SCHOOL OF ENGINEERING

OPERATING SYSTEMS

MODULE 3

Prof. Arpita Paria,
Assistant Professor,
Dept. of CSE

SCHOOL OF ENGINEERING

Theory – Module3

Module	Chapter	Chapter Name
3	3	Process Synchronization

SCHOOL OF
ENGINEERING

Module 3 – Chapter 3 – PROCESS SYNCHRONIZATION

TOPICS

- Process Synchronization
- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Critical Regions
- Monitors

Process Synchronization:

The main objective of process synchronization is to ensure that multiple processes access shared resources without interfering with each other and to prevent the possibility of inconsistent data due to concurrent access.

Types of Process:

On the basis of synchronization, processes are categorized as one of the following two types:

Independent Process :

•The execution of one process does not affect the execution of other processes.

•No shared memory.

Cooperative Process :

• A process that can affect or be affected by other processes executing in the system.

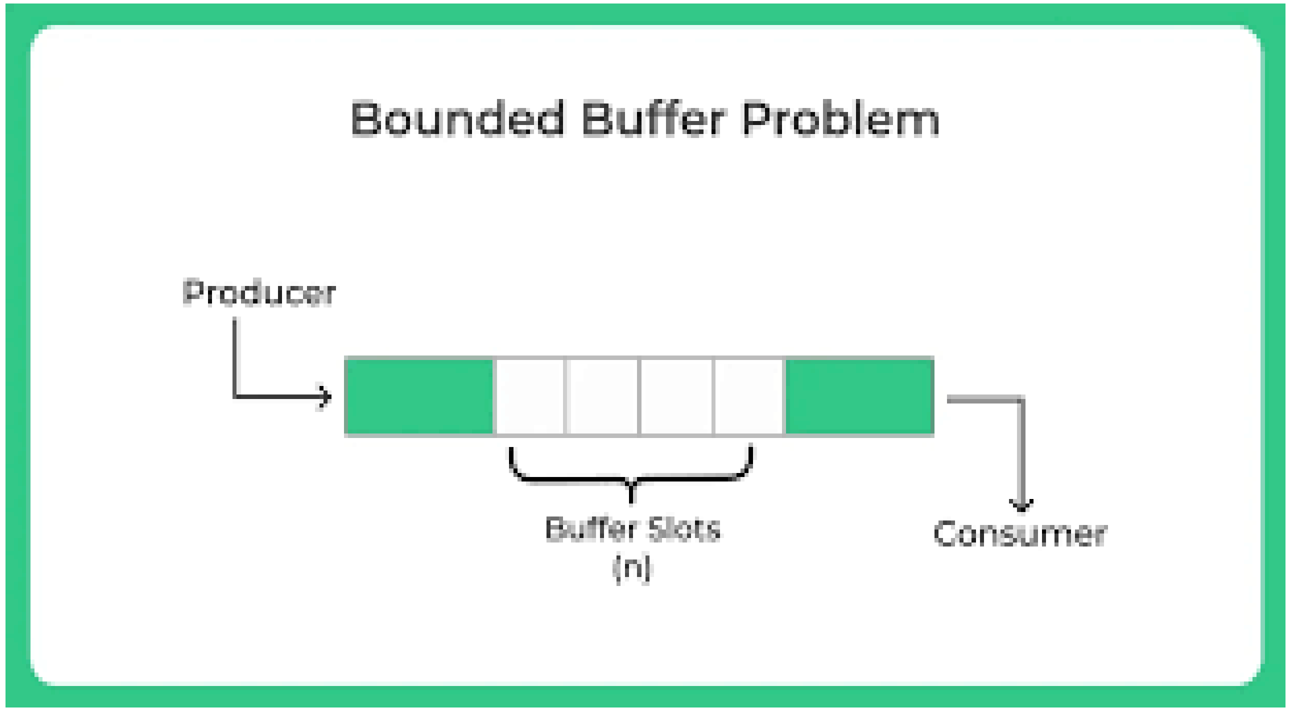
•Shared memory

•Multiple processes shares the common memory.

Background

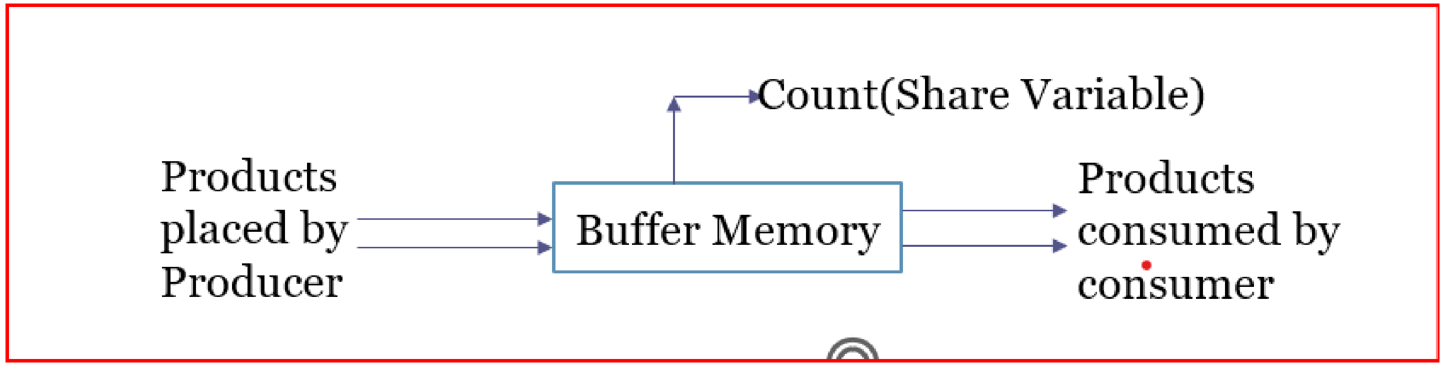
- **Concurrent access to shared data may result in data inconsistency.**
- **Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.**
- Shared-memory solution to bounded-butter problem allows at most
n – 1 items in buffer at the same time. A solution, 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 PROBLEM:



Two Process are there:

1) Producer: Produces product and place it in a buffer(memory).
2) Consumer: Consumes the product which was placed in buffer(memory)by the producer.



Condition at producer side: (Count will be incremented)

1)Before placing the product into the buffer producer have to check buffer is not full.

Condition at Consumer side:(Count will be decremented)

1)Before consuming product from buffer consumer have to check buffer is not empty.

SCHOOL OF ENGINEERING

Contd….

Shared data

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

BOUNDED BUFFER PROBLEM

- Producer process

```
item nextProduced;

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

BOUNDED BUFFER PROBLEM

- Consumer process

```
item nextConsumed;

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

BOUNDED BUFFER PROBLEM

- The statements

  ```
  counter++;
  counter--;
  ```

  must be performed *atomically*.

- Atomic operation means an operation that completes in its entirety without interruption.

Bounded Buffer

- 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

Bounded Buffer

- 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.

Bounded Buffer

- Assume **counter** is initially 5. One interleaving of statements is:

  T0:producer: **register1 = counter** (*register1 = 5*)
  T1:producer: **register1 = register1 + 1** (*register1 = 6*)
  T2:consumer: **register2 = counter** (*register2 = 5*)
  T3:consumer: **register2 = register2 – 1** (*register2 = 4*)
  T4:producer: **counter = register1** (*counter = 6*)
  T5:consumer: **counter = register2** (*counter = 4*)

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

SCHOOL OF ENGINEERING

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 in which the shared data is accessed.
- The process may be changing common variables, updating a table, writing a file and so on.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

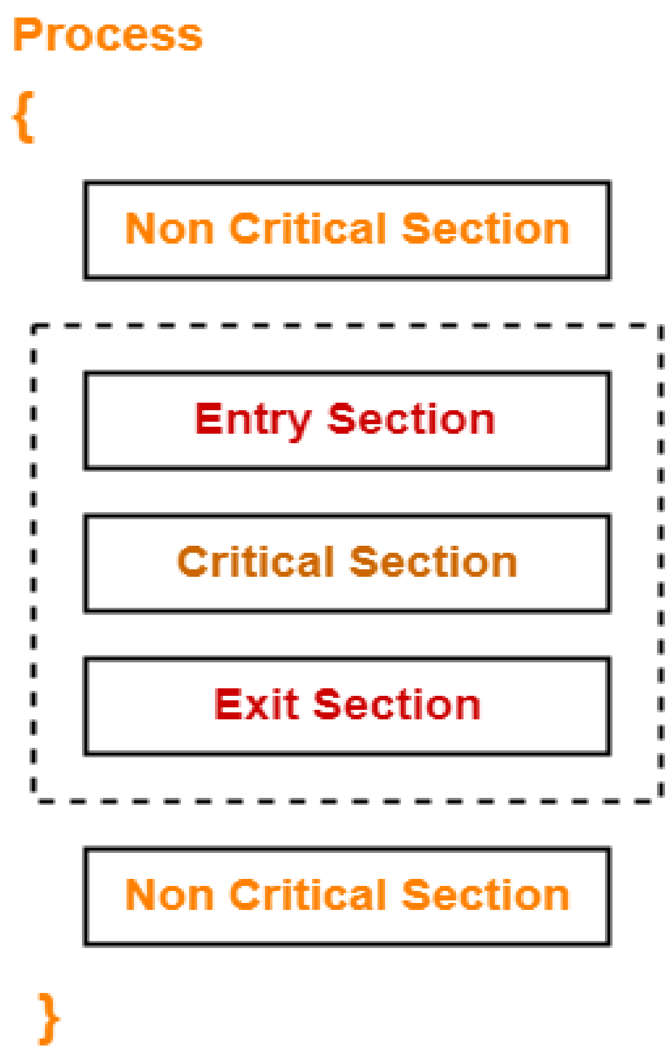
Example:

Fig 1, Process P1 is updating B value. This updated B value need to be used by Process P2 to calculate D value. If no synchronization, before B getting updated by P1, if it is used for calculation of D value in P2, then results will be wrong.

```
P1( )
{
C = B - 1 ;
B = 2 x C ;
}
```

```
P2( )
{
D = 2 x B ;
B = D - 1 ;
}
```

Critical Section Problem:



- Each Process must request permission to enter its **critical section.**
- The section of code implementing this request is the **entry section.**
- The critical section may be followed by an **exit section.**
- The remaining code is the **remainder section.**

Solution to Critical-Section Problem must satisfy the following Three Requirements:

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: Peterson's Solution

Combined shared variables of algorithms 1 and 2.

Process P_i

```
do {
    flag [i]:= true; // process is ready to enter critical section
turn = j; //whose turn is to enter Critical section
while (flag [j] and turn = j) ;
        critical section
    flag [i] = false;
        remainder section
} while (1);
```

Process P_j

```
do {
    flag [j]:= true; // process is ready to enter critical section
turn = i; //whose turn is to enter Critical section
while (flag [i] and turn = i) ;
        critical section
    flag [j] = false;
        remainder section
} while (1);
```

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

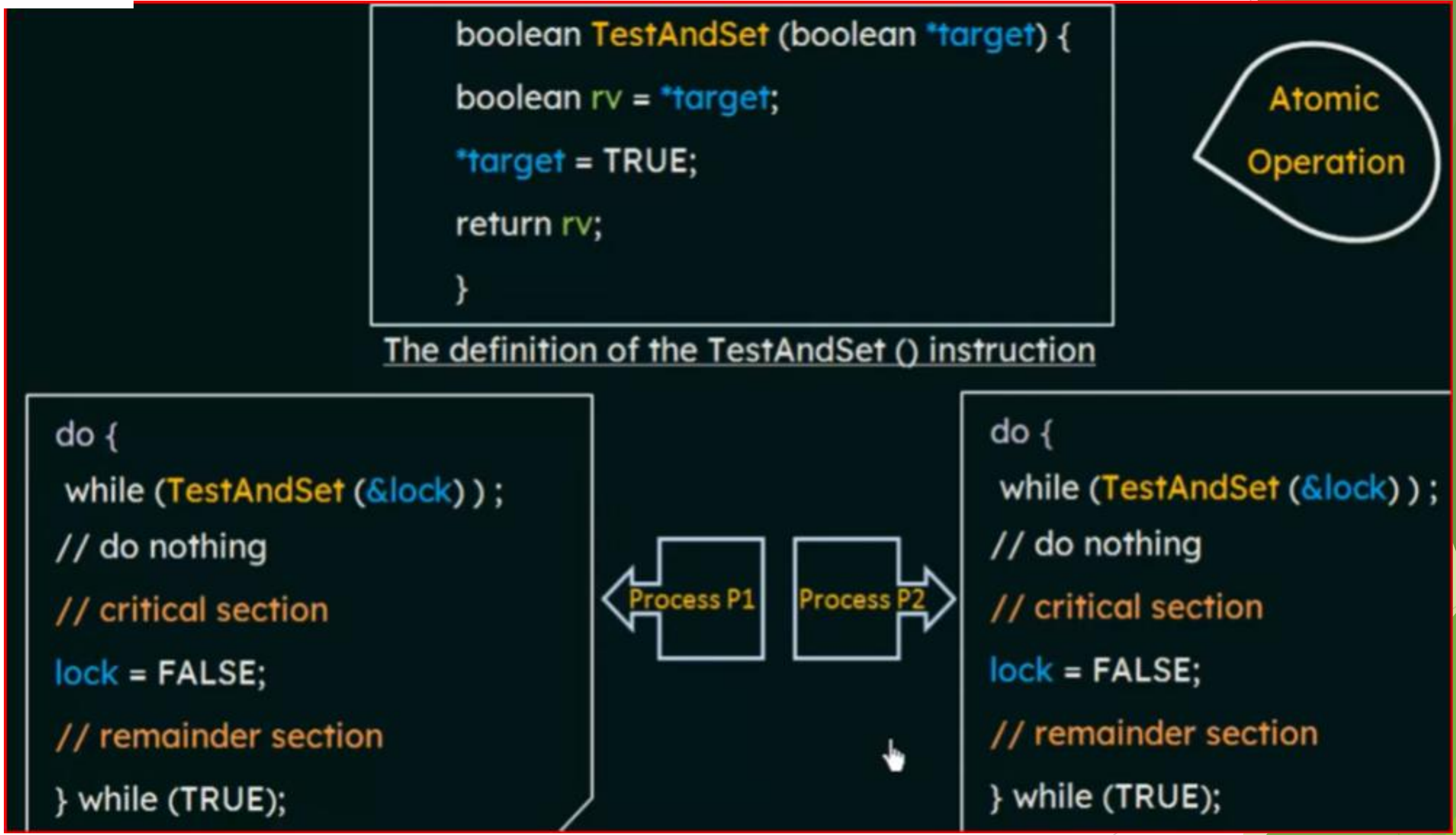
Synchronization Hardware

- Interrupt Disabling
  - A process runs until it invokes an operating system service or until it is interrupted.
  - Disabling interrupts guarantees mutual exclusion.
  - It works on Non-preemptive approach.
  - Will not work in multiprocessor architecture- delays due to message passing to disable interrupt.
  - Uses Non-preemptive approach, hence less process interleaving resulting in reduced system efficiency.

Special Machine Instructions(Test and Set Instruction)

- A hardware solution to the synchronization problem.
- There is a shared lock variable which can take either of the two values,0 or 1.
- Before entering into the critical section, a process inquires about the lock.
- If it is locked.it keeps on waiting till it becomes free.

SCHOOL OF ENGINEERING



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

The definition of the TestAndSet () instruction

```
do {
 while (TestAndSet (&lock) ) ;
// do nothing
// critical section
lock = FALSE;
// remainder section
} while (TRUE);
```

```
do {
 while (TestAndSet (&lock) ) ;
// do nothing
// critical section
lock = FALSE;
// remainder section
} while (TRUE);
```

Working of Test & Set operation

Mutual Exclusion with Swap

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

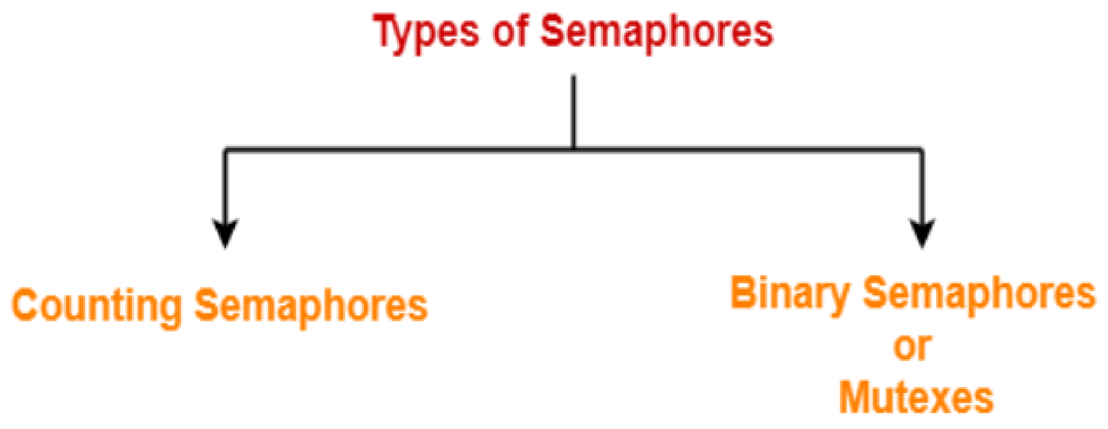
```
boolean lock;
boolean waiting[n];
```

- Process P_i

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

Semaphore

- Semaphore is an integer variable which is used in mutual exclusive manner by various concurrent cooperative processes in order to achieve synchronization.

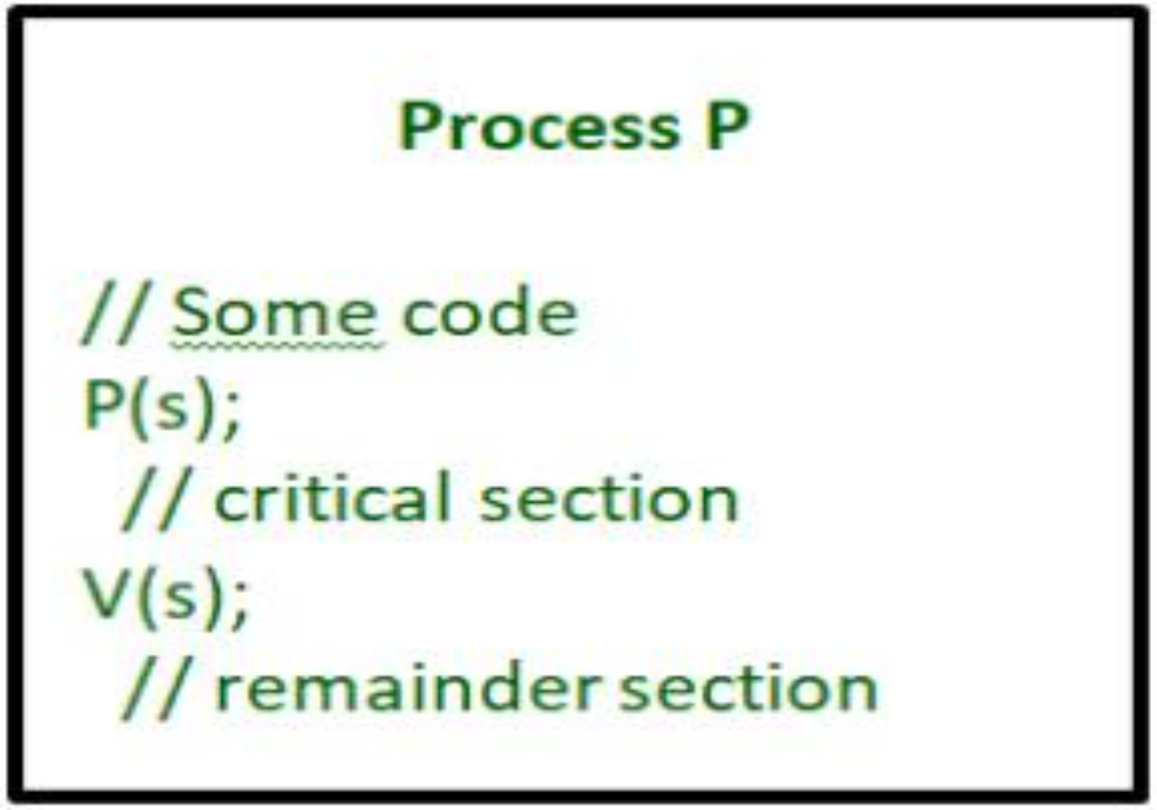


1. Counting Semaphores
2. Binary Semaphores or Mutexes

Binary Semaphore

- This is also known as mutex lock. It can have only two values – 0 and 1.
- Its value is initialized to 1.
- It is used to implement the solution of critical section problems with multiple processes.
- **P operation is also called wait, sleep or down operation** and
- **V operation is also called signal, wake-up or up operation**.
- Both operations are atomic and semaphore(s) is always initialized to one. Here atomic means that variable on which read, modify and update happens at the same time/moment with no pre-emption i.e. in between read, modify and update no other operation is performed that may change the variable.
- A critical section is surrounded by both operations to implement process synchronization.

