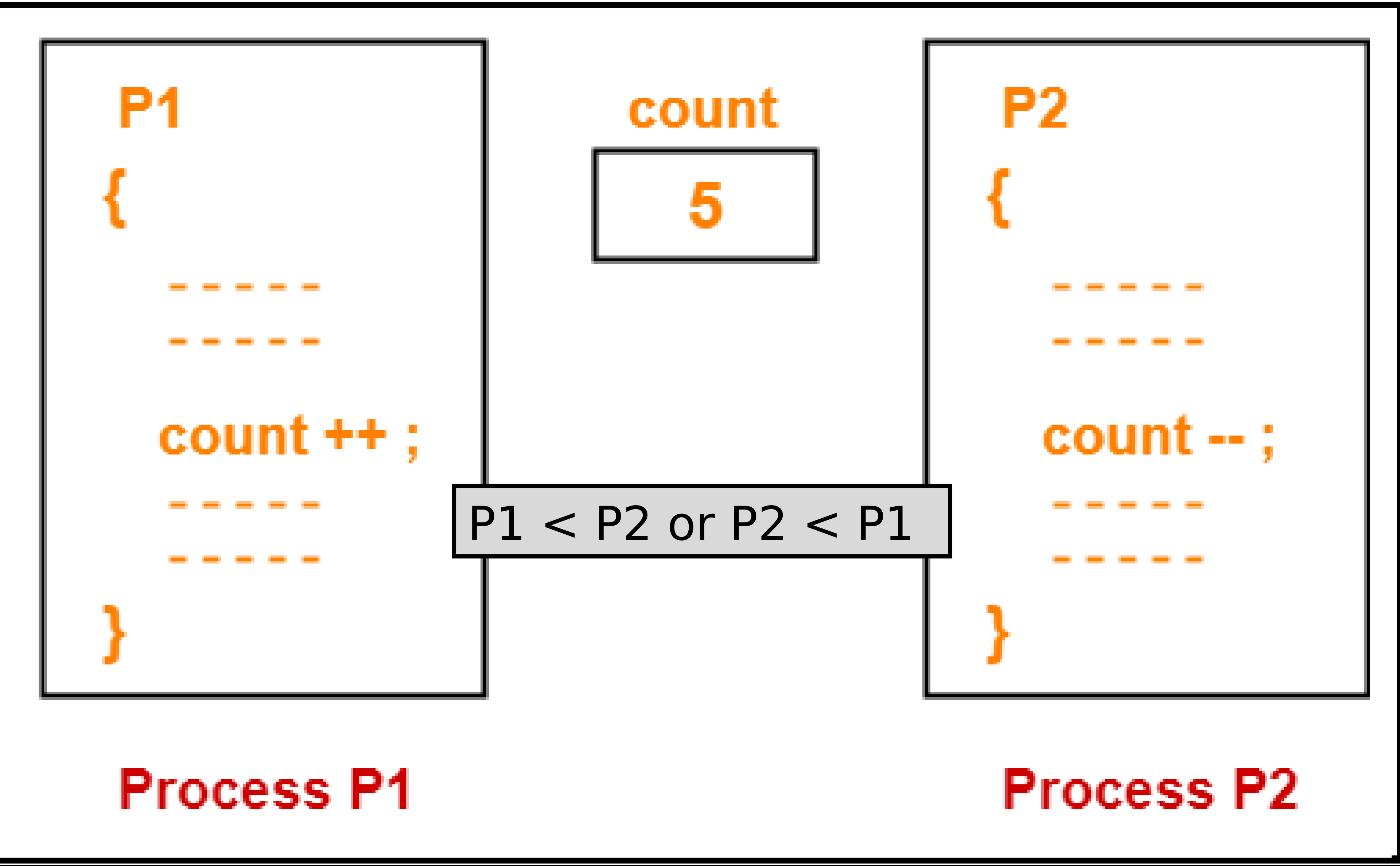


CS-3103 : Operating Systems : Sec-A (NB) : Process Synchronization

Process Synchronization

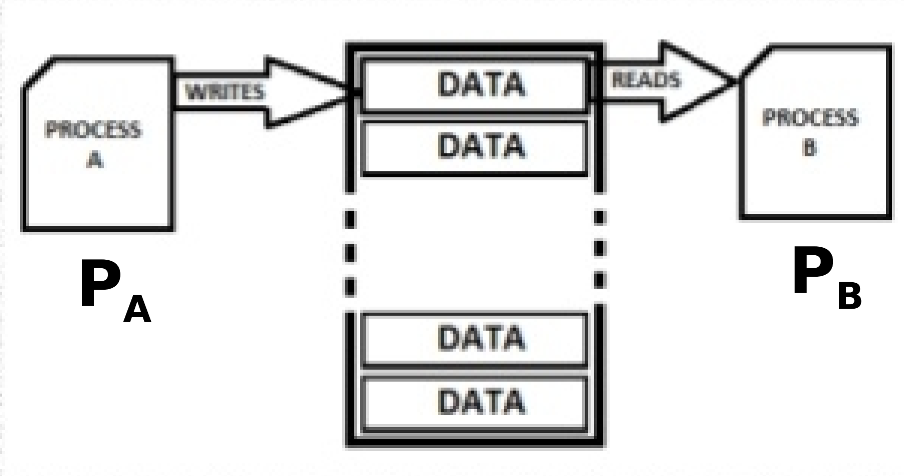
- Background
- The Critical-Section Problem
- Peterson’s Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Alternative Approaches

WHAT IS PROCESS SYNCHRONIZATION?

We have problem dealing with two concurrent processes trying to read and write data at the same time…..

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

State-I: P_A:
a = a + 5 (in the process of being written in the memory location at time instance t1)



State-II: P_B:
Read value of a from the same memory location at the same time instance t1

Race Condition

- Incorrect behaviour of a program due to concurrent execution of critical sections by two or more threads.
- For example, if thread 1 deletes an entry in a linked list while thread 2 is accessing the same entry.

- A **race condition** is where multiple processes/threads concurrently read and write to a shared memory location and the result depends on the order of the execution.
  - This was the cause of a patient death on a radiation therapy machine, the Therac-25
    - http://sunnyday.mit.edu/therac-25.html
    - Yakima Software flow
- Also can happen in bank account database transactions with, say a husband and a wife accessing the same account simultaneously from different ATMs

Race Condition Between Processes

Producer Consumer problem

- The *producer–consumer* problem (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem.
- The problem describes two processes, the *producer* and the *consumer*, who share a *common, fixed-size buffer* used as a *queue*.
- The producer's job is to generate *data*, put it into the *buffer*, and *start again*.
- At the same time, the consumer is *consuming the data (i.e., removing it from the buffer)*, *one piece at a time*.
