {
                . . .
        }
        procedure body P2 (…) {
                . . .
        }
        procedure body Pn (…) {
                . . .
        }
        {
                initialization code
        }
}
```

- To allow a process to wait within the monitor, a **condition** variable must be declared, as
  condition x, y;
- Condition variable can only be used with the operations **wait** and **signal**.
  - The operation
    x.wait();
    means that the process invoking this operation is suspended until another process invokes.
    x.signal();
  - The x.signal operation resumes exactly one suspended process. If no process is suspended, then the signal operation has no effect.

Schematic View of a Monitor



Monitor With Condition Variables



Dining Philosophers Example

```
monitor dp
{
        enum {thinking, hungry, eating} state[5];
        condition self[5];
        void pickup(int i)              // following slides
        void putdown(int i)   // following slides
        void test(int i)                // following slides
        void init() {
                for (int i = 0; i < 5; i++)
                        state[i] = thinking;
                        }
}
void pickup(int i) {
        state[i] = hungry;
        test[i];
        if (state[i] != eating)
                self[i].wait();
                        }
void putdown(int i) {
        state[i] = thinking;
        // test left and right neighbors
        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();
        }}
```

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;     // (initially  = 0)
int next-count = 0;
```

- Each external procedure ***F*** will be replaced by

```
wait(mutex);
   …
  body of F;
   …
if (next-count > 0)
        signal(next)
else
        signal(mutex);
```

- Mutual exclusion within a monitor is ensured.

Monitor Implementation

- 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--;
  ```

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

  ```
  if (x-count > 0) {
  next-count++;
  signal(x-sem);
  wait(next);
  next-count--;
  }
  ```

- *Conditional-wait* construct: **x.wait(c);**
  - **c** – integer expression evaluated when the **wait** operation is executed.
  - value of **c** (a *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.

Solaris 2 Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing.
- Uses *adaptive mutexes* for efficiency when protecting data from short code segments. Uses *condition variables* and *readers-writers* locks when longer sections of code need access to data.
- Uses *turnstiles* to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock.

Windows 2000 Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems.
- Uses *spinlocks* on multiprocessor systems.
- Also provides *dispatcher objects* which may act as wither mutexes and semaphores.
- Dispatcher objects may also provide *events*. An event acts much like a condition variable.

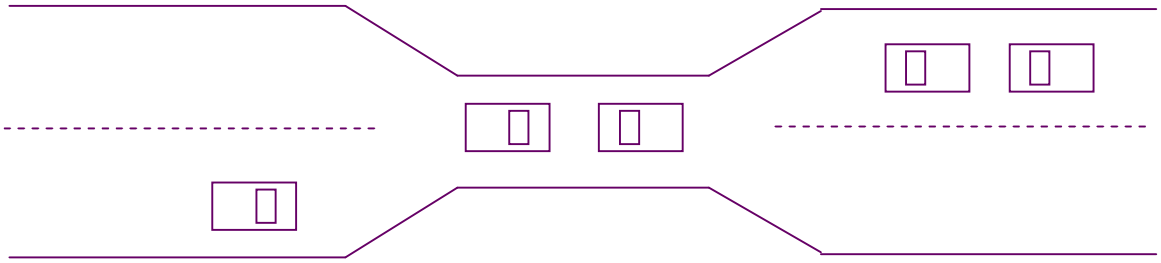
Deadlocks

The Deadlock Problem

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

P_0	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.

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

- **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 $\{P_0, P_1, \ldots, P_0\}$ 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_0 is waiting for a resource that is held by P_0.