Binary Semaphore



Process P

```
// Some code
P(s);
  // critical section
V(s);
  // remainder section
```

Binary Semaphore: (Wait(),Signal())

```
Down(Semaphore S)
 {
   if(S value==1)
     {
        S value=0;
     }

     else
   {
     Block this process
    and place in suspend list,
     sleep();
  }
}
```

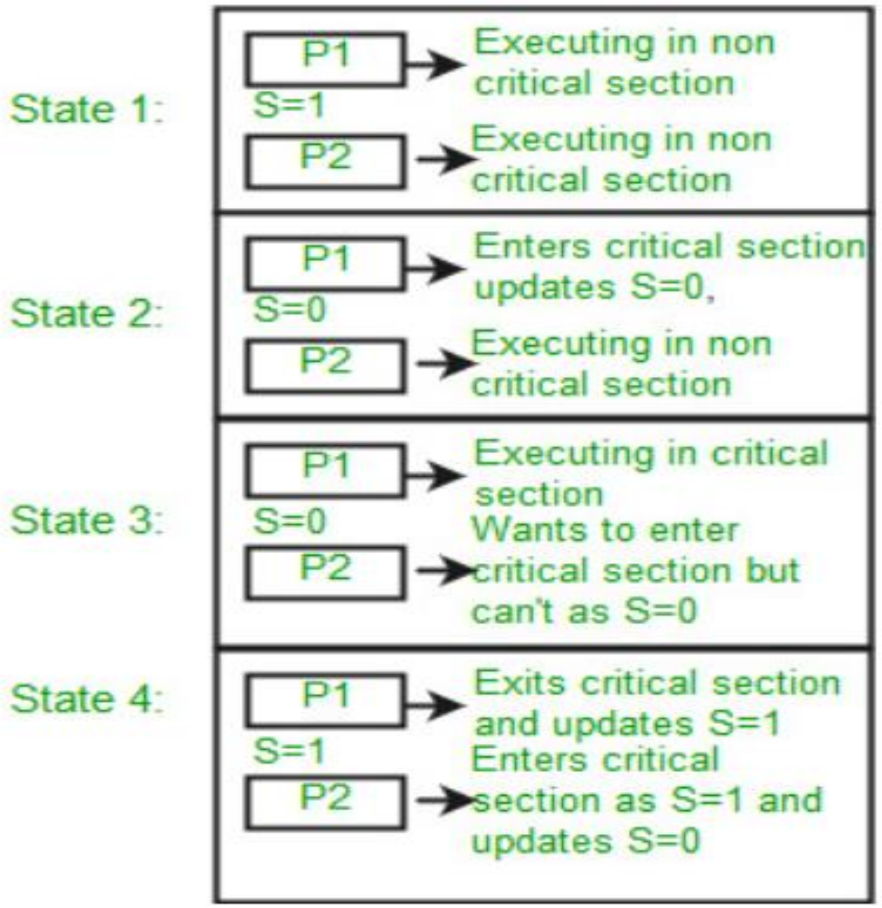
```
Up(Semaphore S)
{
If(Suspend list is empty)
{
S value=1;
}
Else
{
Select a process from Suspend list
And wake up();
}
}
```

Binary Semaphore(Explanation)

- To implement mutual exclusion.
- Let there be two processes P1 and P2 and a semaphore s is initialized as 1.
- Now if suppose P1 enters in its critical section then the value of semaphore s becomes 0.
- Now if P2 wants to enter its critical section then it will wait until s > 0, this can only happen when P1 finishes its critical section and calls V operation on semaphore s.
- This way mutual exclusion is achieved.

SCHOOL OF ENGINEERING

Binary Semaphore


State 1:
P1
Executing in non critical section
S=1
P2
Executing in non critical section
State 2:
P1
Enters critical section updates S=0,
S=0
P2
Executing in non critical section
State 3:
P1
Executing in critical section
S=0
P2
Wants to enter critical section but can't as S=0
State 4:
P1
Exits critical section and updates S=1
S=1
P2
Enters critical section as S=1 and updates S=0

Counting Semaphore

- A counting semaphore has two components-
- An integer value.
- An associated waiting list (usually a queue).
- The value of counting semaphore may be positive or negative.
- Positive value indicates the number of processes that can be present in the critical section at the same time.
- Negative value indicates the number of processes that are blocked in the waiting list.

Counting Semaphore:

```
Down(Semaphore S)
{
S value = S value-1;
If(S value <0)
{
   put process(PCB)in Suspended
   list,
 Sleep();
}
else
Return;
}
```

```
Up(Semaphore S)
{
S value = S value+1;
If(S value <=0)
{
Select a process from Suspended
List &&
Wake up();
}
}
```

Contd....

- The **wait operation** is executed when a process tries to enter the critical section.
- Wait operation decrements the value of counting semaphore by 1.

Then, following two cases are possible-

Case-01: Counting Semaphore Value >= 0

If the resulting value of counting semaphore is greater than or equal to 0, process is allowed to enter the critical section.

Case-02: Counting Semaphore Value < 0

If the resulting value of counting semaphore is less than 0, process is not allowed to enter the critical section.

In this case, process is put to sleep in the waiting list.

Contd…

- The **signal operation** is executed when a process takes exit from the critical section.
- Signal operation increments the value of counting semaphore by 1.

- Then, following two cases are possible-

- **<u>Case-01: Counting Semaphore <= 0</u>**

- If the resulting value of counting semaphore is less than or equal to 0, a process is chosen from the waiting list and wake up to execute.

- **<u>Case-02: Counting Semaphore > 0</u>**

- If the resulting value of counting semaphore is greater than 0, no action is taken.

- **Problem-01:**

A counting semaphore S is initialized to 10. Then, 6 P operations and 4 V operations are performed on S. What is the final value of S?

- **Solution-**

- **P operation** also called as wait operation **decrements** the value of semaphore variable by 1.
- **V operation** also called as signal operation **increments** the value of semaphore variable by 1.

Thus,

- Final value of semaphore variable S
- = 10 – (6 x 1) + (4 x 1)
- = 10 – 6 + 4
- =8

Problem-02:

- A counting semaphore S is initialized to 7. Then, 20 P operations and 15 V operations are performed on S. What is the final value of S?

- **Solution-**

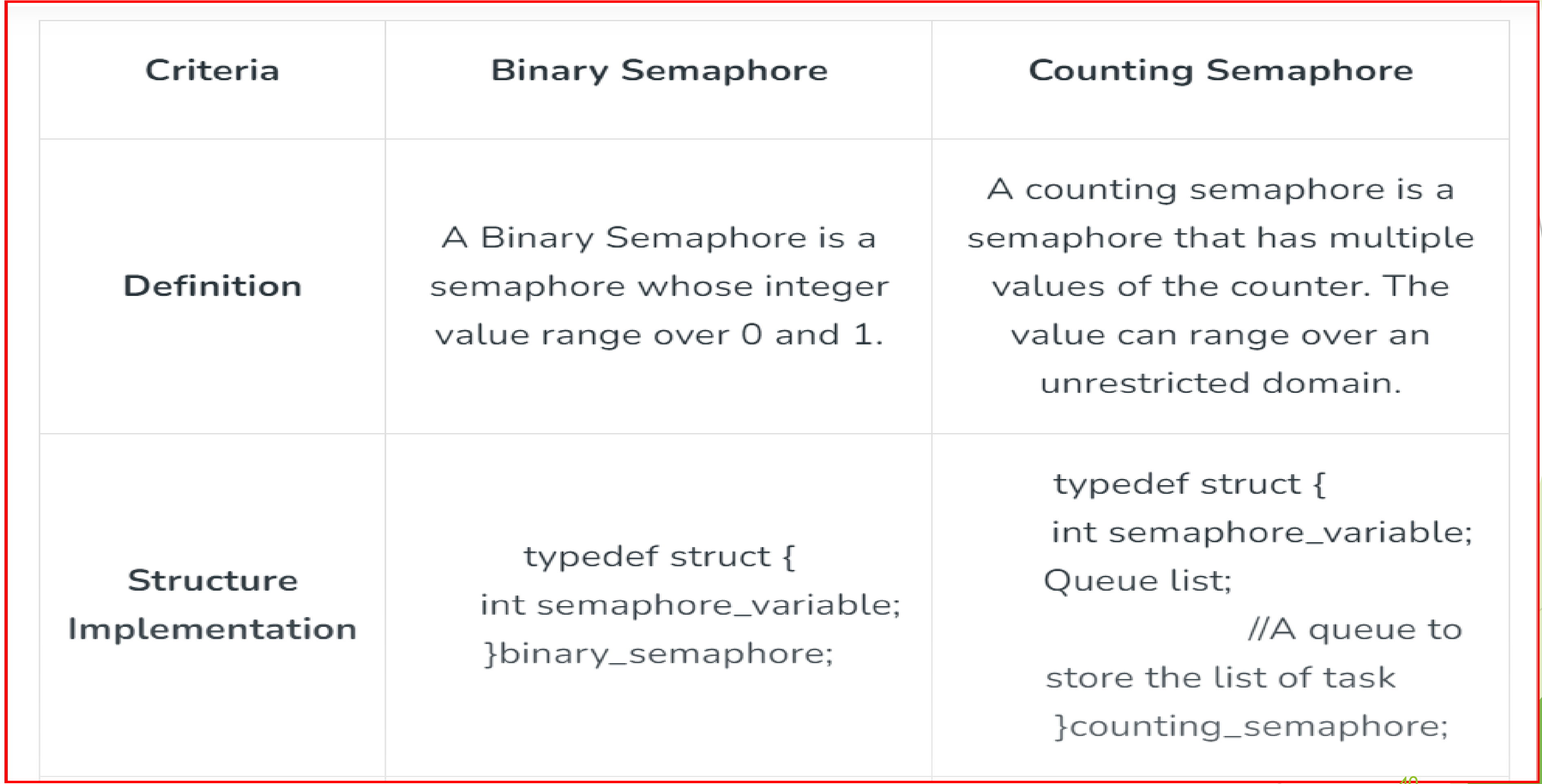
- P operation also called as wait operation decrements the value of semaphore variable by 1.
- V operation also called as signal operation increments the value of semaphore variable by 1.

Thus,

- Final value of semaphore variable S
- = 7 – (20 x 1) + (15 x 1)
- = 7 – 20 + 15
- = 2

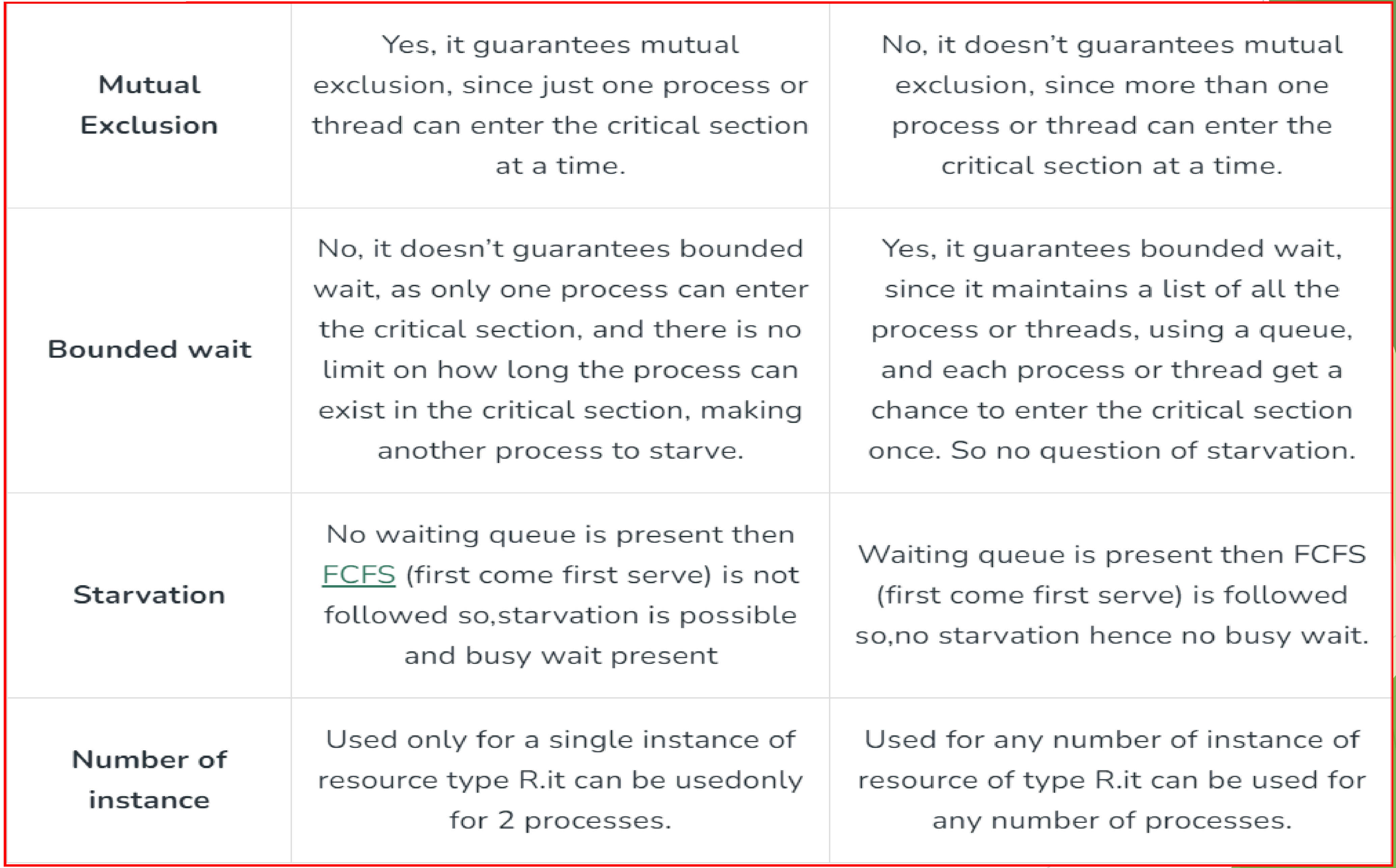
SCHOOL OF ENGINEERING

Difference between Binary Semaphore and Counting Semaphore?



Criteria	Binary Semaphore	Counting Semaphore
Definition	A Binary Semaphore is a semaphore whose integer value range over 0 and 1.	A counting semaphore is a semaphore that has multiple values of the counter. The value can range over an unrestricted domain.

Structure Implementation	typedef struct {
	int semaphore_variable;
	}binary_semaphore;
		typedef struct {
		int semaphore_variable;
		Queue list; //A queue to store the list of task
		}counting_semaphore;



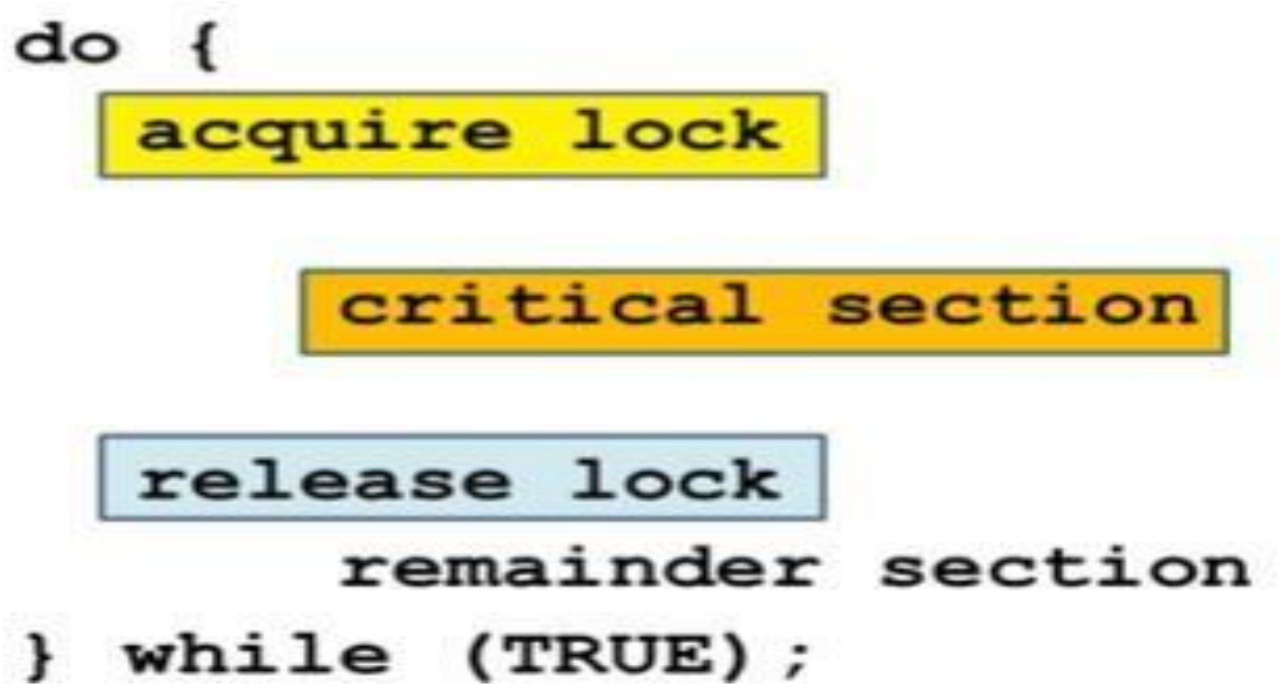
Mutual Exclusion	Yes, it guarantees mutual exclusion, since just one process or thread can enter the critical section at a time.	No, it doesn't guarantees mutual exclusion, since more than one process or thread can enter the critical section at a time.
Bounded wait	No, it doesn't guarantees bounded wait, as only one process can enter the critical section, and there is no limit on how long the process can exist in the critical section, making another process to starve.	Yes, it guarantees bounded wait, since it maintains a list of all the process or threads, using a queue, and each process or thread get a chance to enter the critical section once. So no question of starvation.
Starvation	No waiting queue is present then FCFS (first come first serve) is not followed so,starvation is possible and busy wait present	Waiting queue is present then FCFS (first come first serve) is followed so,no starvation hence no busy wait.
Number of instance	Used only for a single instance of resource type R.it can be usedonly for 2 processes.	Used for any number of instance of resource of type R.it can be used for any number of processes.