- The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

Producer Consumer problem

- ❑ The solution for the producer is to either go to sleep or discard data if the buffer is full.
- ❑ The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again.
- ❑ In the same way, the consumer can go to sleep if it finds the buffer empty.
- ❑ The next time the producer puts data into the buffer, it wakes up the sleeping consumer.
- ❑ The solution can be reached by means of inter-process communication, typically using *semaphores*.
- ❑ An inadequate solution could result in a *deadlock* where both processes are waiting to be awakened.
- ❑ The problem can also be generalized to have multiple producers and consumers.

Race Condition (Producer-Consumer)

- `counter = counter + 1` could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- `counter = counter - 1` could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution *interleaving* with "**count = 5**" initially:

S0: producer execute `register1 = counter` **{register1 = 5}**

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

S2: consumer execute `register2 = counter` **{register2 = 5}**

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

S4: producer execute `counter = register1` **{counter = 6}**

S5: consumer execute `counter = register2` **{counter = 4}**

Race Condition (Cont.)

- How do we solve the race condition?
- We need to make sure that:
  - The execution of

    ```
    counter = counter + 1
    ```

    is done as an "*atomic*" action. That is, while it is being executed, no other instruction can be executed concurrently.
    - actually no other instruction can access **`counter`**
  - Similarly for

    ```
    counter = counter - 1
    ```

- The ability to execute an instruction, or a number of instructions, atomically is crucial for being able to solve many of the synchronization problems.

