GURU NANAK COLLEGE

Department of BCA

Operating System

Process Synchronization

UNIT II: Process Synchronization: Critical-Section problem - Synchronization Hardware – Semaphores – Classic Problems of Synchronization – Critical Region – Monitors.

Deadlock : Characterization – Methods for handling Deadlocks – Prevention, Avoidance, and Detection of Deadlock - Recovery from deadlock.

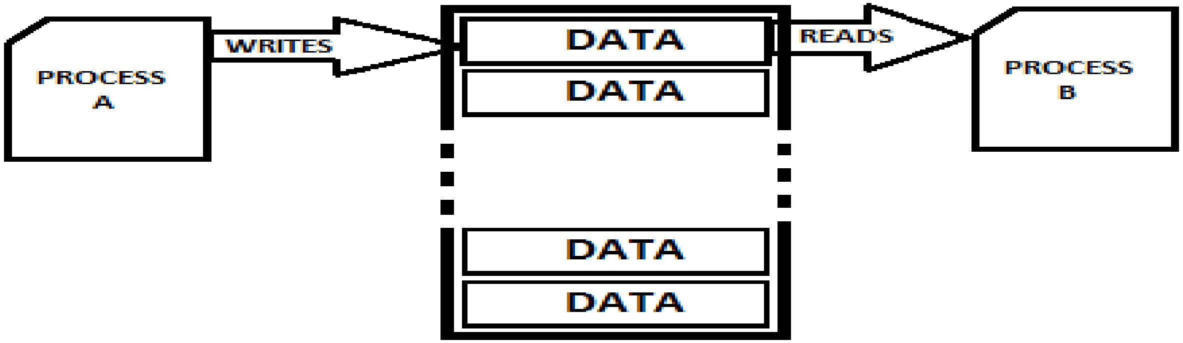
G.Kiruthiga, Associate Professor, Dept of BCA, GNC

Contents

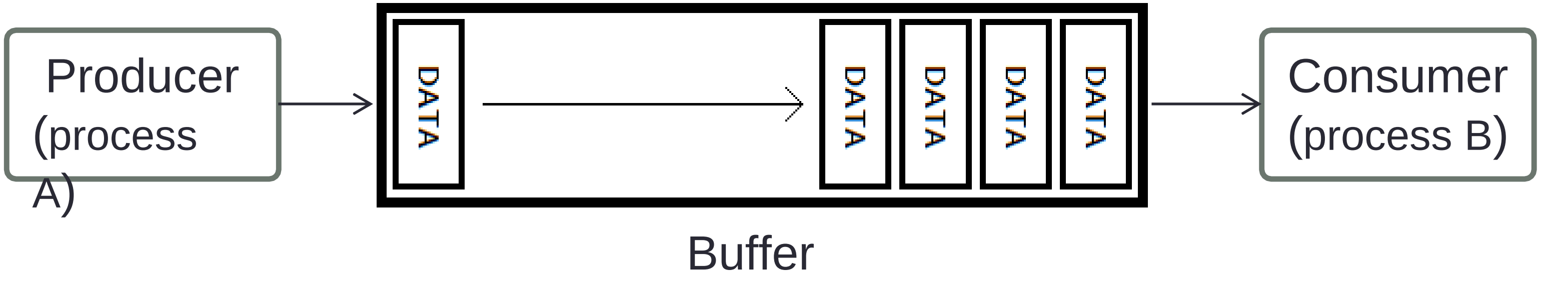
- What is Process Synchronization and why it is needed
- The Critical Section Problem
- Peterson’s Solution
- Synchronization Hardware
- Semaphores
- Applications of Semaphores
- Classical Problems of Synchronizations
- Synchronization Examples
- Monitors
- Atomic Transactions
- References

What is Process Synchronization

- Several Processes run in an Operating System
- Some of them share resources due to which problems like data inconsistency may arise
- For Example: One process changing the data in a memory location where another process is trying to read the data from the same memory location. It is possible that the data read by the second process will be erroneous



Producer Consumer Problem
(or Bounded-Buffer Problem)



- Problem: To ensure that the ***Producer*** should not add DATA when the Buffer is *full* and the ***Consumer*** should not take data when the Buffer is *empty*

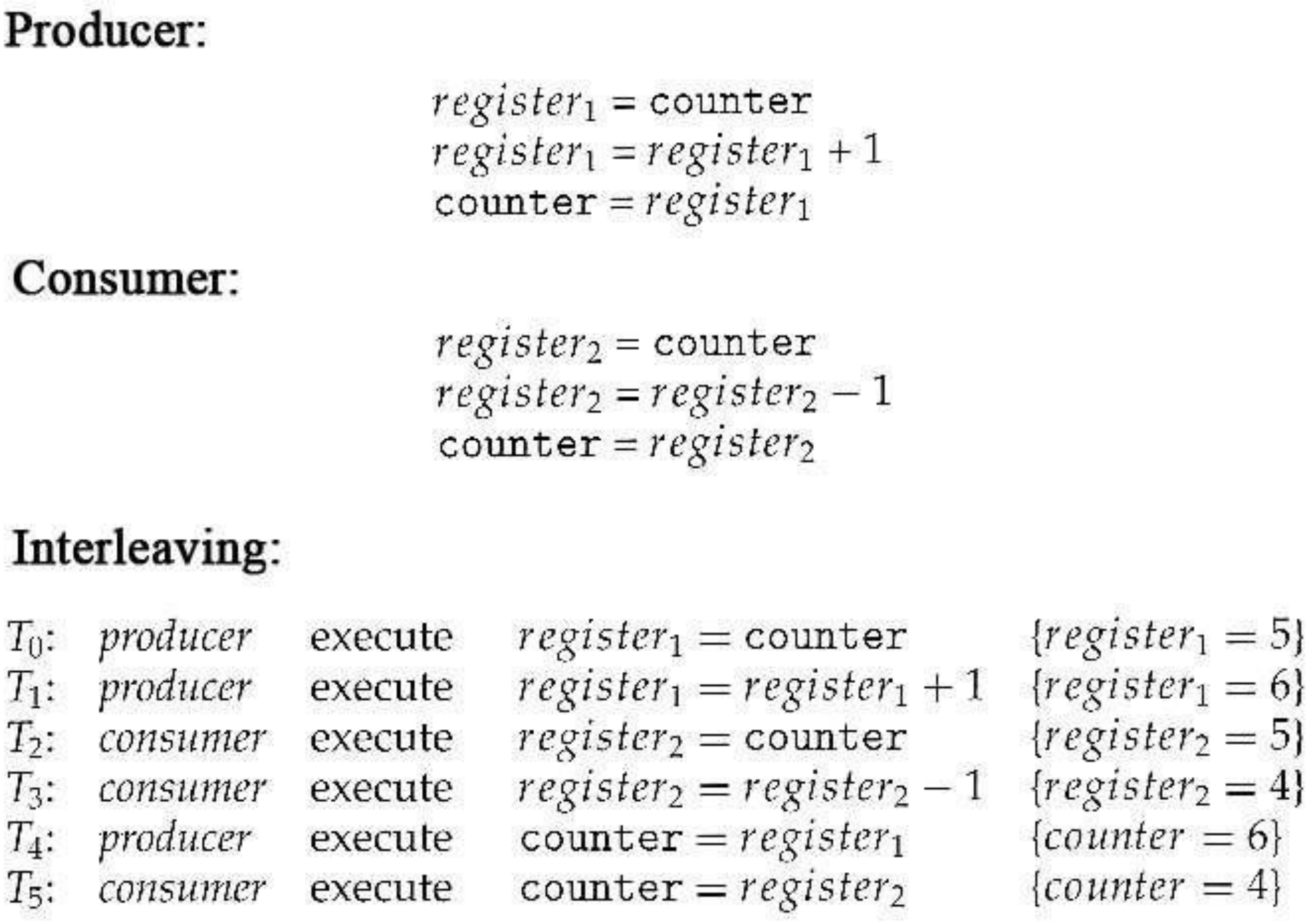
Solution for Producer Consumer Problem

- Using Semaphores (In computer science, particularly in operating systems, a **semaphore** is a variable or abstract data type that is used for controlling access, by multiple processes, to a common resource in a concurrent system such as a multiprogramming operating system.)

- Using Monitors (In concurrent programming, a **monitor** is a synchronization construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become true)

- Atomic Transactions (Atomic read-modify-write access to shared variables is avoided, as each of the two Count variables is updated only by a single thread. Also, these variables stay incremented all the time; the relation remains correct when their values wrap around on an integer overflow)

Race Condition



Producer:

$$register_1 = \texttt{counter}$$
$$register_1 = register_1 + 1$$
$$\texttt{counter} = register_1$$

Consumer:

$$register_2 = \texttt{counter}$$
$$register_2 = register_2 - 1$$
$$\texttt{counter} = register_2$$

Interleaving:

T_0:	*producer*	execute	$register_1 = \texttt{counter}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \texttt{counter}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\texttt{counter} = register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	$\texttt{counter} = register_2$	$\{counter = 4\}$

?

What if T5 happened before T4?

Critical Section Problem

- **A section of code, common to n cooperating processes, in which the processes may be accessing common variables.**

A Critical Section Environment contains:

- **Entry Section** Code requesting entry into the critical section.
- **Critical Section** Code in which only one process can execute at any one time.
- **Exit Section** The end of the critical section, releasing or allowing others in.
- **Remainder Section** Rest of the code AFTER the critical section.