Mutex

Binary semaphore and Mutex are same except following difference.

- Process that locks the mutex must be the one to unlock it.
- In Binary semaphore, one process lock semaphore and another will unlock it.

SCHOOL OF ENGINEERING


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

```
acquire() {
   while (!available); /* busy wait */
   available = false;
 }
```

```
release() {
   available = true;
}
```

Disadvantage: **busy waiting**. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire().

Strong/Weak Semaphore

- ***Strong Semaphores*** use FIFO.
- ***Weak Semaphores*** don’t specify the order of removal from the queue.

SCHOOL OF ENGINEERING

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);
```

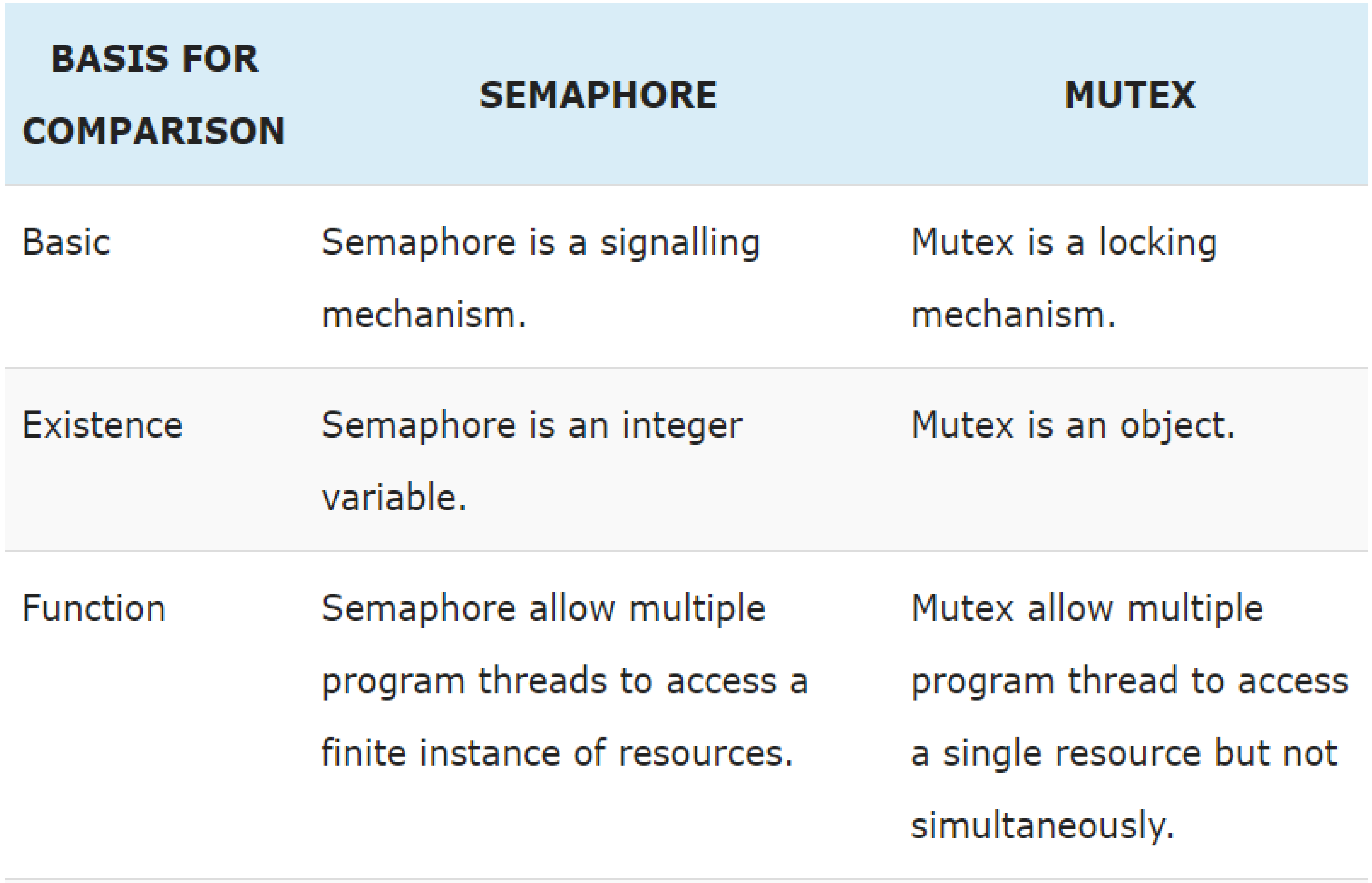
Deadlock and Starvation

- **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

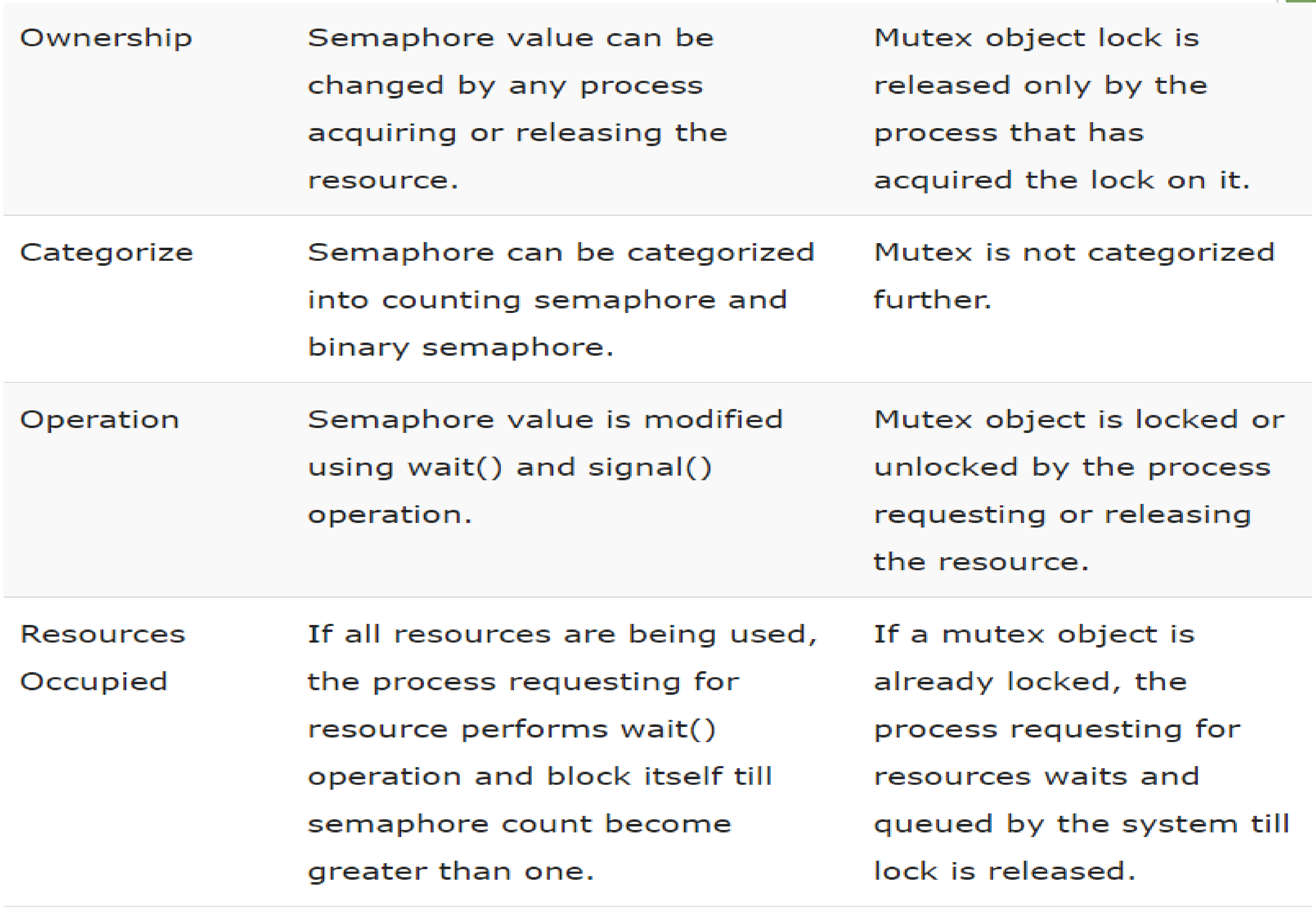
P_0	P_1
$wait(S);$	$wait(Q);$
$wait(Q);$	$wait(S);$
$\vdots$	$\vdots$
$signal(S);$	$signal(Q);$
$signal(Q)$	$signal(S);$

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

Difference Between Semaphore & Mutex



BASIS FOR COMPARISON	SEMAPHORE	MUTEX
Basic	Semaphore is a signalling mechanism.	Mutex is a locking mechanism.
Existence	Semaphore is an integer variable.	Mutex is an object.
Function	Semaphore allow multiple program threads to access a finite instance of resources.	Mutex allow multiple program thread to access a single resource but not simultaneously.

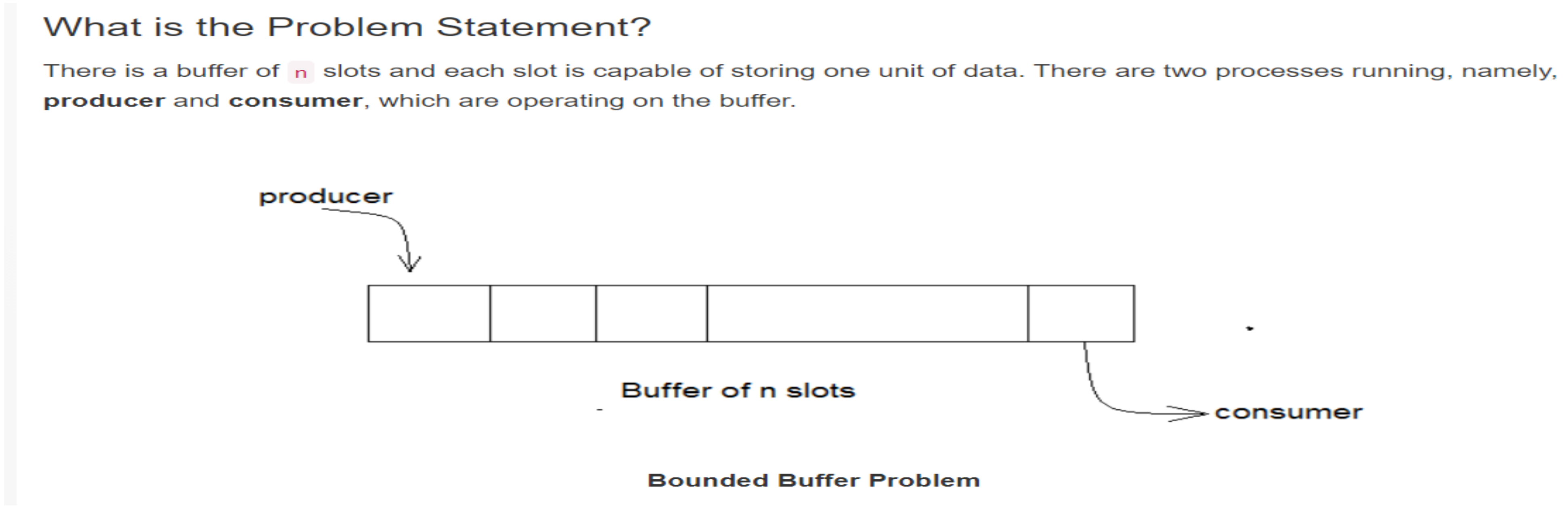


Ownership	Semaphore value can be changed by any process acquiring or releasing the resource.	Mutex object lock is released only by the process that has acquired the lock on it.
Categorize	Semaphore can be categorized into counting semaphore and binary semaphore.	Mutex is not categorized further.
Operation	Semaphore value is modified using wait() and signal() operation.	Mutex object is locked or unlocked by the process requesting or releasing the resource.
Resources Occupied	If all resources are being used, the process requesting for resource performs wait() operation and block itself till semaphore count become greater than one.	If a mutex object is already locked, the process requesting for resources waits and queued by the system till lock is released.

Classical Problems of Synchronization

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

Bounded Buffer (Producer-Consumer) Problem

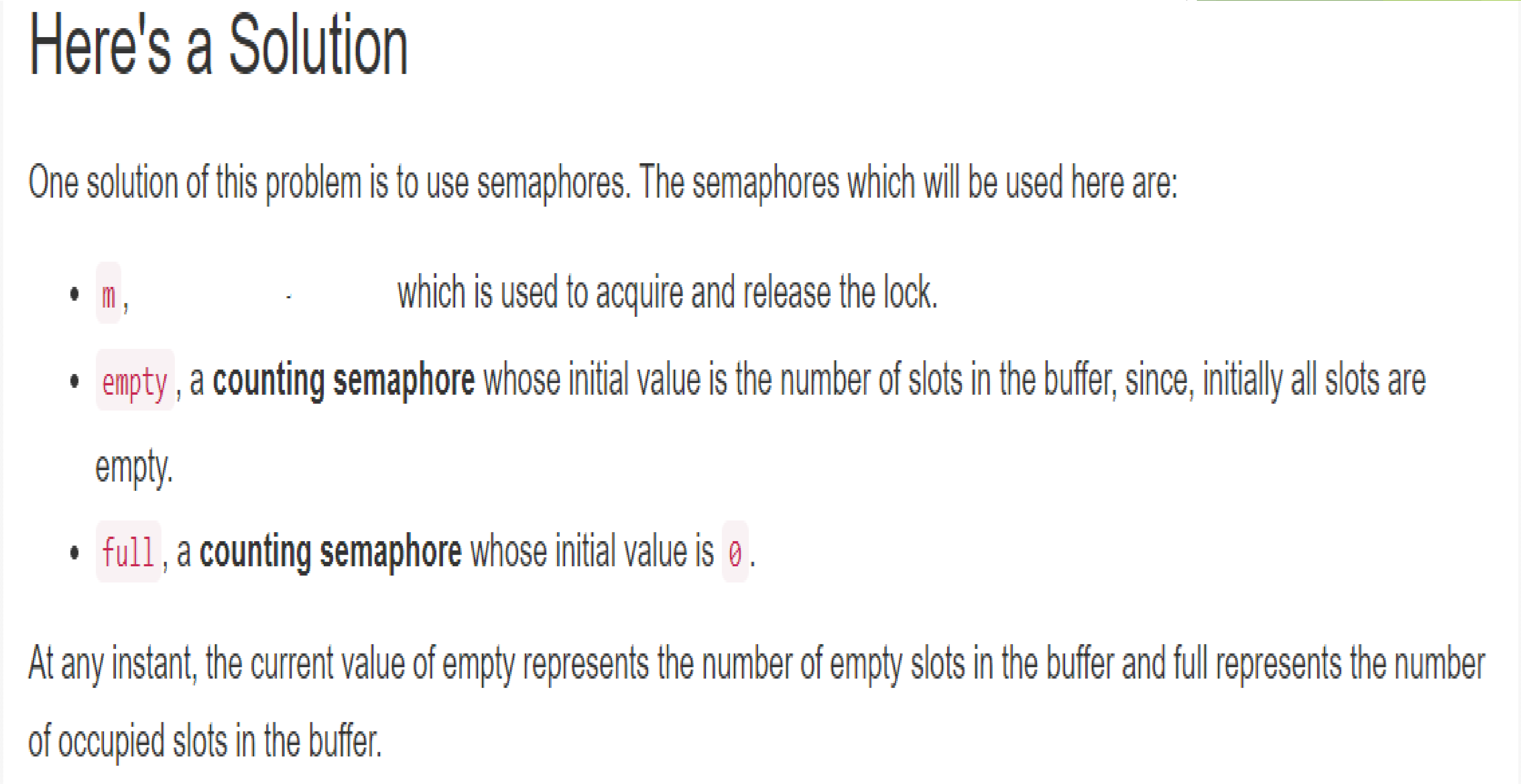


What is the Problem Statement?

There is a buffer of `n` slots and each slot is capable of storing one unit of data. There are two processes running, namely, **producer** and **consumer**, which are operating on the buffer.

Bounded Buffer Problem

- A producer tries to insert data into an empty slot of the buffer. A consumer tries to remove data from a filled slot in the buffer. As you might have guessed by now, those two processes won't produce the expected output if they are being executed concurrently.
- There needs to be a way to make the producer and consumer work in an independent manner.



Here's a Solution

One solution of this problem is to use semaphores. The semaphores which will be used here are:

- `m`, which is used to acquire and release the lock.
- `empty`, a **counting semaphore** whose initial value is the number of slots in the buffer, since, initially all slots are empty.
- `full`, a **counting semaphore** whose initial value is `0`.

At any instant, the current value of empty represents the number of empty slots in the buffer and full represents the number of occupied slots in the buffer.

SCHOOL OF ENGINEERING

Producer operation

- **do {**
- // wait until empty slots > 0 and then decrement 'empty'
- **wait(empty);**
- // acquire lock
- **wait(mutex);**
- /* perform the insert operation in a slot */
- // release lock
- **signal(mutex);**
- // increment 'full'
- **signal(full);**
- **}**
- **while(TRUE)**

Consumer operation

- **do**
- **{**
-         // wait until full > 0 and then decrement 'full'
- **wait(full);**
-           // acquire the lock
- **wait(mutex);**
- /* perform the remove operation in a slot */
- // release the lock
- **signal(mutex);**
- // increment 'empty'
    signal(empty);
- **}**
- **while(TRUE);**

Readers Writers Problem

The Problem statement

- A database is to be shared among several concurrent processes.
- Some of these processes may want only to read the database, whereas others may want to update(that is, read and write) the database.
- We distinguish between these two types of processes by referring to the former as **Readers** and to the latter as **Writers**.
- If two readers access the shared data simultaneously, no adverse affects will result.
- However, if a writer and some other thread(either a reader or a writer) access the database simultaneously, chaos may ensue.
- To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database.
- This synchronization problem is referred to as the readers-writers problem.

Solution to the Readers-Writers Problem using Semaphores

- We will make use of two semaphores and an integer variable:

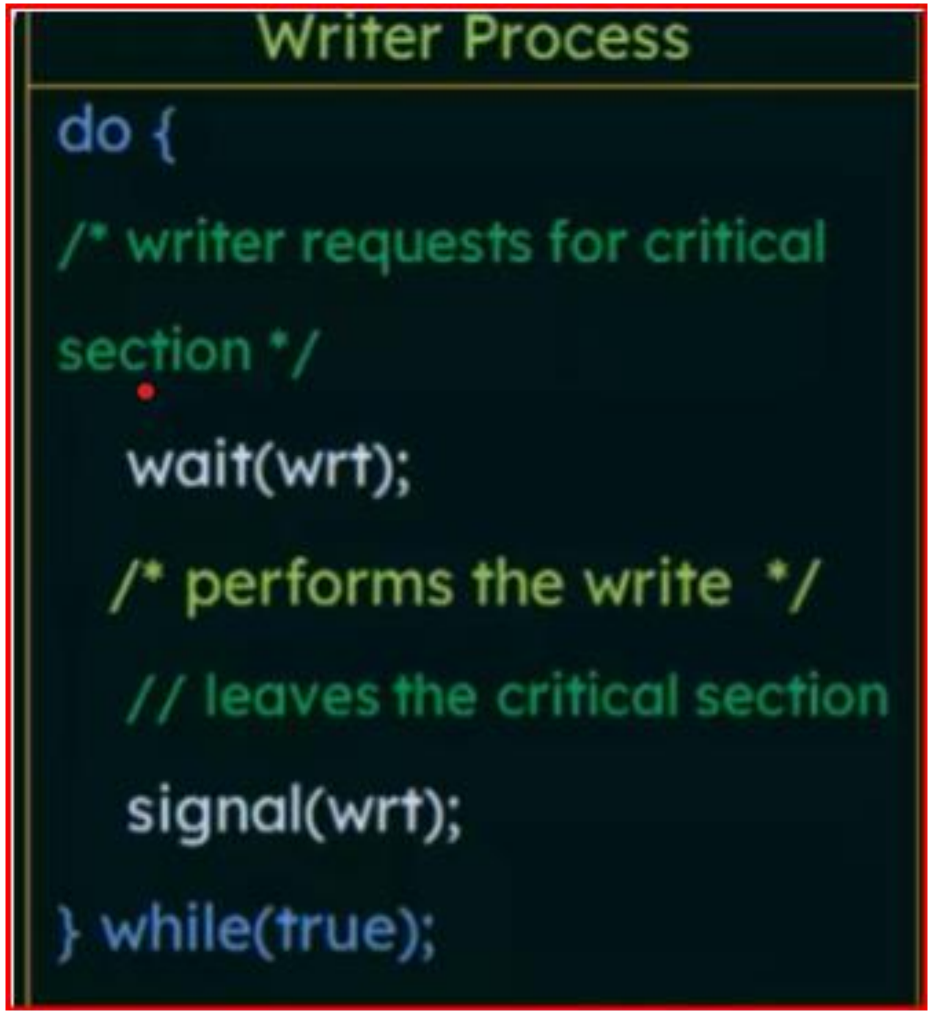
1.**mutex**,a semaphore(initialized to 1)which is used to ensure mutual exclusion when read_count is updated i.e. when any reader enters or exit from the critical section.

2. **wrt**, a semaphore(initialized to 1) common to both reader and writer processes.

3. **read_count**, an integer variable(initialized to 0) that keeps track of how many processes are currently reading the object.

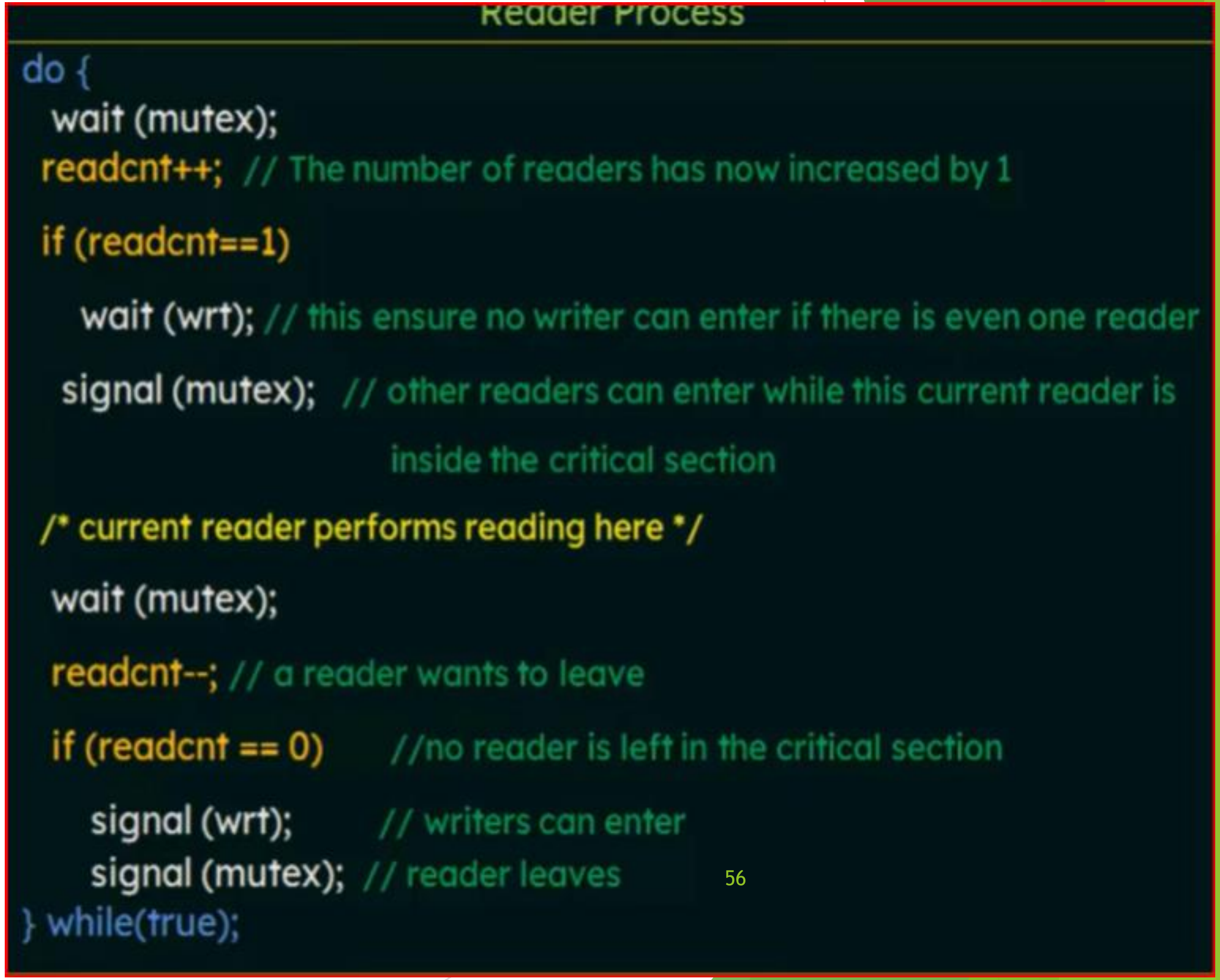
SCHOOL OF ENGINEERING

Readers Writers Problem



Writer Process