Producer-Consumer

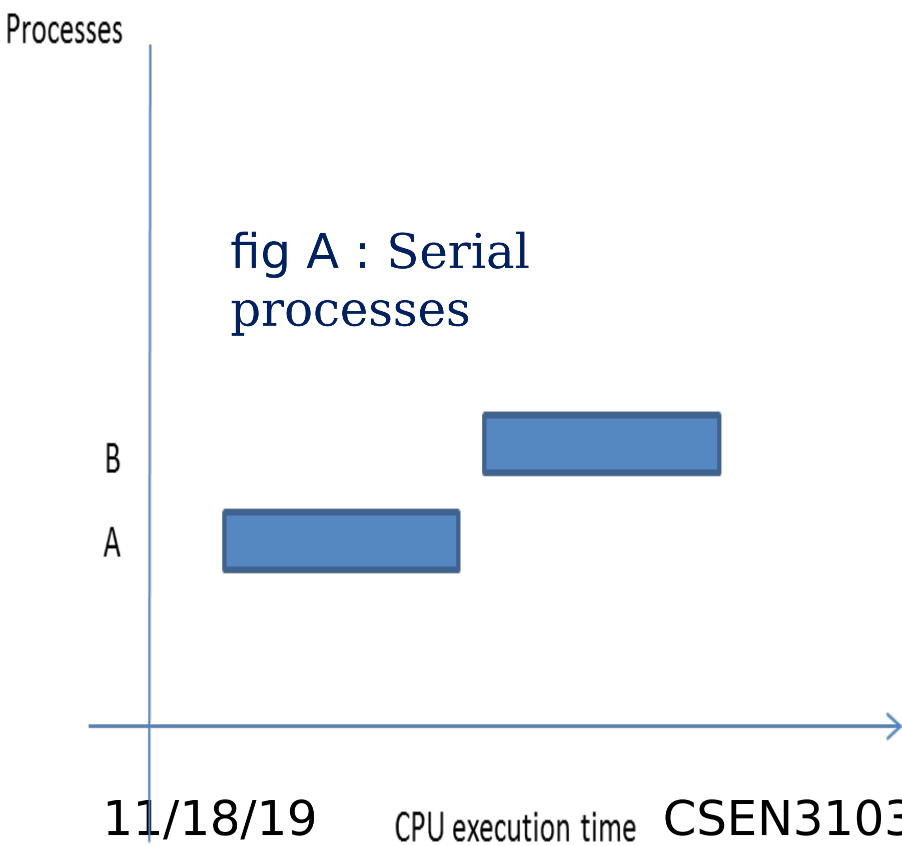
```
int itemCount = 0;

procedure producer()
{
    while (true)
    {
        item = produceItem();

        if (itemCount == BUFFER_SIZE)
        {
            sleep();
        }

        putItemIntoBuffer(item);
        itemCount = itemCount + 1;

        if (itemCount == 1)
        {
            wakeup(consumer);
        }
    }
}
```