Critical Section Problem

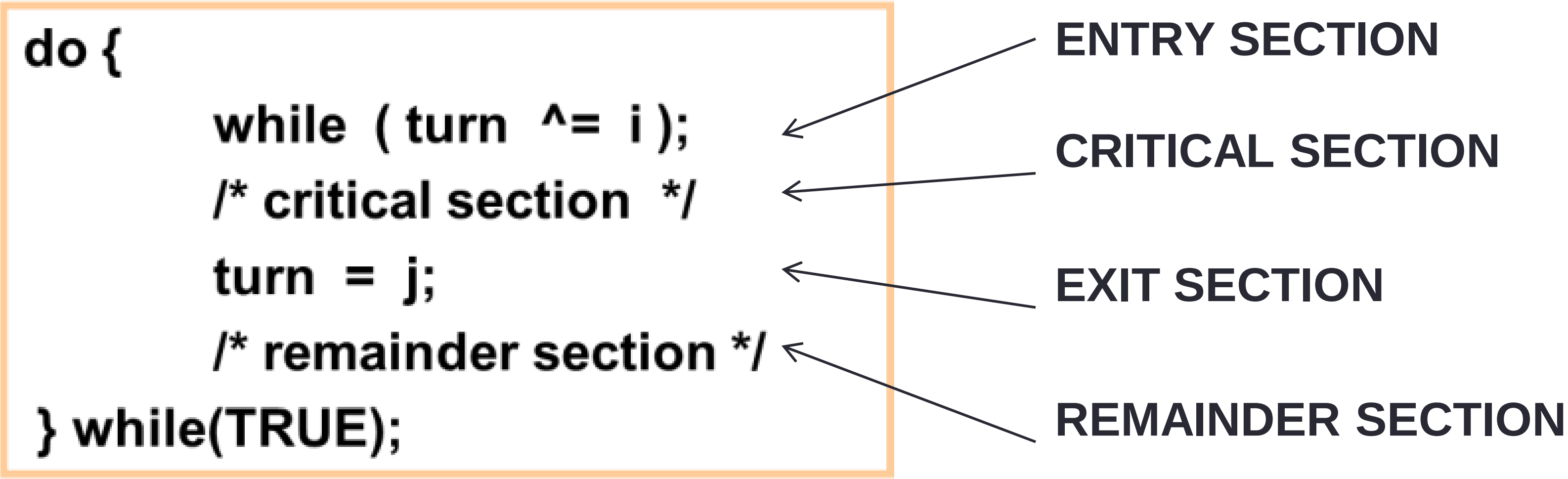
The critical section must ENFORCE ALL THREE of the following rules:

- **Mutual Exclusion:** No more than one process can execute in its critical section at one time.

- **Progress:** If no one is in the critical section and someone wants in, then those processes not in their remainder section must be able to decide in a finite time who should go in.

- **Bounded Wait:** All requesters must eventually be let into the critical section

Critical Section Problem



```
do {
          while  ( turn  ^=  i );
          /* critical section  */
          turn  =  j;
          /* remainder section */
 } while(TRUE);
```

ENTRY SECTION

CRITICAL SECTION

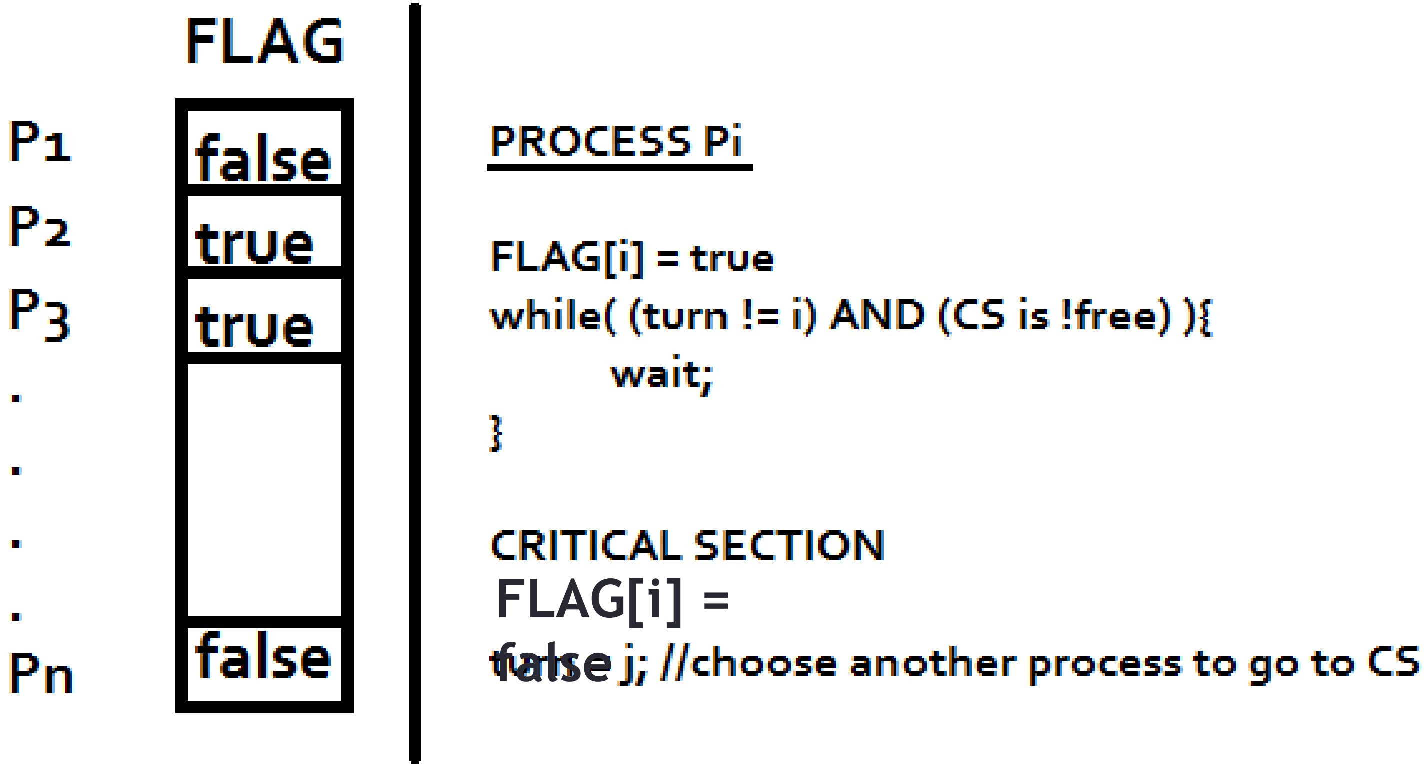
EXIT SECTION

REMAINDER SECTION

Peterson’s Solution

- To handle the problem of Critical Section (CS), Peterson gave an algorithm with a bounded waiting.
- Suppose there are N processes (P1, P2, … PN) and each of them at some point need to enter the Critical Section.
- A FLAG[] array of size N is maintained which is by default false and whenever a process need to enter the critical section it has to set its flag as true, i.e. suppose Pi wants to enter so it will set FLAG[i]=TRUE
- There is another variable called TURN which indicates the process number which is currently to enter into the CS. The process that enters into the CS while exiting would change the TURN to another number from among the list of ready processes

PETERSON’S SOLUTION



	FLAG
P1	false
P2	true
P3	true
.	
.	
.	
.	
Pn	false

PROCESS Pi

```
FLAG[i] = true
while( (turn != i) AND (CS is !free) ){
        wait;
}

CRITICAL SECTION
FLAG[i] = false
turn = j; //choose another process to go to CS
```

- If turn is 2 (say) then P2 enters the CS and while exiting sets the turn as 3 and thus P3 breaks out of wait loop

Synchronization Hardware

- Problems of Critical Section are also solvable by hardware.

- Uniprocessor systems disables interrupts while a Process Pi is using the CS but it is a great disadvantage in multiprocessor systems

- Some systems provide a lock functionality where a Process acquires a lock while entering the CS and releases the lock after leaving it. Thus another process trying to enter CS cannot enter as the entry is locked. It can only do so if it is free by acquiring the lock itself

- Another advanced approach is the ***Atomic Instructions*** (Non-Interruptible instructions).

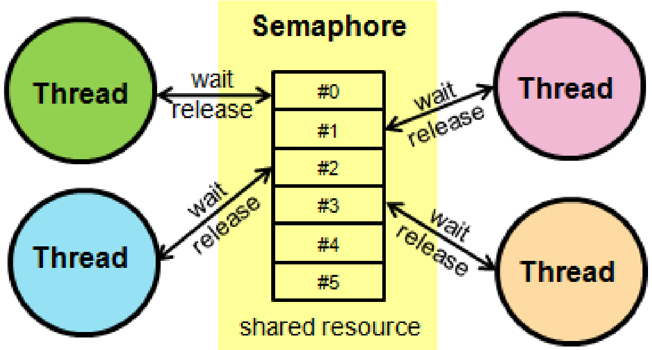
MUTEX LOCKS

- As the synchronization hardware solution is not easy to implement from everyone, a strict software approach called Mutex Locks was introduced. In this approach, in the entry section of code, a LOCK is acquired over the critical resources modified and used inside critical section, and in the exit section that LOCK is released.

- As the resource is locked while a process executes its critical section hence no other process can access it

Semaphores

- It is a integer variable for which only two (atomic) operations are defined, the wait and signal operations.