```
do {
/* writer requests for critical
section */
    wait(wrt);
   /* performs the write  */
    // leaves the critical section
    signal(wrt);
} while(true);
```

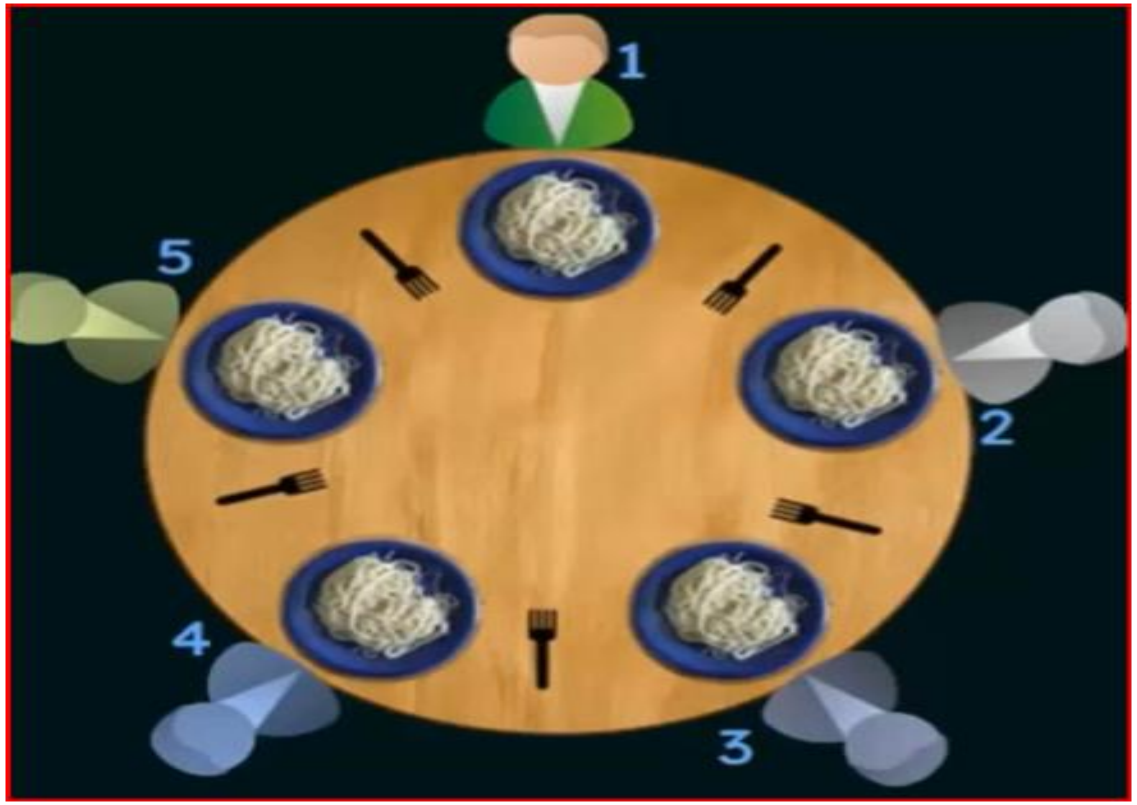


Reader Process

```
do {
  wait (mutex);
 readcnt++;  // The number of readers has now increased by 1
 if (readcnt==1)
     wait (wrt); // this ensure no writer can enter if there is even one reader
   signal (mutex);   // other readers can enter while this current reader is
                         inside the critical section
 /* current reader performs reading here */
  wait (mutex);
  readcnt--; // a reader wants to leave
  if (readcnt == 0)      //no reader is left in the critical section
      signal (wrt);        // writers can enter
      signal (mutex);  // reader leaves
} while(true);
```

SCHOOL OF ENGINEERING

Dining Philosophers Problem



Contd..

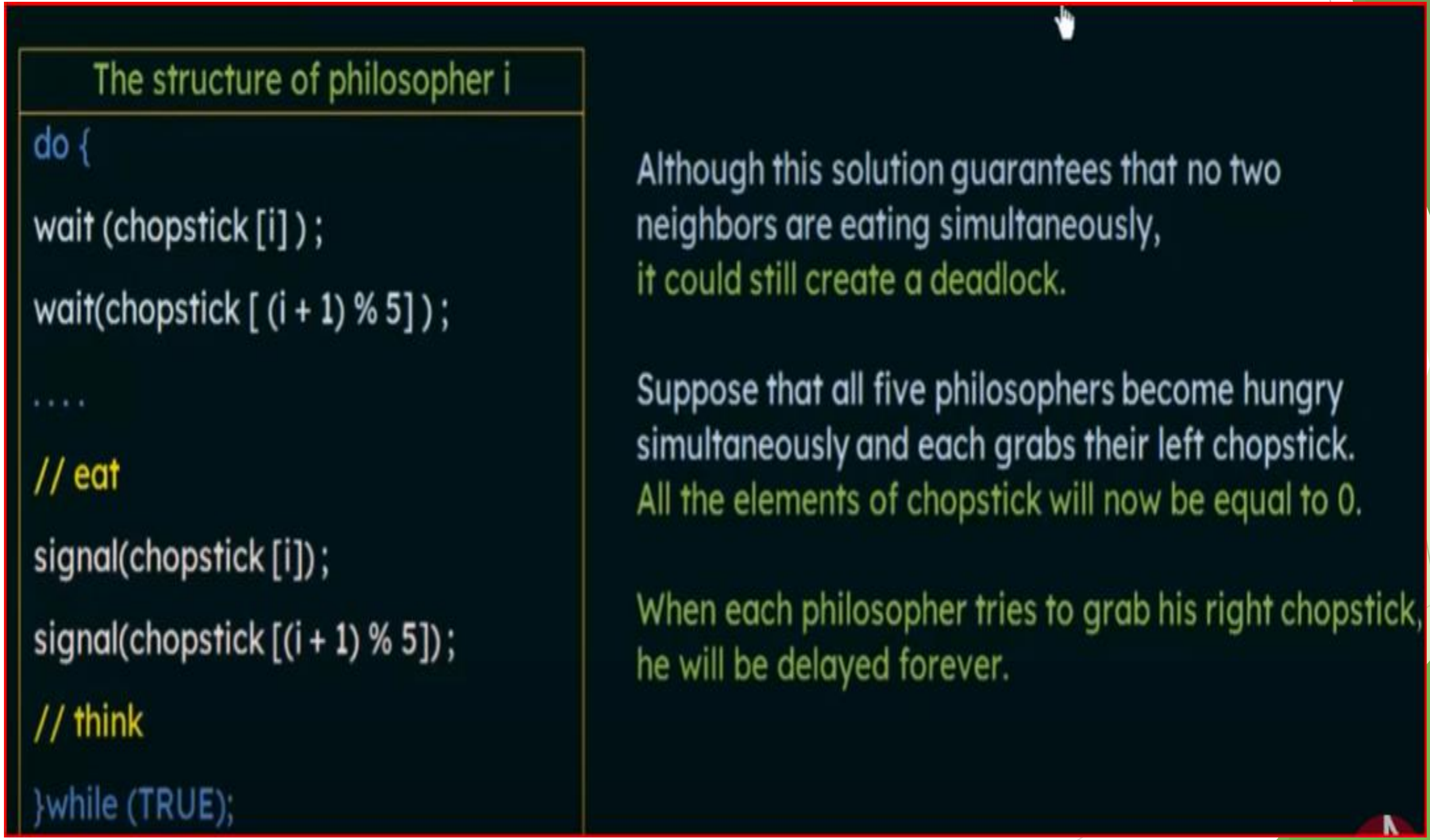
1. Five philosophers are sitting at a rounded dining table.
2. There is one chopstick between each pair of adjacent philosophers.
3. Philosophers are either thinking or eating.
4. Whenever a philosopher wishes to eat, he/she first needs to find two chopsticks.
5. If the hungry philosopher does not have two chopsticks (i.e. one or two of her neighbours already picked up the chopstick) she will have to wait until both chopsticks are available.
6. When a philosopher finishes eating, she puts down both chopsticks to their original places, and resumes thinking.
7. There is an infinite amount of food on their plate, so they only need to worry about the chopsticks.

- **There are a few conditions:**

1. Philosophers are either thinking or eating. They do not talk to each other.
2. Philosophers can only fetch chopsticks placed between them and their neighbours.
3. Philosophers cannot take their neighbours' chopsticks away while they are eating.
4. Hopefully no philosophers should starve to death (i.e. wait over a certain amount of time before he/she acquires both chopsticks).

SCHOOL OF ENGINEERING

Explanation:



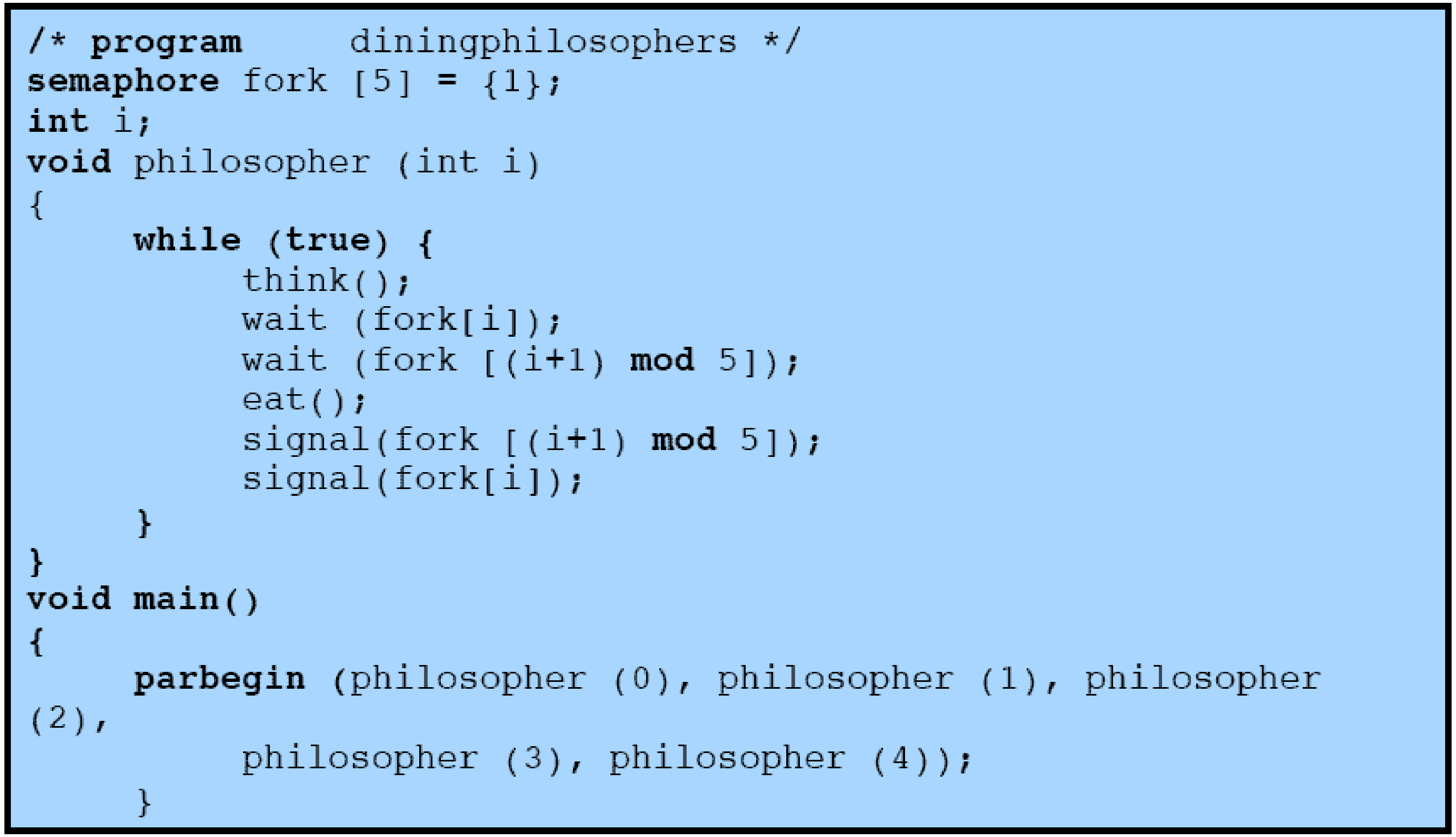
The structure of philosopher i

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

Although this solution guarantees that no two neighbors are eating simultaneously, it could still create a deadlock.

Suppose that all five philosophers become hungry simultaneously and each grabs their left chopstick. All the elements of chopstick will now be equal to 0.

When each philosopher tries to grab his right chopstick, he will be delayed forever.