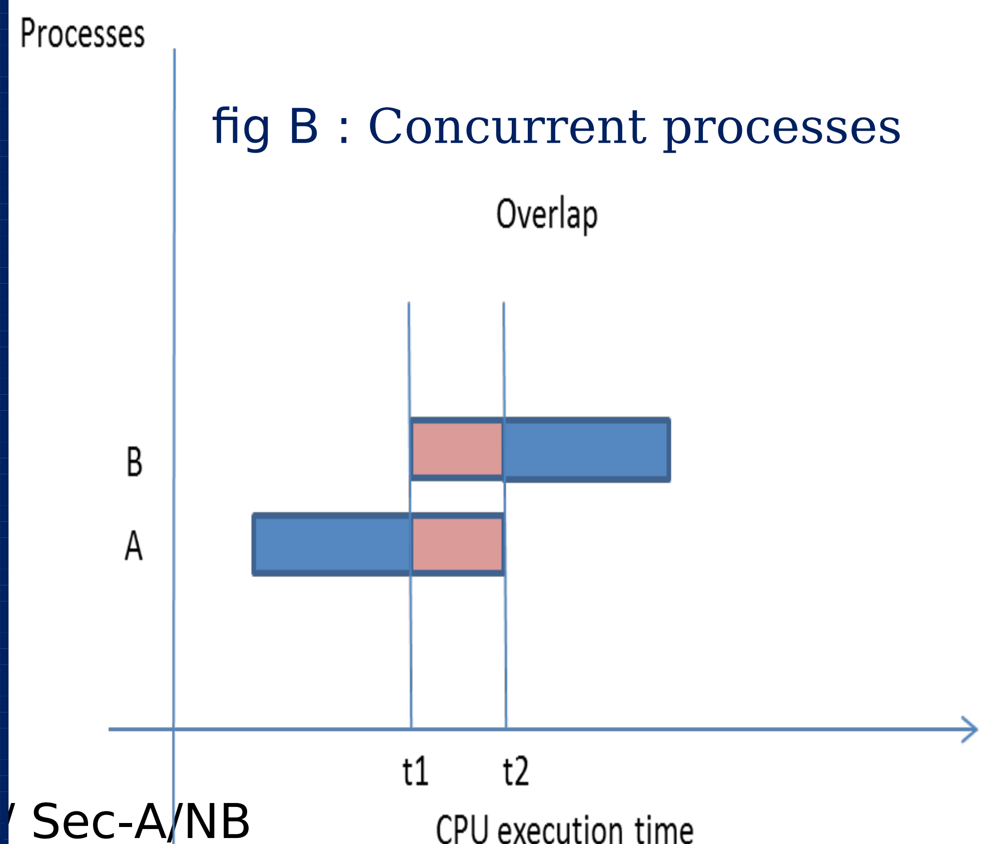
```
procedure consumer()
{
    while (true)
    {

        if (itemCount == 0)
        {
            sleep();
        }

        item = removeItemFromBuffer();
        itemCount = itemCount - 1;

        if (itemCount == BUFFER_SIZE - 1)
        {
            wakeup(producer);
        }

        consumeItem(item);
    }
}
```

- Multiprogramming and multitasking operating systems allow simultaneous residency of processes in main memory.
- The resident processes can be divided into two types:

✓ **Serial processes** : Two processes A and B are said to be serial when the execution of B starts only after the execution of A has stopped (fig A).

✓ **Concurrent processes** have an overlap in their CPU execution time (fig B). It may be noted that for ***time duration t1 to t2***, both processes, A and B, are using the CPU.