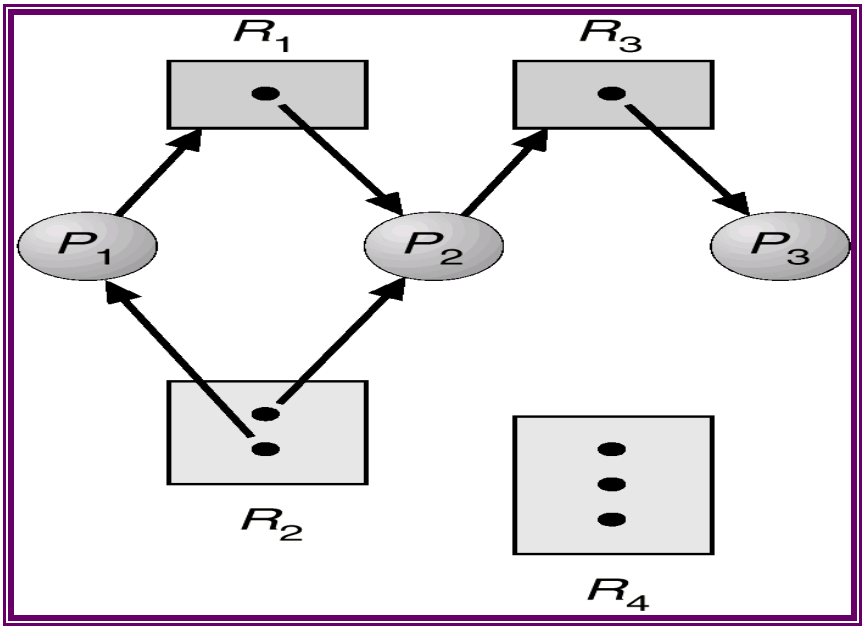
Resource-Allocation Graph

- 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 $P_1 \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$

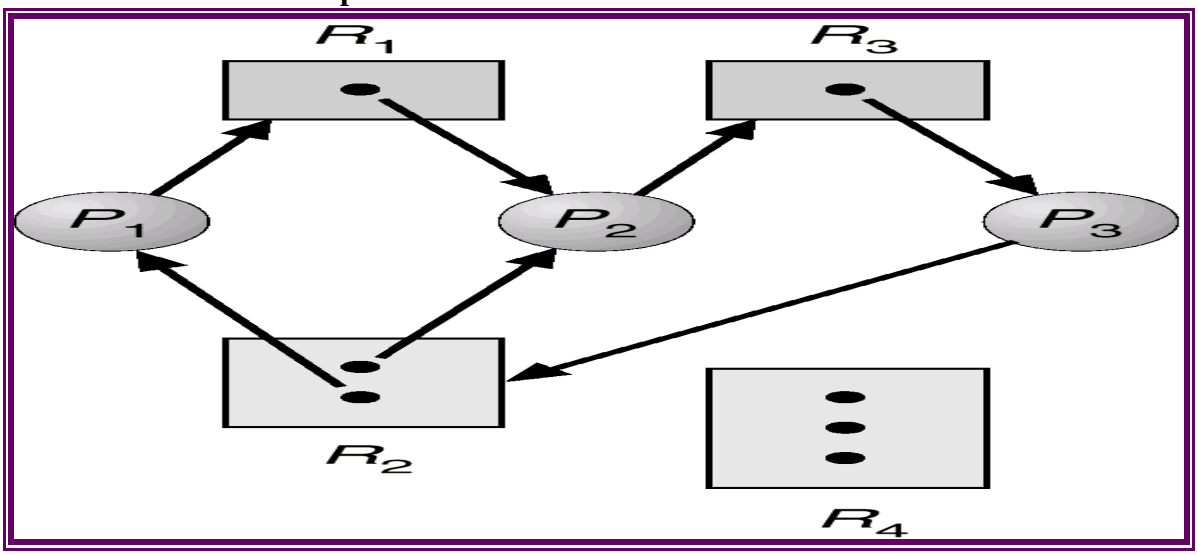
- Process

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

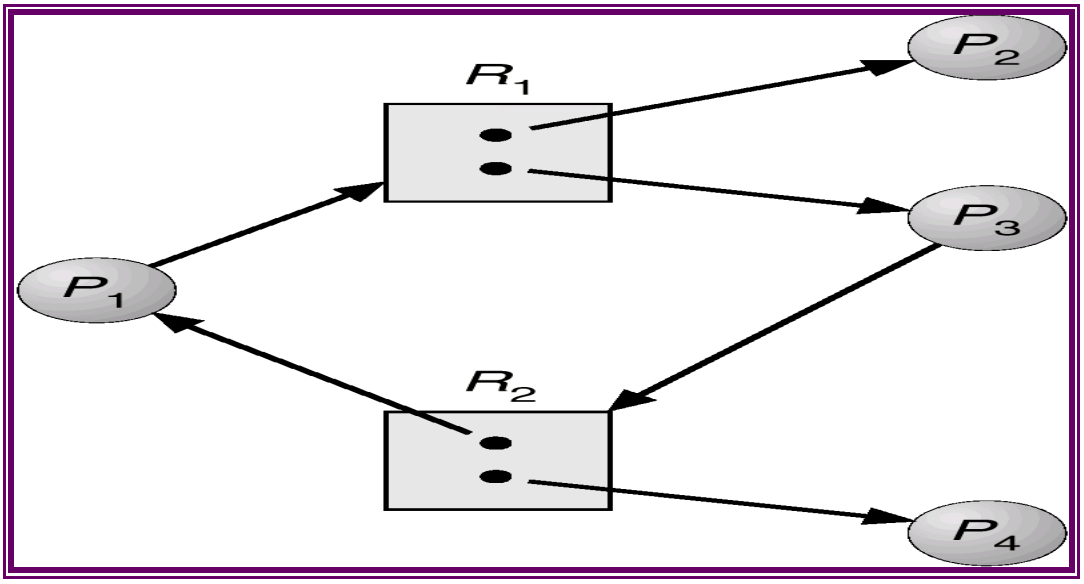
Example of a Resource Allocation Graph



Resource Allocation Graph With



Resource Allocation Graph With A Cycle But No Deadlock



Basic Facts

- 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

- 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

- **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.
