Module 3: PROCESS COORDINATION

-by

Asst Prof Rohini M. Sawant

TYPES OF PROCESSES

- Processes executing concurrently in the operating system may be either independent processes or cooperating processes.
- *Independent* process cannot affect or be affected by the execution of another process.
- *Cooperating* process can affect or be affected by the execution of another process.
- Cooperating processes require an **interprocess communication** (**IPC**) mechanism that will allow them to exchange data— that is, send data to and receive data from each other.

PROCESS COOPERATION

Reasons for Cooperation

- **Information sharing**. Since several applications may be interested in the same piece of information (for instance, copying and pasting), we must provide an environment to allow concurrent access to such information.
- **Computation speedup**. If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing cores.
- **Modularity**. We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.
- **Convenience.**

INTERPROCESS COMMUNICATION

- Cooperating Process require an Interprocess Communication mechanism that will allow them to exchange data and information
- There are two models for IPC:

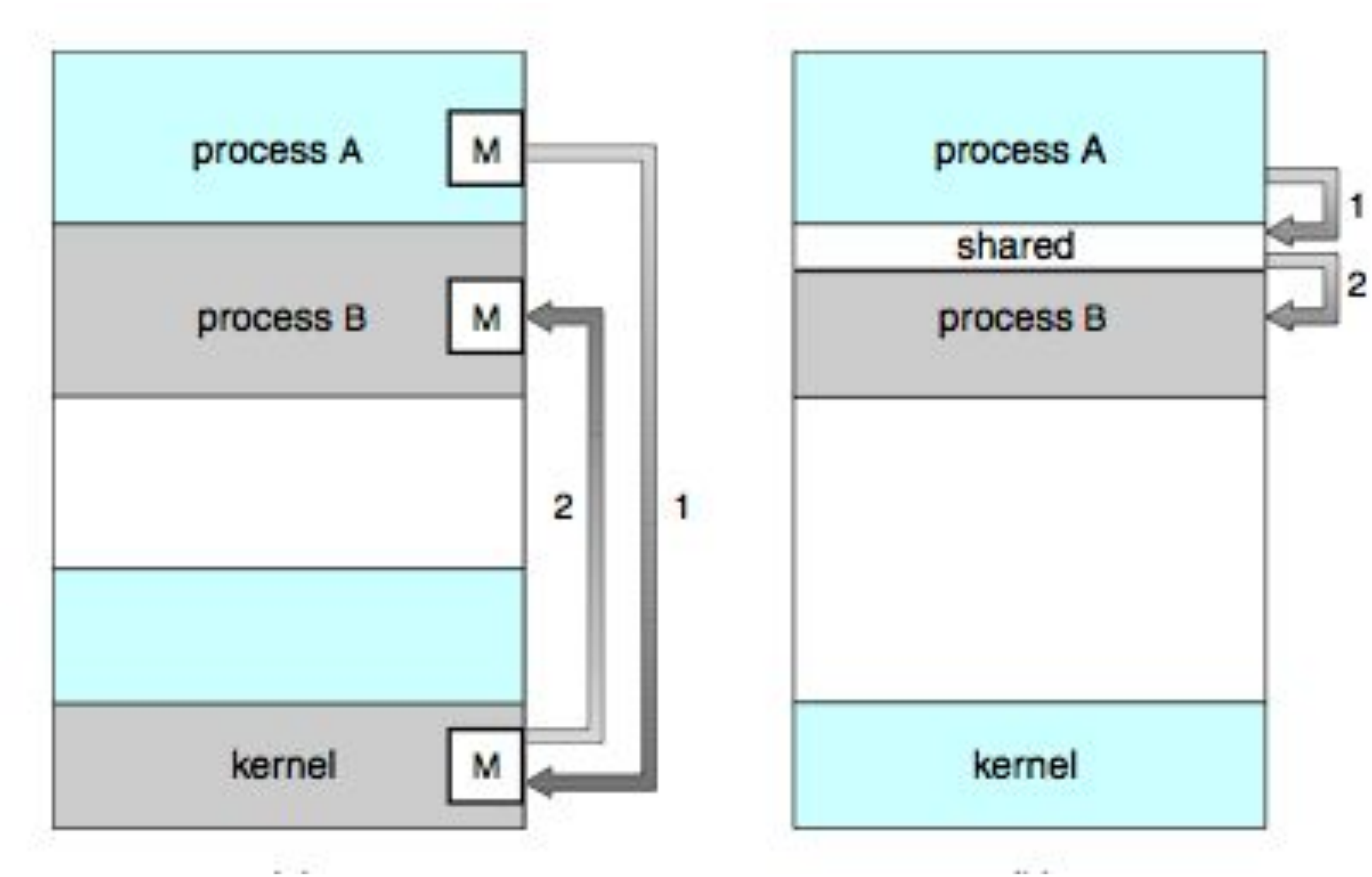
□ **SHARED MEMORY MODEL**

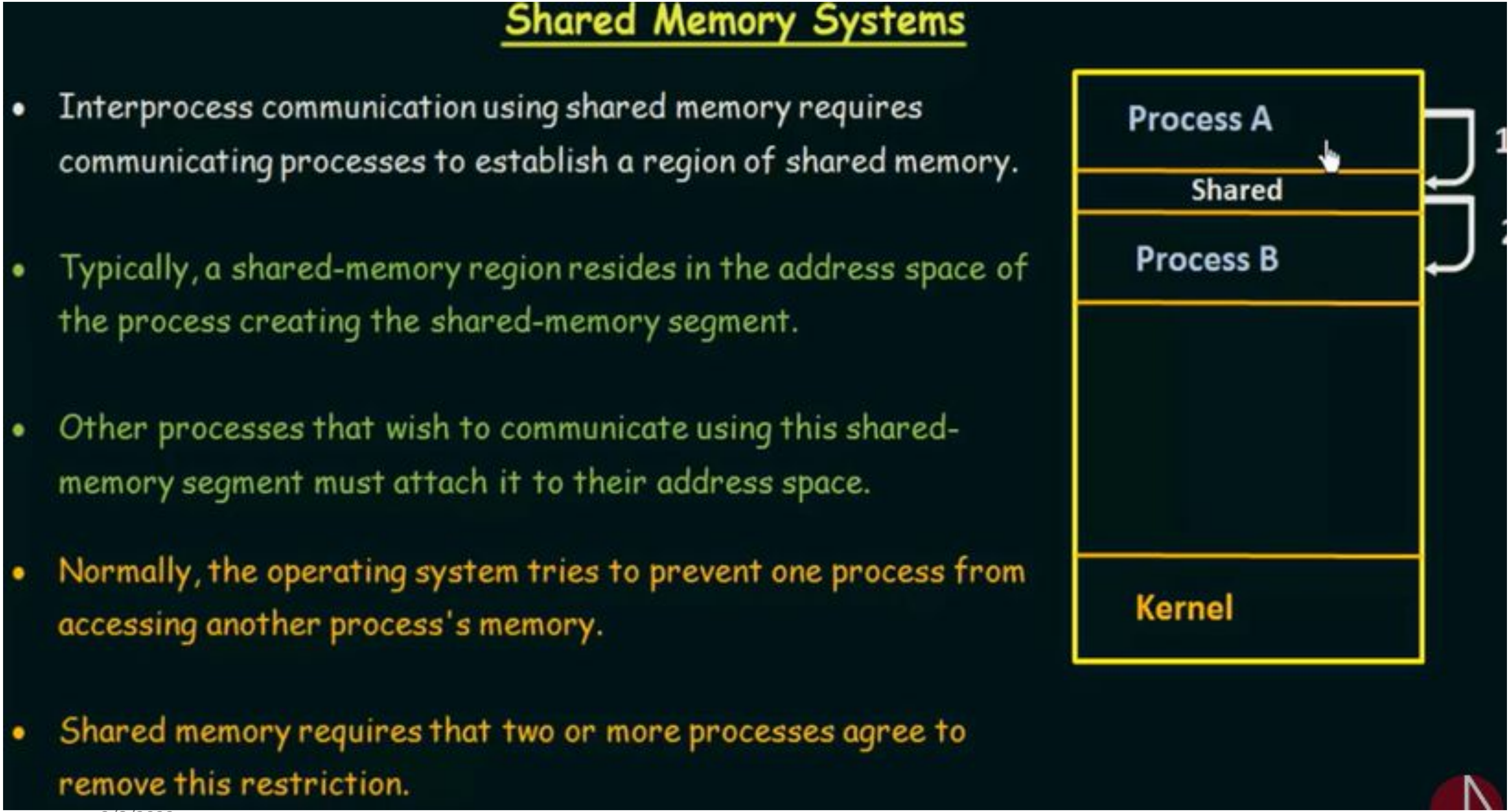
□ **MESSAGE PASSING MODEL**

- In the shared-memory model, a region of memory that is shared by the cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.
- In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.

INTERPROCESS COMMUNICATION

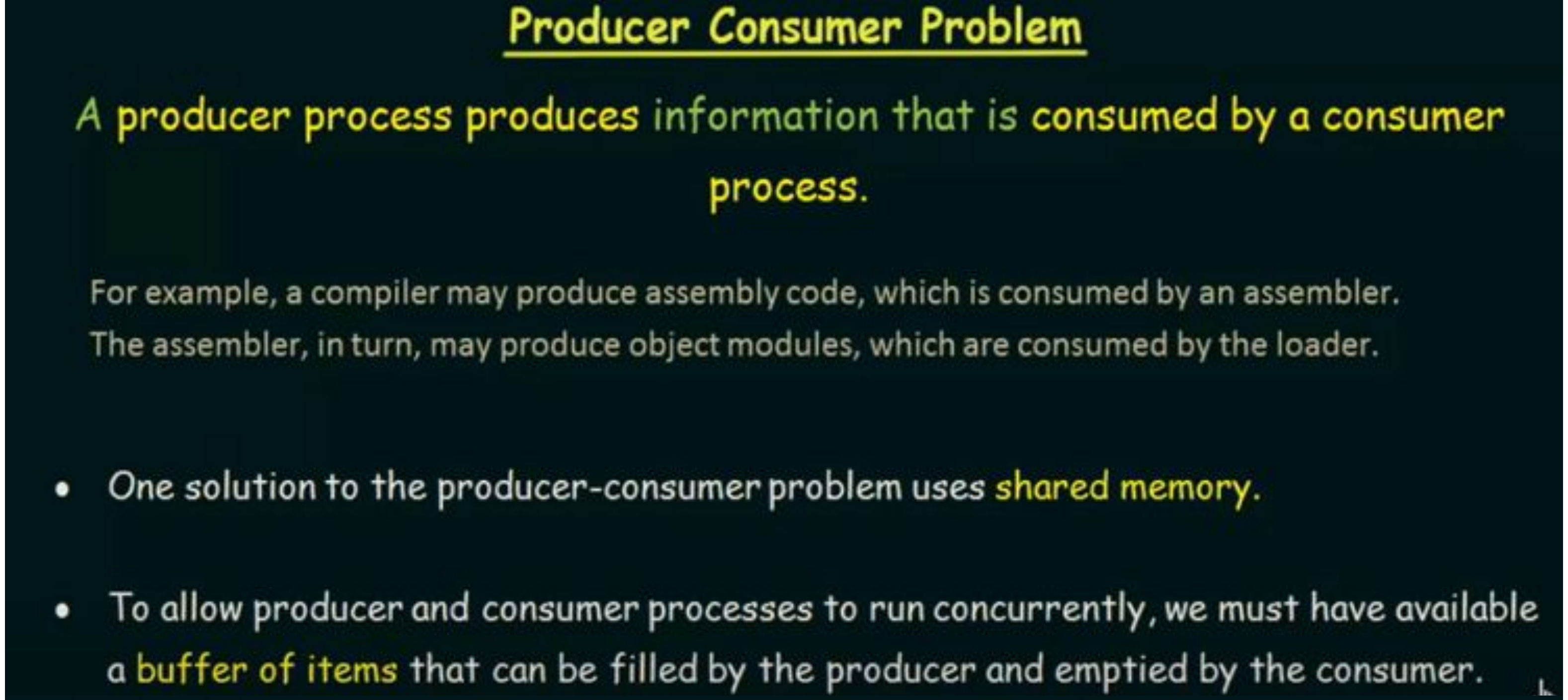
Message Passing **Shared Memory**





Shared Memory Systems

- Interprocess communication using shared memory requires communicating processes to establish a region of shared memory.
- Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment.
- Other processes that wish to communicate using this shared-memory segment must attach it to their address space.
- Normally, the operating system tries to prevent one process from accessing another process's memory.
- Shared memory requires that two or more processes agree to remove this restriction.

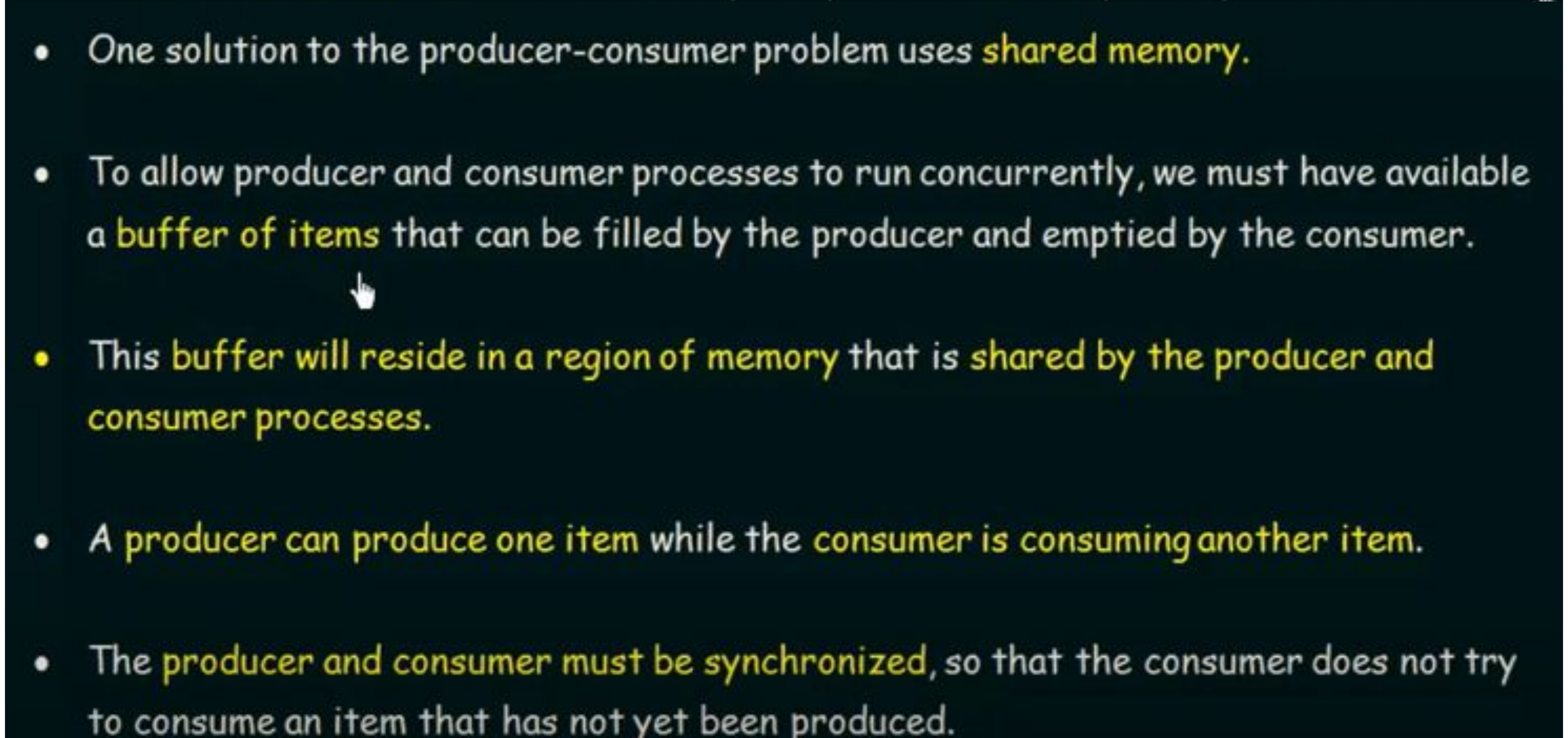


Producer Consumer Problem

A producer process produces information that is consumed by a consumer process.

For example, a compiler may produce assembly code, which is consumed by an assembler.
The assembler, in turn, may produce object modules, which are consumed by the loader.

- One solution to the producer-consumer problem uses shared memory.
- To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.



- One solution to the producer-consumer problem uses shared memory.
- To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.

- This buffer will reside in a region of memory that is shared by the producer and consumer processes.
- A producer can produce one item while the consumer is consuming another item.
- The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

SHARED MEMORY

- Two types of buffers can be used.

 The **unbounded buffer** places no practical limit on the size of the buffer. The consumer may have to wait for new items only when the buffer is empty, but the producer can always produce new items as there is not limit on size.

 The **bounded buffer** assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

MESSAGE PASSING

- Message system – processes communicate with each other without resorting to shared variables
- Message passing facility provides two operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

DIRECT COMMUNICATION

- Processes must name each other explicitly:
    - **send** (*P, message*) – send a message to process P
    - **receive**(*Q, message*) – receive a message from process Q
- Properties of communication link
    - Links are established automatically
    - A link is associated with exactly one pair of communicating processes
    - Between each pair there exists exactly one link
    - The link may be unidirectional, but is usually bi-directional

DIRECT COMMUNICATION

- This scheme exhibits ***symmetry*** in addressing; that is, both the sender process and the receiver process must name the other to communicate.
- A variant of this scheme employs ***asymmetry*** in addressing. the send() and receive() primitives are defined as follows:
- send(P, message)—Send a message to process P.
- receive(id, message)—Receive a message from any process.
- The disadvantage in both these schemes is the **Limited Modularity** of the resulting process definition. Changing the identifier of a process may necessitate examining all other process definitions.