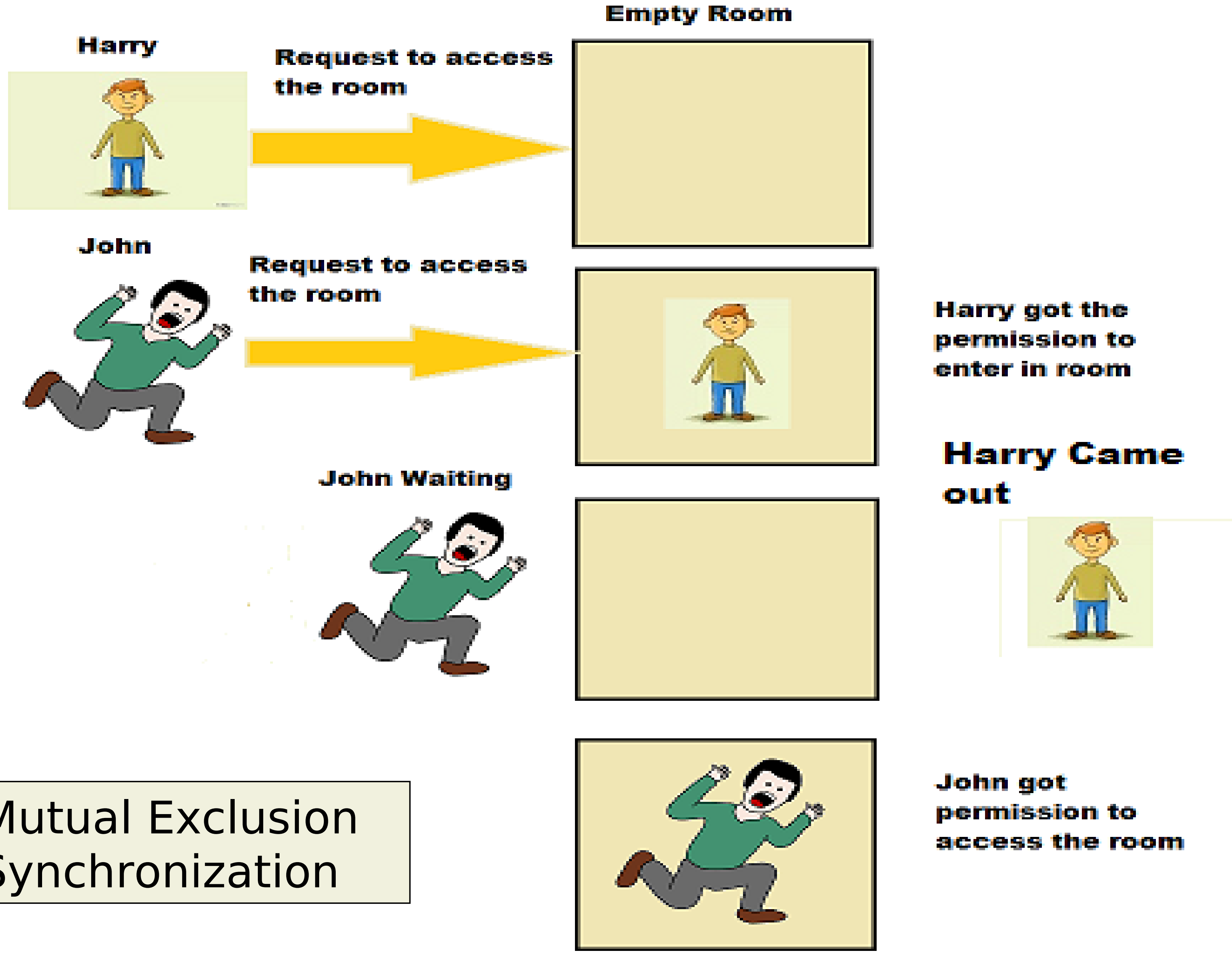
Categories of Concurrent Processes (part of Inter Process Communication (IPC))

- Concurrent processes can be further divided into two categories:
  - ***Fully independent processes*** – two processes are independent if they do not affect each other or do not interact with each other. Example: when an application program does not share data with other executing processes, its called independent process.
  - ***Co-operating processes*** - when execution of processes is affected by each other and they share data (or Shared Memory in operating system). Example: Producer-Consumer.