```
/* program     diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
     while (true) {
          think();
          wait (fork[i]);
          wait (fork [(i+1) mod 5]);
          eat();
          signal(fork [(i+1) mod 5]);
          signal(fork[i]);
     }
}
void main()
{
     parbegin (philosopher (0), philosopher (1), philosopher
(2),
          philosopher (3), philosopher (4));
     }
```

Figure 6.12 **A First Solution to the Dining Philosophers Problem**

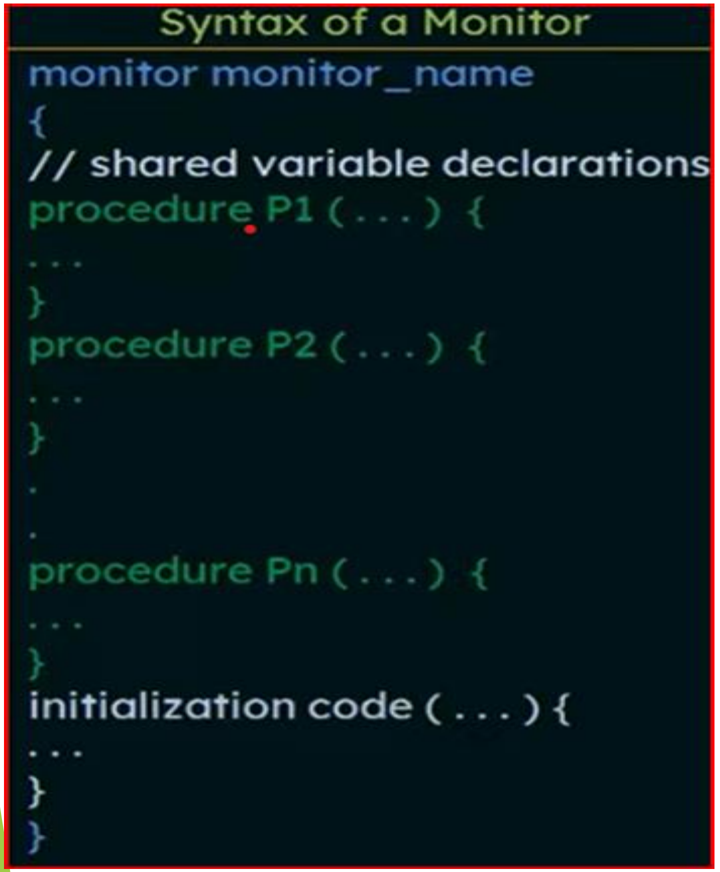
Some possible remedies to avoid deadlocks

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up his chopsticks only if both chopsticks are available.
- Use an asymmetric solution; that is, an odd philosopher picks up first his left chopsticks, whereas an even philosopher picks up his right chopstick and then his left chopstick.

Monitors

- High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.
- The Monitor type contains shared variables and the set of procedures that operate on the shared variable.
- When any process wishes to access the shared variables in the monitor, it needs to access it through the procedures.
- These processes line up in a queue and are only provided access when the previous process release the shared variables.
- Only one process can be active in a monitor at a time.
- Monitor has condition variables.

Explanation

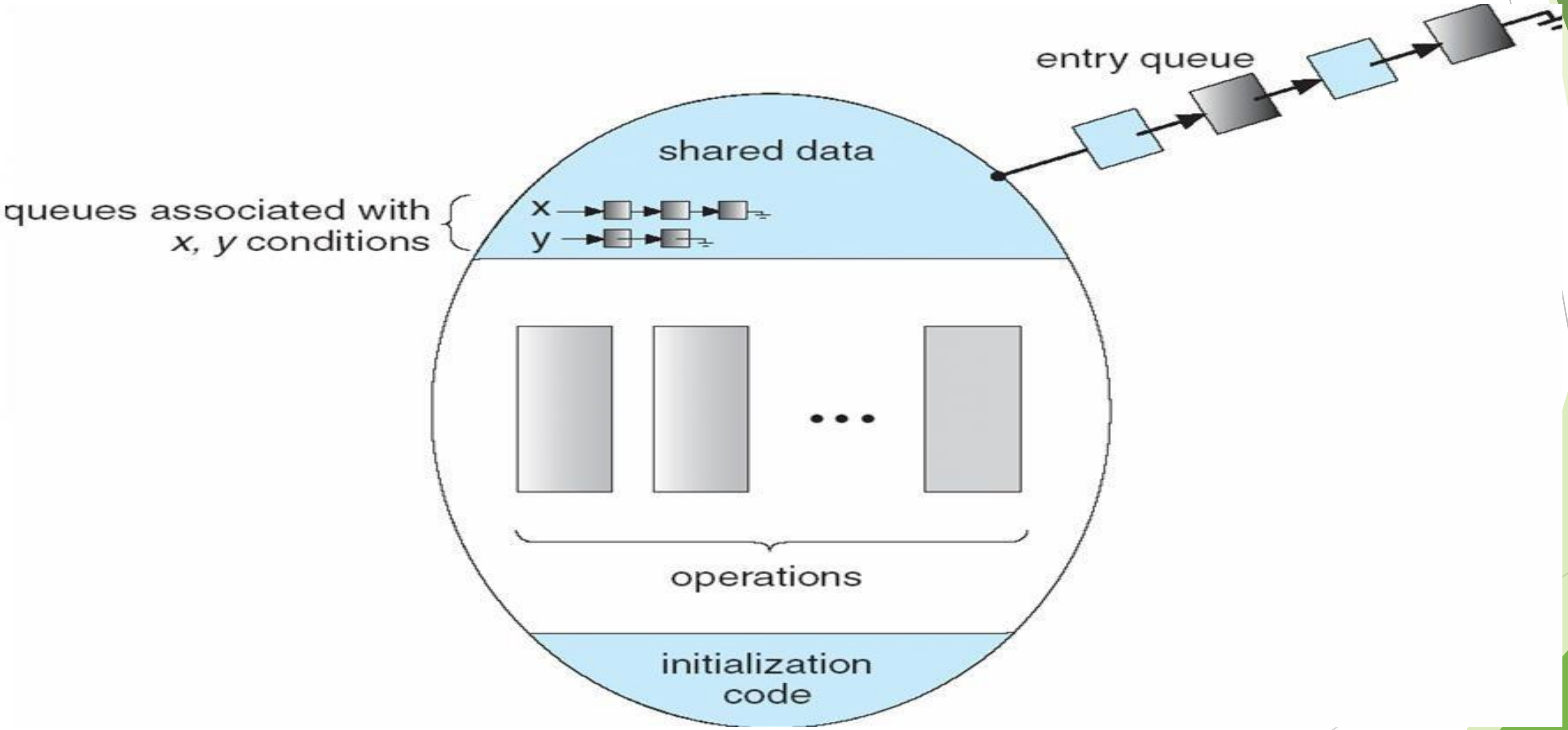


```
Syntax of a Monitor
monitor monitor_name
{
// shared variable declarations
procedure P1 (...) {
...
}
procedure P2 (...) {
...
}
.
.
procedure Pn (...) {
...
}
initialization code (...){
...
}
}
```

- A Procedure defined within a monitor can access only those variables declared locally within a monitor and its formal parameters.
- The local variables of a monitor can be accessed by only the local procedures.
- The monitor construct ensures that only one process at a time can be active within the monitor.
- Condition Construct: condition x,y; The operation that can be invoked on a condition variable are 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.

SCHOOL OF ENGINEERING

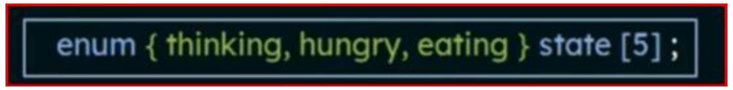
Monitor with condition variables



Schematic view of a Monitor

Dining-Philosophers Solution Using Monitors

- This solution imposes the restriction that a philosopher may pick up his chopsticks only if both of them are available.
- To solve this problem, we need to distinguish among three states in which we may find a philosopher. For this purpose, we are following the data structure.

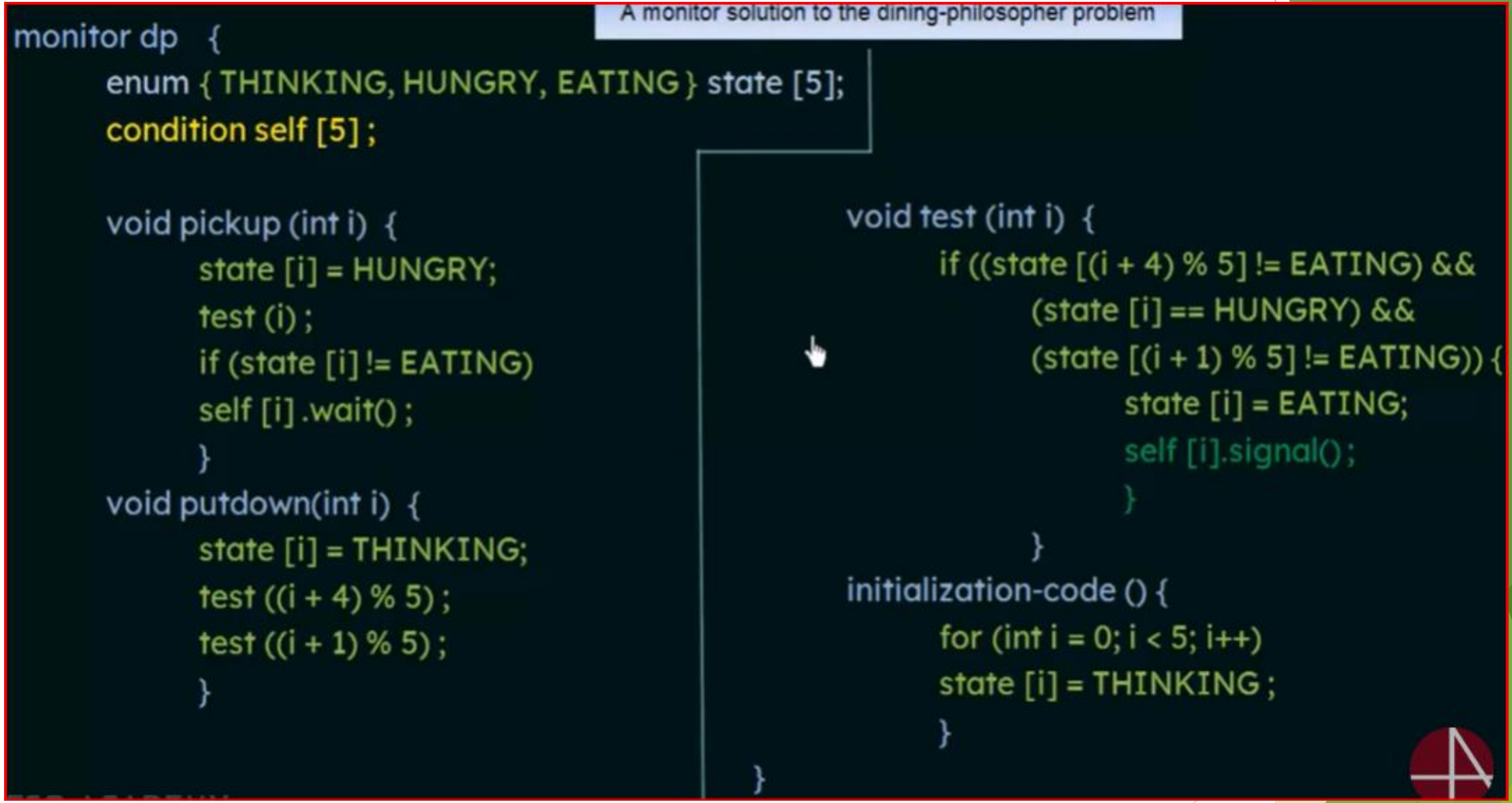


```
enum { thinking, hungry, eating } state [5] ;
```

- Philosopher i can set the variable **state[i] = eating** only if his two neighbors are not eating: **(state[(i+4)%5]!=eating) and (state[(i+1)%5]!=eating).**

- We also need to declare **condition self[5];** where philosopher i can delay himself when he is hungry but is unable to obtain the chopsticks he need.

SCHOOL OF ENGINEERING



A monitor solution to the dining-philosopher problem

```
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 solution to the dining-philosopher problem

Deadlocks

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

System Model

- A system consists of a finite number of resources to be distributed among a number of competing processes.
- Under the normal mode of operation, a process may utilize a resource in only
- the following sequence:

1. **Request:** The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
2. **Use:** The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
3. **Release**: The process releases the resource.

- **System calls:** request() and release() device, open() and close() file, and allocate() and free() memory system calls. wait() and signal() operations on semaphores or through acquire() and release() of a mutex lock

Deadlock

- **Deadlock:** If two or more processes are waiting on happening of some event, which never happens, then we say these processes are involved in deadlock.

- A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a **Deadlock**.
- Deadlocks are a set of blocked processes each holding a resource and waiting to acquire a resource held by another process.

The Deadlock Problem

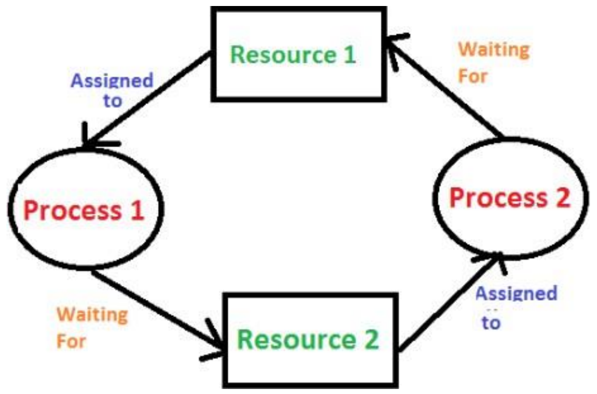
- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Example
  - System has 2 disk drives.
  - P_1 and P_2 each hold one disk drive and each needs another one.

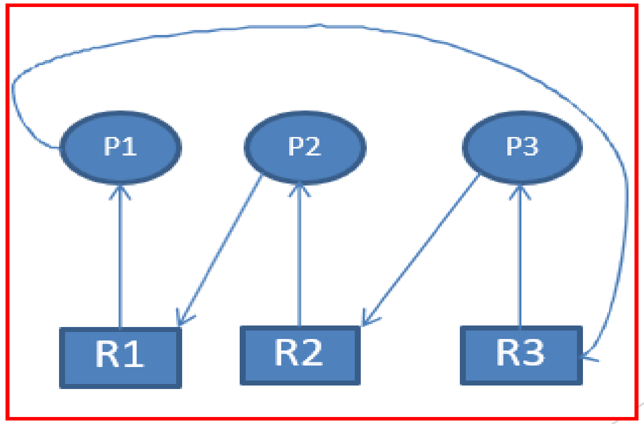
Deadlock characterization:

Necessary Conditions

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

- ***Mutual Exclusion:*** At least one resource must be held in a *nonsharable mode*; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
- ***Hold and Wait:*** A process is holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
- ***No Preemption:*** A resource cannot be taken from a process unless the process releases the resource.
- ***Circular Wait:*** A set of processes are waiting for each other in circular form.





Possibility of Deadlock

These three conditions are necessary but not sufficient for deadlock.

- Mutual Exclusion
- No preemption
- Hold and wait

Resource-Allocation Graph (RAG)

- In our system, how the resources are allocated to the processes and how the processes have been assigned multiple resources are represented using RAG.
- Deadlocks can be described more precisely in terms of a directed graph Whether there is a deadlock is there or not – RAG is used
- P = {P1, P2, ..., Pn}, the set consisting of all the active processes in the system, and R = {R1, R2, ..., Rm}, the set consisting of all resource types in the system.
- A directed edge Pi → Rj is called a **request edge**; a directed edge Rj → Pi is called an **assignment edge**.
- we represent each **process** Pi as a **circle** and each **resource** type Rj as a **rectangle**.

If the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

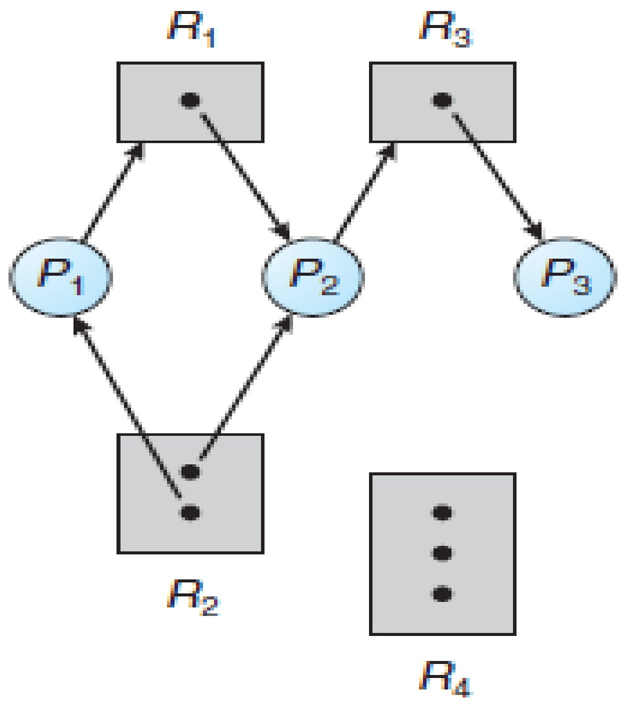
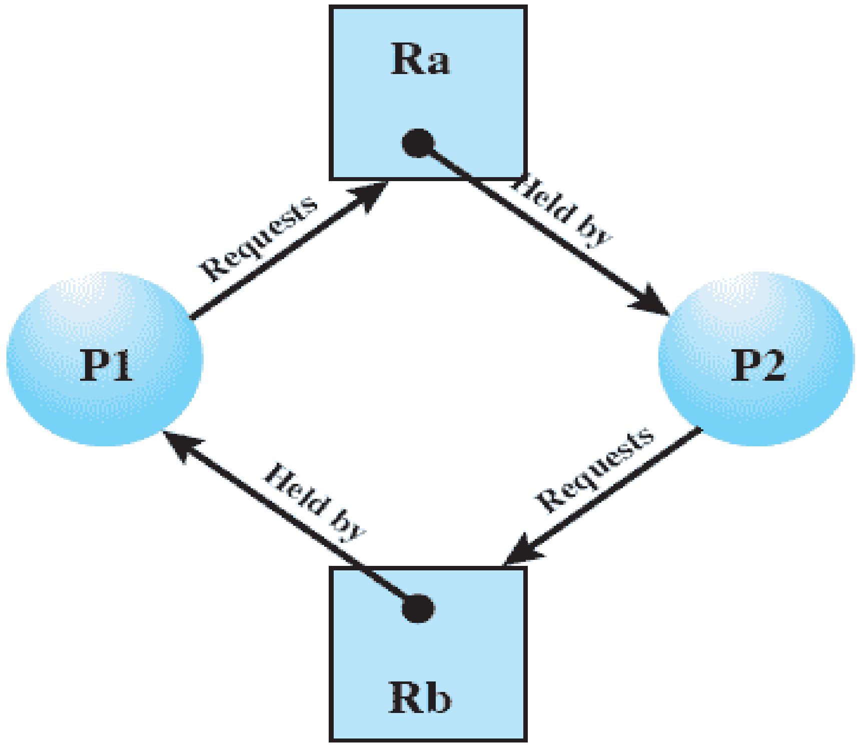


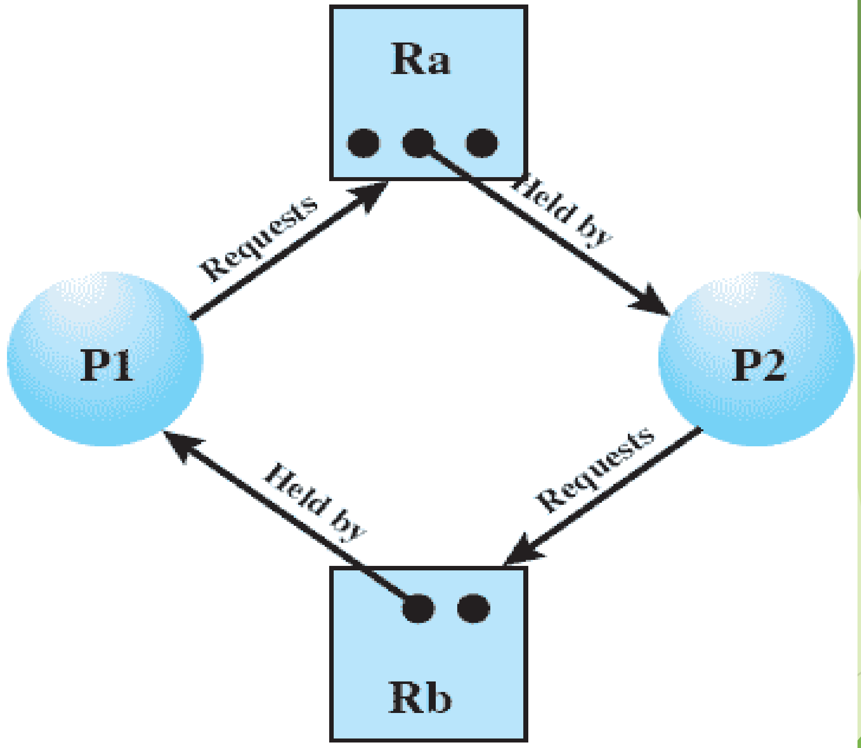
Figure 7.1 Resource-allocation graph.