INDIRECT COMMUNICATION

- Messages are directed and received from mailboxes (also referred to as ports).
- A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.
  - Each mailbox has a unique id.
  - Processes can communicate only if they share a mailbox.

- Properties of communication link
  - Link established only if processes share a common mailbox.
  - A link may be associated with many processes.
  - Each pair of processes may share several communication links.
  - Link may be unidirectional or bi-directional.

INDIRECT COMMUNICATION

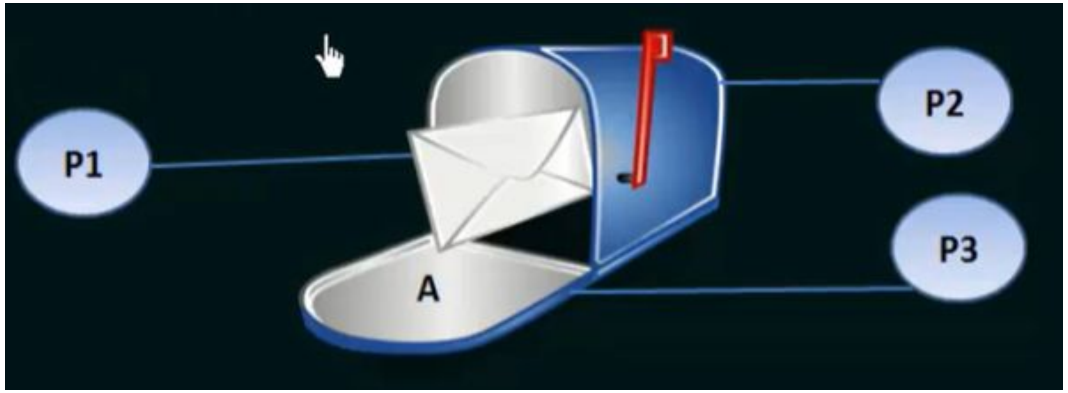
- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:

send(*A, message*) – send a message to mailbox A

receive(*A, message*) – receive a message from mailbox A.

INDIRECT COMMUNICATION

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?



- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was. The system also may define an algorithm for selecting which process will receive message (Eg Roundrobin)

SYNCHRONOUS AND ASYNCHRONOUS

- **Synchronization**
- Communication between processes takes place through calls to send() and receive() primitives.
- There are different design options for implementing each primitive.
- Message passing may be either **blocking** or **nonblocking**—also known as **synchronous** and **asynchronous**.

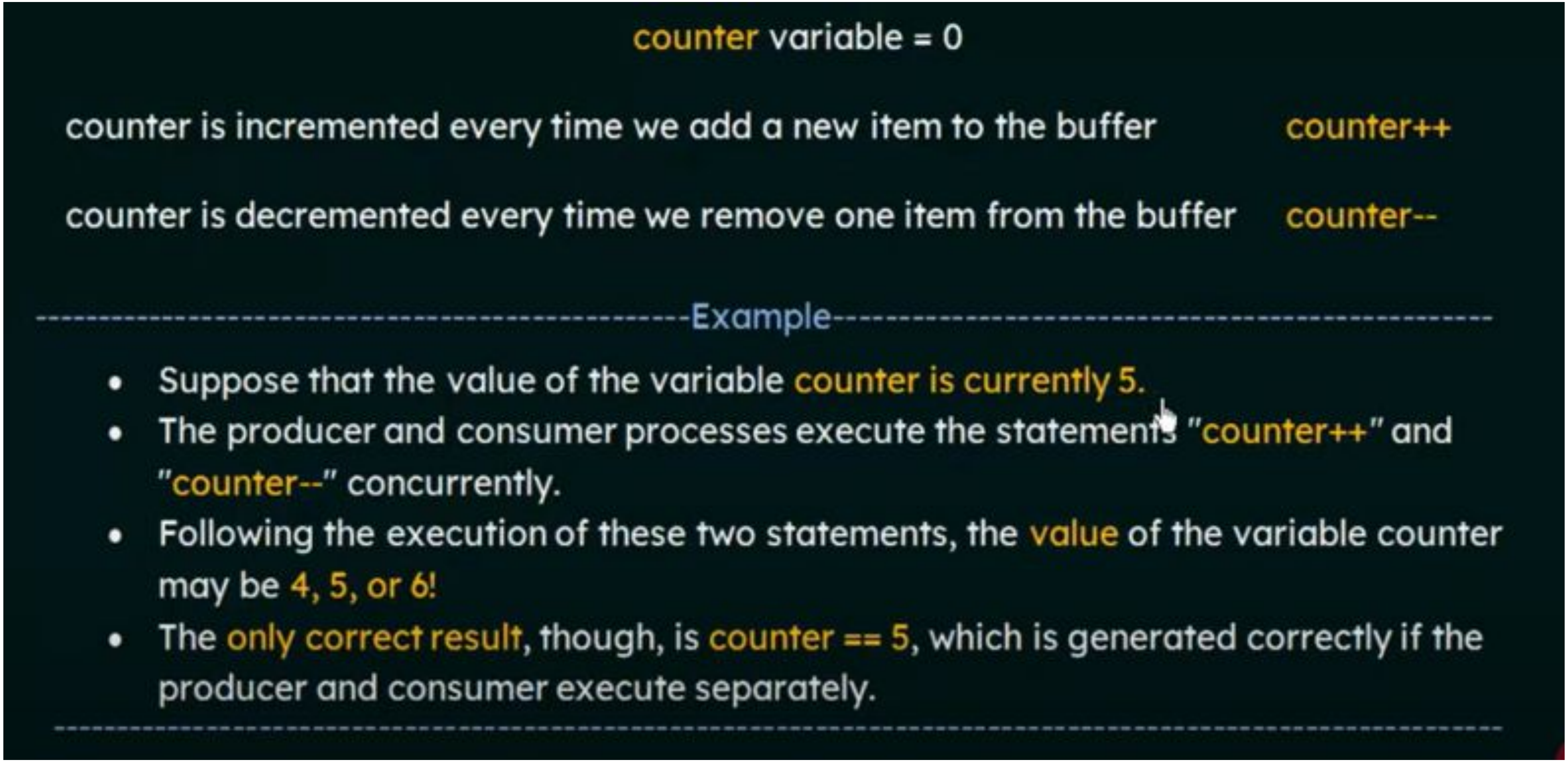
SYNCHRONOUS AND ASYNCHRONOUS

- **Blocking send**. The sending process is blocked until the message is
- received by the receiving process or by the mailbox.
- **Non-blocking send**. The sending process sends the message and resumes operation.
- **Blocking receive**. The receiver blocks until a message is available.
- **Non-blocking receive**. The receiver retrieves either a valid message or a null.

BUFFERING

- Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:
- **Zero capacity**. The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
- **Bounded capacity**. The queue has finite length *n;* thus, at most *n* messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue, and the sender can continue execution without waiting. The link's capacity is finite, however. If the link is full, the sender must block until space is available in the queue.
- **Unbounded capacity**. The queue's length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.
- The zero-capacity case is sometimes referred to as a message system with no buffering. The other cases are referred to as systems with automatic buffering.

PROCESS SYNCHRONIZATION



counter variable = 0

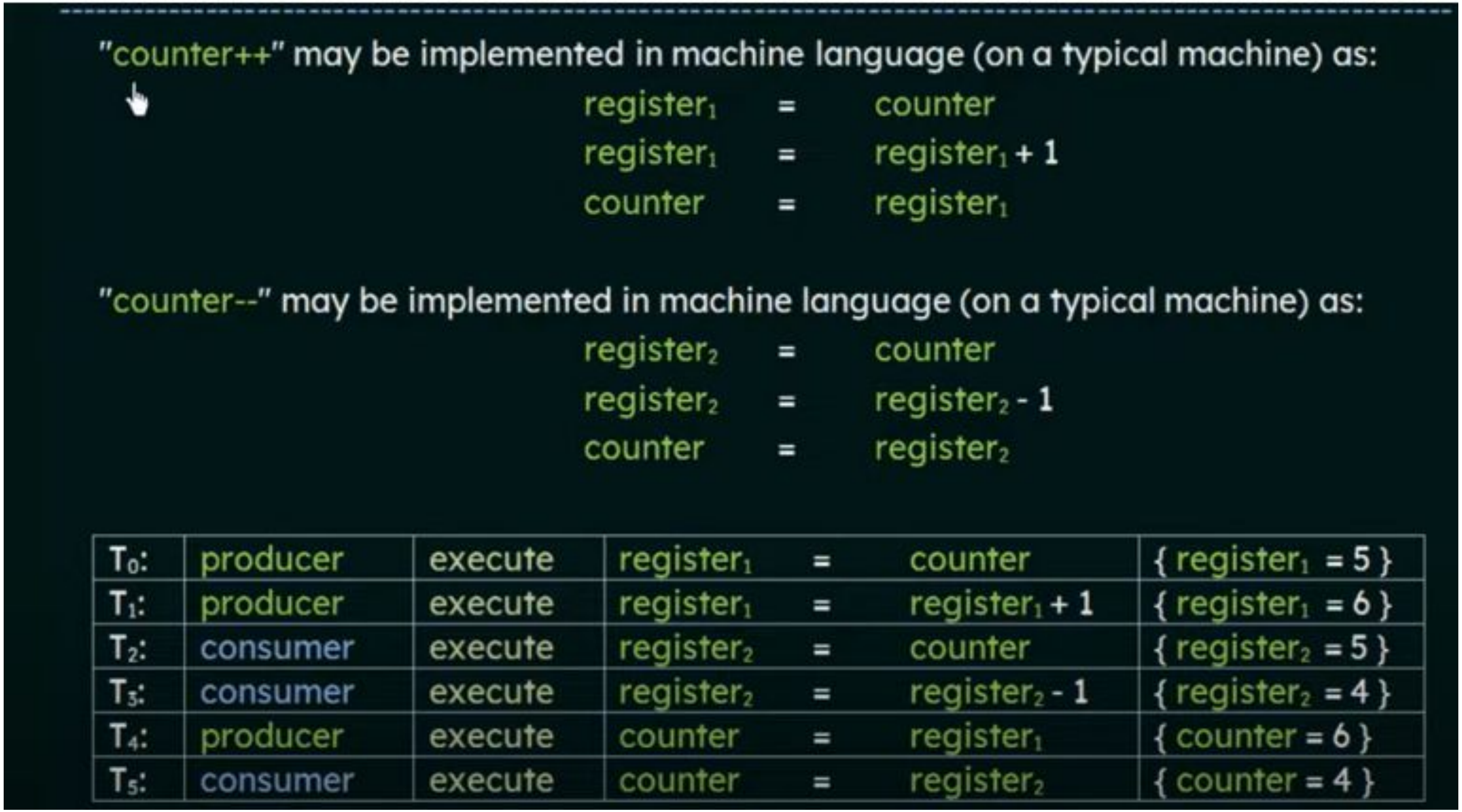
counter is incremented every time we add a new item to the buffer counter++

counter is decremented every time we remove one item from the buffer counter--

--Example--

- Suppose that the value of the variable counter is currently 5.
- The producer and consumer processes execute the statements "counter++" and "counter--" concurrently.
- Following the execution of these two statements, the value of the variable counter may be 4, 5, or 6!
- The only correct result, though, is counter == 5, which is generated correctly if the producer and consumer execute separately.

PROCESS SYNCHRONIZATION



"counter++" may be implemented in machine language (on a typical machine) as:

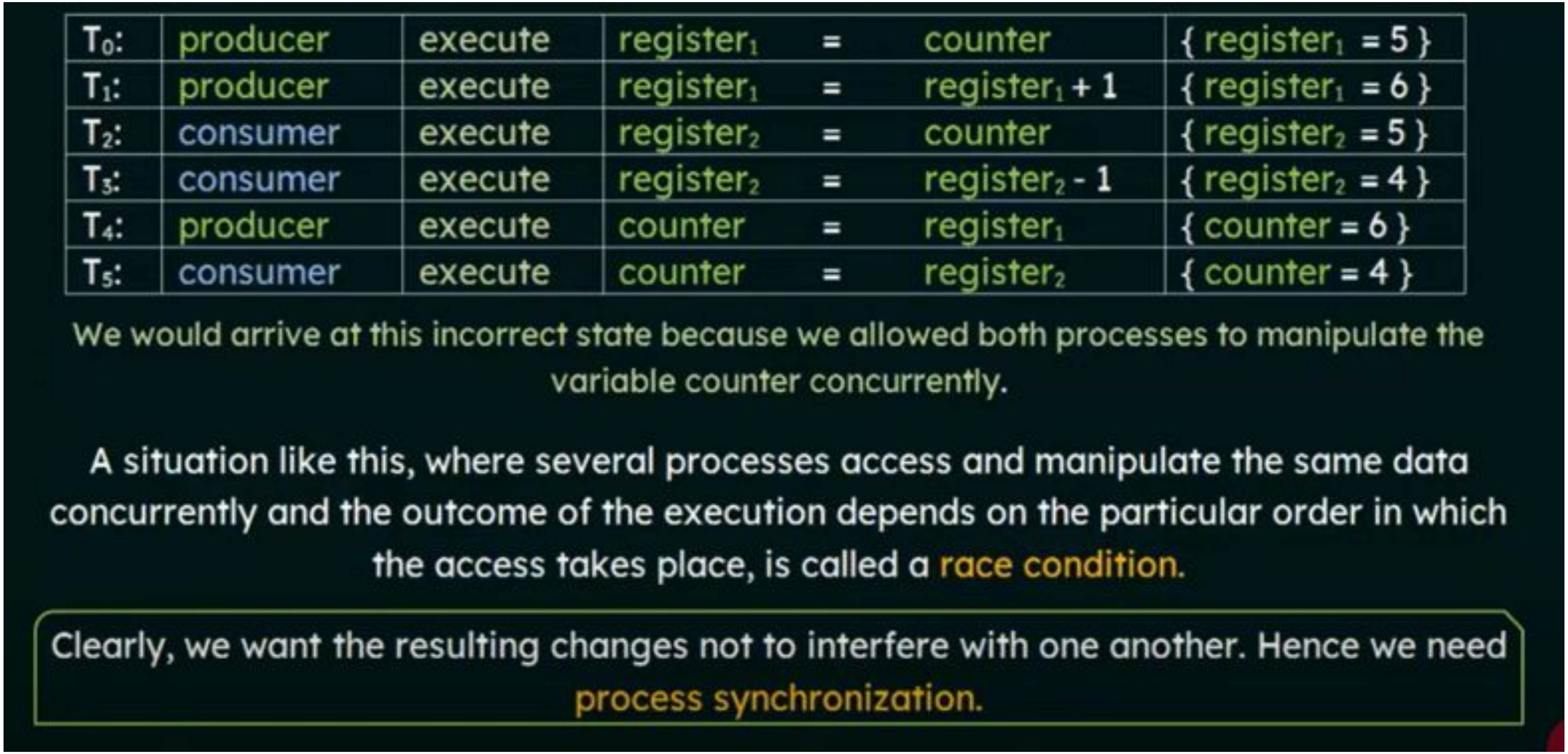
$$\begin{aligned} register_1 &= counter \\ register_1 &= register_1 + 1 \\ counter &= register_1 \end{aligned}$$

"counter--" may be implemented in machine language (on a typical machine) as:

$$\begin{aligned} register_2 &= counter \\ register_2 &= register_2 - 1 \\ counter &= register_2 \end{aligned}$$

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

PROCESS SYNCHRONIZATION



T_0:	producer	execute	register$_1$	=	counter	{ register$_1$ = 5 }
T_1:	producer	execute	register$_1$	=	register$_1$ + 1	{ register$_1$ = 6 }
T_2:	consumer	execute	register$_2$	=	counter	{ register$_2$ = 5 }
T_3:	consumer	execute	register$_2$	=	register$_2$ - 1	{ register$_2$ = 4 }
T_4:	producer	execute	counter	=	register$_1$	{ counter = 6 }
T_5:	consumer	execute	counter	=	register$_2$	{ counter = 4 }

We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently.

A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition.

Clearly, we want the resulting changes not to interfere with one another. Hence we need process synchronization.

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**.
- To guard against Race condition above, we need to ensure that only one process at a time can be manipulating the variable counter.
- To make such a guarantee, we require that the processes be synchronized in some way.
- Synchronization is the mechanism to ensure orderly execution of cooperating processes that share a logical address space.

RACE CONDITION

- count++ could be implemented as

```
    register1 = count
    register1 = register1 + 1
    count = register1
```

- count-- could be implemented as

```
    register2 = count
    register2 = register2 - 1
    count = register2
```

RACE CONDITION

- 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.
- A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition.