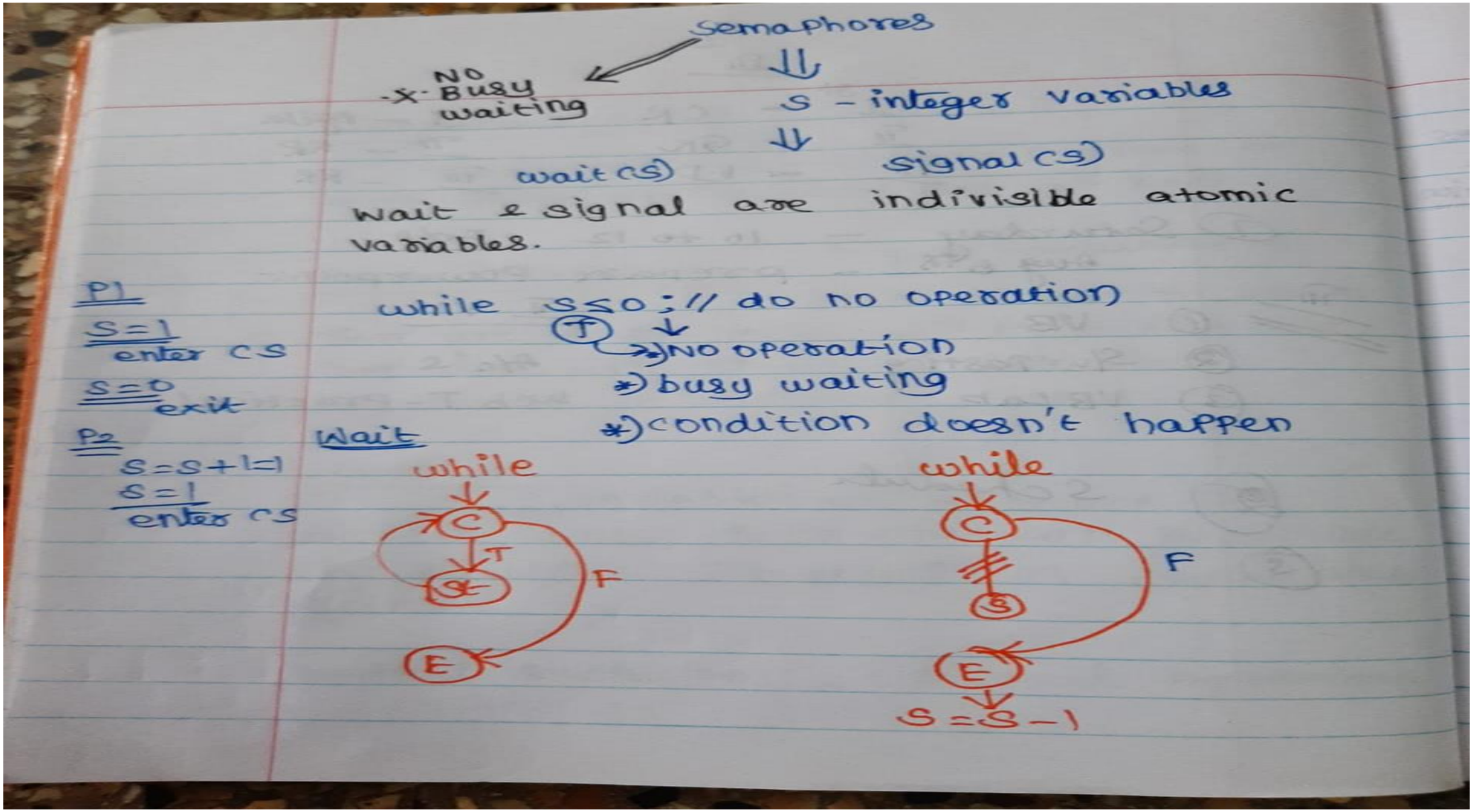


- Variable or abstract data type used to control access to a common resource by multiple processes in concurrent systems.
- Synchronization tool, does not require busy waiting.

```
WAIT ( S ):
     while              ( S <= 0 ); //do no
     operation
     S = S - 1;
```

```
SIGNAL ( S ):
     S = S + 1;
```

Semaphores (Cont..)



Semaphores

-*- No Busy waiting

⇓

S – integer variables

⇓

wait (S)   signal (S)

wait & signal are indivisible atomic variables.

P1

S=1

enter CS

S=0

exit

P2

S=S+1=1

S=1

enter CS

while S≤0; // do no operation

(T) ↓

→ No operation

*) busy waiting

*) condition doesn't happen

Wait

while
C
T
S
F
E
while
C
S
F
E
S=S−1

Semaphores as general Synchronization Tool

- **TYPES**

Semaphores are mainly of two types:

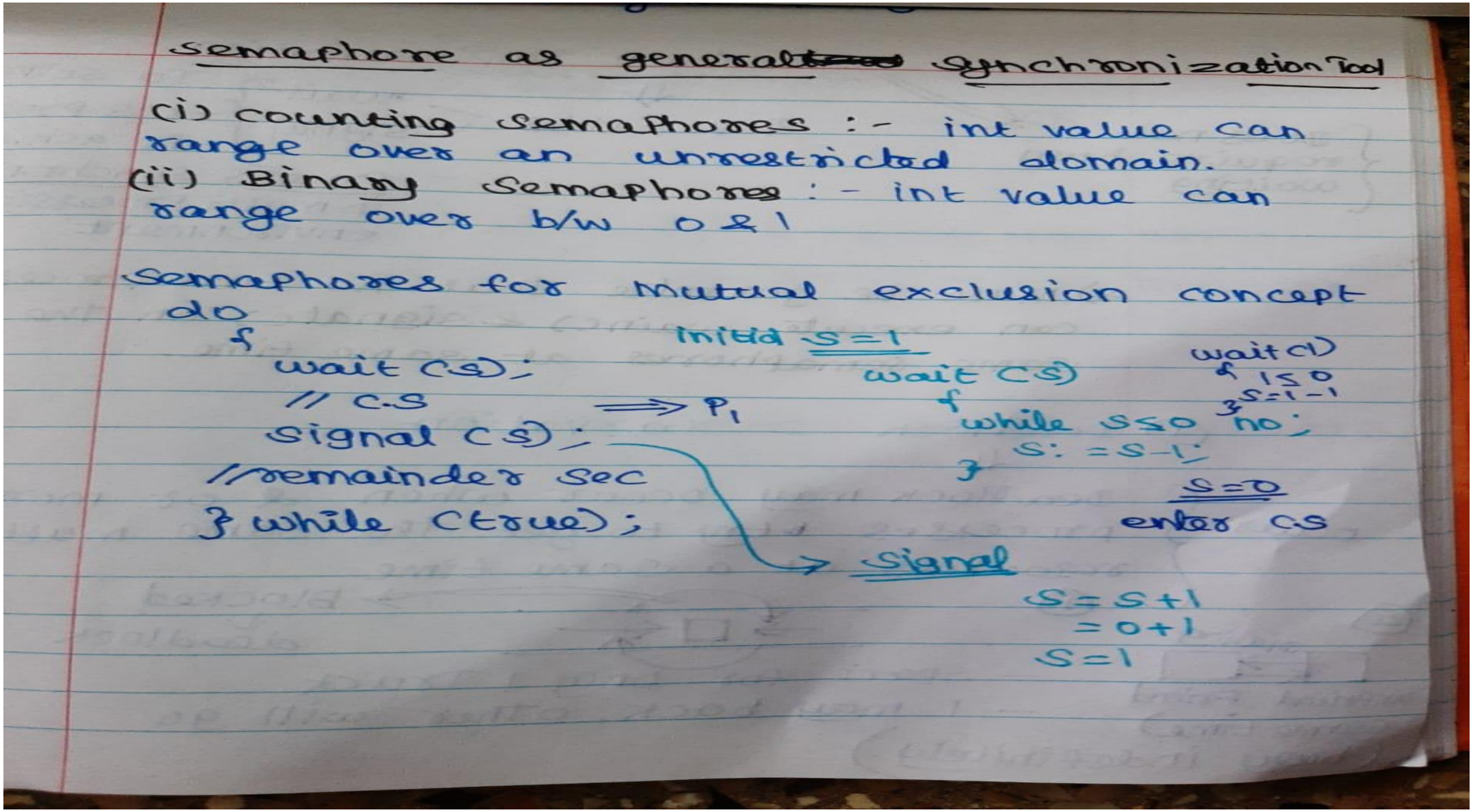
1. Binary Semaphore

   It is a special form of semaphore used for implementing mutual exclusion, hence it is often called Mutex. A binary semaphore is initialized to 1 and only takes the value 0 and 1 during execution of a program.

2. Counting Semaphores

   These are used to implement bounded concurrency.

Synchronization Tool (Cont..)



Semaphore as general synchronization Tool

(i) counting semaphores :- int value can range over an unrestricted domain.

(ii) Binary semaphores :- int value can range over b/w 0 & 1

Semaphores for mutual exclusion concept

```
do
  {
    wait (S);
    // C.S                ⟹ P1
    signal (S);
  //remainder sec
  } while (true);
```

Initial $\underline{S=1}$

```
wait (S)
{
  while S≤0 no;
  S: = S-1;
}
```

wait (1)
{ 1≤0
} S=1-1

$\underline{S=0}$

enter CS

Signal

$S=S+1$
$=0+1$
$S=1$

Semaphores

- **FORMAT:**

```
wait ( mutex );          // Mutual exclusion: mutex init to 1.
CRITICAL SECTION
signal( mutex );
REMAINDER
```

- Note that not only must the variable-changing steps (S-- and S++) be indivisible, it is also necessary that for the wait operation when the test proves false that there be no interruptions before S gets decremented. It S okay, however, for the busy loop to be interrupted when the test is true, which prevents the system from hanging forever

Semaphores

- **Properties (Characteristics)**

1. Simple
2. Works with many processes
3. Can have many different critical sections with different semaphores
4. Each critical section has unique access semaphores
5. Can permit multiple processes into the critical section at once, if desirable.

Semaphores

- Semaphores can be used to force synchronization (precedence) if the **preceding** process does a signal at the end, and the **follower** does wait at beginning. For example, here we want P1 to execute before P2.

```
P1:                              P2:
    statement 1;                     wait (synch );
    signal ( synch );                statement 2;
```

- To prevent looping, we redefine the semaphore structure as:

```
typedef struct {
            int value;
            struct process *list;      //linked list of PTBL waiting on
} SEMAPHORE;                                      S
```

Semaphores

```
SEMAPHORE s;
wait(s) {
    s.value = s.value - 1;
    if ( s.value < 0 ) {
            add this process to s.L;
            block;
    }
}
```

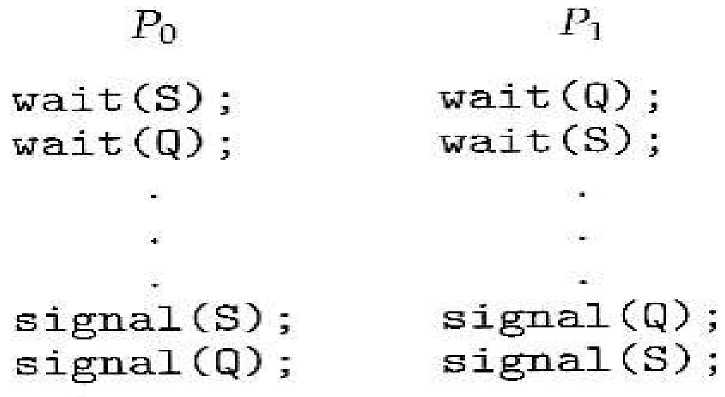
```
SEMAPHORE s;
signal(s) {
    s.value = s.value + 1;
    if ( s.value <= 0 ) {
            remove a process P from s.L;
            wakeup(P);
    }
}
```

- It's critical that these be atomic - in uniprocessors we can disable interrupts, but in multiprocessors other mechanisms for atomicity are needed.