Communication between processes, Producer & Consumer

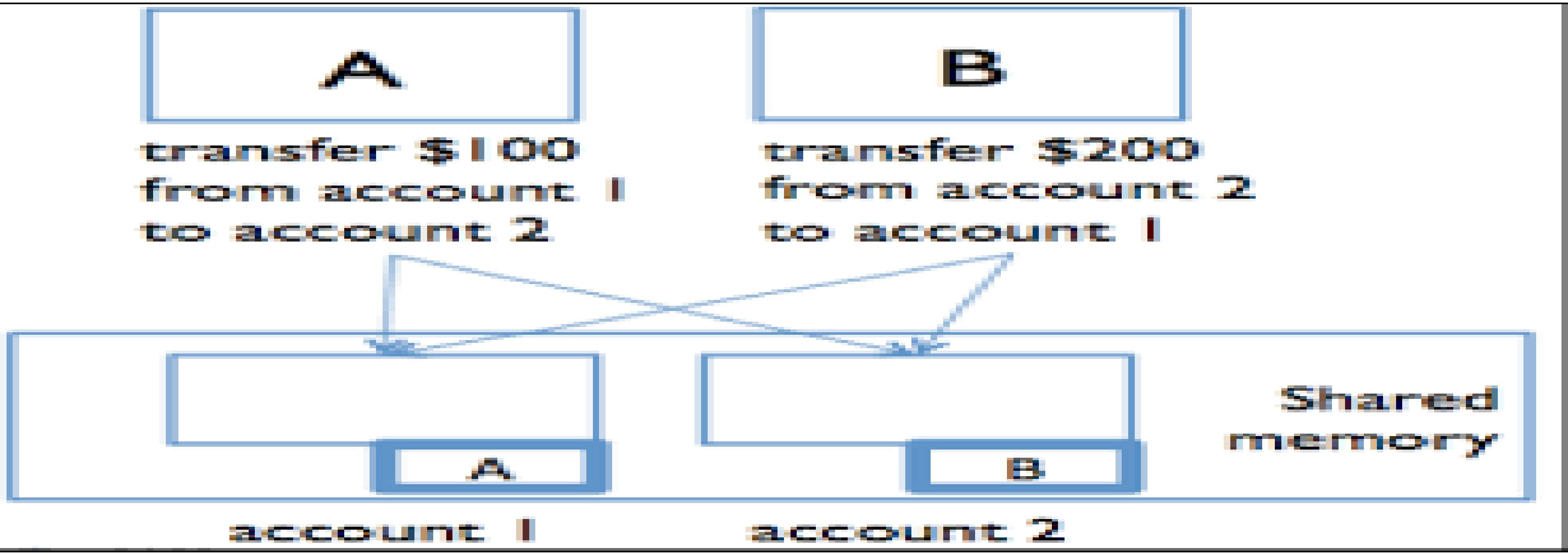
- There are two processes: Producer and Consumer.
- Producer produces some item and Consumer consumes that item.
- The two processes shares a common space or memory location known as buffer where the item produced by Producer is stored and from where the Consumer consumes the item if needed.
- There are two version of this problem: first one is known as unbounded buffer problem in which Producer can keep on producing items and there is no limit on size of buffer, the second one is known as bounded buffer problem in which producer can produce up to a certain amount of item and after that it starts waiting for consumer to consume it.
- We have discussed the bounded buffer problem earlier…..

In fact, the communication between concurrent processes becomes necessary in two situations,
1) Mutual Exclusion, 2) Synchronization.



1) Mutual Exclusion
2) Synchronization

How to Synchronize processes A and B



Processes A and B are accessing bank accounts A & B (that are in the shared memory) at the same time. We have to issue locking mechanisms when processes A & B are writing in these accounts. Actions of these concurrent processes have to be synchronized by mutual exclusion and critical sections….

Key terms related to concurrency

atomic operation — A function or action implemented as a sequence of one or more instructions that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation. The sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state. Atomicity guarantees isolation from concurrent processes.

critical section — A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.

deadlock

livelock

Livelock occurs when two or more processes continually repeat the same interaction in response to changes in the other processes without doing any useful work. These processes are not in the waiting state, and they are running concurrently. This is different from a deadlock because in a deadlock all processes are in the waiting state.

mutual exclusion — The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.

race condition — A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.

starvation — A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

Mutual Exclusion and Synchronization

- In a computer system, the resources can also be divided into shareable and non-shareable resources.
- **Shareable resources**: The resource that can be used by multiple processes concurrently. For example, CPU, read-only files, library files. A file in read mode can be concurrently read by many readers. The CPU can be time shared by the processes (RR-scheduling).
- **Non-shareable resources**: Peripheral devices such as printer, plotter and writable files. Example: A printer cannot be shared among processes. Unless the job of one process is completed, the job of another process cannot be taken by the printer. Similarly, two users cannot be allowed to write on the same file at the same time.

Mutual Exclusion and Synchronization

- A sharable ($\mathbf{d_s}$) or non-sharable resource should be protected so that only one process is able to access it at a time, and other processes must wait for their turn to use the resource.
- The operating system guards access to shareable and non-shareable resources using a piece of code called **critical section (CS)**. This means that when a process desires to access a resource, it must execute the code written within the CS that guards the resource as shown in Fig. X.