RACE CONDITION

- Consider this execution interleaving with “count = 5” initially:

  S0: producer execute register1 = count {register1 = 5}

  S1: producer execute register1 = register1 + 1 {register1 = 6}

  S2: consumer execute register2 = count {register2 = 5}

  S3: consumer execute register2 = register2 - 1 {register2 = 4}

  S4: producer execute count = register1 {count = 6 }

  S5: consumer execute count = register2 {count = 4}

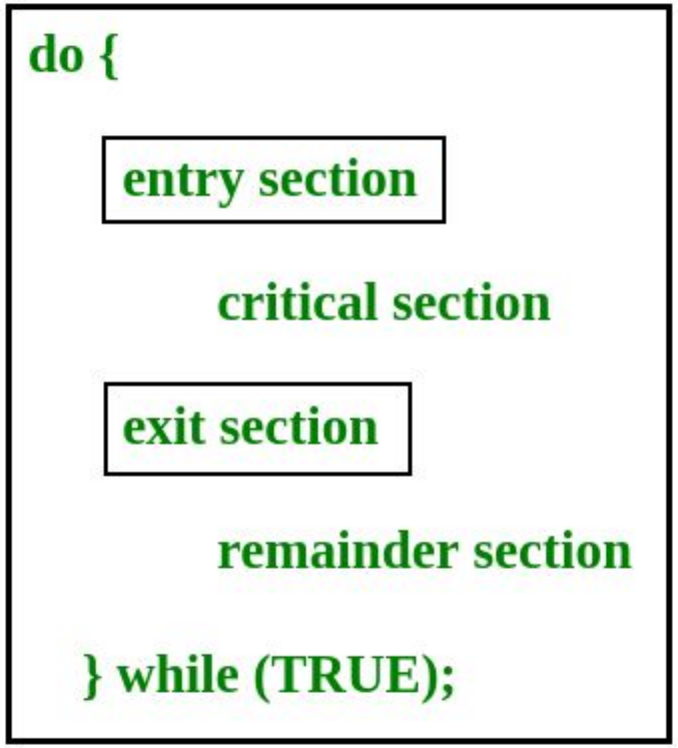
- The value of **count** may be either 4 or 6, where the correct result should be 5.

CRITICAL SECTION

- Consider a system consisting of n processes {P0, P1, ..., Pn−1}.
- Each process has a segment of code, called a critical section, in which the process may be accessing — and updating — data that is shared with at least one other process.
- In Critical Section the process may be changing common variables, updating a table, writing a file and so on
- The important feature of the system is that, when one process is executing in its critical section, no other process is allowed to execute in its critical section.
- That is no two process are executing in their Critical Section at the same time.
- The Critical Section Problem is to design a protocol that the processes can use to cooperate.

RULES OF CRITICAL SECTION

- 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**.



```
do {

    entry section

        critical section

    exit section

        remainder section

  } while (TRUE);
```

Solution to Critical-Section Problem:

A **Solution to Critical-Section Problem** must satisfy these 3 requirements:

- ❖ Mutual Exclusion
- ❖ Progress
- ❖ Bounded Waiting

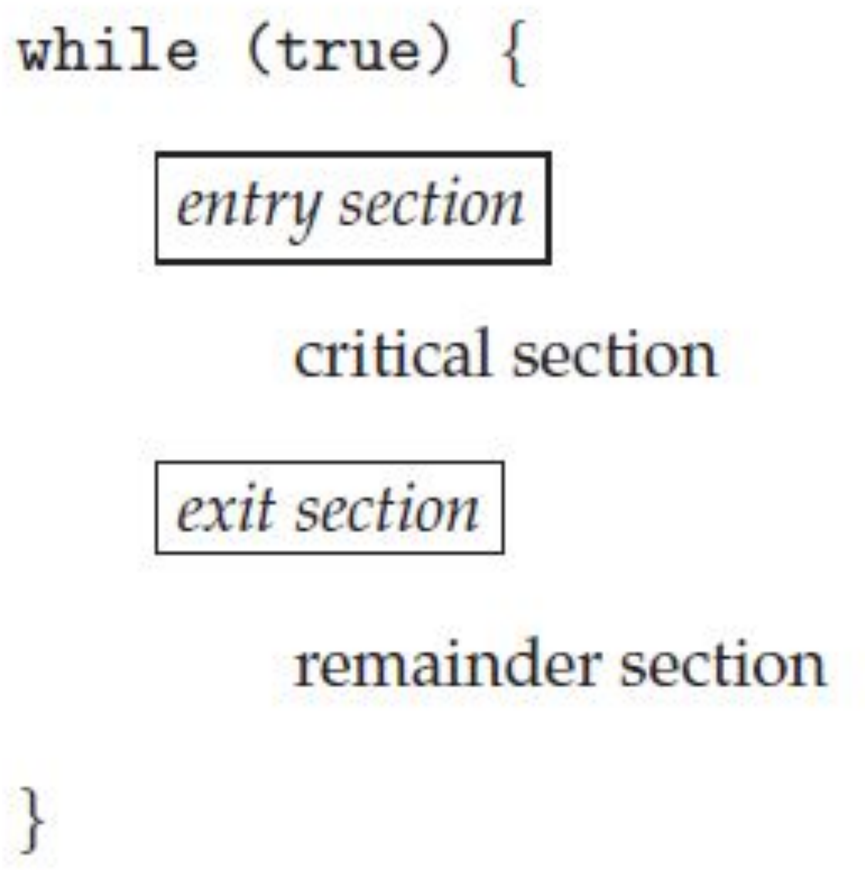
1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections

Solution to Critical-Section Problem:

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.

Solution to Critical-Section Problem:



```
while (true) {

    entry section

        critical section

    exit section

        remainder section

}
```

PETERSON’S ALGORITHM

- **Peterson's algorithm** (or **Peterson's solution**) is a concurrent programming algorithm for mutual exclusion that allows two or more processes to share a single-use resource without conflict, using only shared memory for communication.
- It was formulated by Gary L. Peterson in 1981.
- While Peterson's original formulation worked with only two processes.

PETERSON’S ALGORITHM

- The algorithm uses two variables, flag and turn.
- A flag[n] value of true indicates that the process n wants to enter the critical section(intension).
- Turn variable value decides which process enters in critical section.
- Entrance to the critical section is granted for process P0 if P1 does not want to enter its critical section or if P1 has given priority to P0 by setting turn to 0.

Peterson’s Solution

- Process P_i

```
do {

      flag [i]:= true;

      turn = j;

      while (flag [j] and turn = j) ;

         critical section

      flag [i] = false;

         remainder section

} while (1);
```

Process Pj:

```
do {
flag [j]:= true;
turn = i;
while (flag [i] and turn =
i) ;

          critical section

          flag [j] = false;

remainder section
                    } while;
```

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

PETERSON’S ALGORITHM

- Given 2 process i and j, you need to guarantee mutual exclusion between the two without any additional hardware support.
- Basically, Peterson’s algorithm provides guaranteed mutual exclusion by using only the shared memory.
- It uses two ideas in the algorithm,

1. Willingness to acquire lock.
2. Turn to acquire lock.

PETERSON’S ALGORITHM

- The idea is that first Pi process expresses its desire to acquire lock and sets

flag[self] = 1 and then gives the other Process

Pj a chance to acquire the lock.

- If process Pj desires to acquire the lock, then, it gets the lock and then passes the chance to the Pi process again.
- If it does not desire to get the lock then the while loop breaks and the Pj process gets the chance.

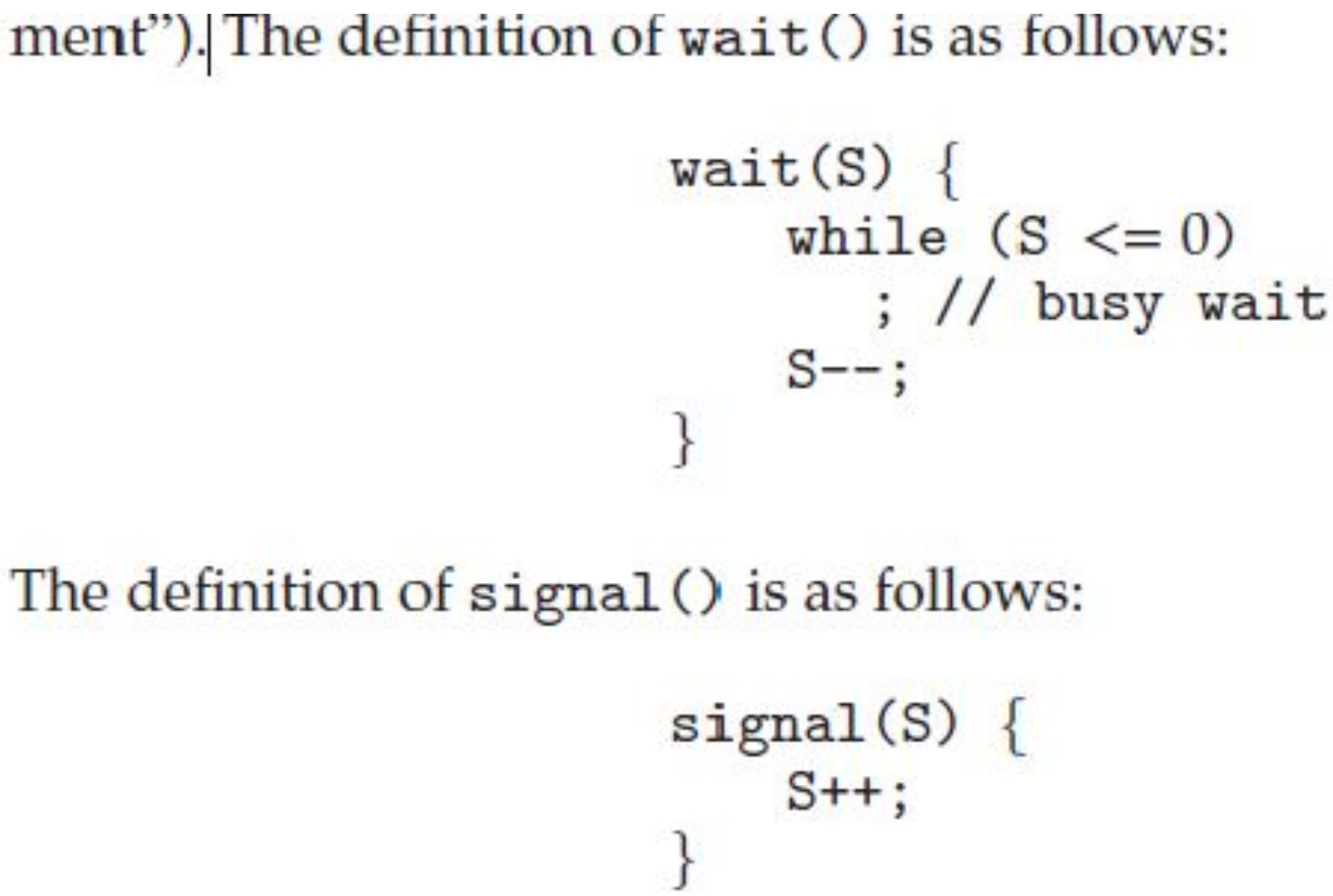
PETERSON’S ALGORITHM

Disadvantages:

- It works for 2 processes.
- It does not guarantee to work on modern computer architecture.

SEMAPHORES

- Special variable called a semaphore is used for signaling
- A process is suspended until, it has received a specific signal.
- Semaphore is a variable that has an integer value
- May be initialized to a nonnegative number.
- Two standard operations modify S: wait() and signal()
- Semaphores were introduced by the Dutch computer scientist Edsger Dijkstra, and such,
  the wait() operation was originally termed P (from the Dutch proberen, “to test”);
  the signal() was originally called V (from verhogen, “to increment”).
- Wait operation decrements the semaphore value
- Signal operation increments semaphore value



ment”). The definition of wait() is as follows:

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

The definition of signal() is as follows:

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

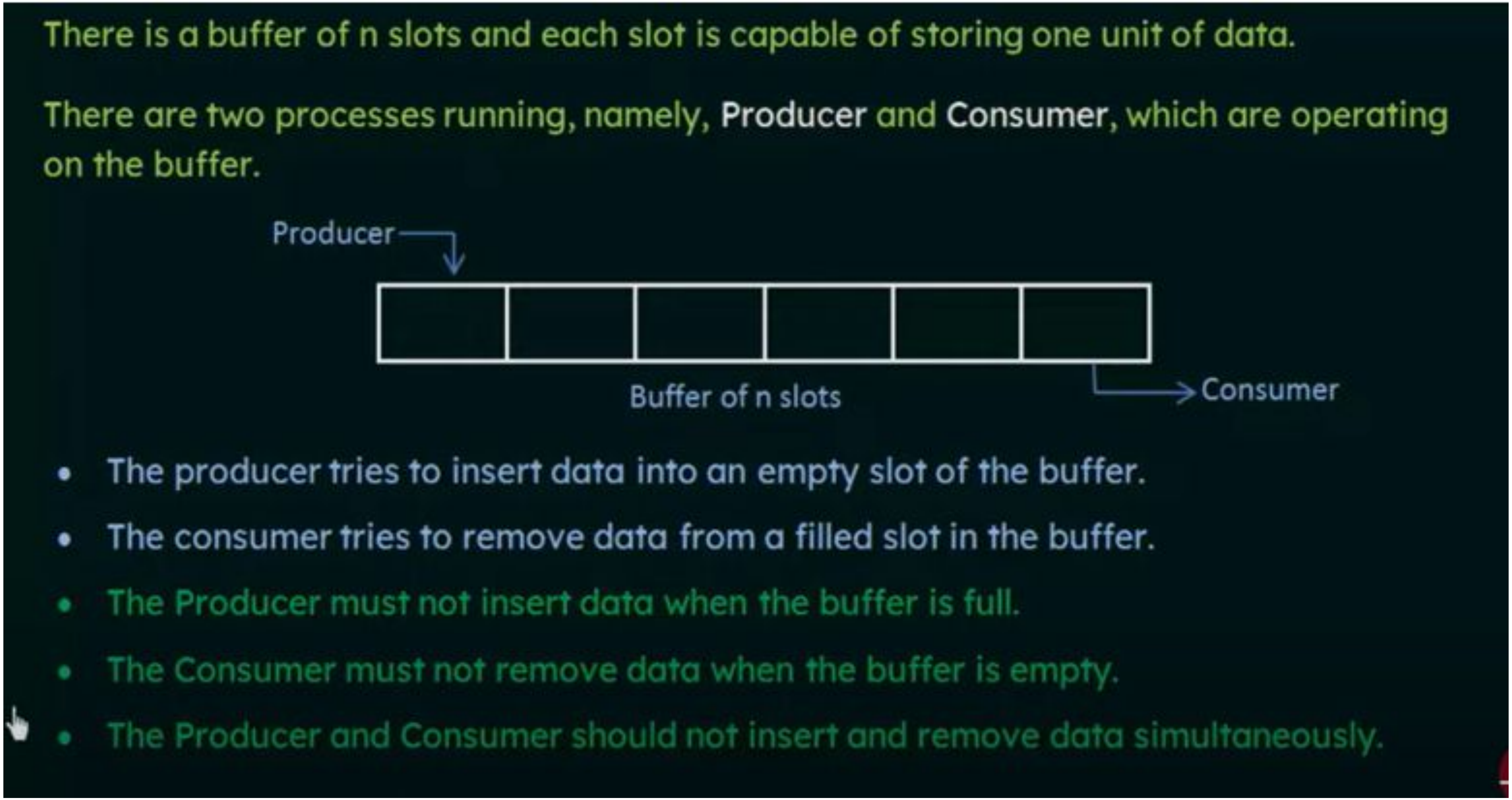
SEMAPHORES

- Operating systems often has two types of Semaphores: Counting and Binary semaphores.
- The value of a counting semaphore can range over an unrestricted domain.
- The value of a binary semaphore can range only between 0 and 1. Thus, binary semaphores behave similarly to mutex locks.
- Counting semaphores can range over an unrestricted domain & can be used to control access to a given resource consisting of a finite number of instances.
- The semaphore is initialized to the number of resources available.
- Each process that wishes to use a resource performs a wait() operation on the semaphore.
- When a process releases a resource, it performs a signal() operation.
- When the count for the semaphore goes to 0, all resources are being used.

CLASSIC PROBLEMS OF SYNCHRONIZATION

- **Bounded Buffer** problem is also called **Producer Consumer problem.**
- This problem is generalized in terms of the Producer-Consumer problem.
- Solution to this problem is, creating three counting semaphores “mutex”, “full” and “empty” to keep track of the current number of full and empty buffers respectively.
- Producers produce a product and consumers consume the product, but both use of one of the containers each time.
- Sleep-wake up system calls is used this situation as another solution to avoid race conditions.
- Producers should produce and go to sleep when the buffer is full and consumer should wake up when producer puts data in buffer. It should sleep until the producer produces data.