- **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

- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes.

Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a *safe state*.
- System is in safe state if there exists a safe sequence of all processes.
- Sequence $<P_1, P_2, \ldots, P_n>$ is safe if for each P_i, the resources that *Pi* can still request can be satisfied by currently available resources + resources held by all the P_j, with $j<I$.
  - 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,

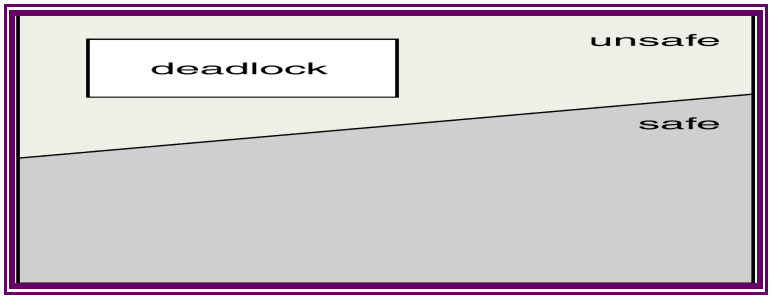
and terminate.

- When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

Basic Facts

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

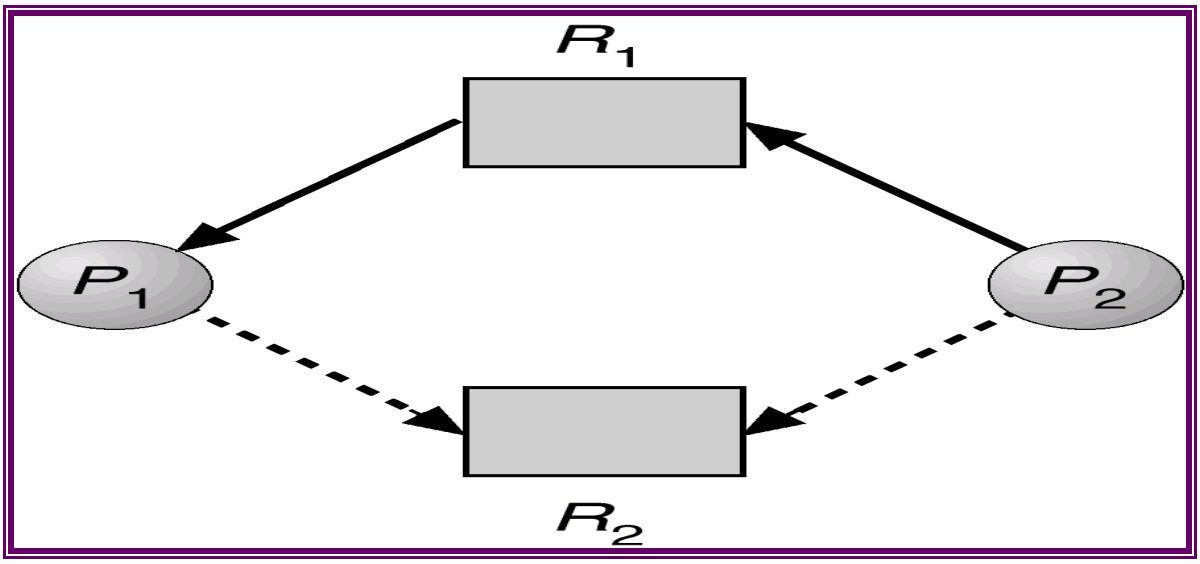
Safe, Unsafe , Deadlock State



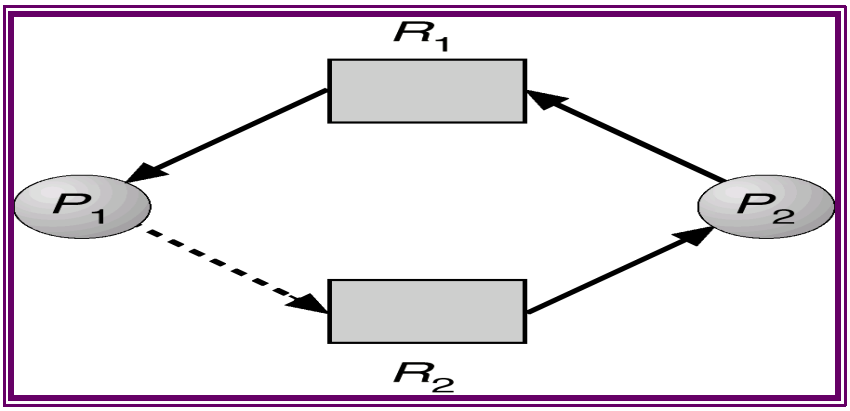
Resource-Allocation Graph Algorithm

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

Resource-Allocation Graph For Deadlock Avoidance



Unsafe State In Resource-Allocation Graph



Banker's Algorithm

- 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

- *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* - 1,3, …, *n.*
2. Find and *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:

   $Available = Available = Request_i;$
   $Allocation_i = Allocation_i + Request_i;$
   $Need_i = Need_i – Request_{i;;}$

   - *If safe* $\Rightarrow$ *the resources are allocated to* P_i.