Semaphores

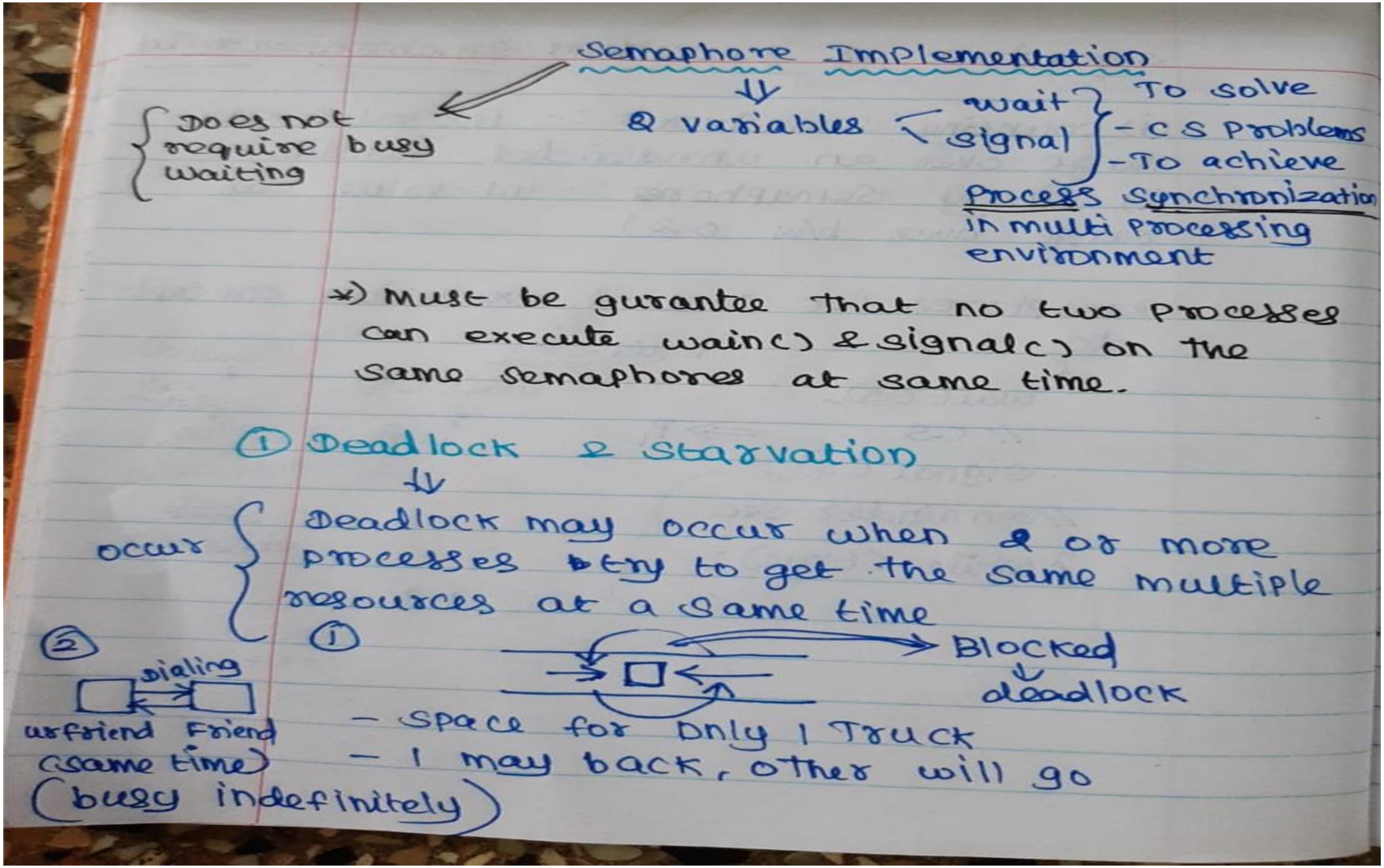
- **DEADLOCK**

- One important problem that can arise when using semaphores to block processes waiting for a limited resource is the problem of **deadlocks**, which occur when multiple processes are blocked, each waiting for a resource that can only be freed by one of the other (blocked) processes, as illustrated in the following example.



```
        P0                  P1

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

Deadlock



Semaphore Implementation

⇓

2 variables — wait, signal } To solve – C S Problems – To achieve Process Synchronization in multi processing environment

{ Does not require busy waiting

*) Must be gurantee that no two processes can execute wain() & signal() on the same semaphores at same time.

① Deadlock & Starvation

⇓

occur { Deadlock may occur when 2 or more processes try to get the same multiple resources at a same time

①

Blocked

⇓

deadlock

- Space for Only 1 Truck
- 1 may back, other will go

②

dialing

urfriend Friend (same time)

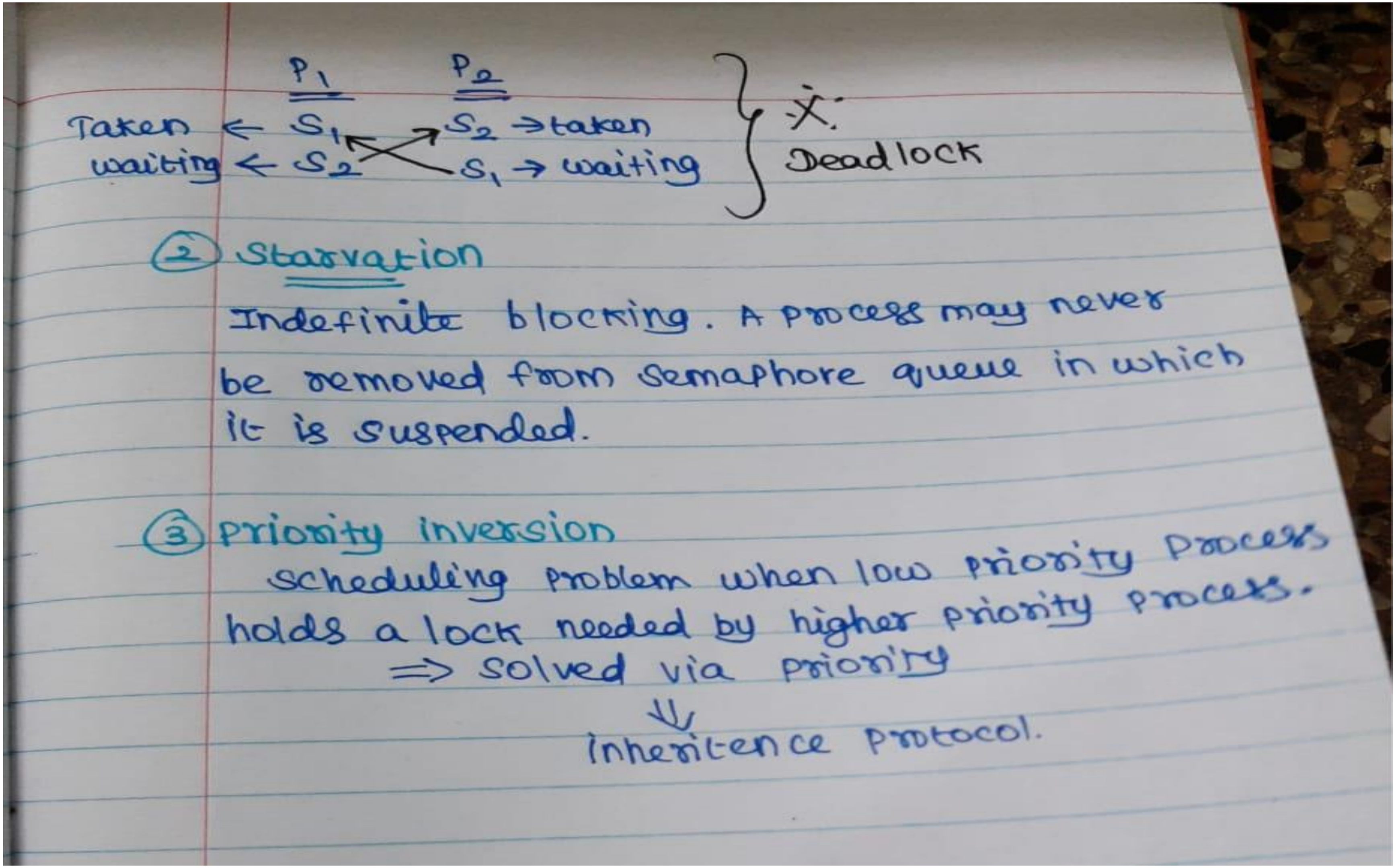
(busy indefinitely)

Semaphores

STARVATION

- Another problem to consider is that of **starvation**, in which one or more processes gets blocked forever, and never get a chance to take their turn in the critical section. For example, in the semaphores above, we did not specify the algorithms for adding processes to the waiting queue in the semaphore in the wait( ) call, or selecting one to be removed from the queue in the signal( ) call. If the method chosen is a FIFO queue, then every process will eventually get their turn, but if a LIFO queue is implemented instead, then the first process to start waiting could starve.

Starvation



P_1 P_2

Taken ← S_1 S_2 → taken

waiting ← S_2 S_1 → waiting

✗ Deadlock

(2) Starvation

Indefinite blocking. A process may never be removed from semaphore queue in which it is suspended.

(3) Priority inversion

scheduling problem when low priority process holds a lock needed by higher priority process.

⇒ solved via priority

⇓

inheritence protocol.

G.Kiruthiga, Associate Professor, Dept of BCA, GNC

Classical Problems of Synchronization

- Bounded buffer problem
- Readers and writers problem
- Dinning philosopher problem

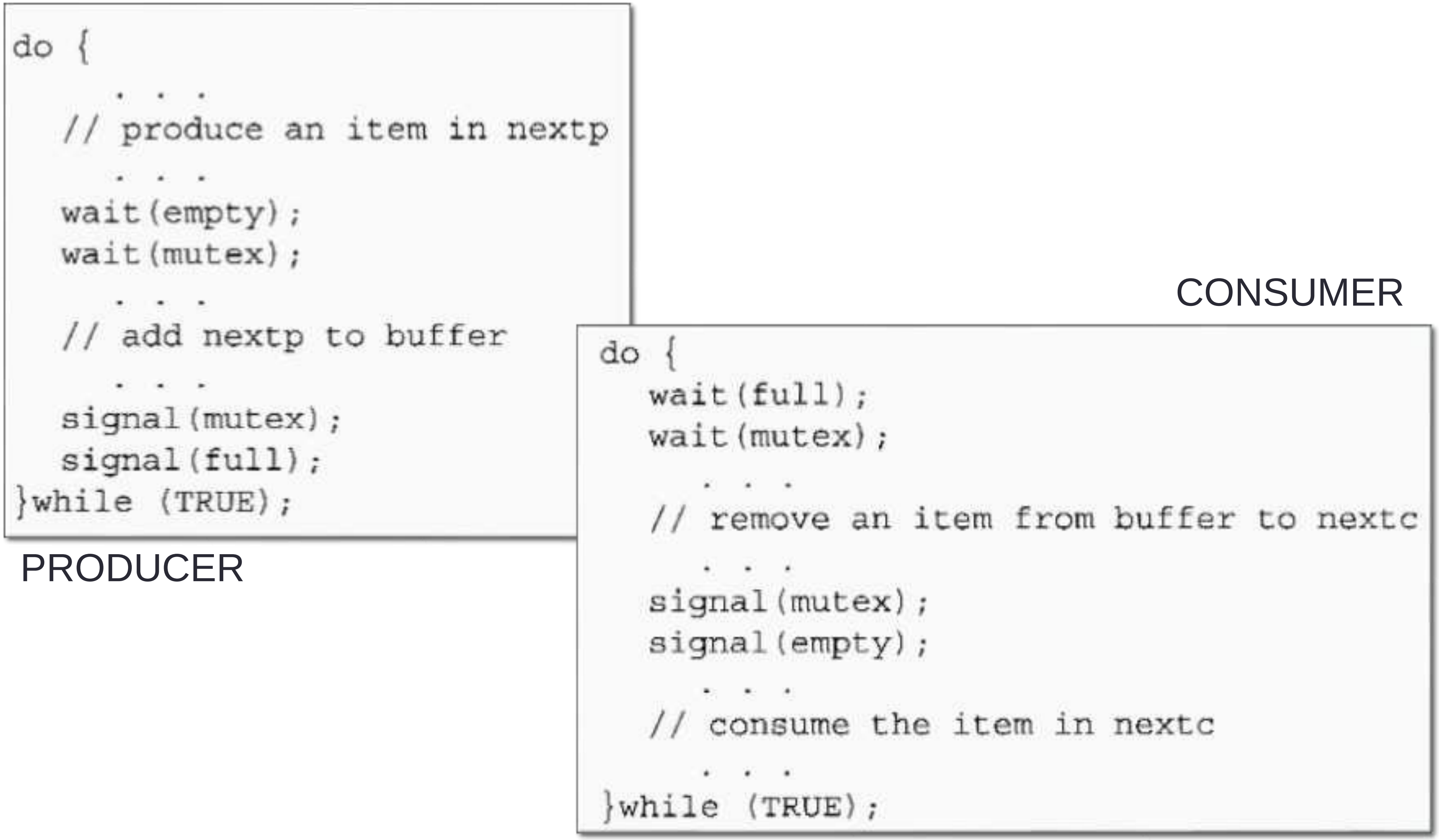
- This is the main 3 problem occur the problem is in synchronization.
- These problems solved by using Semaphore variable.

The Bounded-buffer Problem

- This is a generalization of the producer-consumer problem where in access is controlled to a shared group of buffers of a limited size. (Producer consumer problem to solve bounded buffer problem)
- In this solution, the two counting semaphores "full" and "empty" keep track of the current number of full and empty buffers respectively (and initialized to 0 and N respectively.) The binary semaphore mutex controls access to the critical section. The producer and consumer processes are nearly identical - One can think of the producer as producing full buffers, and the consumer producing empty buffers.
- N buffers, each can hold one item.
- Semaphore, mutex initialized=1.
- Semaphore, full=0, empty=N

Full=0,
mutex=1,
empty=1

The Bounded-buffer Problem



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

PRODUCER

CONSUMER