BOUNDED BUFFER

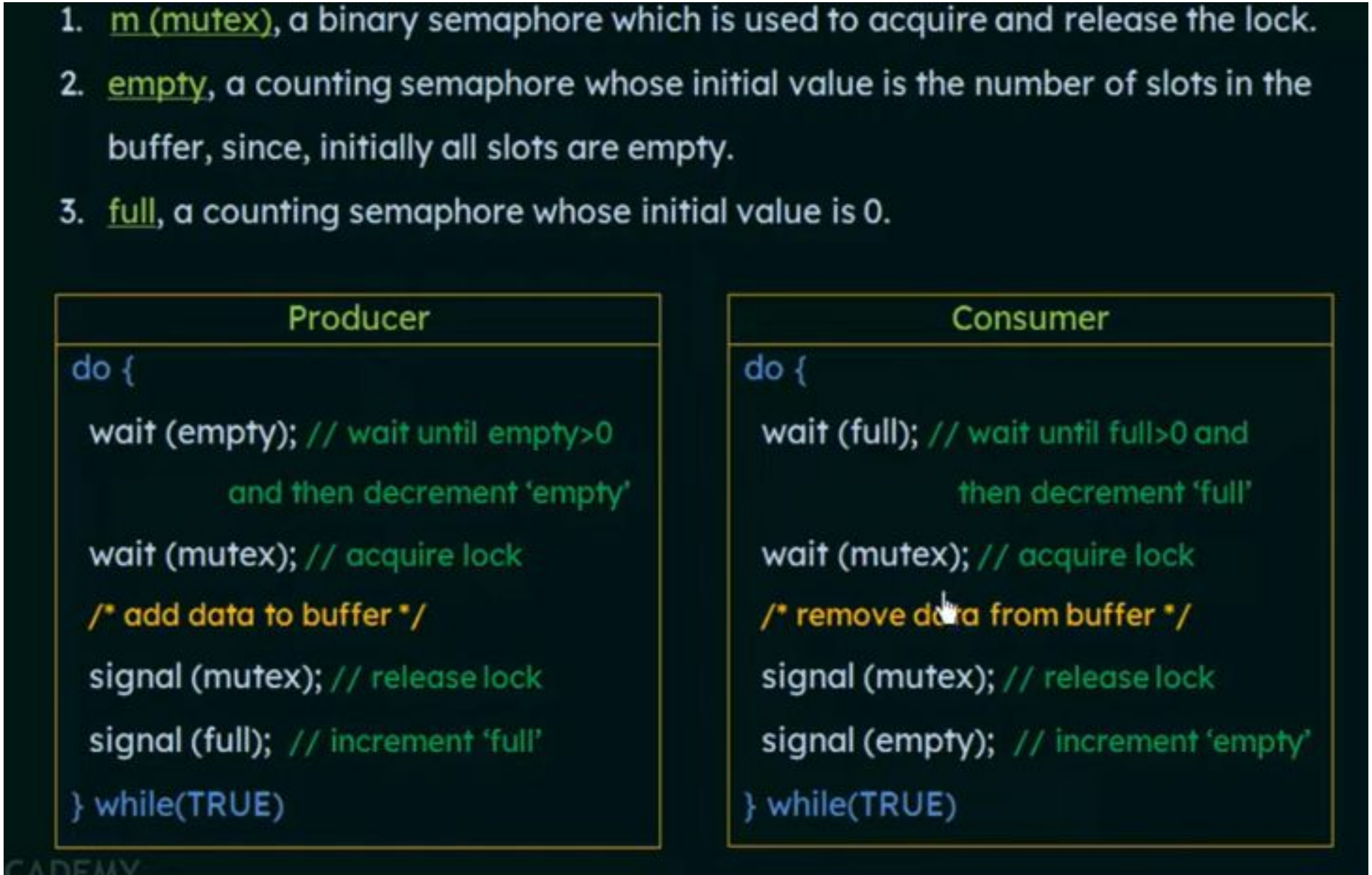


There is a buffer of n slots and each slot is capable of storing one unit of data.

There are two processes running, namely, Producer and Consumer, which are operating on the buffer.

- The producer tries to insert data into an empty slot of the buffer.
- The consumer tries to remove data from a filled slot in the buffer.
- The Producer must not insert data when the buffer is full.
- The Consumer must not remove data when the buffer is empty.
- The Producer and Consumer should not insert and remove data simultaneously.

BOUNDED BUFFER



1. m (mutex), a binary semaphore which is used to acquire and release the lock.
2. empty, a counting semaphore whose initial value is the number of slots in the buffer, since, initially all slots are empty.
3. full, a counting semaphore whose initial value is 0.

Producer

```
do {
  wait (empty); // wait until empty>0
                and then decrement 'empty'
  wait (mutex); // acquire lock
  /* add data to buffer */
  signal (mutex); // release lock
  signal (full);  // increment 'full'
} while(TRUE)
```

Consumer

```
do {
  wait (full); // wait until full>0 and
               then decrement 'full'
  wait (mutex); // acquire lock
  /* remove data from buffer */
  signal (mutex); // release lock
  signal (empty);  // increment 'empty'
} while(TRUE)
```

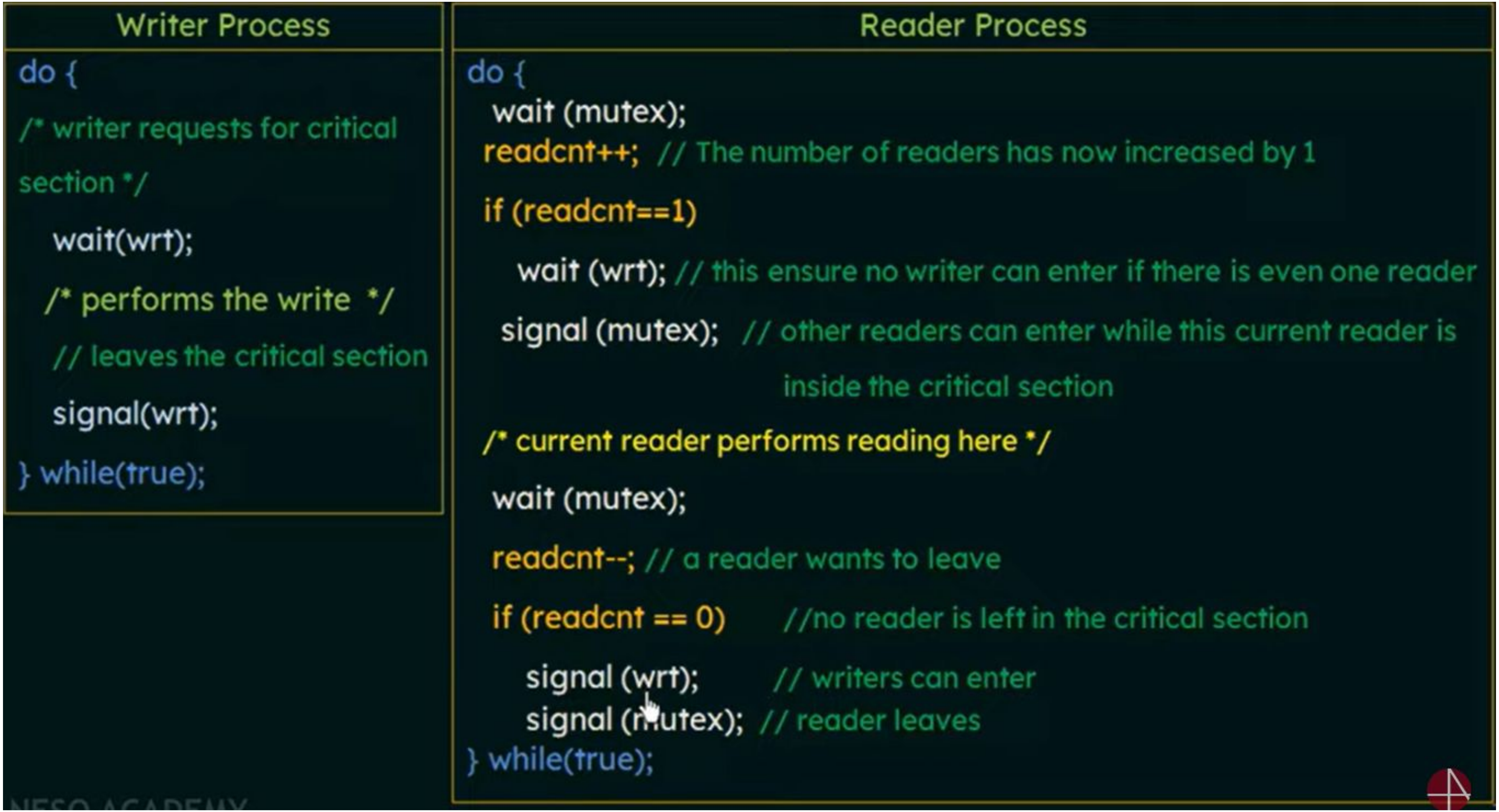
READERS/WRITERS PROBLEM

Two types of Processes:

- ❖ **Writers:** modify a shared object.
- ❖ **Readers:** they just read. They do not modify shared object.
- Any number of readers may simultaneously read the file
- Only one writer at a time may write to the file
- If a writer is writing to the file, no reader may read it or no writer may write it.
- Reader-writer problem has several variations, involving priorities:

1.Reader have priority: No reader will be waiting if no writer writing.(writer may suffer starvation.)

2.Writer have priority: if writer is writing no process can read. (reader's starvation.)

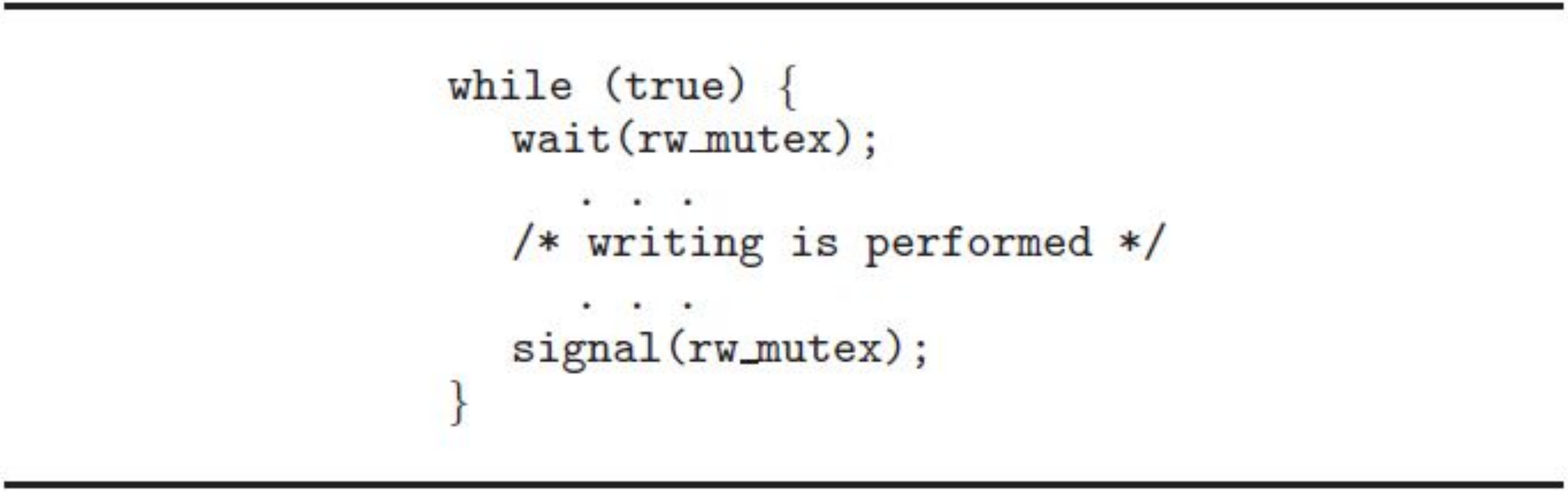


Writer Process

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

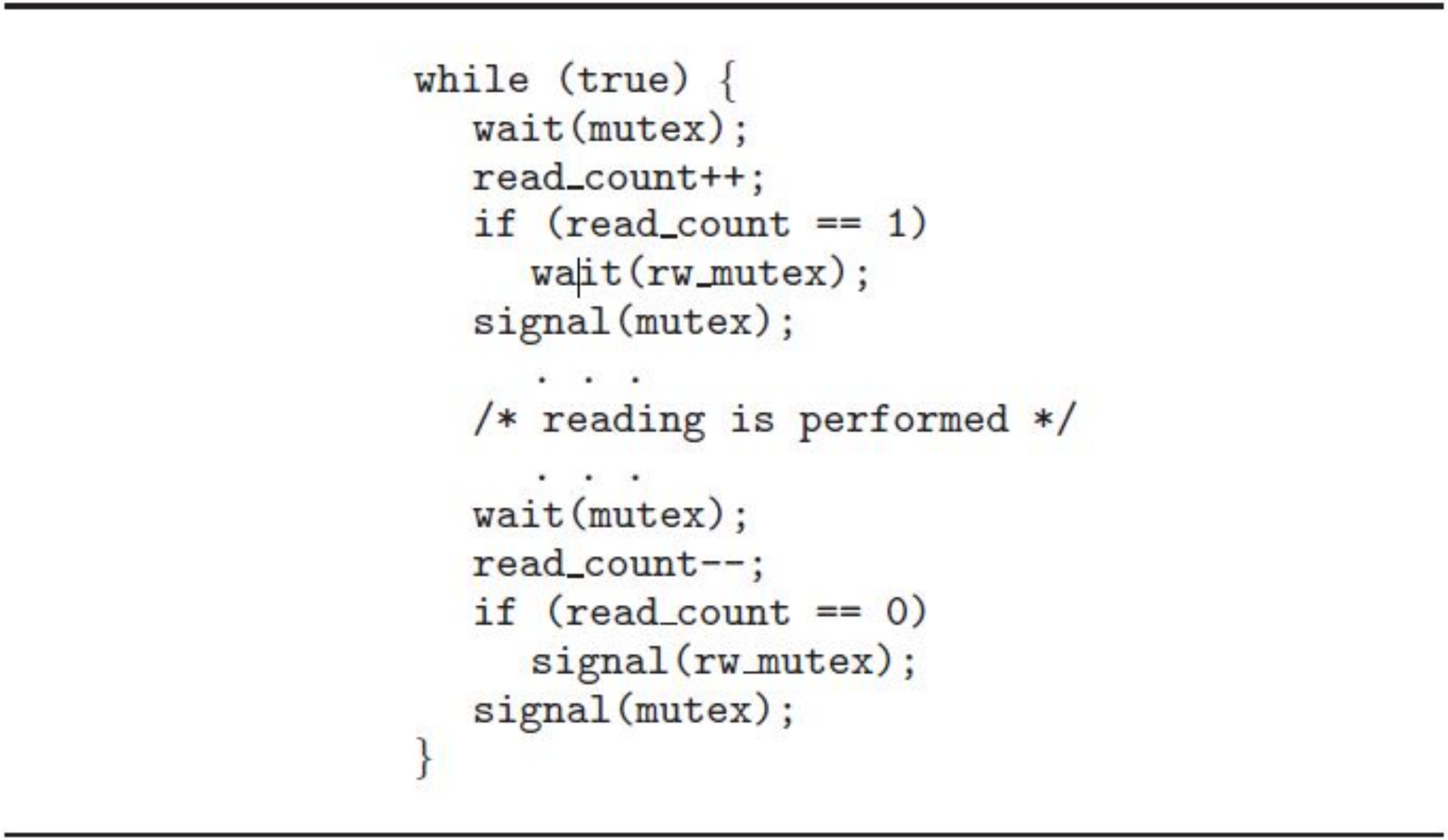
Reader Process

```
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);
```



```
while (true) {
   wait(rw_mutex);
      . . .
   /* writing is performed */
      . . .
   signal(rw_mutex);
}
```

Figure 7.3 The structure of a writer process.

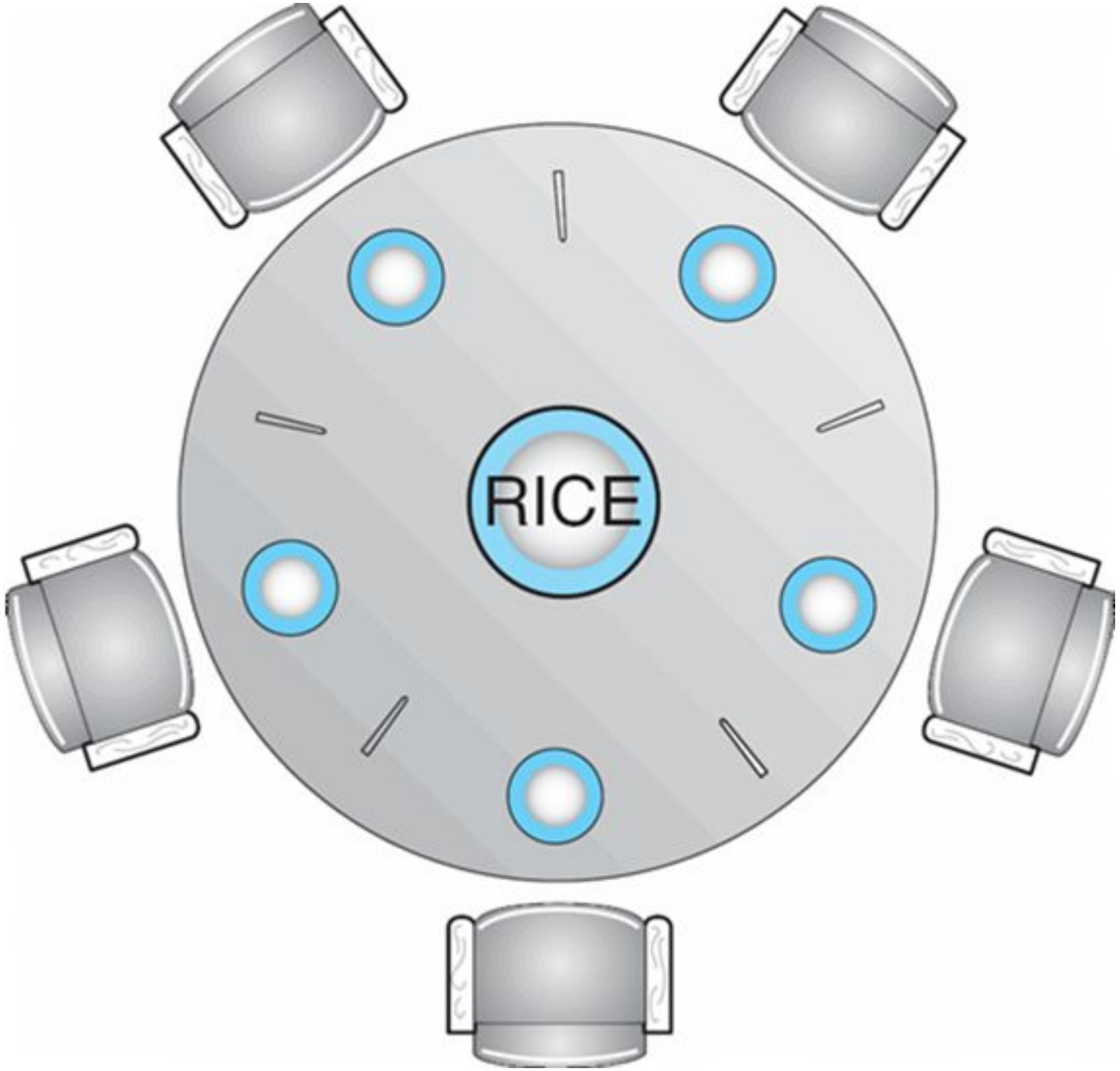


```
while (true) {
  wait(mutex);
  read_count++;
  if (read_count == 1)
    wait(rw_mutex);
  signal(mutex);
    . . .
  /* reading is performed */
    . . .
  wait(mutex);
  read_count--;
  if (read_count == 0)
    signal(rw_mutex);
  signal(mutex);
}
```

Figure 7.4 The structure of a reader process.