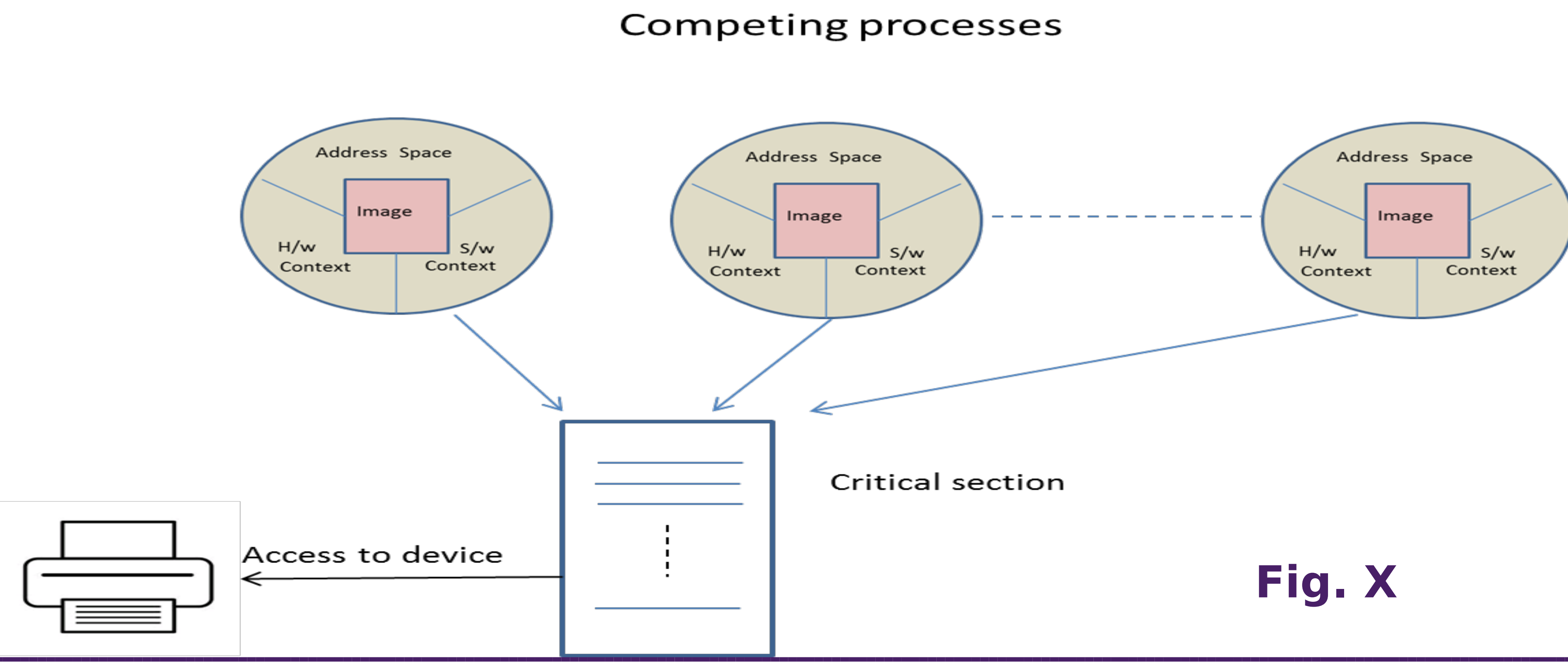


Fig. X

Management of Critical Sections (CS)

- A ***critical section*** is a piece of code that accesses shared resources. The resources can be variables, data structures, and devices. The CS is executed as an atomic action, i.e., if two processes P1 and P2 both want to execute the CS, then only one is allowed to execute it, and the other is made to wait.
- <u>The general format is given below</u>