Process: P1, P2, P3
Resources: R1, R2,R3,R4
R1- 1 instance
R2-2 instance
R3-1 instance
R4- 3 instance

Resource Allocation Graphs



(c) Circular wait



(d) No deadlock

SCHOOL OF ENGINEERING

Resource Allocation Graphs

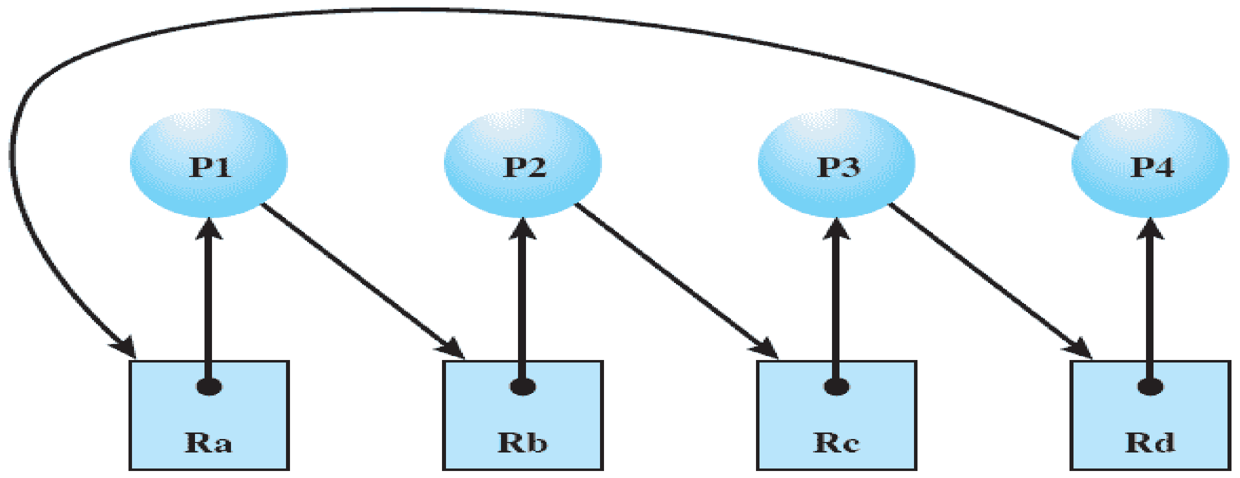
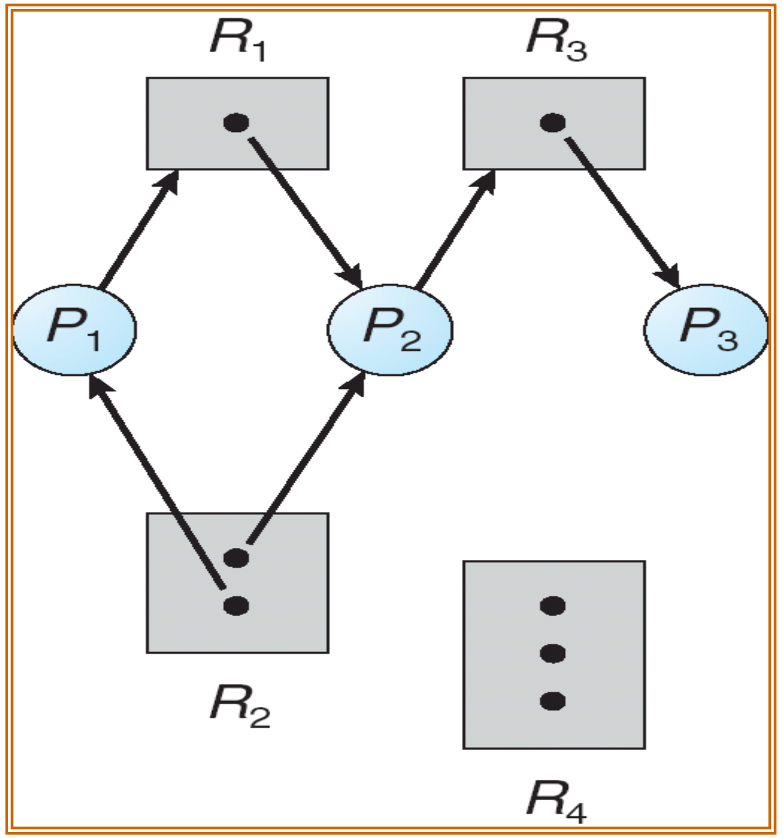


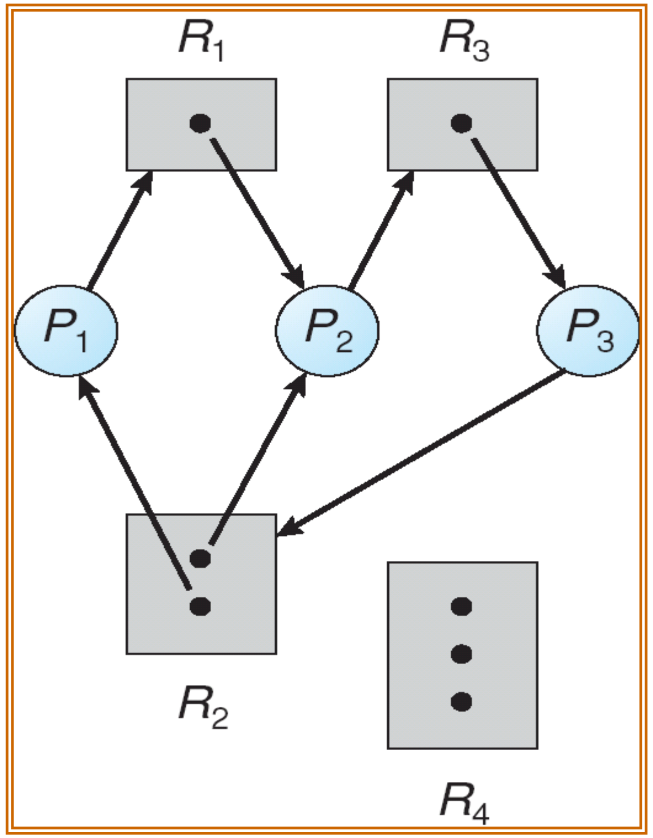
Figure 6.6 Resource Allocation Graph for Figure 6.1b

SCHOOL OF ENGINEERING

Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Methods for Handling Deadlocks

- Deadlock prevention: Prevent deadlock by eliminating one of the deadlock condition.
- Deadlock Avoidance : Ensure that the system will *never* enter a deadlock state.
- Deadlock detection: Allow the system to enter a deadlock state and then recover.

1. Deadlock Prevention

Design a system in such a way that the possibility of deadlock is excluded.

- Techniques followed to prevent conditions

1. Mutual Exclusion
   - For Mutual exclusion condition - at least one resource must be non- sharable.
   - By allowing resource sharing between the processes we can make mutual exclusion false.

   ex: Read-only files are a good example of a sharable resource.
   - We cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable.
   - ex: Printer

2. 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.

Disadvantage:

- **Resource utilization may be low**: since resources may be allocated but unused for a long period.
- **Starvation is possible**: A process that needs several popular resources may have to wait indefinitely.

3. 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.

4. Circular Wait -

Define a linear ordering of resource types
Associate index with each resources
Resource Ri precedes Rj in the ordering i<j
Circular wait prevention is also inefficient
# slowing down processes
# denying resource access unnecessarily

Ex: R1, R2, R3, R4, R5, R6

Deadlock Avoidance

- A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock
- Allows the three necessary conditions.
    - Mutual Exclusion
    - Hold and Wait
    - No Preemption
- Assures that deadlock point never reached.
- Do not start a process if its demands might lead to deadlock
- Requires knowledge of future process resource requests.

Deadlock Avoidance

- Safe State
- Resource-Allocation-Graph Algorithm
- Banker’s Algorithm

Deadlock Avoidance

- **Safe State**

- A state is safe if the system can allocate resources to each process (up to its maximum) in **some order** and still avoid a deadlock

- A system is in a **safe state** only if there exists a **safe sequence**.

Safe State

- A state is safe if the system can allocate resources to each process (up to its maximum) in **some order** and still avoid a deadlock
- A system is in a **safe state** only if there exists a **safe sequence**.

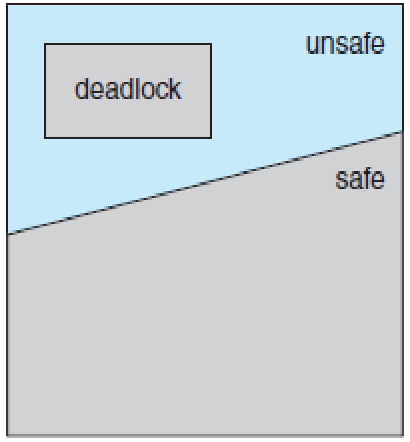


Figure 7.6 Safe, unsafe, and deadlocked state spaces.

- A safe state (safe sequence) is not a deadlocked state.
- A deadlocked state is an unsafe state (no safe sequence).
- Not all unsafe states are deadlocks, however, an unsafe state may lead to a deadlock.
- As long as the state is safe, the operating system can avoid unsafe (and deadlocked) states.

12 magnetic tape drives, 3 processes P0,P1,P2

	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

Initially, System is safe, Safe Sequence <P1, P0, P2>

12 magnetic tape drives, 4 processes P1,P2,P3,P4

Process	Max. Need	Current Allocation
P1	8	3
P2	10	4
P3	5	2
P4	3	1

Safe Sequence < P4, P3, P1, P2 >

Deadlock Avoidance

Disadvantages:

- Maximum resource requirement must be stated in advance.
- Processes under consideration must be independent, no synchronization requirements.
- There must be a fixed number of resources to allocate.
- Claim matrix needs to be known in advance.
- It is still conservative because the safe state calculation assumes that the process will come for all the remaining requests at the same time.

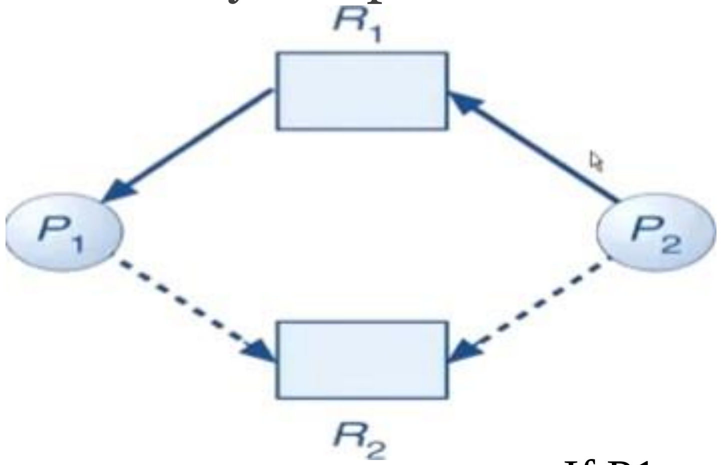
Resource-Allocation-Graph Algorithm

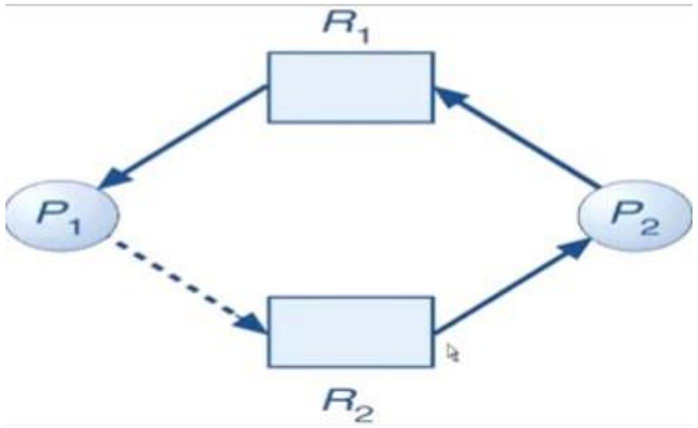
- Deadlock avoidance – single instance of a resource
- In addition to the request and assignment edges, a new type of edge, called a **claim edge**.
- Claim edge *Pi* → *Rj* indicated that process *Pi* may request resource *Rj;* represented by **dashed line**.
- **All claim edges** must appear in the graph, **initiall**y.
- Claim edge *Pi* → *Rj* converts to request edge when a process requests a resource
- When a resource is released by a process, assignmentedge *Rj*→ *Pi*

reconverts to a claim edge *Pi* → *Rj*.

Deadlock Avoidance

- Suppose *Pi* request *Rj*
- Request edge *Pi* → *Rj* is converted to assignment edge *Rj*→ *Pi*
- Graph checked for cycles using **cycle detection algorithm**
- If no cycle – system is in safe state, resource can be allocated to *Pi*.
- If cycle –process has to wait.

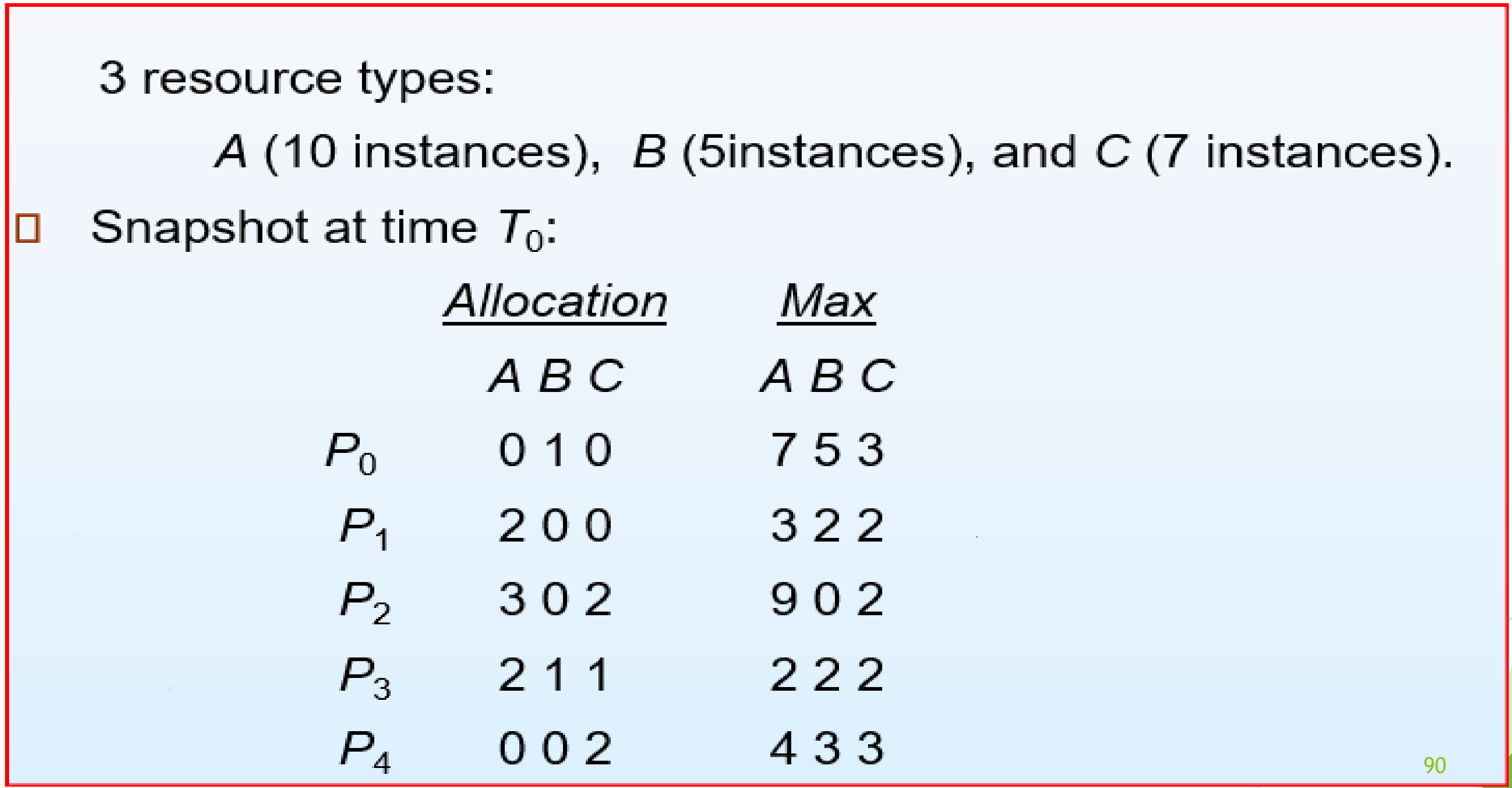




If P1 requests R2, and P2 requests R1, then deadlock will occur.

SCHOOL OF ENGINEERING

By applying Banker's Algorithm, check whether the given system is in safe state ? If yes write the sequence of process that complete execution.