```
do {
   wait(full);
   wait(mutex);
     . . .
   // remove an item from buffer to nextc
     . . .
   signal(mutex);
   signal(empty);
     . . .
   // consume the item in nextc
     . . .
}while (TRUE);
```

The Readers-writers Problem

- **THE READERS/WRITERS PROBLEM:**

- This is the same as the Producer / Consumer problem except - we now can have many concurrent readers and one exclusive writer.

- Locks: are shared (for the readers) and exclusive (for the writer).

Two possible (contradictory) guidelines can be used:

1. No reader is kept waiting unless a writer holds the lock (the readers have precedence).

2. If a writer is waiting for access, no new reader gains access (writer has precedence).

- (NOTE: starvation can occur on either of these rules if they are followed rigorously.)

The Readers-writers Problem

```
Reader:
do {
        wait( mutex );                              /* Allow 1 reader in
        readcount = readcount + 1;                  entry*/
        if readcount == 1 then wait(wrt );          /* 1st reader locks writer */
        signal( mutex );
                /*    reading is performed */
         wait( mutex );
         readcount = readcount - 1;
         if readcount == 0 then signal(wrt );        /*last reader frees writer */
         signal( mutex );
} while(TRUE);

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

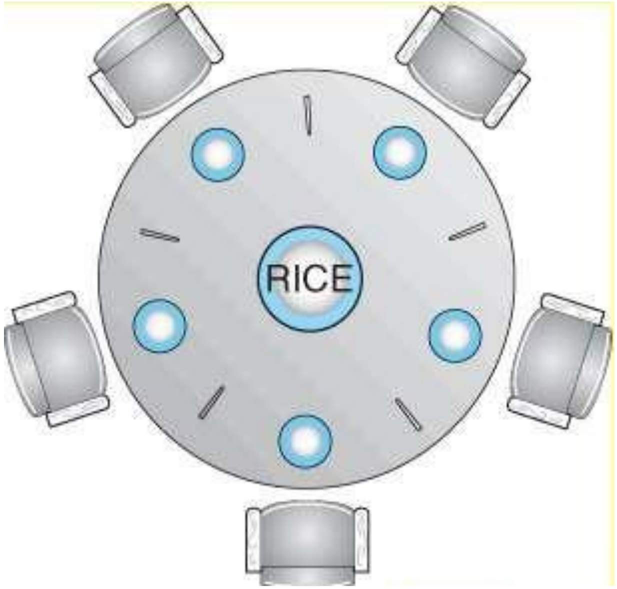
THE READERS/WRITERS PROBLEM:

```
BINARY_SEMAPHORE    wrt          = 1;
BINARY_SEMAPHORE    mutex        = 1;
BINARY_SEMAPHOR     readcount = 0;
E
int
```

Dining Philosophers Problem

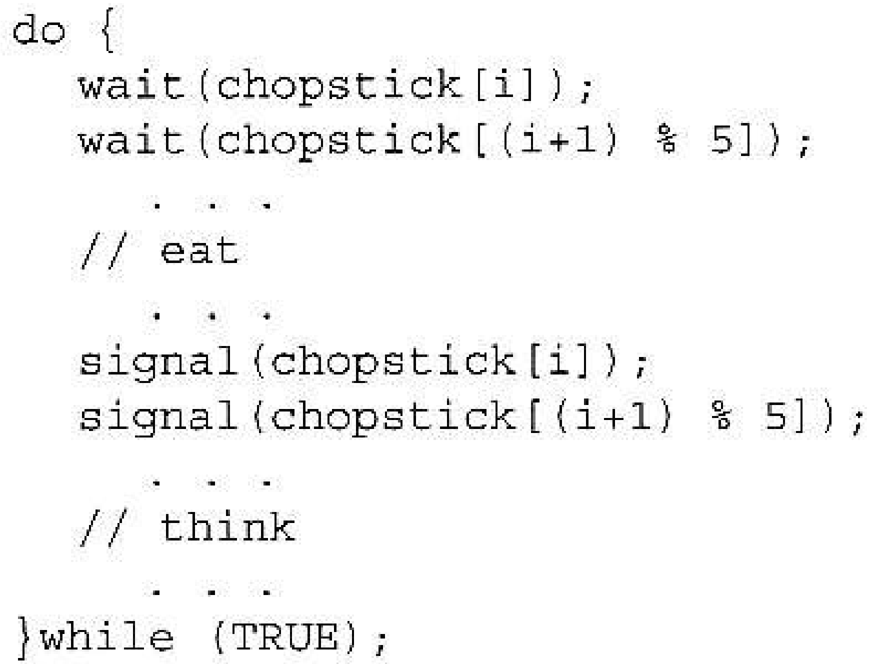
- Five philosophers sitting at a round dining table to eat
- Each one has got a plate and one chopstick at the right side of each plate
- To start eating each of them need a plate and two chopsticks
- Thus, clearly there will be a deadlock and starvation because of the limited number of chopsticks

- This can be solved by either:
  - Allow only 4 philosophers to be hungry at a time
  - Allow pickup only if both chopsticks are available. (Done in critical section)
  - Odd # philosopher always picks up left chopstick 1st, even # philosopher always picks up right chopstick 1st