The Dining-Philosophers Problem

- Consider five philosophers who spend their lives thinking and eating.
- The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher.
- In the center of the table is a bowl of rice, and the table is laid with five single chopsticks.
- When a philosopher thinks, she does not interact with her colleagues.
- From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors).
- A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks.
- When she is finished eating, she puts down both chopsticks and starts thinking again.


RICE

The Dining-Philosophers Problem

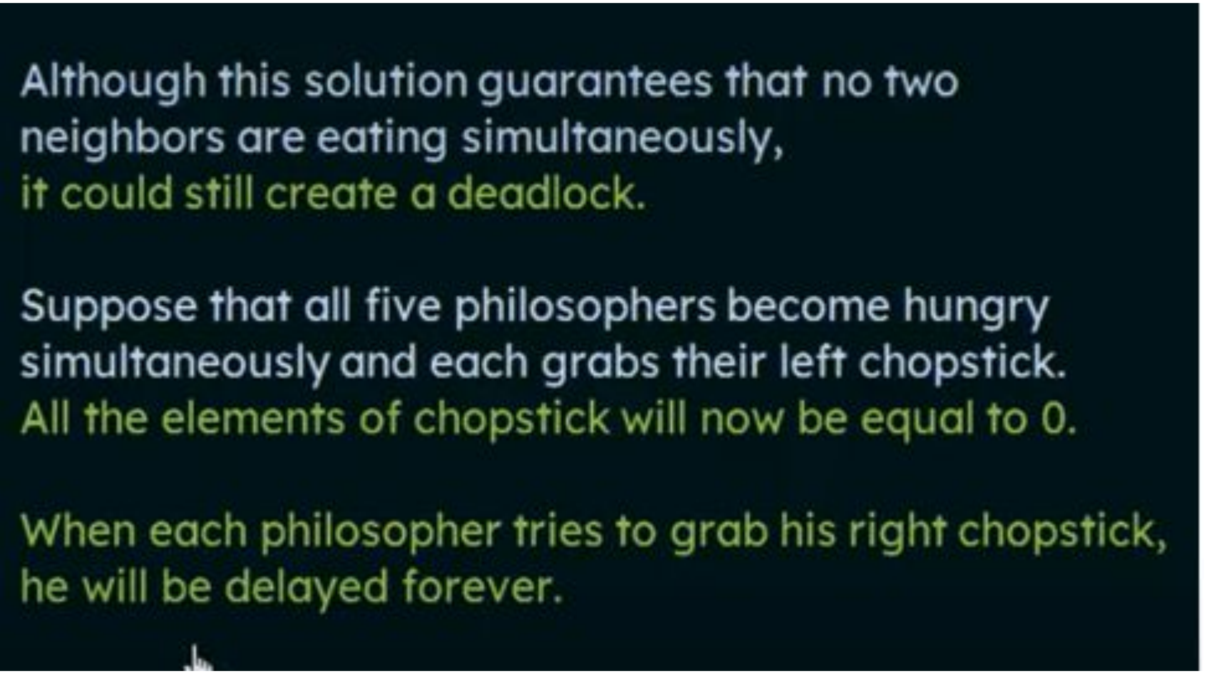
- One simple solution is to represent each chopstick with semaphore.
- A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore.
- She releases her chopsticks by executing the signal() operation on the appropriate semaphores. Thus, the shared data are

```
semaphore chopstick[5];
```

```
while (true) {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for a while */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
}
```

Figure 7.6 The structure of philosopher *i*.

The Dining-Philosophers Problem



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.

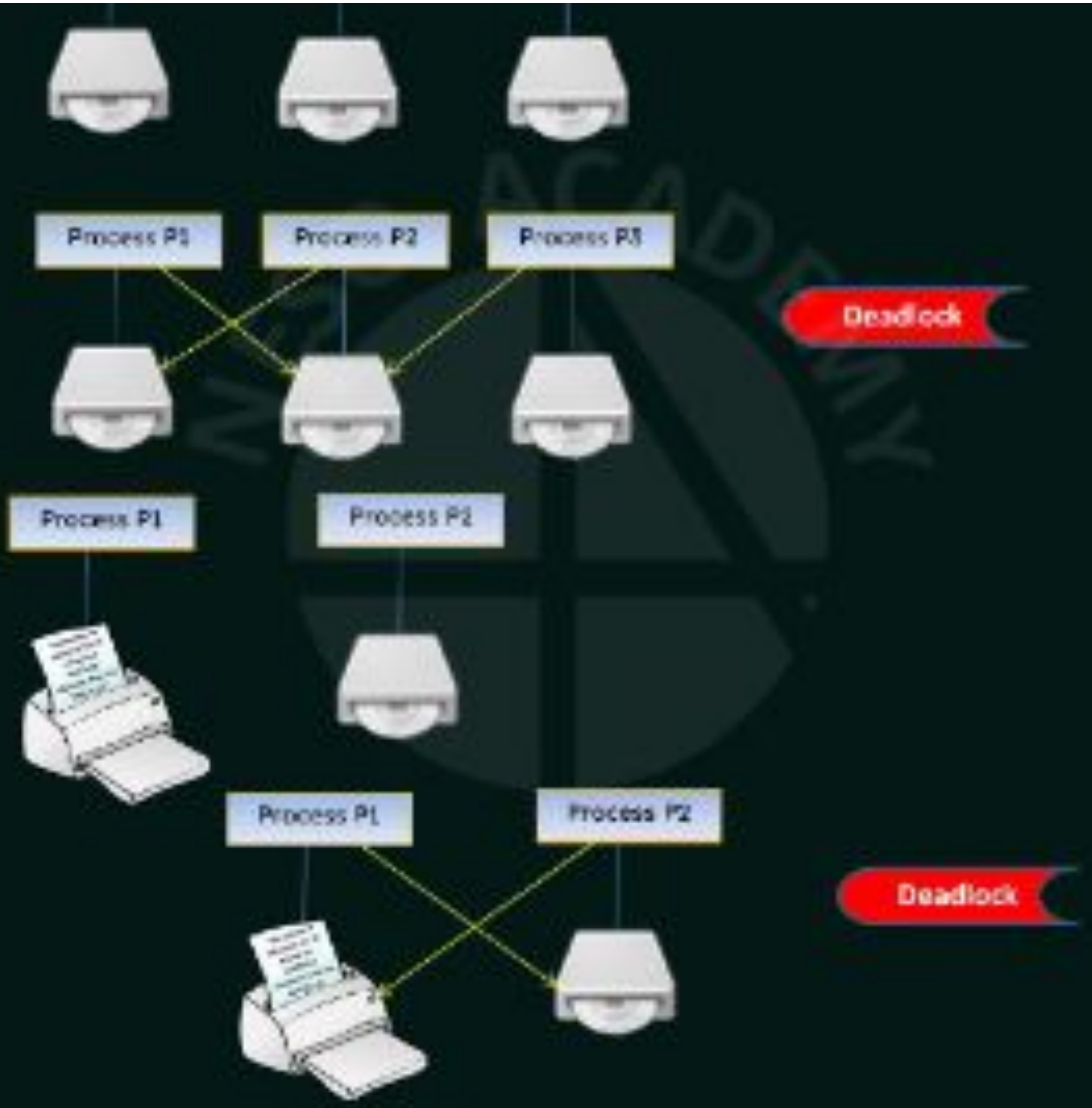
When each philosopher tries to grab his right chopstick, he will be delayed forever.

Several possible remedies to the deadlock problem are the following:

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
- Use an asymmetric solution—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even numbered philosopher picks up her right chopstick and then her left chopstick.

DEADLOCKS

- In a multiprogramming system, processes request resources.
- If those resources are being used by other processes then the process enters a waiting state.
- However, if other processes are also in a waiting state, we have deadlock. The formal definition of deadlock is as follows:
- Definition: A set of processes is in a deadlock state if every process in the set is waiting for an event (release) that can only be caused by some other process in the same set.
- Example 5.1 Process-1 requests the printer, gets it
- Process-2 requests the tape unit, gets it
- Process-1 requests the tape unit, waits
- Process-2 requests the printer, waits
- Process-1 and Process-2 are deadlocked.



Process P1

Process P2

Process P3

Deadlock

Process P1

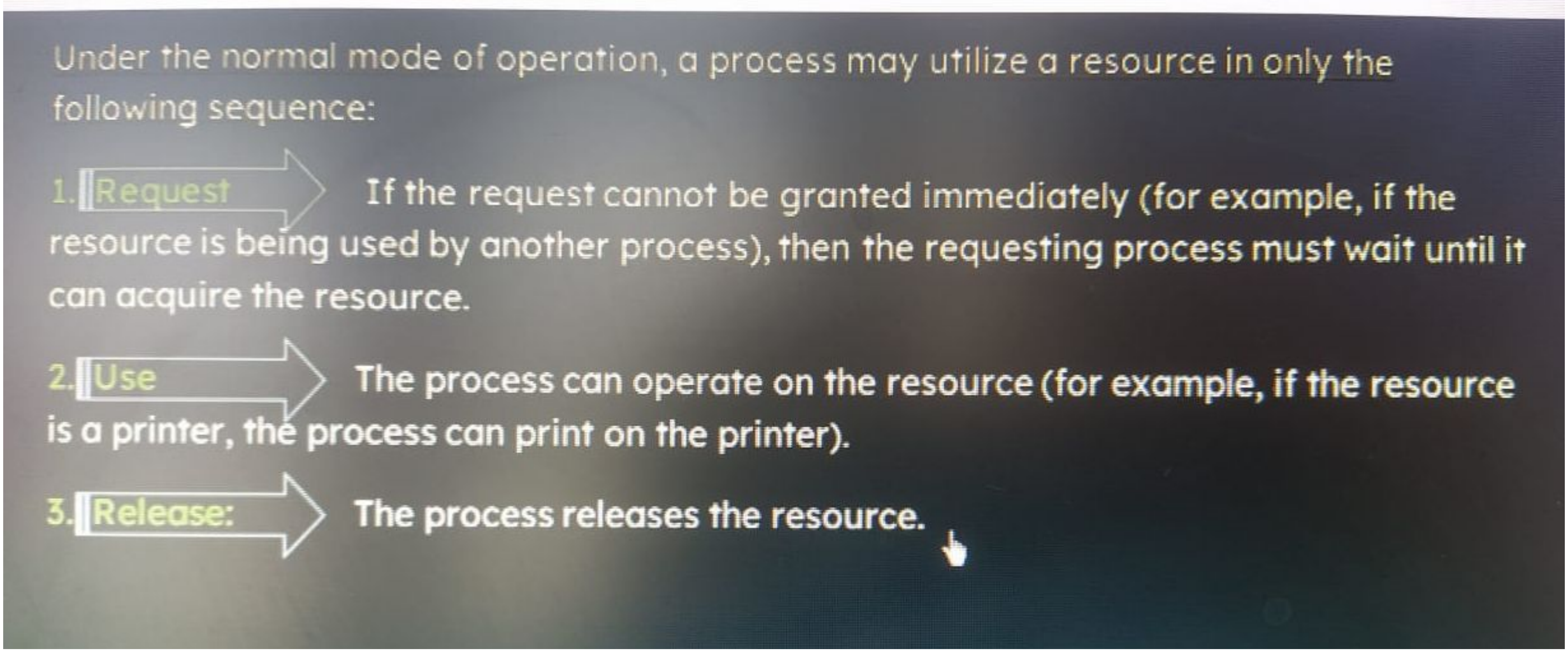
Process P2

Process P1

Process P2

Deadlock

DEADLOCK-SYSTEM MODEL



Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. Request → 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.

DEADLOCK CHARACTERIZATION

Deadlock can arise if four conditions hold simultaneously.

●**Mutual exclusion:** only one process at a time can use a resource.

●**Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes.

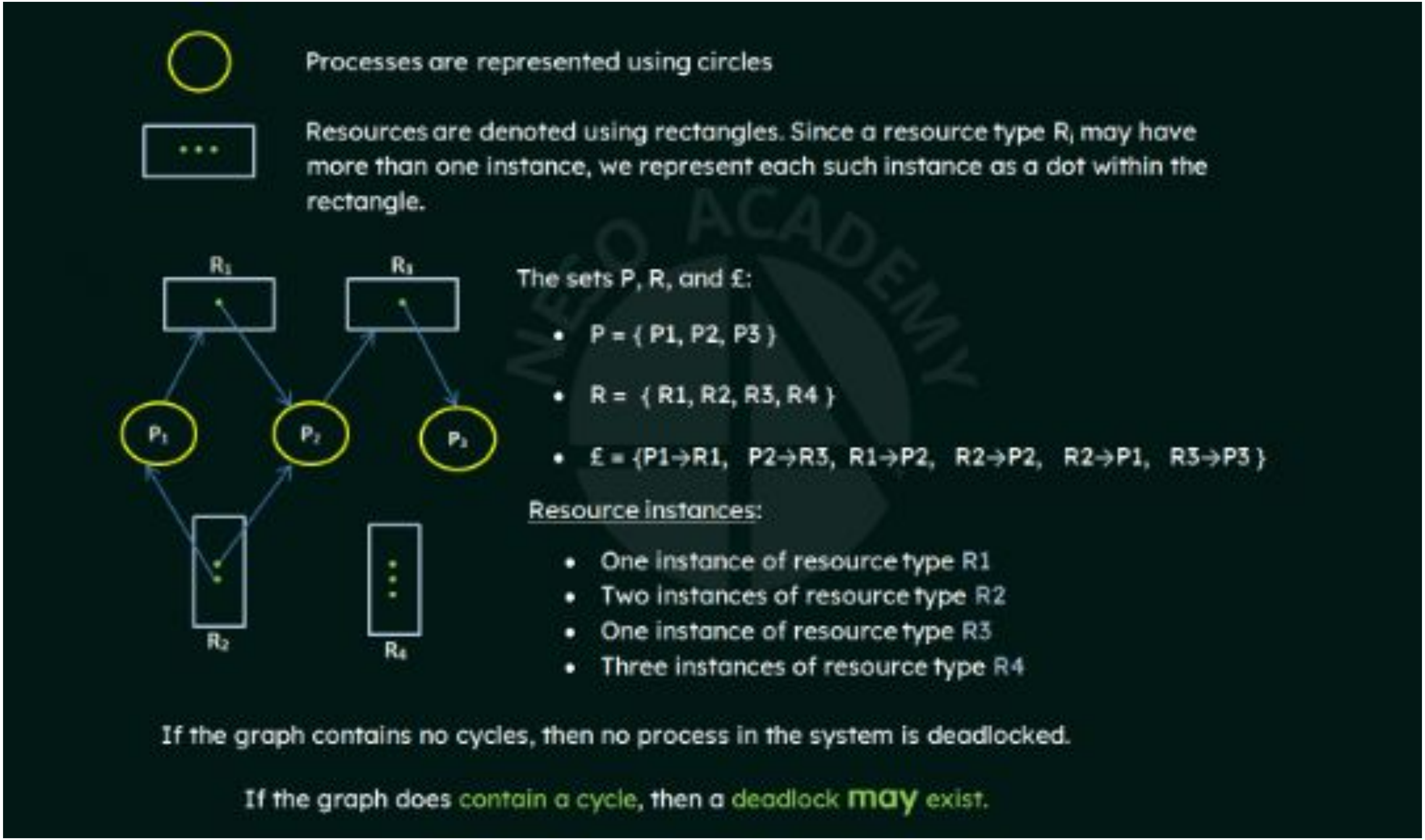
●**No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task.

●**Circular wait:** there exists a set {P_0, P_1, …, P_n} of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

Resource-Allocation Graph

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

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



Processes are represented using circles

Resources are denoted using rectangles. Since a resource type R_i may have more than one instance, we represent each such instance as a dot within the rectangle.

The sets P, R, and £:

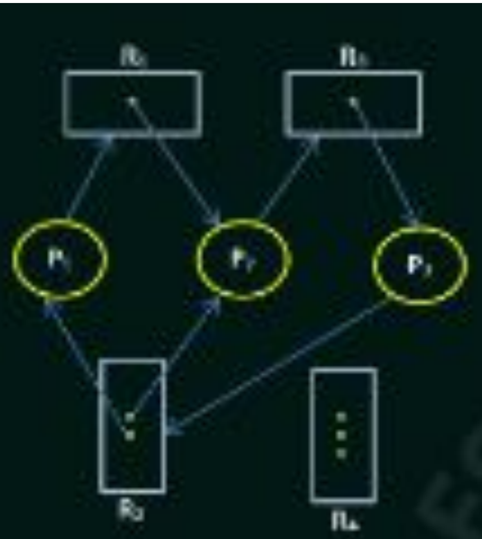
- P = { P1, P2, P3 }
- R = { R1, R2, R3, R4 }
- £ = {P1→R1, P2→R3, R1→P2, R2→P2, R2→P1, R3→P3 }

Resource instances:

- One instance of resource type R1
- Two instances of resource type R2
- One instance of resource type R3
- Three instances of resource type R4

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.