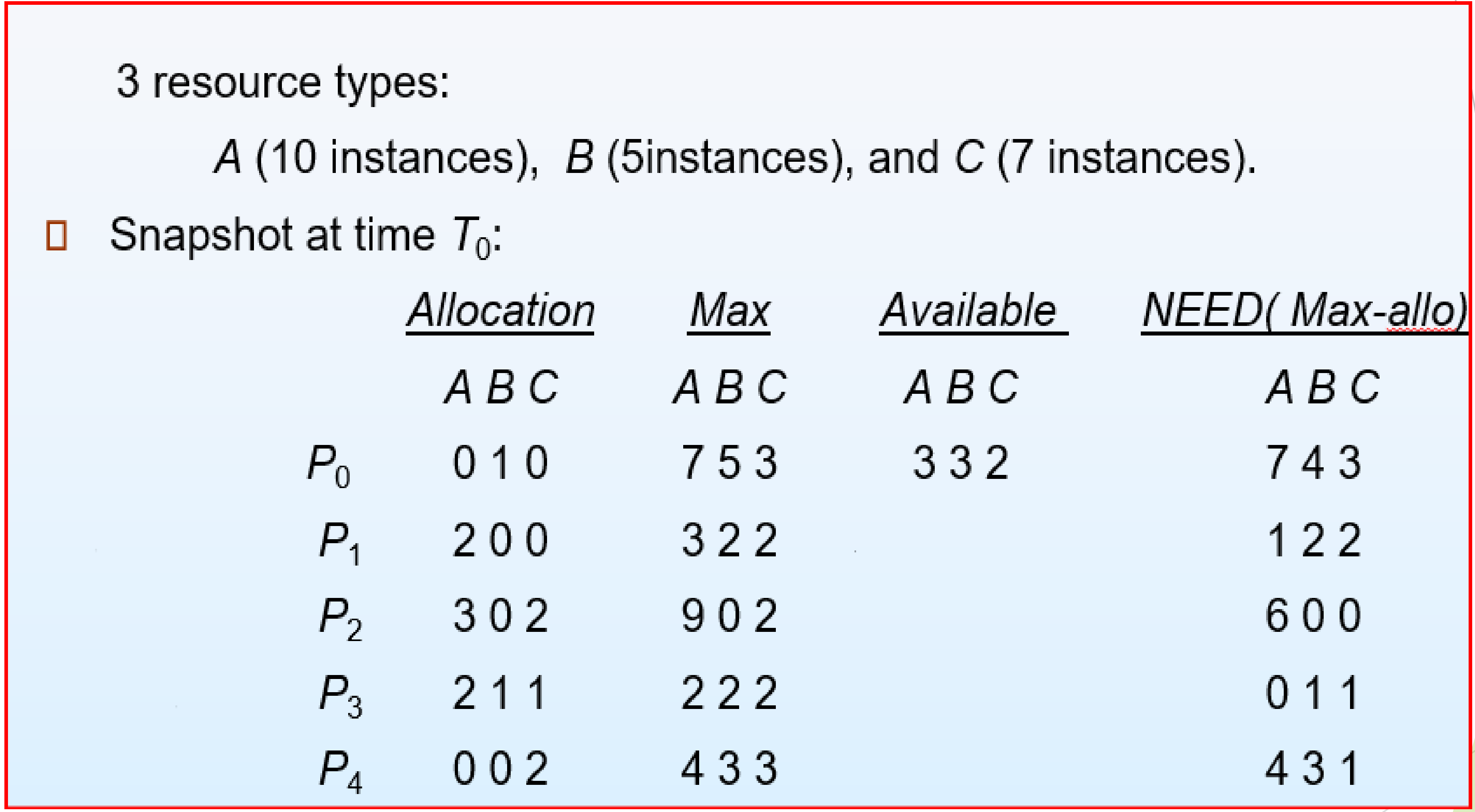


3 resource types:

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

- Snapshot at time T_0:

	Allocation	Max
	A B C	*A B C*
P_0	0 1 0	7 5 3
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



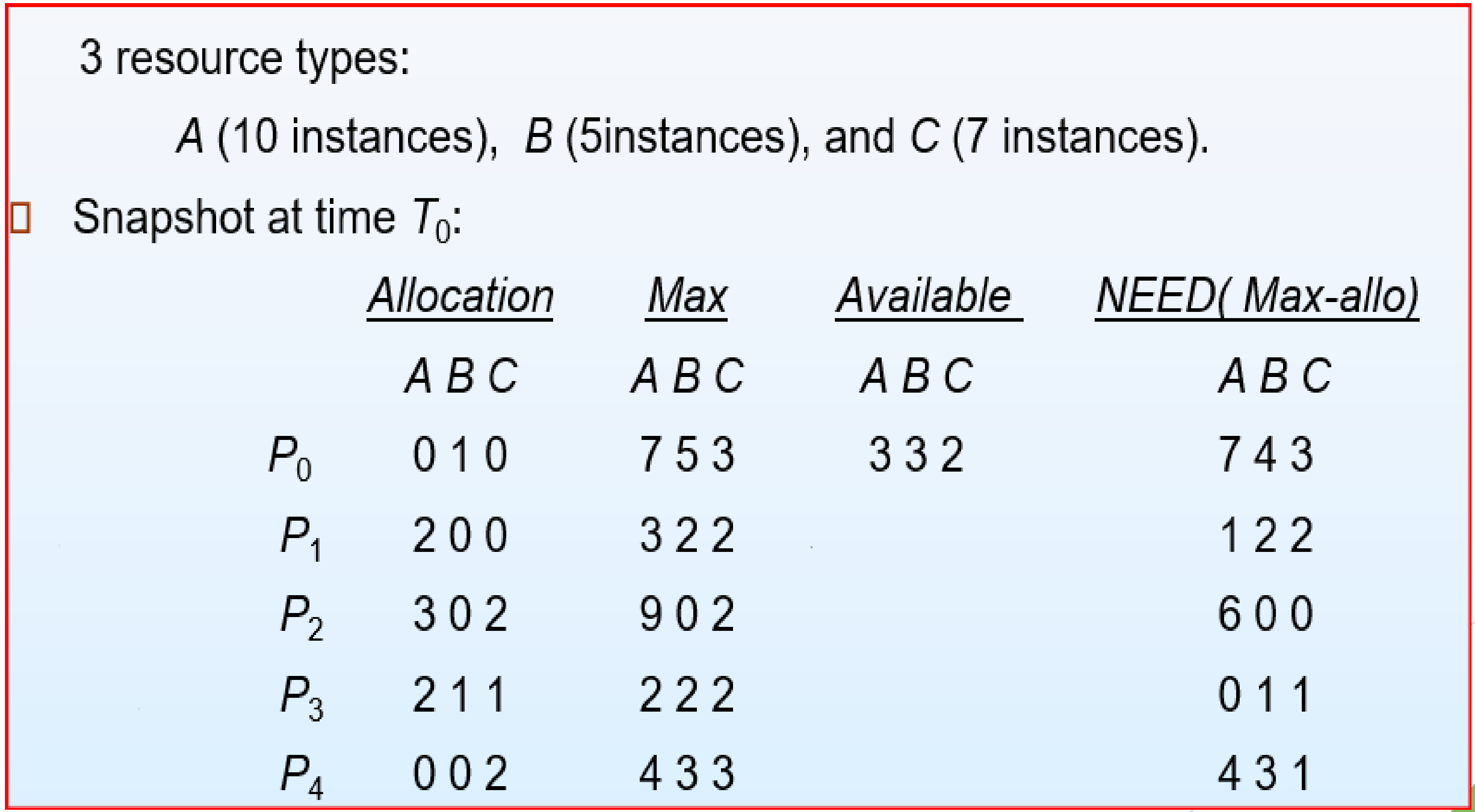
3 resource types:

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

- Snapshot at time T_0:

	Allocation	*Max*	*Available*	*NEED(Max-allo)*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

By applying Banker's Algorithm, check whether the given system is in safe state ? If yes write the sequence of process that complete execution.



3 resource types:

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

- Snapshot at time T_0:

	Allocation	*Max*	*Available*	*NEED(Max-allo)*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

By applying Banker’s Algorithm, check whether the given system is in safe state ? If yes write the sequence of process that complete execution.

P1 goes for completion

	Allocation	Max	Available	NEED(Max-allo)
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	5 3 2	7 4 3
P_1	0 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

By applying Banker's Algorithm, check whether the given system is in safe state ? If yes write the sequence of process that complete execution.

P3 goes for completion

	Allocation	Max	Available	NEED(Max-allo)
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	7 4 3	7 4 3
P_1	0 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	0 0 0	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

By applying Banker's Algorithm, check whether the given system is in safe state ? If yes write the sequence of process that complete execution.

P4 goes for completion

	Allocation	Max	Available	NEED(Max-allo)
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	7 4 5	7 4 3
P_1	0 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	0 0 0	2 2 2		0 1 1
P_4	0 0 0	4 3 3		4 3 1

SCHOOL OF ENGINEERING

By applying Banker’s Algorithm, check whether the given system is in safe state ? If yes write the sequence of process that complete execution.

P2 goes for completion

	Allocation	Max	Available	NEED(Max-allo)
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	10 4 7	7 4 3
P_1	0 0 0	3 2 2		1 2 2
P_2	0 0 0	9 0 2		6 0 0
P_3	0 0 0	2 2 2		0 1 1
P_4	0 0 0	4 3 3		4 3 1

SCHOOL OF ENGINEERING

By applying Banker's Algorithm, check whether the given system is in safe state ? If yes write the sequence of process that complete execution.

P0 goes for completion

	Allocation	Max	Available	NEED(Max-allo)
	A B C	A B C	A B C	A B C
P_0	0 0 0	7 5 3	10 5 7	7 4 3
P_1	0 0 0	3 2 2		1 2 2
P_2	0 0 0	9 0 2		6 0 0
P_3	0 0 0	2 2 2		0 1 1
P_4	0 0 0	4 3 3		4 3 1

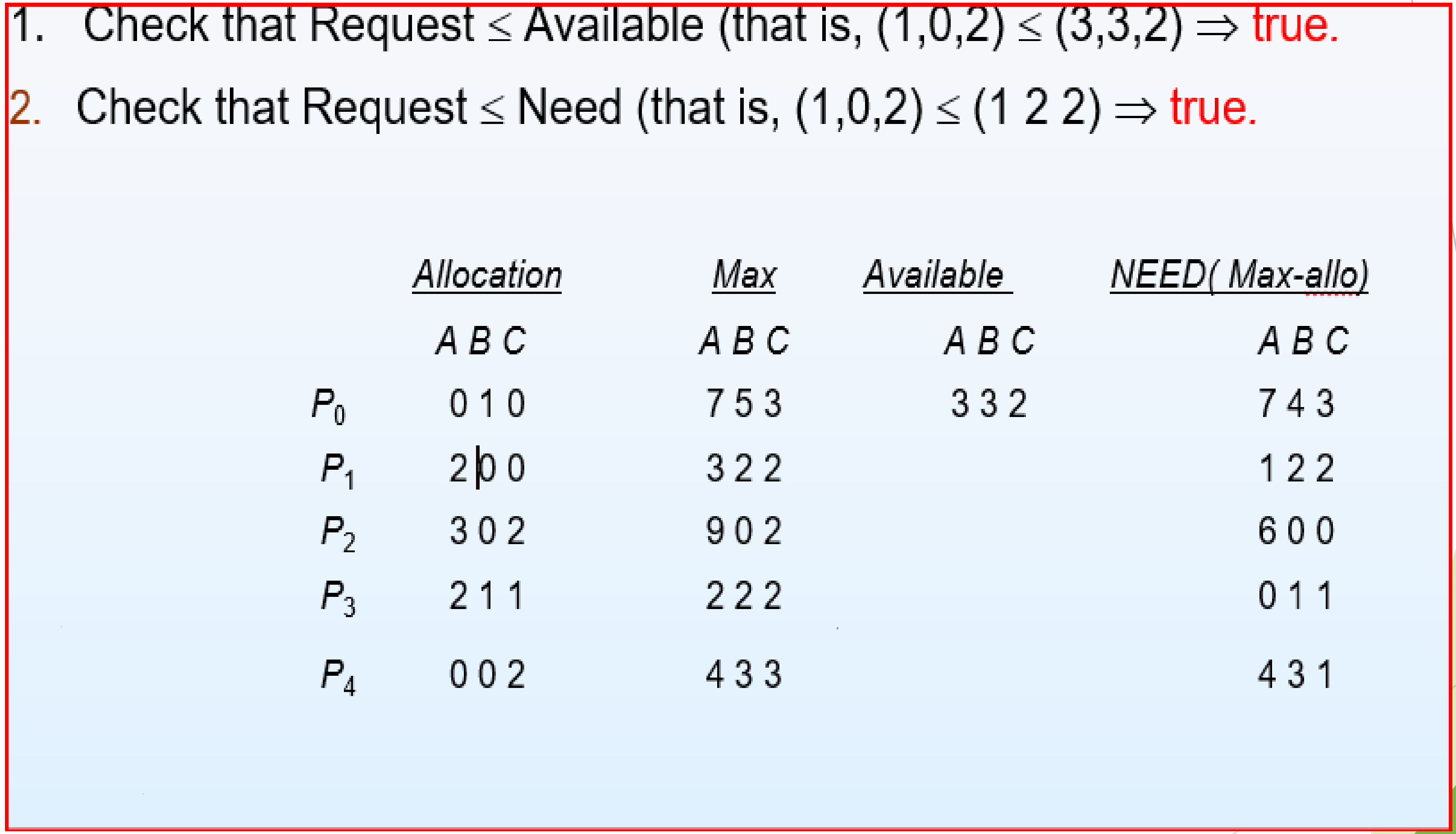
SCHOOL OF ENGINEERING

Example (Cont.)

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

SCHOOL OF ENGINEERING

Example: P_1 Request (1,0,2)



1. Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true.
2. Check that Request ≤ Need (that is, (1,0,2) ≤ (1 2 2) ⇒ true.

	Allocation	*Max*	*Available*	*NEED(Max-allo)*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

SCHOOL OF ENGINEERING

Example: P_1 Request (1,0,2)

1. Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true.
2. Check that Request ≤ Need (that is, (1,0,2) ≤ (3 2 2) ⇒ true.

	Allocation	*Max*	*Available*	*NEED(Max-allo)*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	2 3 0	7 4 3
P_1	3 0 2	3 2 2		0 2 0
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

Example: P_1 Request (1,0,2)

Executing safety algorithm shows that sequence < P1, P3, P4, P0, P2> satisfies safety requirement.

Deadlock Detection

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

Deadlock Detection Algorithm

Initially all processes are unmarked.

Then following steps are followed:

Step 1: Mark each process that has a row in the allocation matrix of all ZERO.

Step 2: Initialize a temporary vector **W** to equal to the Available vector.

Step 3: Find Index i such that process i is currently unmarked and ith row of Q <=W

$Q_{ik} <= W_k$ for $1<=k<=m$

If no such row is found , terminate the algorithm

Deadlock Detection Algorithm

Step 4: if such row is found , mark process i and add the corresponding row of the allocation matrix to W.

That is

set $W_k = W_k + A_{ik}$ for $1 \le k \le m$.

Return to step 3

Conclusion: If all process are marked, they the system is in safe state(No Deadlock)

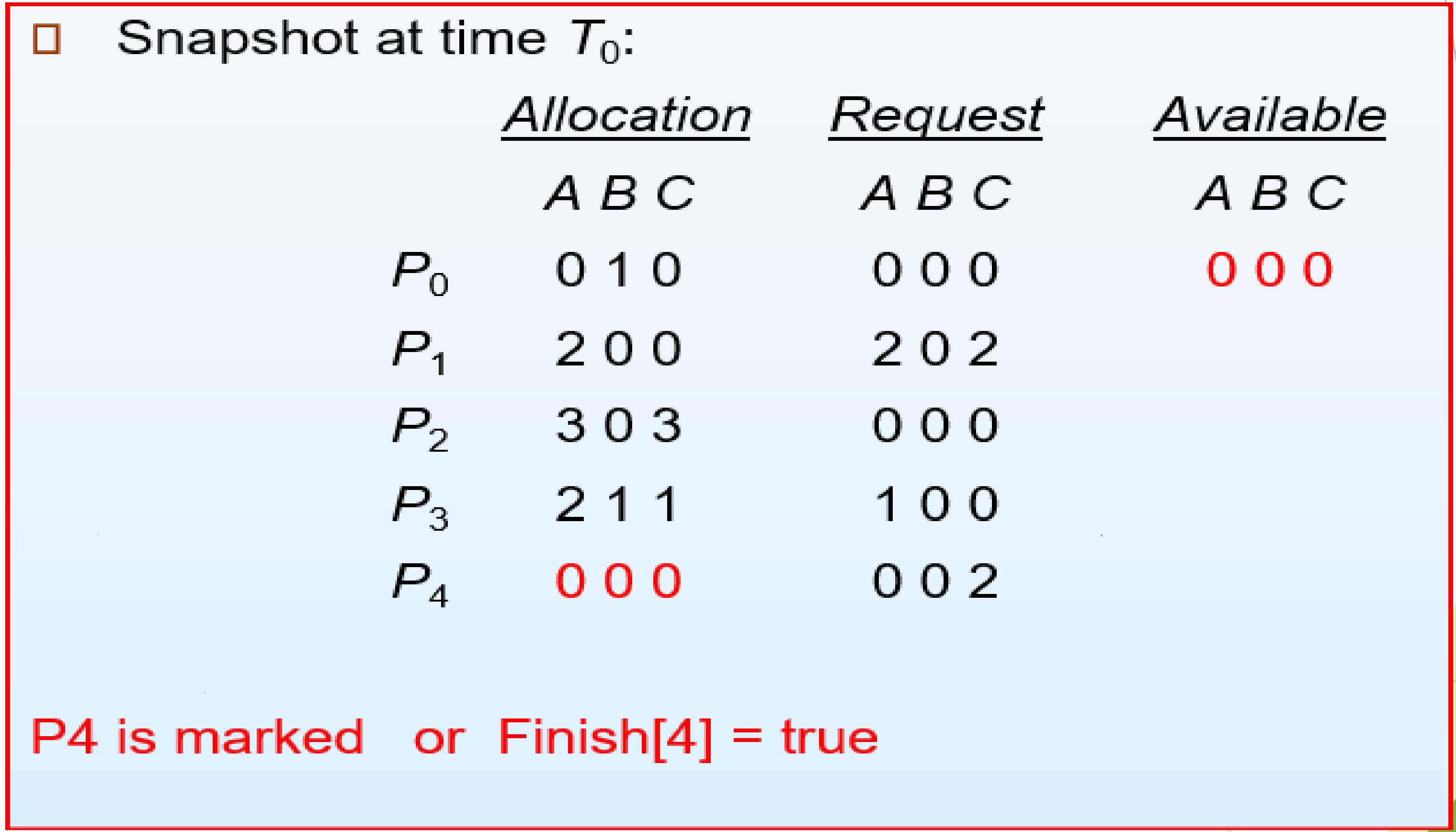
Example of Detection Algorithm

With given Data, check weather system is in safe state or not by applying Deadlock 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 0	0 0 2	