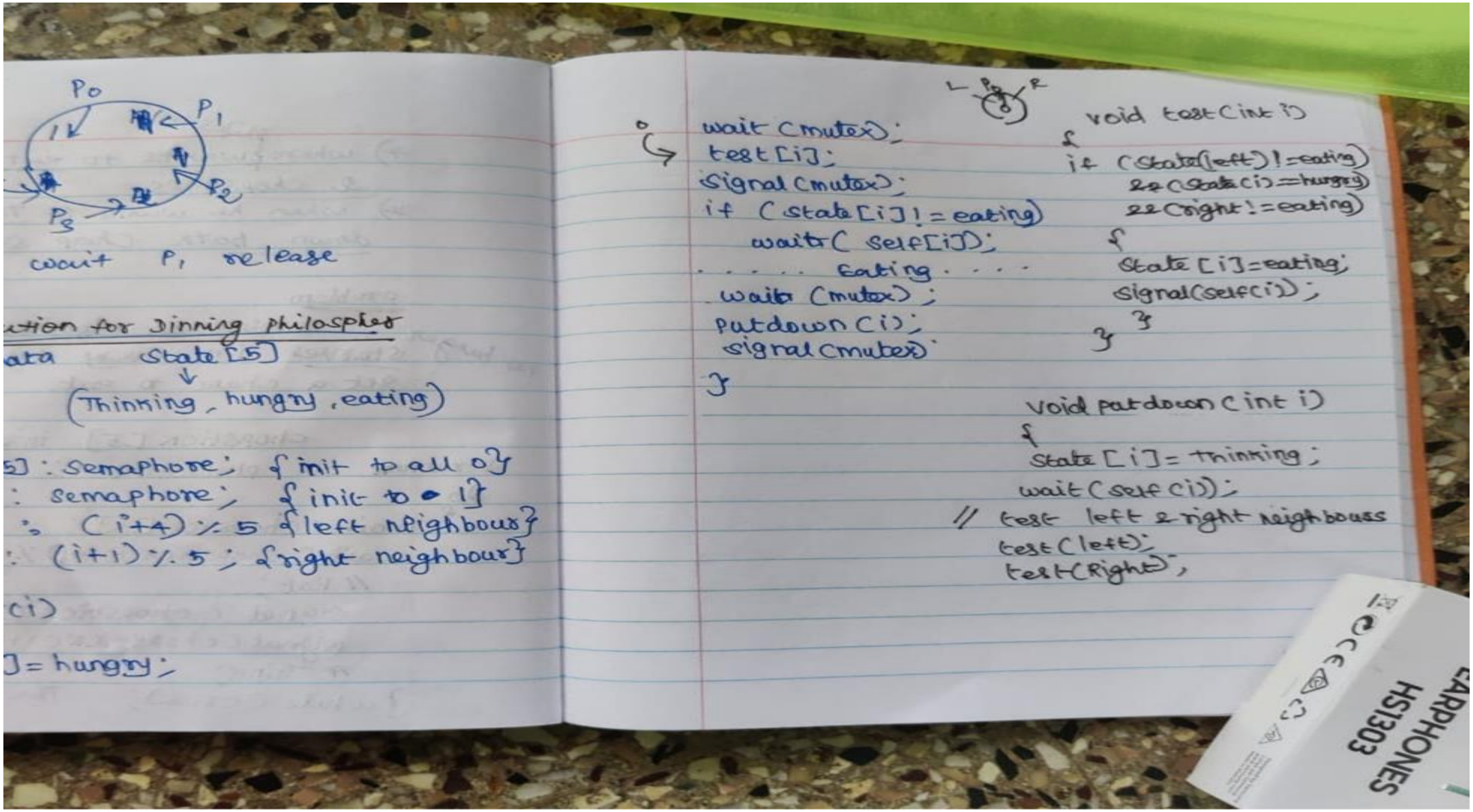
Dining Philosophers Problem

- One possible solution, as shown in the following code section, is to use a set of five semaphores (chopsticks[5]), and to have each hungry philosopher first wait on their left chopstick (chopsticks[i]), and then wait on their right chopstick (chopsticks[(i + 1) % 5])



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

Solution for Dinning philosopher Problem



Po
P1
P2
P3
1
L
R
EARPHONES
HS1303

```
wait   P1   release

ution for Dinning philospher
ata     State [5]
            ↓
   (Thinking , hungry , eating)

5] : semaphore;  {init to all 0}
 : semaphore;  {init to 1}
 :  (i+4) % 5  {left neighbour}
 :  (i+1) % 5 ; {right neighbour}

(i)

]= hungry;
```

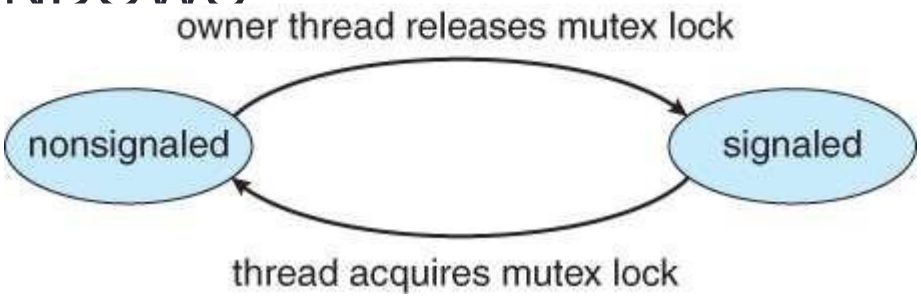
```
wait (mutex);
test[i];
signal (mutex);
if (state[i] != eating)
    wait( self[i]);
. . . . . . Eating . . . .
wait (mutex);
putdown (i);
signal (mutex)
}
```

```
void test(int i)
{
if (State(left) != eating)
    && (State(i) == hungry)
    && (right != eating)
  {
  State [i]=eating;
  signal(self(i));
  }
}
```

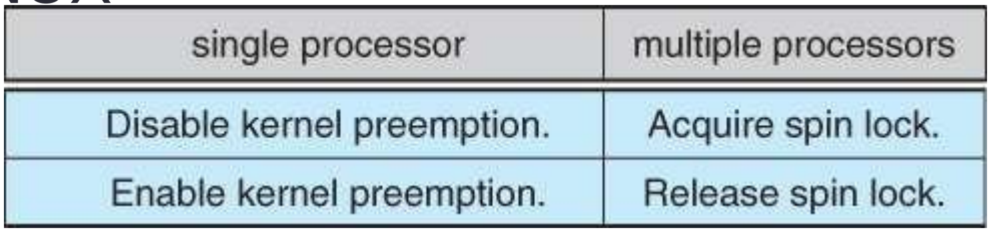
```
void putdown (int i)
{
 State [i]= thinking ;
 wait (self (i));
// test left & right neighbours
 test (left);
 test (Right);
```

Synchronization Examples

- SYNCHRONIZATION IN WINDOWS



- SYNCHRONIZATION IN LINUX



single processor	multiple processors
Disable kernel preemption.	Acquire spin lock.
Enable kernel preemption.	Release spin lock.

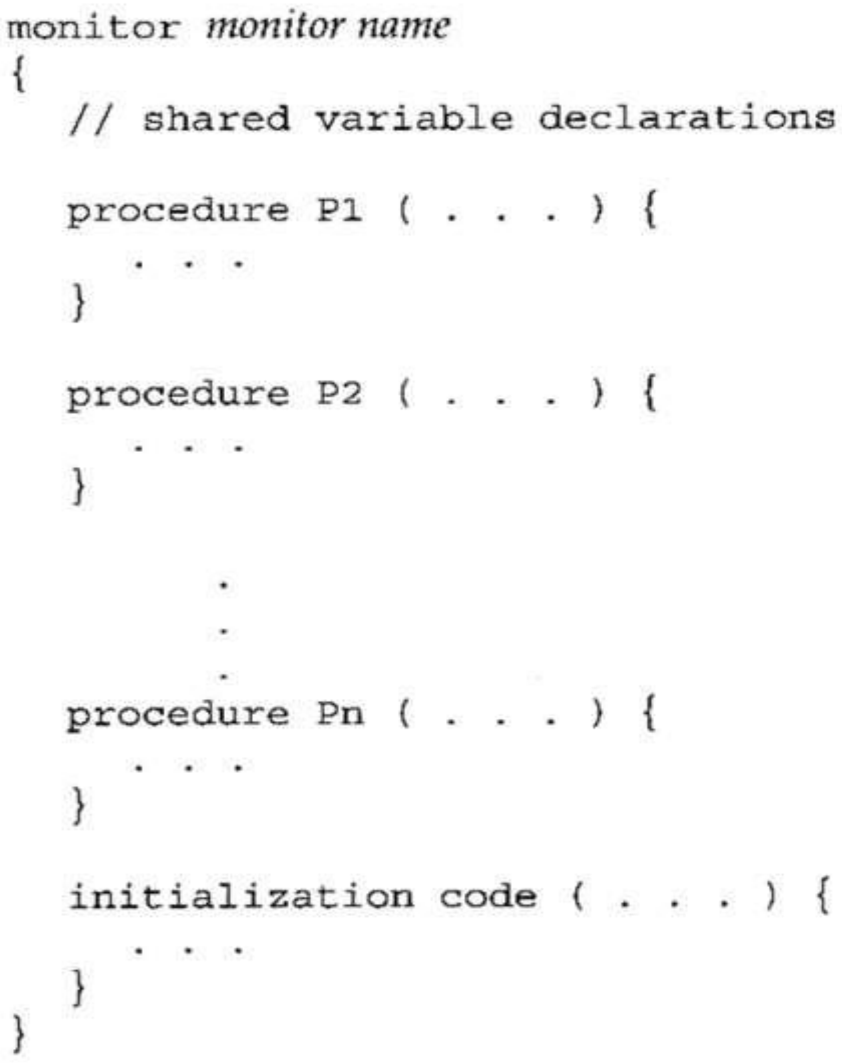
Monitors

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

- Semaphores can be very useful for solving concurrency problems, ***but only if programmers use them properly.*** If even one process fails to abide by the proper use of semaphores, either accidentally or deliberately, then the whole system breaks down.

Monitors

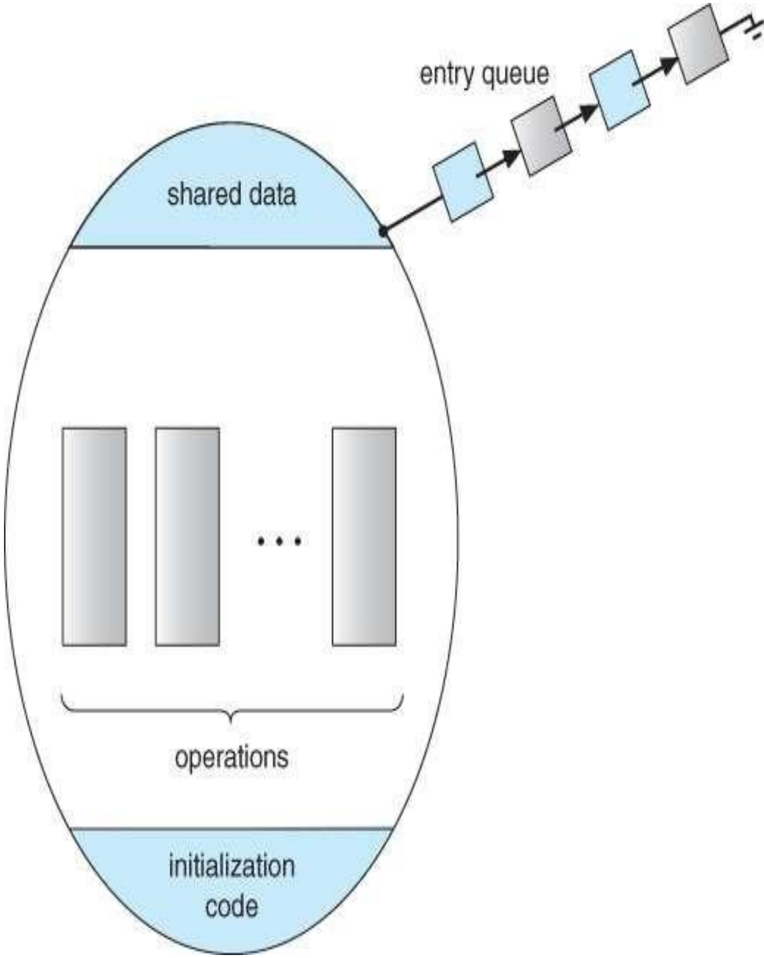
- A monitor is essentially a class, in which all data is private, and with the special restriction that only one method within any given monitor object may be active at the same time.
- An additional restriction is that monitor methods may only access the shared data within the monitor and any data passed to them as parameters, i.e. they cannot access any data external to the monitor.