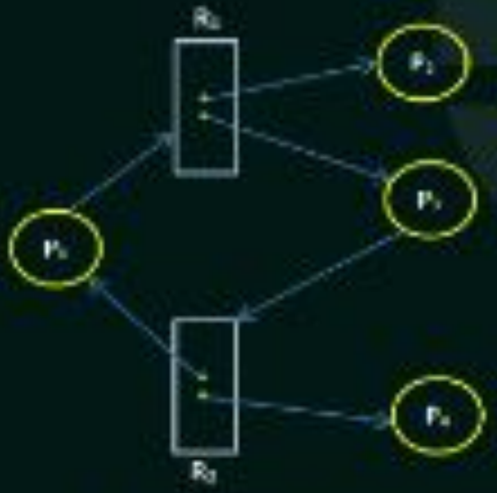


At this point, two minimal cycles exist in the system:

P1 → R1 → P2 → R3 → P3 → R2 → P1

P2 → R3 → P3 → R2 → P2

Processes P1, P2, and P3 are deadlocked.



In this example also we have a cycle:

P1 → R1 → P3 → R2 → P1

However, there is no deadlock.

Observe that process P4 may release its
instance of resource type R2.
That resource can then be allocated
to P3, breaking the cycle.

If a resource-allocation graph does not have a cycle, then the system is not in a deadlocked state.

If there is a cycle, then the system may or may not be in a deadlocked state.

Methods for Handling Deadlocks

- Ensure that the system will ***never*** enter a deadlock state:
- Deadlock prevention
- Deadlock avoidance
- 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:

1. **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources.
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, or allow process to request resources only when the process has none allocated to it.
- Low resource utilization; starvation possible

Deadlock Prevention

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

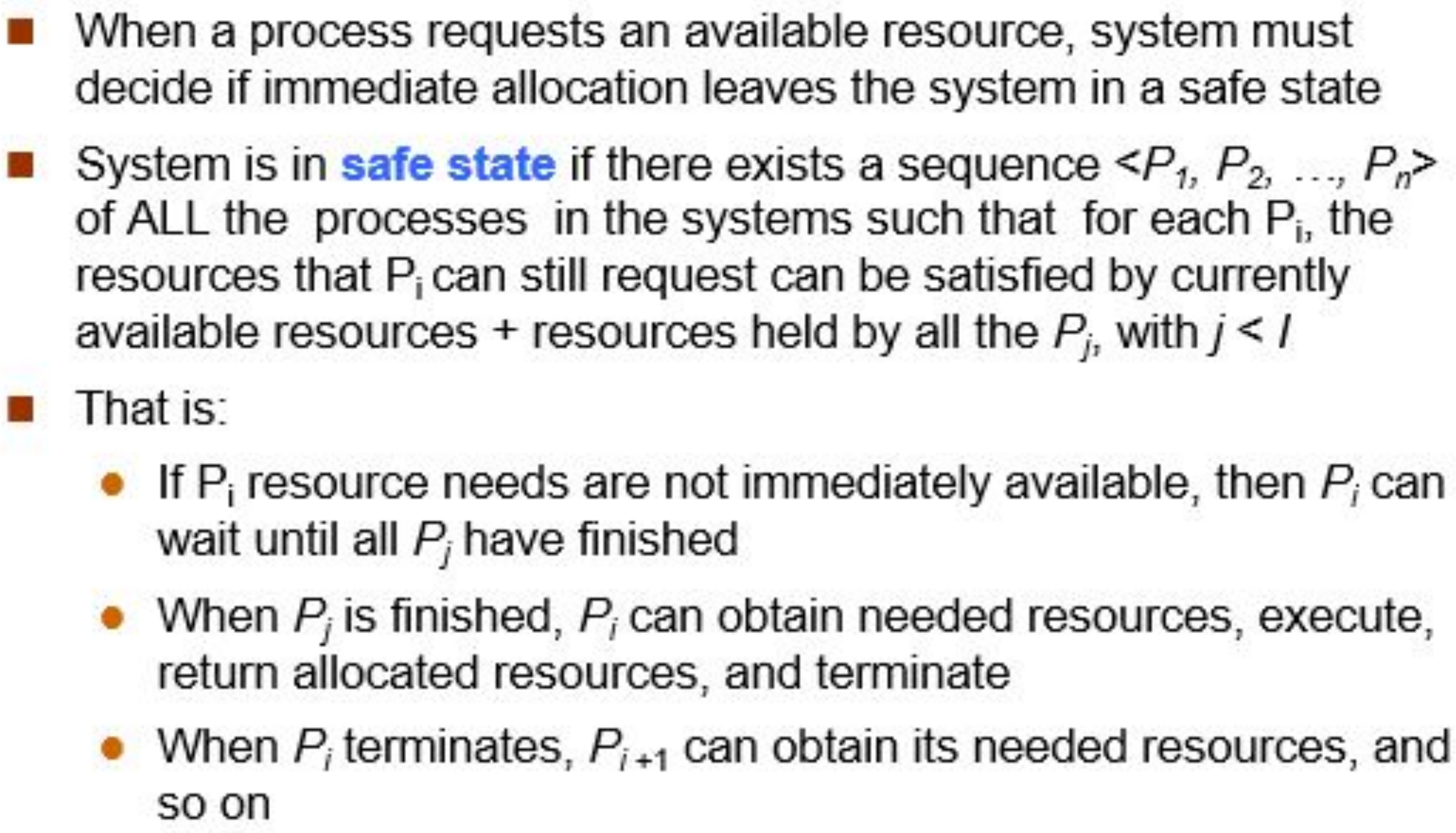
4. Circular Wait – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Deadlock Avoidance

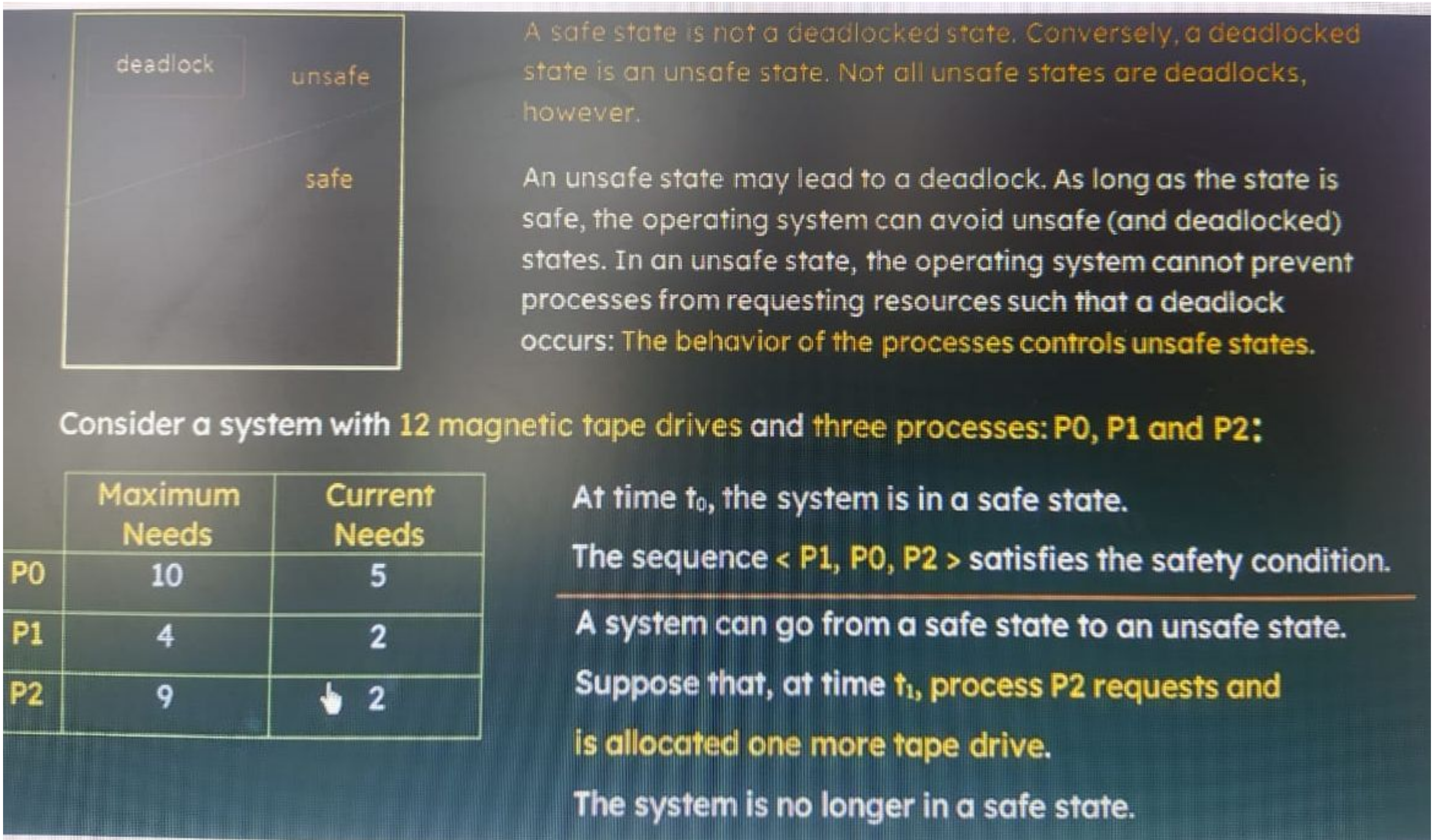
Requires that the system has some additional ***a priori*** information available

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

Safe State



- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < I$
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on


deadlock
unsafe
safe

A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. 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. In an unsafe state, the operating system cannot prevent processes from requesting resources such that a deadlock occurs: The behavior of the processes controls unsafe states.

Consider a system with 12 magnetic tape drives and three processes: P0, P1 and P2:

	Maximum Needs	Current Needs
P0	10	5
P1	4	2
P2	9	2

At time t_0, the system is in a safe state.

The sequence < P1, P0, P2 > satisfies the safety condition.

A system can go from a safe state to an unsafe state.

Suppose that, at time t_1, process P2 requests and is allocated one more tape drive.

The system is no longer in a safe state.

Resource-Allocation Graph Scheme

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

Resource-Allocation Graph Scheme

Suppose that process P_i requests a resource R_j

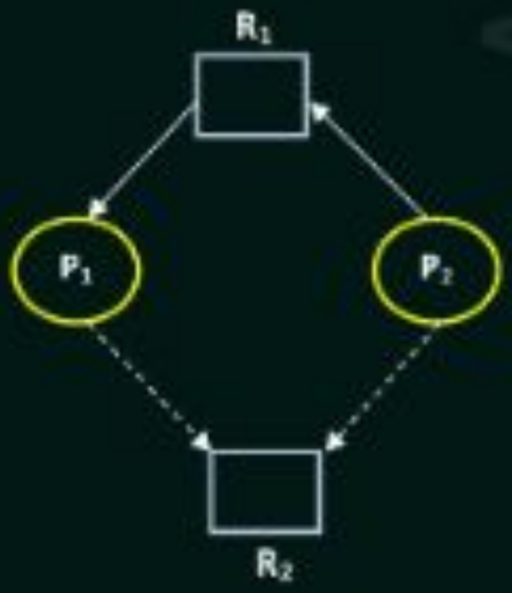
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph.
- If no cycles exists, then allocation of resource will leave the system in a safe state.
- If cycle exists, then allocation of resource will leave the system in an unsafe state.
- **Always remember that Resource Allocation Graph works only when then there is Single Instance of Resource available.**

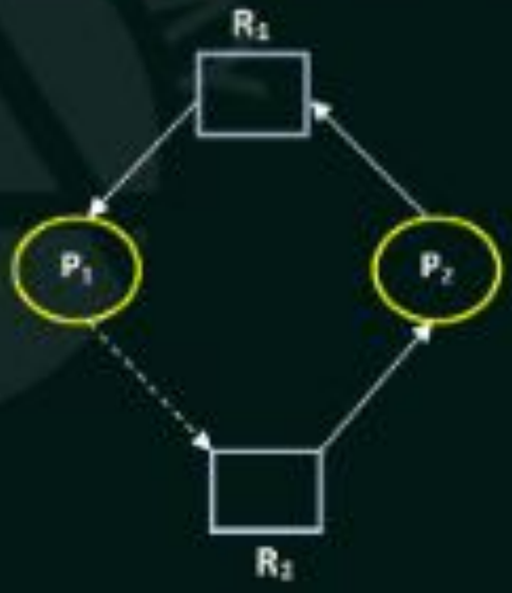
Suppose that process Pi requests resource Rj. The request can be granted only if converting the request edge Pi → Rj to an assignment edge Rj → Pi does not result in the formation of a cycle in the resource-allocation graph.

Note that we check for safety by using a cycle-detection algorithm.

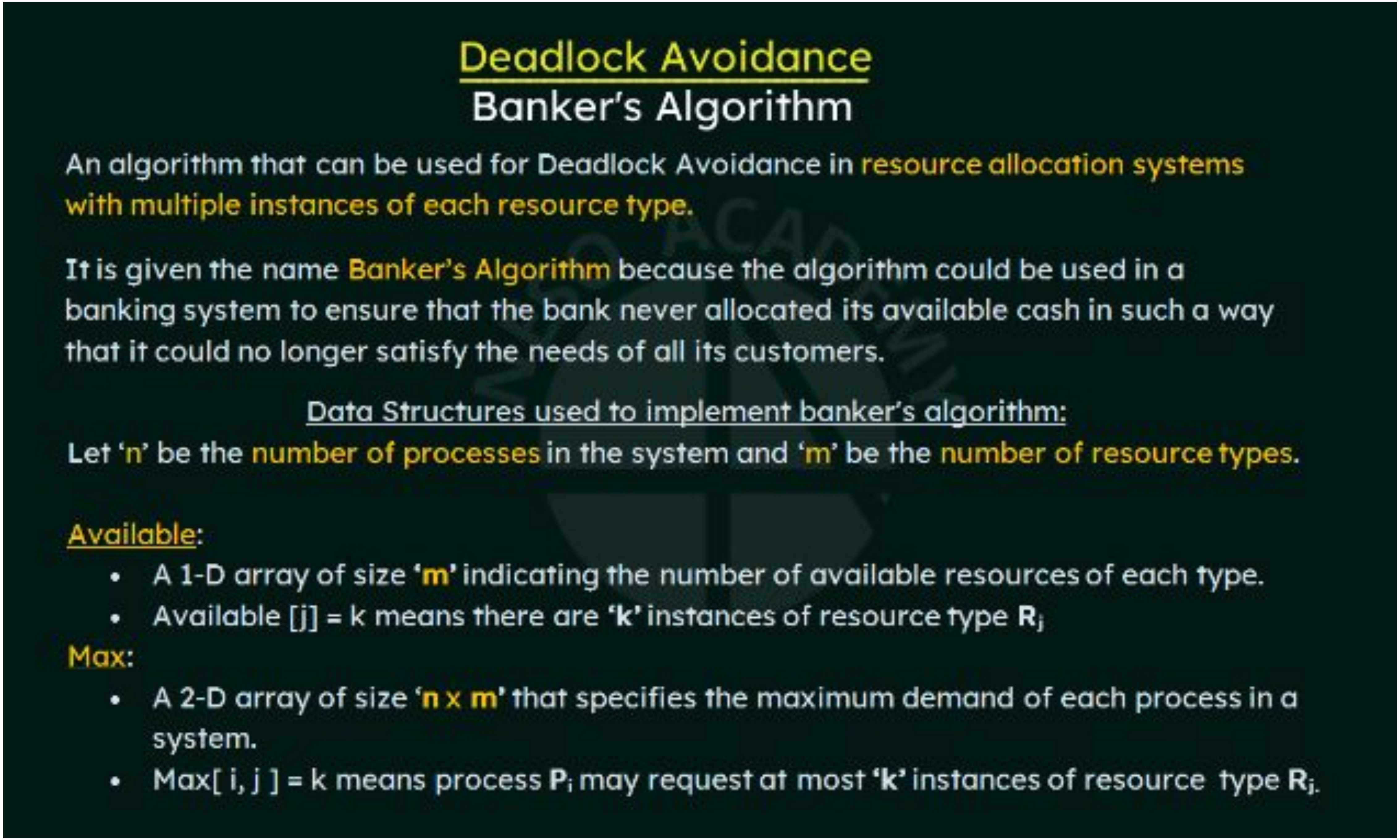
If no cycle exists, then the allocation of the resource will leave the system in a safe state.

If a cycle is found, then the allocation will put the system in an unsafe state.





Unsafe State



Deadlock Avoidance

Banker's Algorithm

An algorithm that can be used for Deadlock Avoidance in resource allocation systems with multiple instances of each resource type.

It is given the name Banker's Algorithm because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.