   - *If unsafe* $\Rightarrow$ 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	

- The conte
- nt 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) (Cont.)

- 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 1	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?

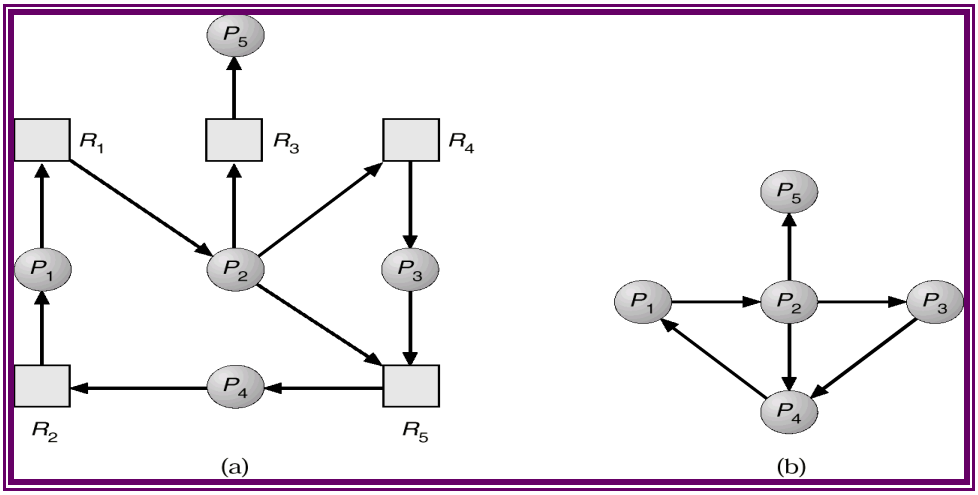
Deadlock Detection

- 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.
- 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



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 $Allocation_i \neq 0$, 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.
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.

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*.
- P_2 requests an additional instance of type *C*.

	Request
	A B C
P_0	0 0 0
P_1	2 0 1
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 process needs to complete.
  - How many processes will need to be terminated.
  - Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

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

Combined Approach to Deadlock Handling

- Combine the three basic approaches
  - Prevention
  - Avoidance
  - Detection

  allowing the use of the optimal approach for each of resources in the system.

- Partition resources into hierarchically ordered classes.
- Use most appropriate technique for handling deadlocks within each class.