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

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

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

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

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

Monitors

- In order to fully realize the potential of monitors, we need to introduce one additional new data type, known as a condition.
  - A variable of type condition has only two legal operations, wait and signal. I.e. if X was defined as type condition, then legal operations would be X.wait( ) and X.signal( )
  - The wait operation blocks a process until some other process calls signal, and adds the blocked process onto a list associated with that condition.
  - The signal process does nothing if there are no processes waiting on that condition. Otherwise it wakes up exactly one process from the condition's list of waiting processes. (Contrast this with counting semaphores, which always affect the semaphore on a signal call.)



Atomic Transactions

- Database operations frequently need to carry out atomic transactions, in which the entire transaction must either complete or not occur at all.

- The classic example is a transfer of funds, which involves withdrawing funds from one account and depositing them into another - Either both halves of the transaction must complete, or neither must complete.

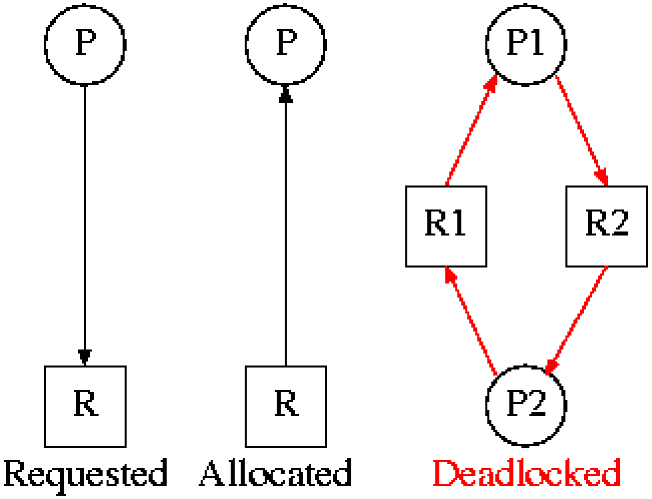
- Operating Systems can be viewed as having many of the same needs and problems as databases, in that an OS can be said to manage a small database of process-related information. As such, OSs can benefit from emulating some of the techniques originally developed for databases.

Atomic Transactions

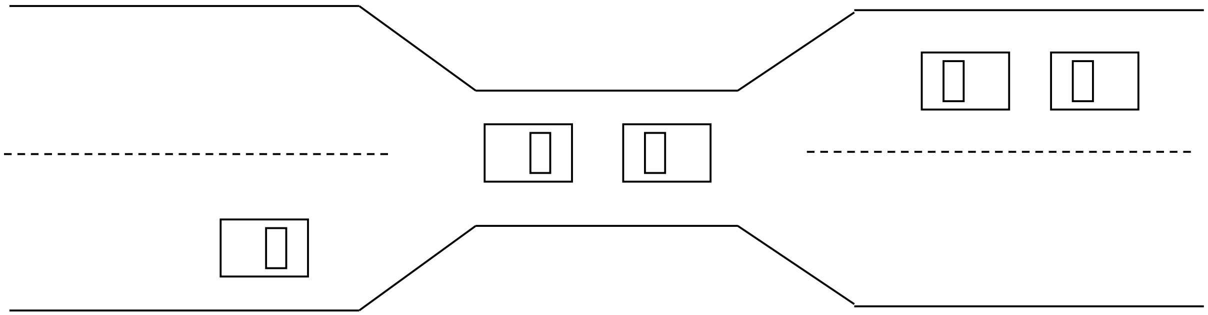
- **FUNDAMENTAL PRINCIPLES – A C I D**
- Atomicity – to outside world, transaction happens indivisibly
- Consistency – transaction preserves system invariants
- Isolated – transactions do not interfere with each other
- Durable – once a transaction “commits,” the changes are permanent

Deadlocks

- A Process request the resources, the resources are not available at that time, so the process enter into the waiting state.
- The requesting resources are held by another waiting process both are in waiting state, this situation is said



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.

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.
  - P1 and P2 each hold one tape drive and each needs another one.
- Example
  - semaphores A and B, initialized to 1

```
        P0                P1
      wait (A);                wait(B)
      wait (B);                wait(A)
```

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 {P0, P1, …, P0} of waiting processes such that P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by

  P2, …, Pn–1 is waiting for a resource that is held by Pn, and P0 is waiting for a resource that is held by P0.

System Model

- Resource types R_1, R_2, . . ., 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

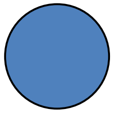
Resource-Allocation Graph

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

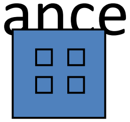
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- request edge – directed edge $P_1 \rightarrow R_j$
- assignment edge – directed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

Resource-Allocation Graph (Cont.)