Data Structures used to implement banker's algorithm:

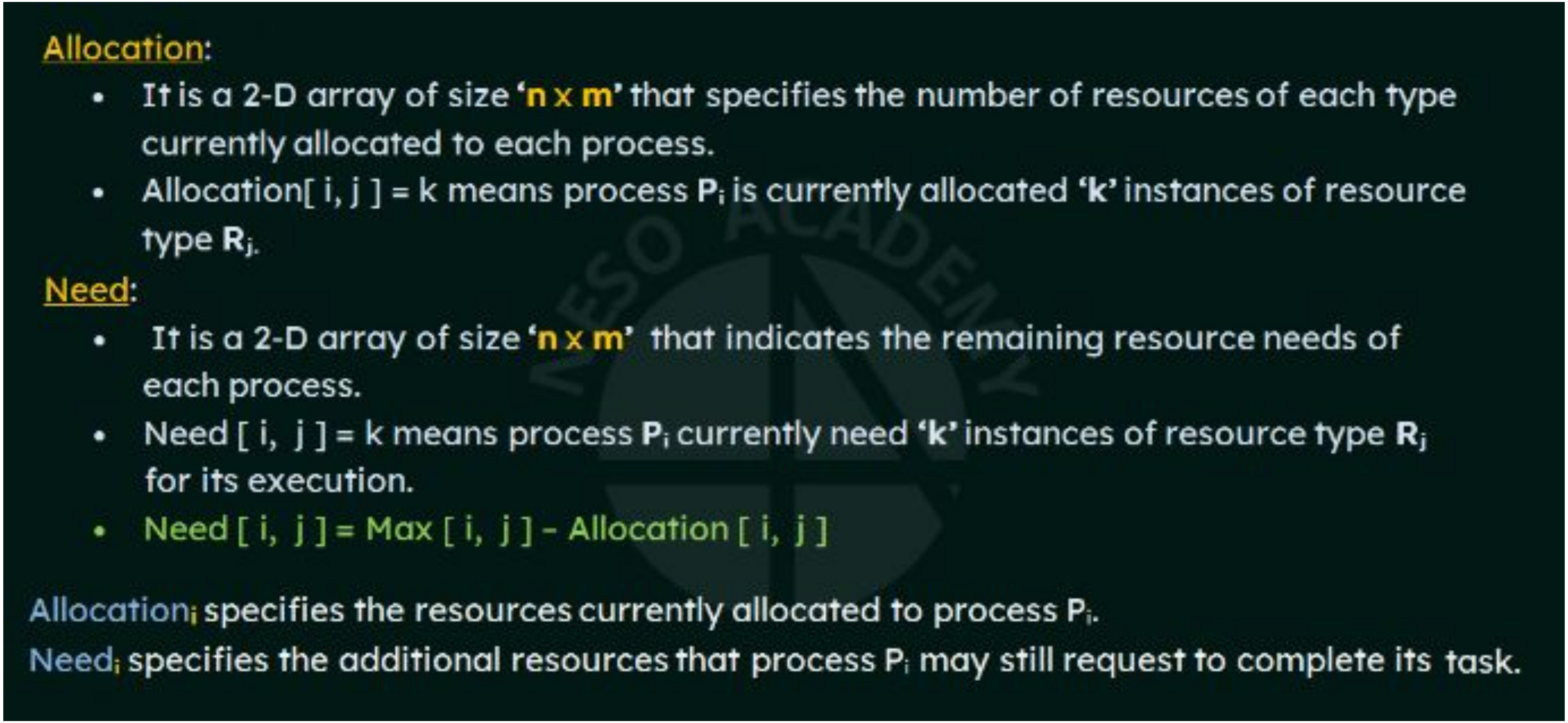
Let 'n' be the number of processes in the system and 'm' be the number of resource types.

Available:

- A 1-D array of size **'m'** indicating the number of available resources of each type.
- Available [j] = k means there are **'k'** instances of resource type $\mathbf{R}_j$

Max:

- A 2-D array of size '**n x m**' that specifies the maximum demand of each process in a system.
- Max[i, j] = k means process $\mathbf{P}_i$ may request at most **'k'** instances of resource type $\mathbf{R}_j$.



Allocation:

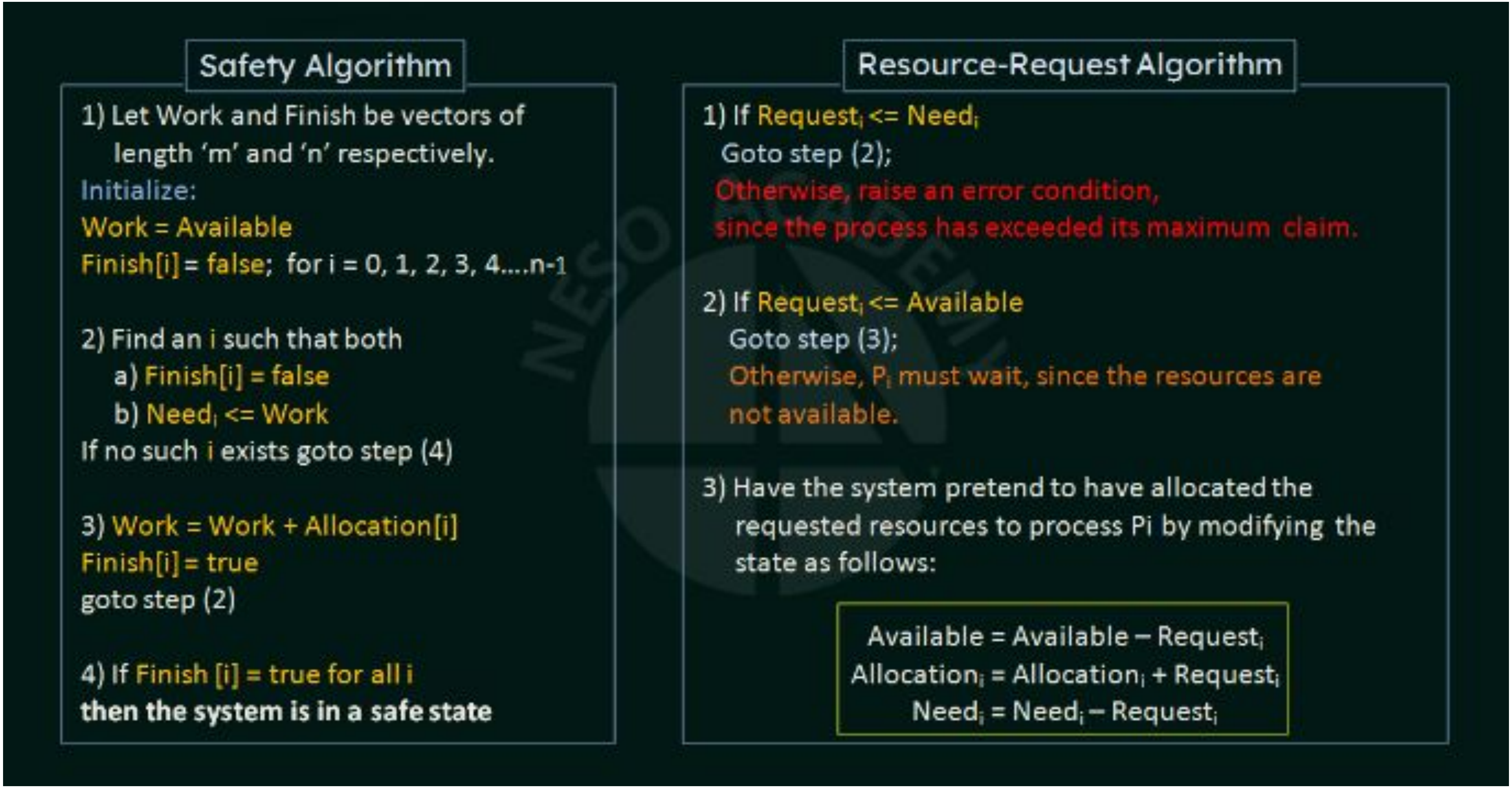
- It is a 2-D array of size **‘n x m’** that specifies the number of resources of each type currently allocated to each process.
- Allocation[i, j] = k means process **P_i** is currently allocated **‘k’** instances of resource type **R_j**.

Need:

- It is a 2-D array of size **‘n x m’** that indicates the remaining resource needs of each process.
- Need [i, j] = k means process **P_i** currently need **‘k’** instances of resource type **R_j** for its execution.
- Need [i, j] = Max [i, j] - Allocation [i, j]

Allocation_i specifies the resources currently allocated to process P_i.

Need_i specifies the additional resources that process P_i may still request to complete its task.



Safety Algorithm

```
1) Let Work and Finish be vectors of
   length ‘m’ and ‘n’ respectively.
Initialize:
Work = Available
Finish[i] = false;  for i = 0, 1, 2, 3, 4….n-1

2) Find an i such that both
   a) Finish[i] = false
   b) $Need_i$ <= Work
If no such i exists goto step (4)

3) Work = Work + Allocation[i]
Finish[i] = true
goto step (2)

4) If Finish [i] = true for all i
then the system is in a safe state
```

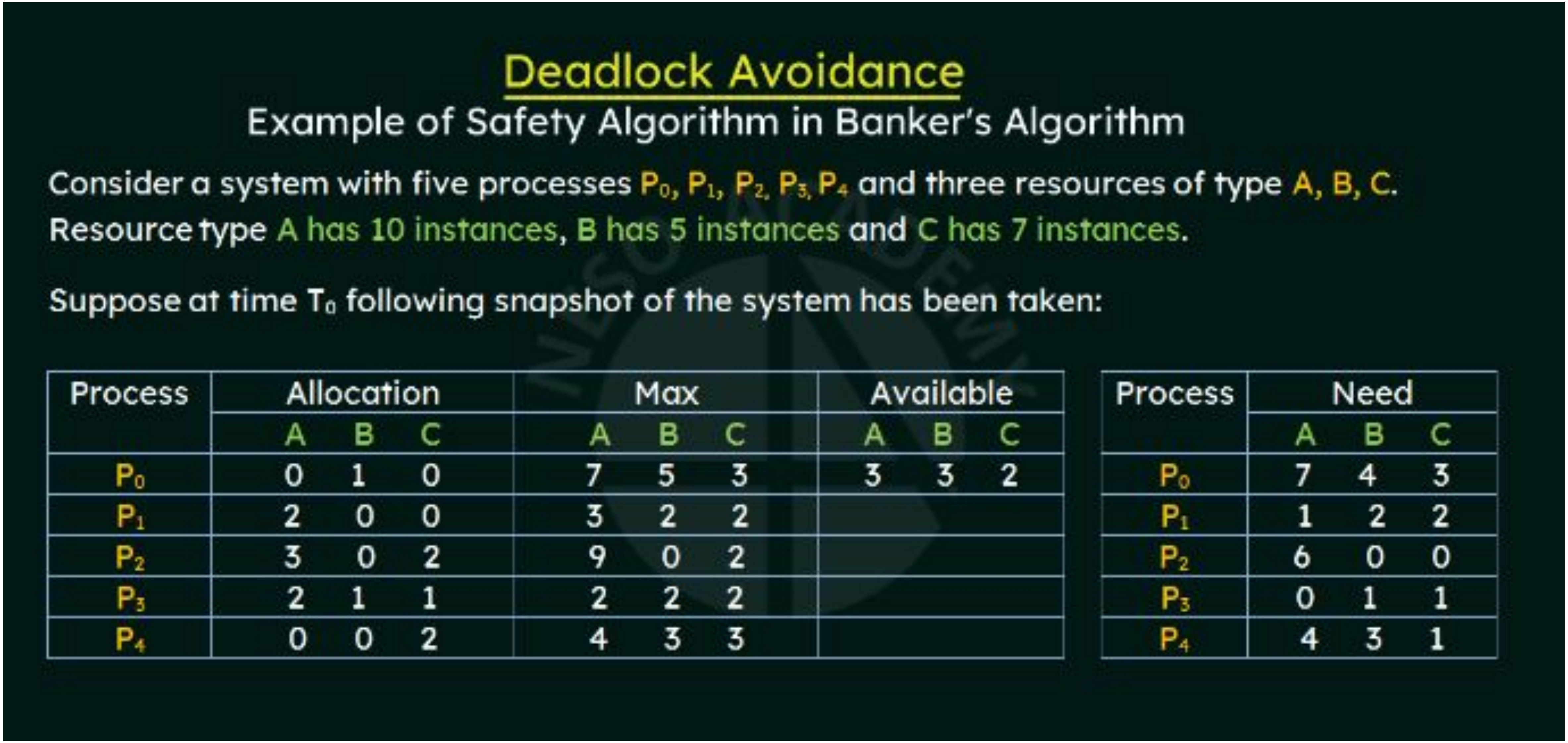
Resource-Request Algorithm

```
1) If $Request_i$ <= $Need_i$
   Goto step (2);
   Otherwise, raise an error condition,
   since the process has exceeded its maximum claim.

2) If $Request_i$ <= Available
   Goto step (3);
   Otherwise, $P_i$ must wait, since the resources are
   not available.

3) Have the system pretend to have allocated the
   requested resources to process Pi by modifying the
   state as follows:

      Available = Available – $Request_i$
      $Allocation_i$ = $Allocation_i$ + $Request_i$
      $Need_i$ = $Need_i$ – $Request_i$
```



Deadlock Avoidance

Example of Safety Algorithm in Banker's Algorithm

Consider a system with five processes P_0, P_1, P_2, P_3, P_4 and three resources of type A, B, C. Resource type A has 10 instances, B has 5 instances and C has 7 instances.

Suppose at time T_0 following snapshot of the system has been taken:

Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

Process	Need		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

Step 1	m=3, n=5				
Work = Available					
Work =	3		3		2
Process	0	1	2	3	4
Finish =	false	false	false	false	false

Step 2	P_0 For i = 0		
$Need_0$ =	7	4	3
7,4,3 > 3,3,2			
Finish [0] = false and $Need_0$ > Work			
P_0 must wait			

Step 2	P_1 For i = 1		
$Need_1$ =	1	2	2
1,2,2 < 3,3,2			
Finish [1] = false and $Need_1$ < Work			
P_1 can be kept in safe sequence			

Step 3	P_1				
Work =	Work + Allocation				
	3,3,2 + 2,0,0				
	5		3		2
Process	0	1	2	3	4
Finish =	false	true	false	false	false

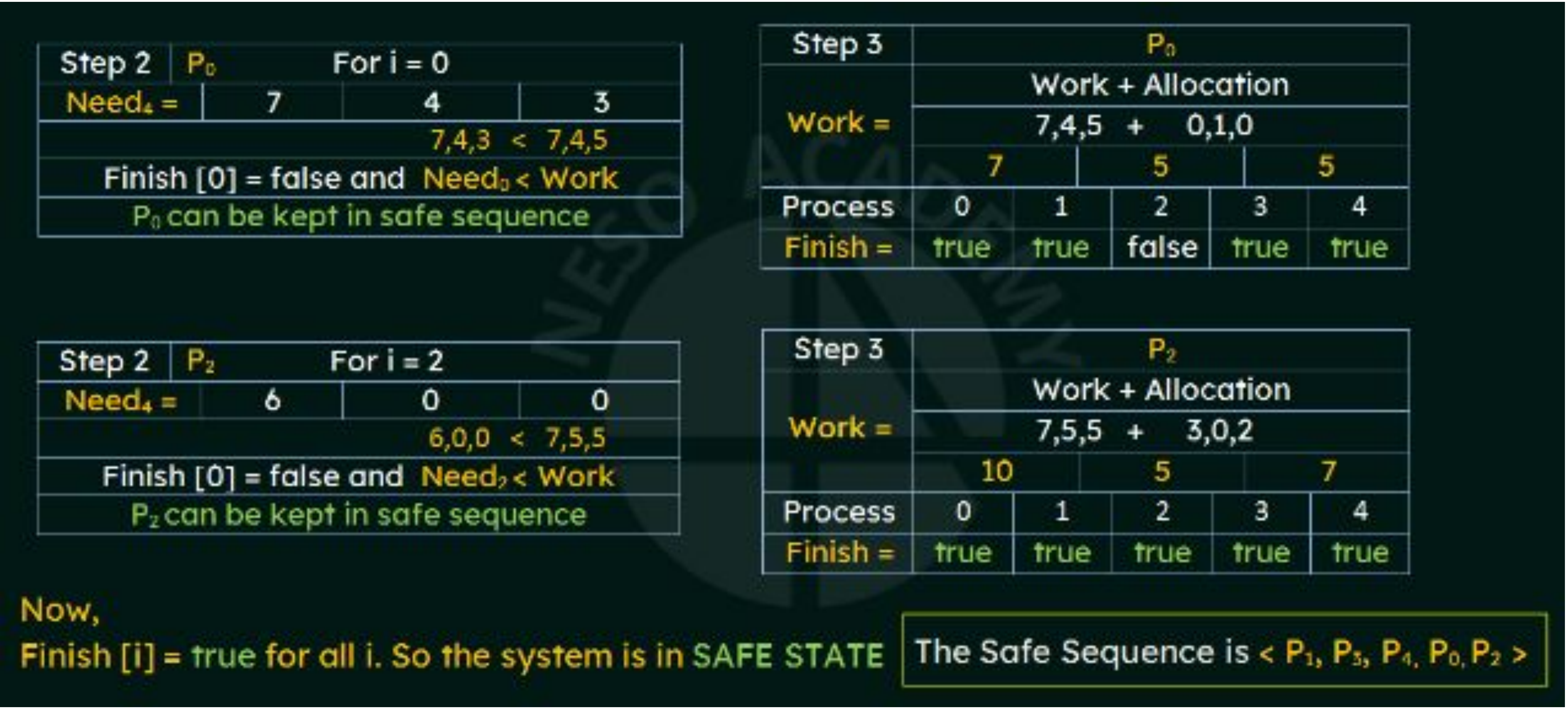
Step 2	P_2	For i = 2	
$Need_2$ =	6	0	0
		6,0,0 > 5,3,2	
Finish [2] = false and $Need_2$ > Work			
P_2 must wait			

Step 2	P_3	For i = 3	
$Need_3$ =	0	1	1
		0,1,1 < 5,3,2	
Finish [3] = false and $Need_3$ < Work			
P_3 can be kept in safe sequence			

Step 2	P_4	For i = 4	
$Need_4$ =	4	3	1
		4,3,1 < 7,4,3	
Finish [4] = false and $Need_4$ < Work			
P_4 can be kept in safe sequence			

Step 3	P_3				
Work =	Work + Allocation				
	5,3,2 + 2,1,1				
	7		4		3
Process	0	1	2	3	4
Finish =	false	true	false	true	false

Step 3	P_4				
Work =	Work + Allocation				
	7,4,3 + 0,0,2				
	7		4		5
Process	0	1	2	3	4
Finish =	false	true	false	true	true



Step 2	P_0	For i = 0	
$Need_4$ =	7	4	3
		7,4,3 < 7,4,5	
Finish [0] = false and $Need_0$ < Work			
P_0 can be kept in safe sequence			

Step 3	P_0				
Work =	Work + Allocation				
	7,4,5 + 0,1,0				
	7		5		5
Process	0	1	2	3	4
Finish =	true	true	false	true	true

Step 2	P_2	For i = 2	
$Need_4$ =	6	0	0
		6,0,0 < 7,5,5	
Finish [0] = false and $Need_2$ < Work			
P_2 can be kept in safe sequence			

Step 3	P_2				
Work =	Work + Allocation				
	7,5,5 + 3,0,2				
	10		5		7
Process	0	1	2	3	4
Finish =	true	true	true	true	true

Now,

Finish [i] = true for all i. So the system is in SAFE STATE

The Safe Sequence is < P_1, P_3, P_4, P_0, P_2 >

Deadlock Avoidance

Example of Resource-Request Algorithm in Banker's Algorithm

Suppose now that process **P_1** requests one additional instance of resource type A and two instances of resource type C, so $Request_1 = (1,0,2)$.