Example of Detection Algorithm



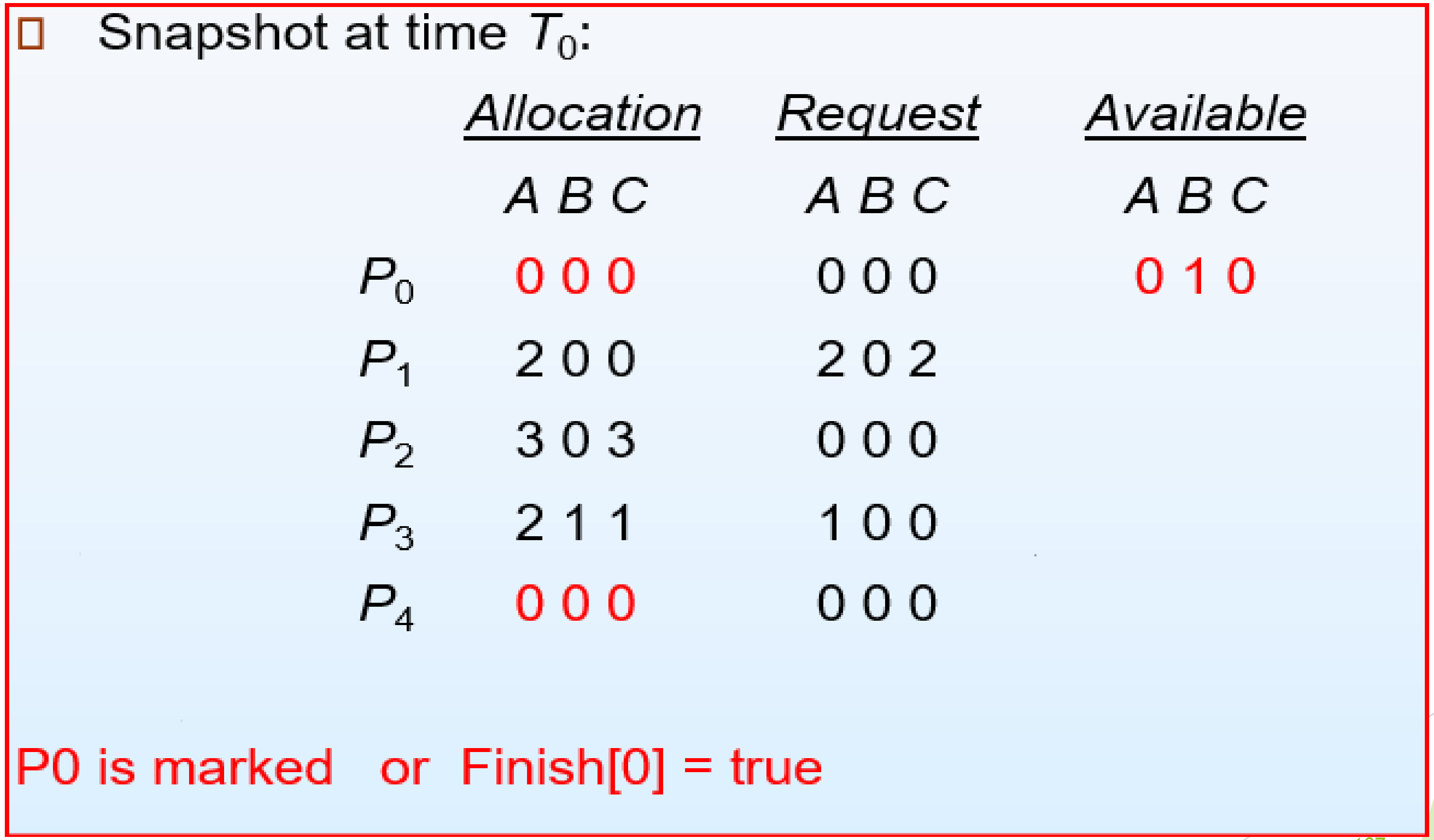
- Snapshot at time T_0:

	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 0	0 0 2	

P4 is marked or Finish[4] = true

SCHOOL OF ENGINEERING

Example of Detection Algorithm



- Snapshot at time T_0:

	Allocation	*Request*	*Available*
	A B C	A B C	A B C
P_0	0 0 0	0 0 0	0 1 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 0	0 0 0	

P0 is marked or Finish[0] = true

SCHOOL OF ENGINEERING

Example of Detection Algorithm

- Snapshot at time T_0:

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

P2 is marked or Finish[2] = true

SCHOOL OF ENGINEERING

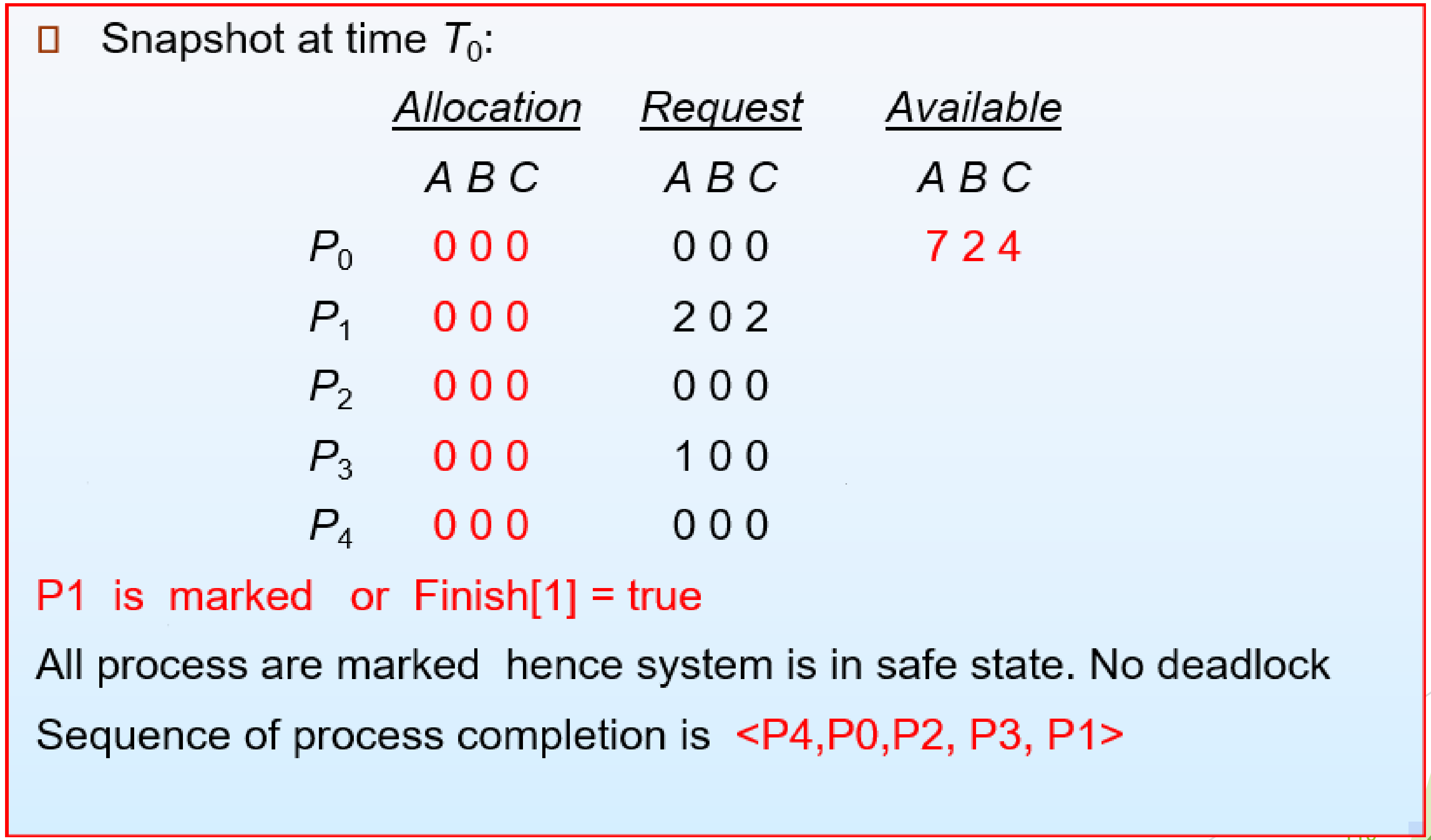
Example of Detection Algorithm

- Snapshot at time T_0:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 0 0	0 0 0	5 2 4
P_1	2 0 0	2 0 2	
P_2	0 0 0	0 0 0	
P_3	0 0 0	1 0 0	
P_4	0 0 0	0 0 0	

P3 is marked or Finish[3] = true

Example of Detection Algorithm



- Snapshot at time T_0:

	Allocation	*Request*	*Available*
	A B C	A B C	A B C
P_0	0 0 0	0 0 0	7 2 4
P_1	0 0 0	2 0 2	
P_2	0 0 0	0 0 0	
P_3	0 0 0	1 0 0	
P_4	0 0 0	0 0 0	

P1 is marked or Finish[1] = true

All process are marked hence system is in safe state. No deadlock

Sequence of process completion is <P4,P0,P2, P3, P1>

Write Solution as written below:

Step 1: Mark P4, because P4 has no allocated resources

Step 2: Set available vector W=(0 0 0)

Step 3: Find a process that is unmarked and whose request is less than or equal to available

Step 4: a) Request of P0 is less than or equal to W, So mark P0 and set W=W+(0 1 0)= (0 0 0) + (0 1 0) = (0 1 0)

b) Request of P2 is less than or equal to W, So mark P2 and set

W= (0 1 0) + (3 0 3) = (3 1 3)

c) Request of P3 is less than or equal to W, So mark P3 and set

W= (3 1 3) + (2 1 1) = (5 2 4)

d) Request of P1 is less than or equal to W, So mark P1 and set

W= (5 2 4) + (2 0 0) = (7 2 4)

All process are marked and hence System is in safe state, no deadlock

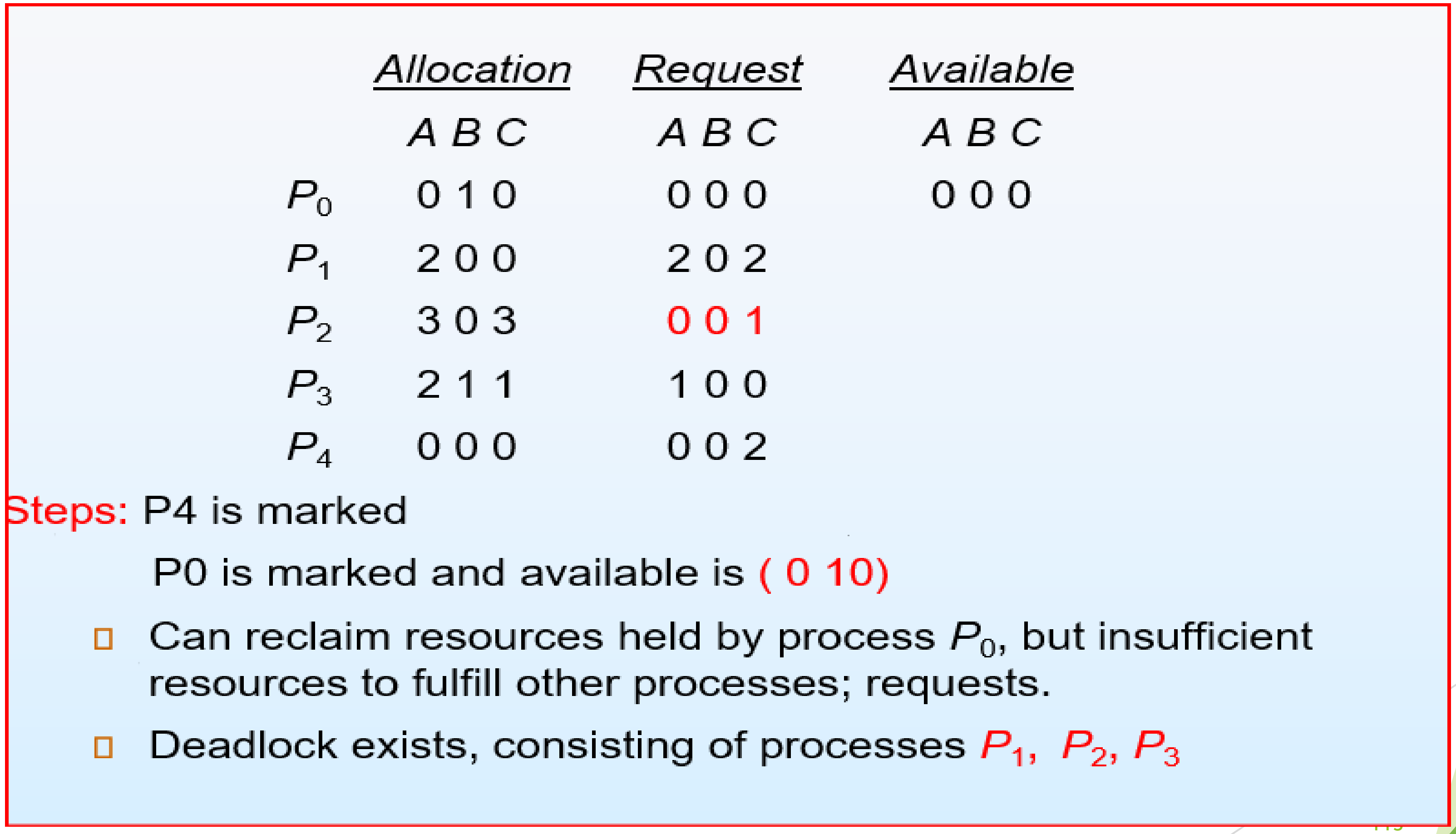
Example (Cont.)

- 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 0

- State of system?

Example of Detection Algorithm



	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 1	
P_3	2 1 1	1 0 0	
P_4	0 0 0	0 0 2	

Steps: P4 is marked

P0 is marked and available is (0 10)

- 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

Detection-Algorithm Usage

- If deadlock occur frequently – invoke detection algorithm frequently
- Invoke a detection algorithm every time a request for allocation cannot be granted immediately.
- Invoke the detection algorithm at less frequent intervals
  - Once per hour
  - CPU utilization falls below 40 percent
  - There may be many cycles in the graph
  - May not be able to tell which process caused the deadlock

Recovery from Deadlock

- Inform the operator that a deadlock has occurred
  - Operator deals with the deadlock manually.
- Let the system recover from the deadlock automatically.
- Break the deadlock
  - **abort** one or more processes to break the circular wait
  - **preempt** some resources from one or more of the deadlocked processes

Recovery from Deadlock

- **To eliminate deadlocks by aborting a process:**

- **Abort all deadlocked processes**
  - will break the deadlock cycle
  - Results of these partial computations must be discarded and recomputed later

- **Abort one process at a time until the deadlock cycle is eliminated**
  - considerable overhead
  - After each process is aborted, a deadlock-detection algorithm must be invoked

To eliminate deadlocks by aborting a process:

In which order should we choose a process to abort?

- Priority of the process (least priority first)
- Resources the process has used (many resources used–abort)
- How many more resources the process needs in order to complete (many resources yet required – abort)
- How long the process has computed and how much longer to completion.
- How many processes will need to be terminated.

Resource Preemption

- **Selecting a victim**
  - Order of preemption to minimize cost
  - Cost factors may include
    - number of resources a deadlocked process is holding
    - amount of time a deadlocked process has thus far consumed during its execution.
- **Starvation**
  - Same process may always be picked as victim (low priority process)
  - Include number of rollbacks in cost factor
- **Rollback**
  - If we preempt a resource from a process, what should be done with that process?
  - cannot continue, missing some needed resource
  - Rollback to some safe state, restart process for that state
  - Difficult to determine a safe state – total rollback – abort the process and restart.

SCHOOL OF
ENGINEERING

Advantages and Disadvantages:

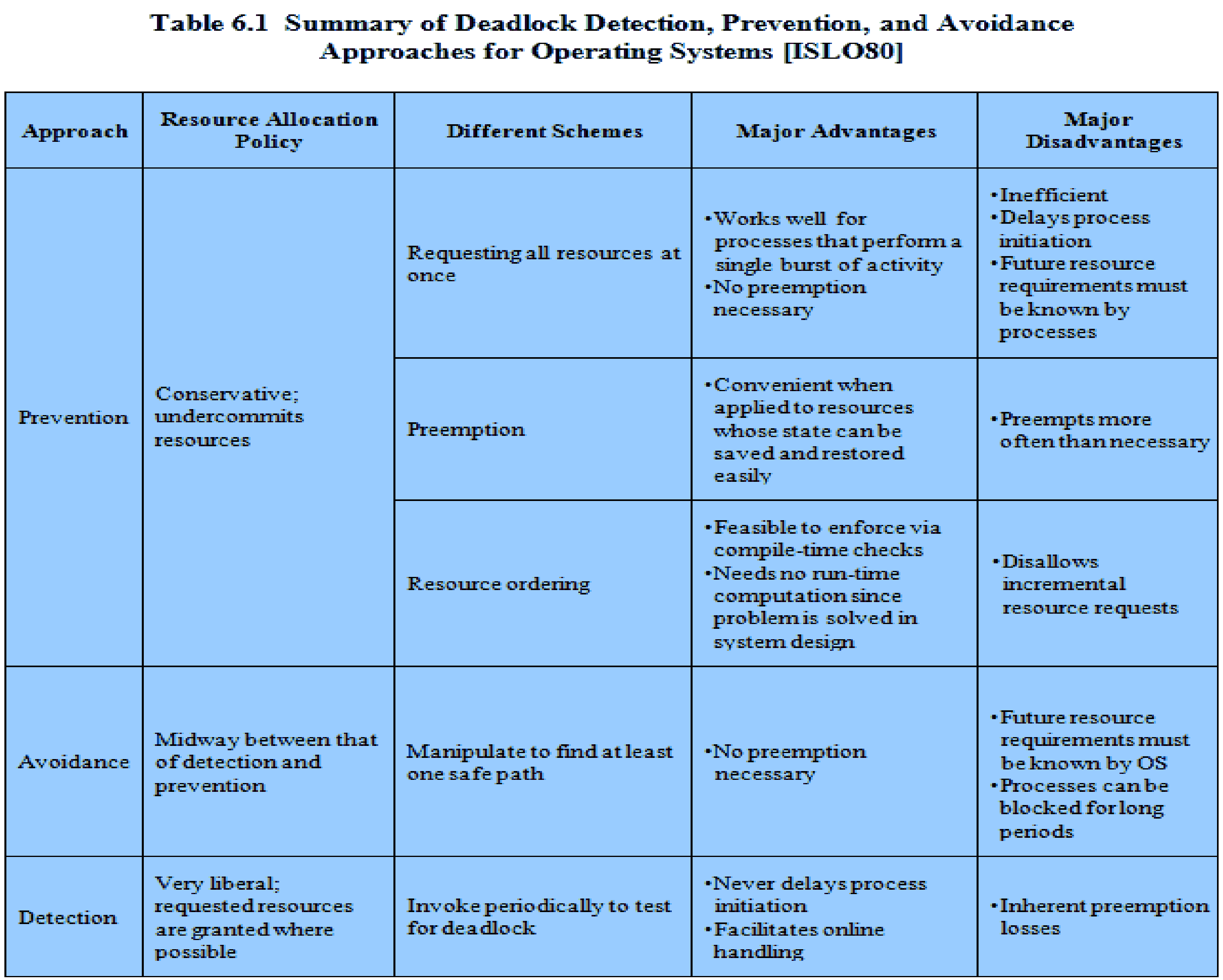


Table 6.1 Summary of Deadlock Detection, Prevention, and Avoidance Approaches for Operating Systems [ISLO80]

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	•Works well for processes that perform a single burst of activity •No preemption necessary	•Inefficient •Delays process initiation •Future resource requirements must be known by processes
		Preemption	•Convenient when applied to resources whose state can be saved and restored easily	•Preempts more often than necessary
		Resource ordering	•Feasible to enforce via compile-time checks •Needs no run-time computation since problem is solved in system design	•Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	•No preemption necessary	•Future resource requirements must be known by OS •Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	•Never delays process initiation •Facilitates online handling	•Inherent preemption losses

SCHOOL OF ENGINEERING

END OF MODULE 3