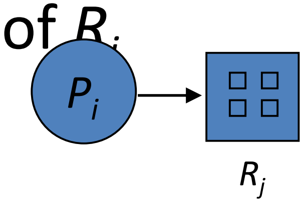
- Process



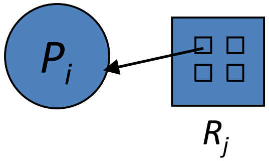
- Resource Type with 4 instances



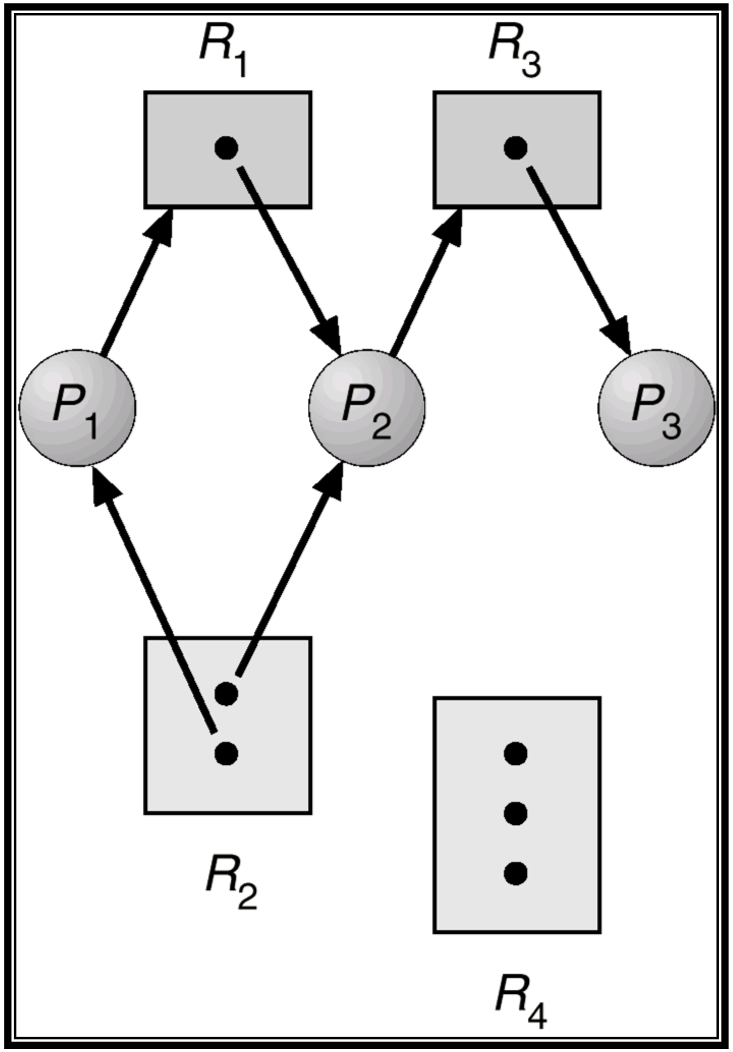
- P_i requests instance of R_j



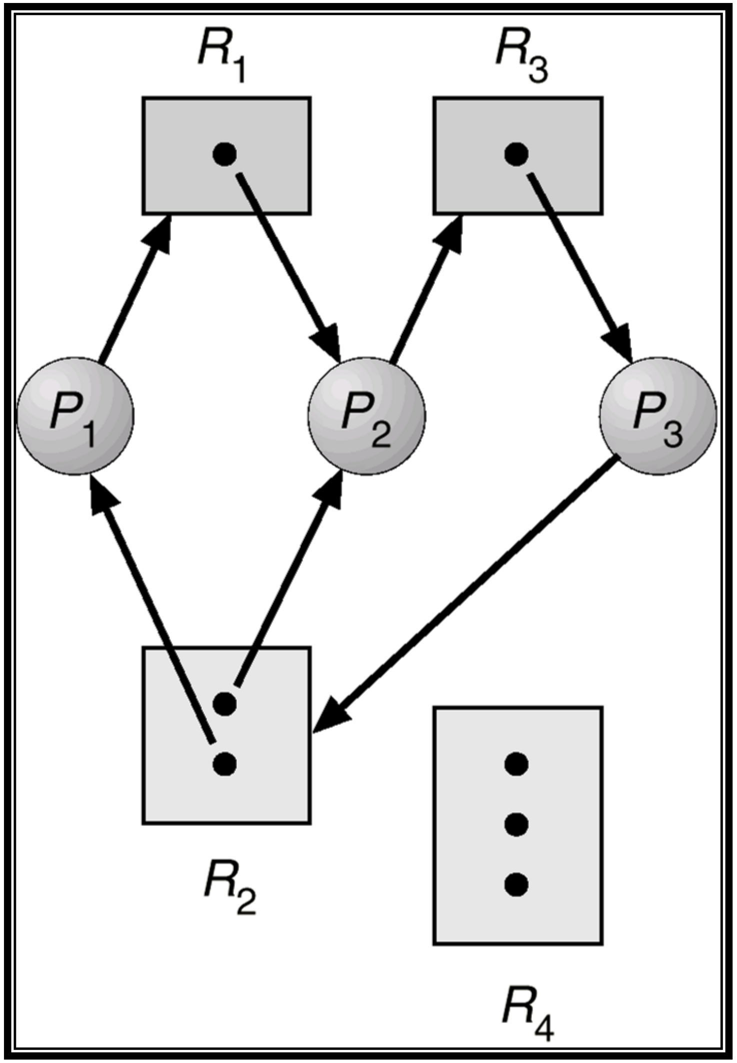
- P_i is holding an instance of R_j



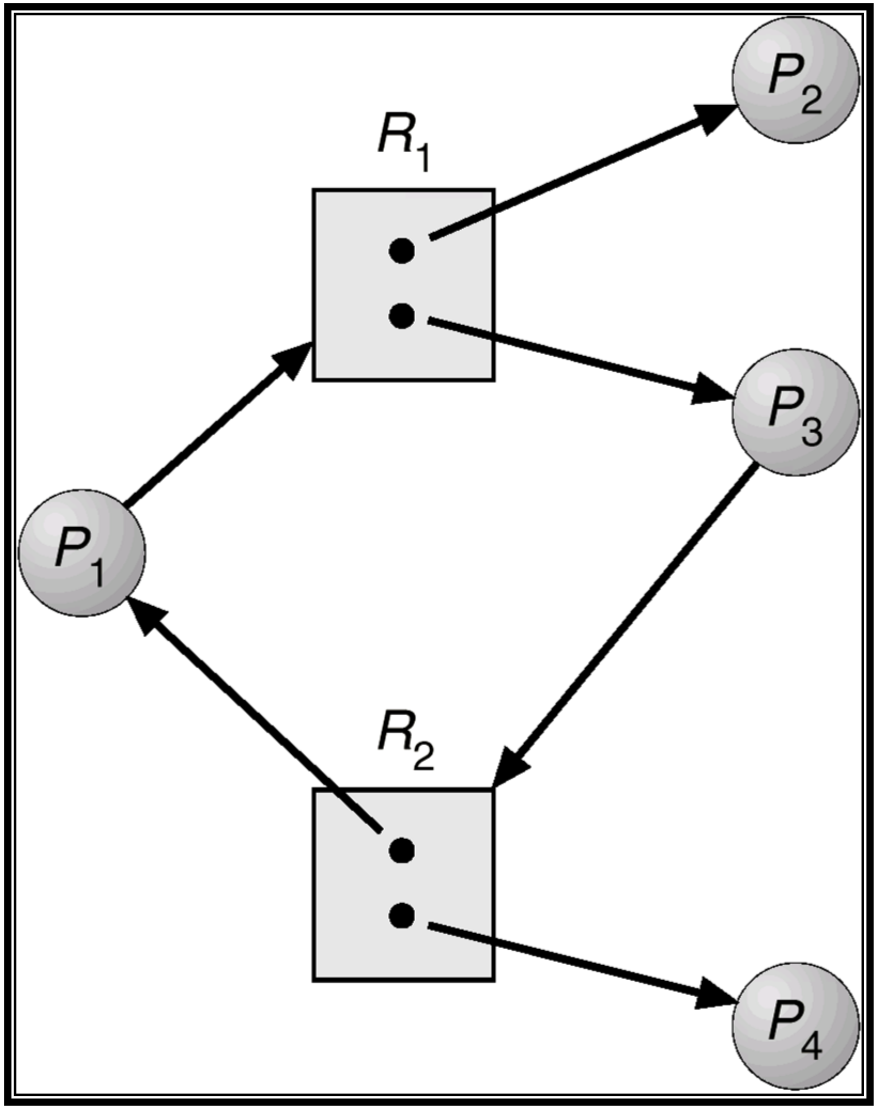
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Resource Allocation Graph With A Cycle But No Deadlock



Basic Facts

- If graph contains no cycles ⇒ 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

Restrain the ways request can be made.

- **Mutual Exclusion** – not required for sharable resources; 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.
  - 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.
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.

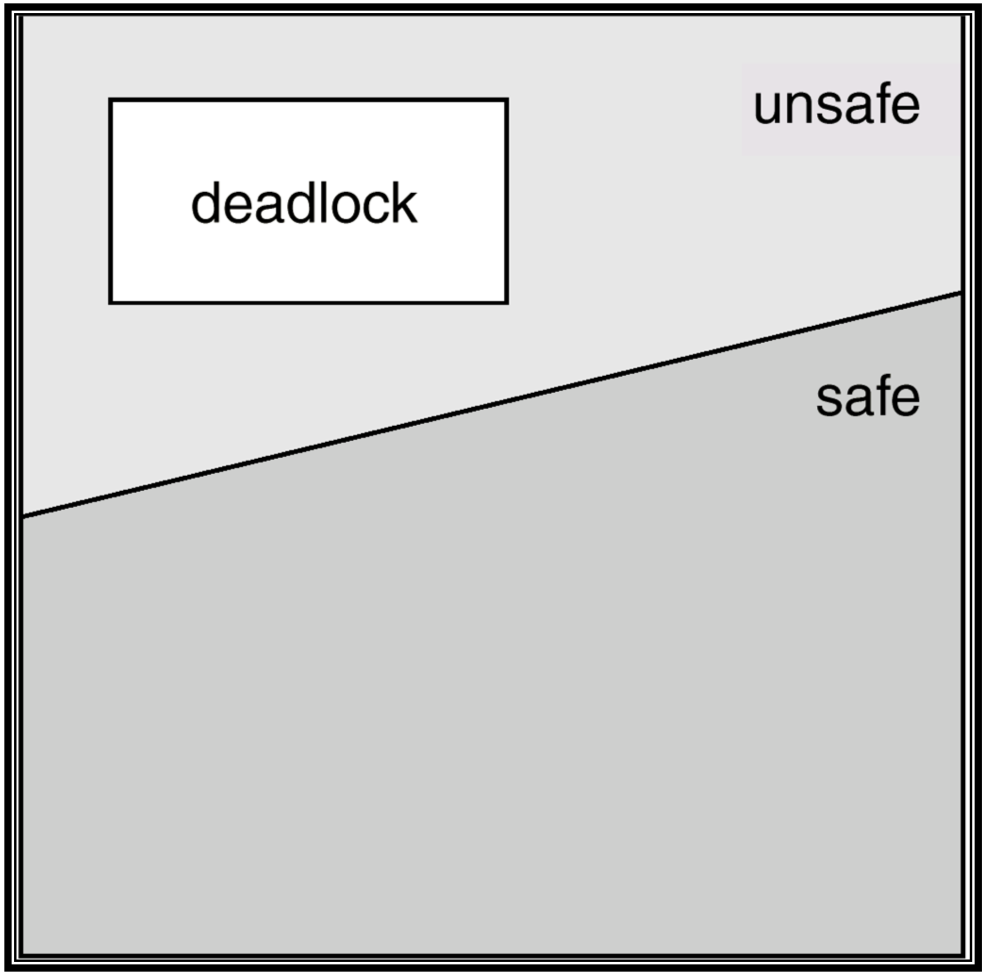
Safe State

- 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 <P1, P2, …, Pn> is safe if for each Pi, the resources that Pi can still request can be satisfied by currently available resources + resources held by all the Pj, with j<I.
  - If Pi resource needs are not immediately available, then Pi can wait until all Pj have finished.
  - When Pj is finished, Pi can obtain needed resources, execute, return allocated resources, and terminate.
  - When Pi terminates, Pi+1 can obtain its needed

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



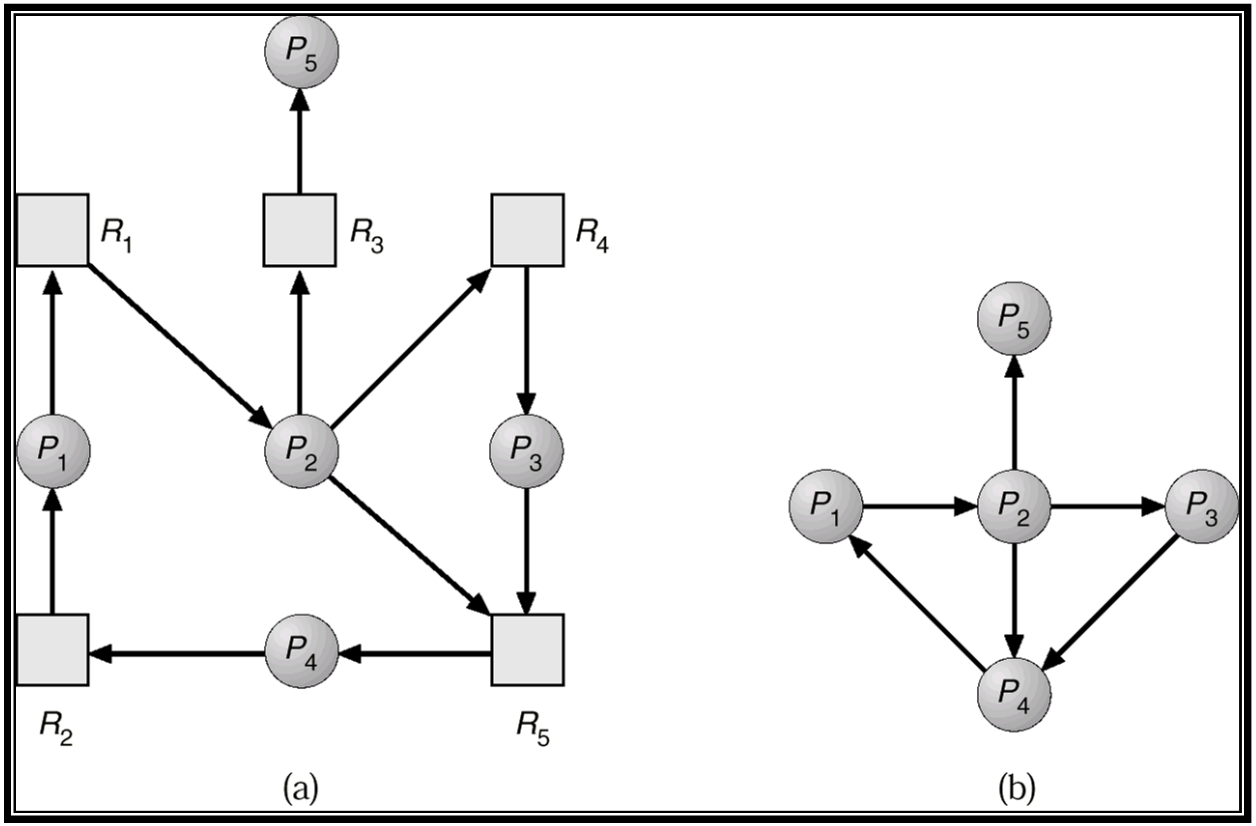
Deadlock Detection

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

Single Instance of Each Resource Type

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

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph

Corresponding wait-for graph

Several Instances of a Resource Type

- *Available:* A vector of length m indicates the number of available resources of each type.

- *Allocation:* An n 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.
- Complexity o(m * n^2)

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.

Detection Algorithm (Cont.)

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	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?

- 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.

Traffic Deadlock for Exercise 8.4