Can this request be immediately granted ?

Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

Process	Need		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

Step 1	Check condition
	1, 0, 2 <= 1, 2, 2
	Request_1 <= Need_1
	Condition Satisfied

Step 2	Check condition
	1, 0, 2 <= 3, 3, 2
	Request_1 <= Available
	Condition Satisfied

Step 3

$\text{Available} = \text{Available} - \text{Request}_1$

$\text{Allocation}_1 = \text{Allocation}_1 + \text{Request}_1$

$\text{Need}_1 = \text{Need}_1 - \text{Request}_1$

Process	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	2	3	0
P_1	3	0	2	0	2	0			
P_2	3	0	2	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

We must determine whether this new system state is safe. To do so, we execute our safety algorithm again and try to find if this system state is safe and if it is safe what is the safe sequence.

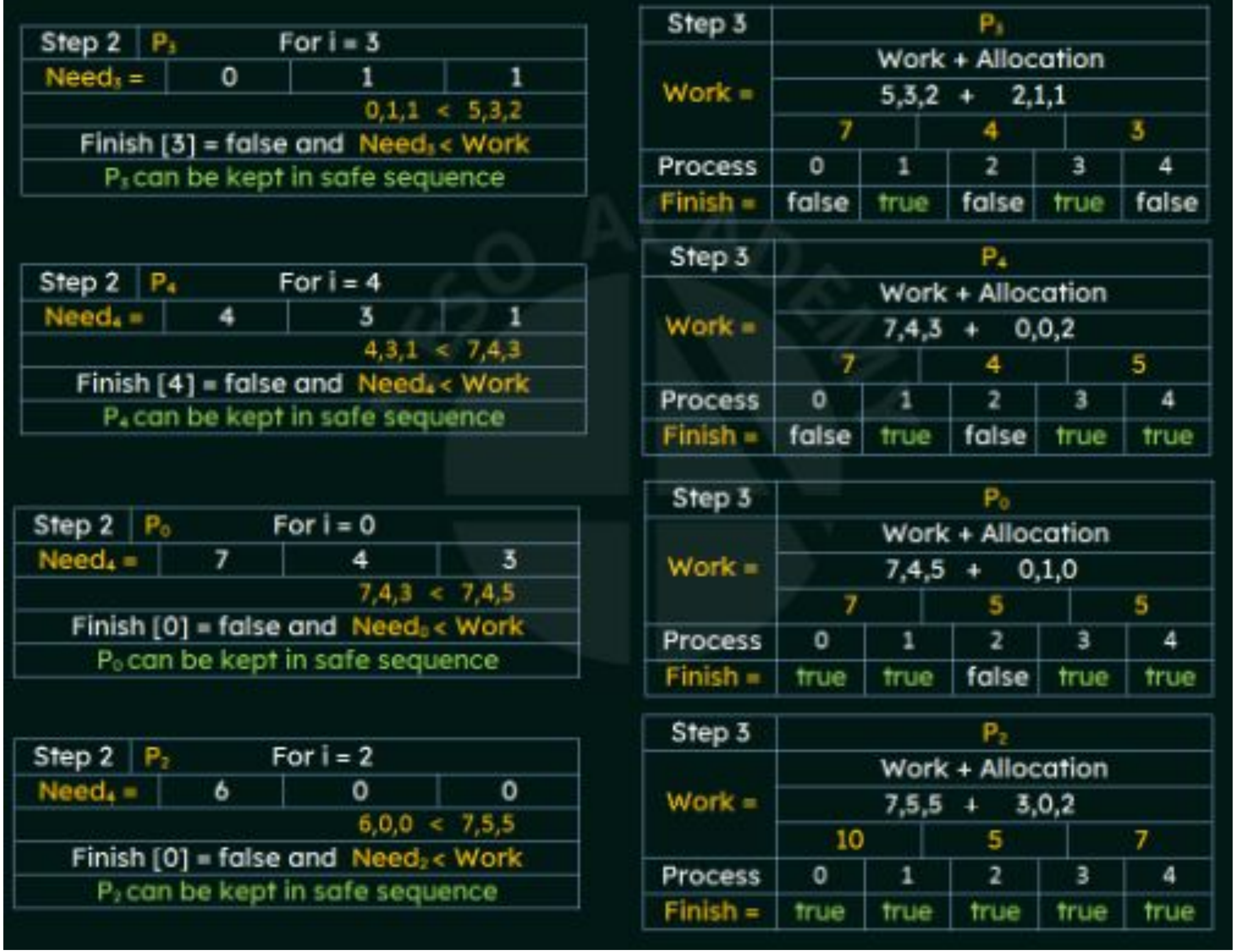
Step 1	m=3, n=5				
Work = Available					
Work =	2		3		0
Process	0	1	2	3	4
Finish =	false	false	false	false	false

Step 2	P_0	For i = 0	
$Need_0$ =	7	4	3
		7,4,3 > 2,3,0	
Finish [0] = false and $Need_0$ > Work			
P_0 must wait			

Step 2	P_1	For i = 1	
$Need_1$ =	0	2	0
		0,2,2 < 2,3,0	
Finish [1] = false and $Need_1$ < Work			
P_1 can be kept in safe sequence			

Step 2	P_2	For i = 2	
$Need_2$ =	6	0	0
		6,0,0 > 5,3,2	
Finish [2] = false and $Need_2$ > Work			
P_2 must wait			

Step 3	P_1				
Work =	Work + Allocation				
	2,3,0 + 3,0,2				
	5		3		2
Process	0	1	2	3	4
Finish =	false	true	false	false	false



Step 2	P_3	For i = 3	
Need$_3$ =	0	1	1
		0,1,1 < 5,3,2	
Finish [3] = false and Need$_3$ < Work			
P_3 can be kept in safe sequence			

Step 2	P_4	For i = 4	
Need$_4$ =	4	3	1
		4,3,1 < 7,4,3	
Finish [4] = false and Need$_4$ < Work			
P_4 can be kept in safe sequence			

Step 2	P_0	For i = 0	
Need$_4$ =	7	4	3
		7,4,3 < 7,4,5	
Finish [0] = false and Need$_0$ < Work			
P_0 can be kept in safe sequence			

Step 2	P_2	For i = 2	
Need$_4$ =	6	0	0
		6,0,0 < 7,5,5	
Finish [0] = false and Need$_2$ < Work			
P_2 can be kept in safe sequence			

Step 3	P_3				
Work =	Work + Allocation				
	5,3,2 + 2,1,1				
	7		4		3
Process	0	1	2	3	4
Finish =	false	true	false	true	false

Step 3	P_4				
Work =	Work + Allocation				
	7,4,3 + 0,0,2				
	7		4		5
Process	0	1	2	3	4
Finish =	false	true	false	true	true

Step 3	P_0				
Work =	Work + Allocation				
	7,4,5 + 0,1,0				
	7		5		5
Process	0	1	2	3	4
Finish =	true	true	false	true	true

Step 3	P_2				
Work =	Work + Allocation				
	7,5,5 + 3,0,2				
	10		5		7
Process	0	1	2	3	4
Finish =	true	true	true	true	true

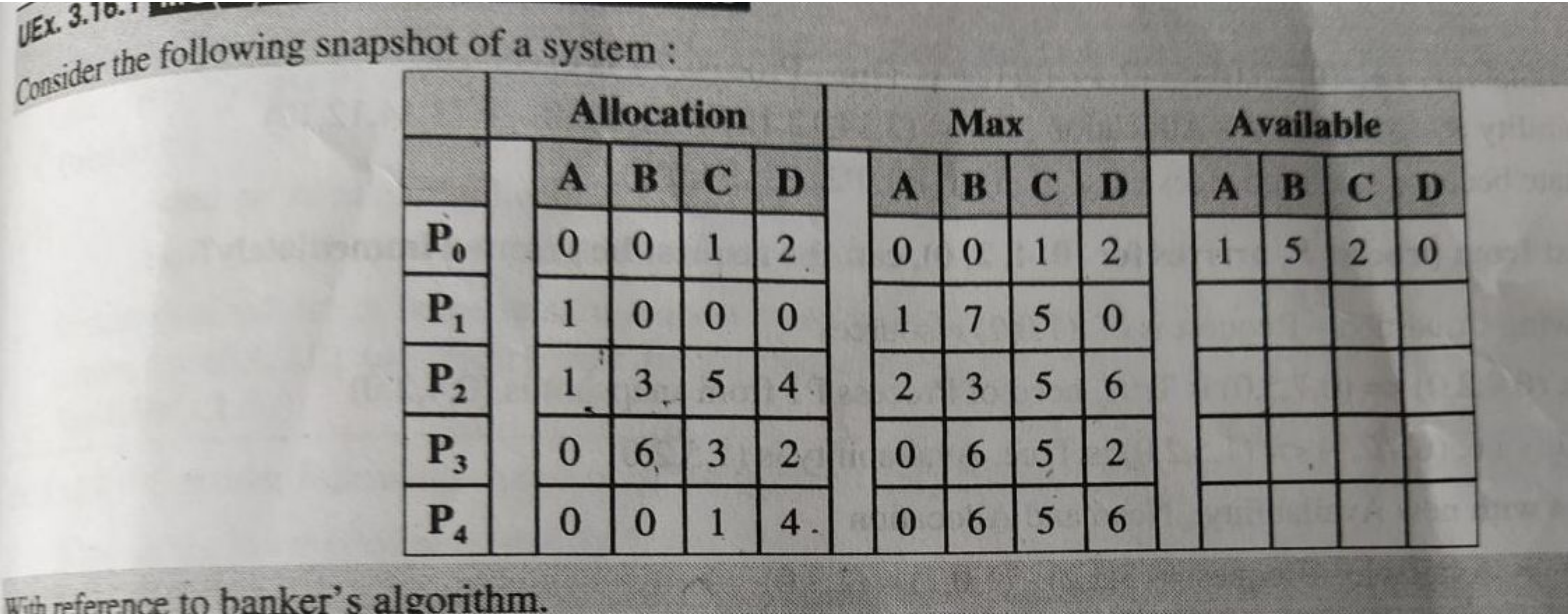
Now,

Finish [i] = true for all i. So the system is in SAFE STATE

The Safe Sequence is < P_1, P_3, P_4, P_0, P_2 >

So, by applying the Resource-Request Algorithm and by checking the state of the system using the Safety Algorithm, we find that granting the request of Process P_1 still keeps the system in a safe state and hence will not lead to a deadlock.

Hence, we can immediately grant the request of process P_1.



UEx. 3.16.1

Consider the following snapshot of a system :

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	1	5	2	0
P_1	1	0	0	0	1	7	5	0				
P_2	1	3	5	4	2	3	5	6				
P_3	0	6	3	2	0	6	5	2				
P_4	0	0	1	4	0	6	5	6				

With reference to banker's algorithm.

With reference to Banker's Algorithm:

i) Find the Need Matrix

ii) Is the system in a Safe State?

iii) If a request from process P1 arrives for (0,4,2,0) can the request be granted immediately.

Q. Consider the following snapshot of a system:

Process	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	0	0	1	2	0	0	1	2	1	5	2	0
P1	1	0	0	0	1	7	5	0				
P2	1	3	5	4	2	3	5	6				
P3	0	6	3	2	0	6	5	2				
P4	0	0	1	4	0	6	5	6				

With reference to Banker's Algorithm: Find the need matrix, Is the system is in a Safe State. & If a request from process P1 arrives for (0, 4, 2, 0) can it be granted immediately?

Solution step 1 | m = 4 n = 5

Work = Available

Work =	1	5	2	0
Process = 0	1	2	3	4
Finish = F	F	F	F	F

Need = Max − Allocation

Need

	A	B	C	D
P0	0	0	0	0
P1	0	7	5	0
P2	1	0	0	2
P3	0	0	2	0
P4	0	6	4	2

Q. Step 2 | P0 For i = 0

Need	0	0	0	0

0,0,0,0 < 1,5,2,0

Finish [0] = False & $Need_0 < Work$

P0 can be in safe sequence

Step 3 P_0

Work = Work + Allocation

= 1520 + 0012 = 1532

Process	0	1	2	3	4
Finish	T	F	F	F	F

Step 2 | P1 For i = 1

Need	0	7	5	0

0750 > 1532

Finish [1] = False Need1 > Work

P1 has to wait

Step 2 | P2 for i = 2

Need	1	0	0	2

1 0 02 < 1532

Finish [2] = False $Need_2 < Work$

P2 can be in safe sequence

Step 3 P2

Work = Work + Allocation

= 1532 + 1354 = 2886

Process	0	1	2	3	4
Finish	T	F	T	F	F

For P3 For i = 3

Need	0	0	2	0

0 0 2 0 < 2886

Finish [3] = False $Need_3 < Work$

P 3 can be in safe sequence

Step 3 P3

Work = Work + Allocation

= 2886 + 0632 = 2 14 11 8

Process	0	1	2	3	4
Finish	T	F	T	T	F

Step 2 P4 For i = 4

Need	0	6	4	2

0 642 < 2 14 11 8

Finish [4] = False $Need_4 < Work$

P4 can be in sequence

Step 3 P4

Work = Work + Allocation

= 2 14 11 8 + 0 014 = 2 14 12 12

Process	0	1	2	3	4
Finish	T	F	T	T	T

Step 1 For i = 1

Need	0	7	5	0

2 14 12 12

0 7 5 0 < ~~2 14 12 12~~

P1 can be in safe state

step2 P1

Work = Work + Allocation

= ~~2 14 11 8 + 0 0 1~~

= 2 14 12 12 + 1 0 0 0

= 3 14 12 12

Process	0	1	2	3	4
Finish	T	T	T	T	T

So safe sequence is P0, P2, P3, P4, P1

For request from P1 for (0, 4, 2, 0)

Step 1 For i = 1

~~Need | 0 | 7 | 5 | 0~~

step1 = check condition

Request ≤ Need

0,4,2,0 ≤ 0,7,5,0

Condition satisfied

step2 = check condition

request ≤ Available

0,4,2,0 ≤ 1,5,2,0

Condition satisfied

Update Values =

Available = Available − Request = 1,5,2,0 − 0,4,2,0 = 1,1,0,0

Allocation = Allocation + Request = 1,0,0,0 + 0,4,2,0 = 1,4,2,0

Need = Need − Request = 0,7,5,0 − 0,4,2,0 = 0,3,3,0

	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P0	0	0	1	2	2	0	0	1	1	1	0	0	0	0	0	0
P1	1	4	2	0	1	7	5	0					0	3	3	0
P2	1	3	5	4	2	3	3	6					1	0	0	2
P3	0	6	3	2	0	6	3	2					0	0	2	0
P4	0	0	1	4	0	6	5	6					0	6	4	2

step 2 | P0 For i = 0

Need	0	0	0	0

0,0,0,0 < 1,1,0,0

Finish[0] = False & $Need_0$ < Work

P0 can be in safe sequence

step3 P0

Work = Work + Allocation

= ~~1520~~ 1100 + 0012

= 1112

Process	0	1	2	3	F
Finish	T	F	F	F	4

step 2	P1 i=1			
Need	0	3	3	0

$0,3,3,0 \not< 1,1,1,2$

P1 has to wait

Step 2	For i=2			
Need	1	0	0	2

$1,0,0,2 < 1,1,1,2$

P2 is in safe sequence

Work = Work + Allocation
= 1,1,1,2 + 1,3,5,4
= 2,4,6 6

Process	P0	1	2	3	4
Finish	T	f	T	F	f

Step3	For i=3			
Need	0	0	2	0

$0,0,2,0 < 2,4,6,6$

P3 can be in safe sequence

Work = Work + Allocation
= 2,4,6,6 + 0,6,3 2 = 2,10,9,8

Process	0	1	2	3	4
Finish	T	F	T	T	F

Step 4	For i=4			
Need	0	6.	4	2

$0,6,4,2 < 2,10,9,8$

P4 can be in safe sequence

Work = Work + Allocation
2,10,9,8 + 0,0,14 = 2,10,10,12

Process	0	1	2	3	4
Finish	T	F	T	T	T

Step 5	For i=1			
Need	0	3	3	0

$0,3,3,0 < 2,10,10,12$

P1 can be in safe sequence

Work = Work + Allocation
2,10,10,12 + 1,4,2,0=
= 3,14,12,12

Process	0	1	2	3	4
Finish	T	T	T	T	T

So safe sequence is P0, P2, P3, P4, P1

Recovery from Deadlock: Process Termination

- **Abort all deadlocked processes**
- **Abort one process at a time until the deadlock cycle is eliminated**
- **In which order should we choose to abort?**

1. Priority of the process
2. How long process has computed, and how much longer to completion
3. Resources the process has used
4. Resources process needs to complete
5. How many processes will need to be terminated
6. Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

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