{

Non-critical section

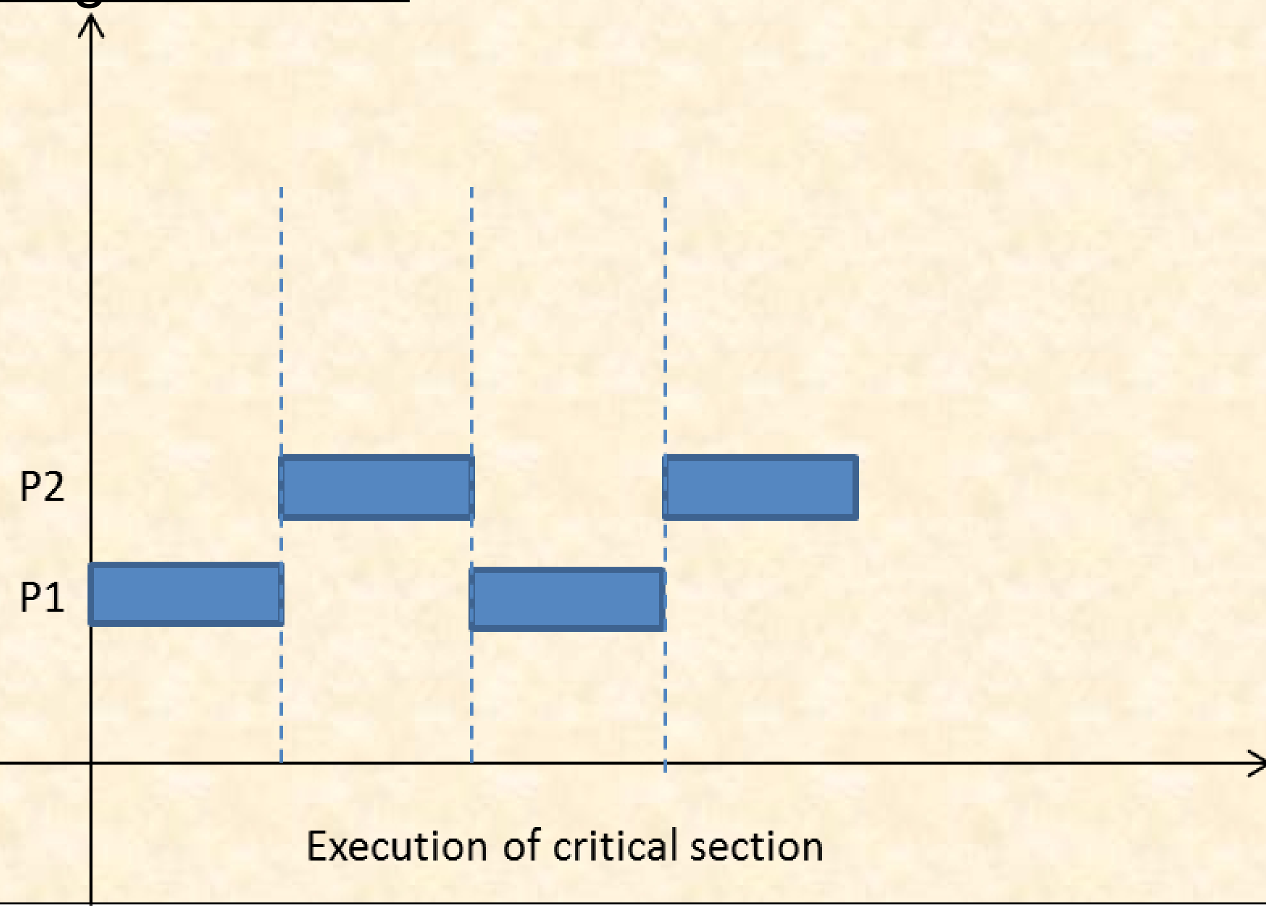
<entry section>

Critical Section

<exit section>

Non-critical section

}



- Consider system of ***n*** processes {$P_0, P_1, \ldots P_{n-1}$}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- e this
- enter critical section in the critical section; once ection it enters the **exit** s the **remainder section**

- General structure of process ***P_i***

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

CS must satisfy the following correctness conditions:

- **Mutual exclusion**: Only one process is allowed to enter into the critical section that guards a shared resource. At any moment, at MOST one process may execute a CS for a data item d_s.
- **Progress:** If critical section is available for execution then only the processes can participate in the decision as to which process will enter into the critical section next. When no process is executing a CS for a data item d_s, one of the processes wishing to enter a CS for d_s will be granted entry.
- **Bounded wait:** The policy has to be fair in the sense that no process should wait forever. After a process P_i has indicated its desire to enter a CS for d_s, the number of times other

The progress and bounded wait properties together prevent starvation. Apart from correctness, a CS implementation should also guarantee that *any process wishing to enter a CS would not be delayed indefinitely, i.e., starvation would not occur*.

Properties of a Critical Section Implementation

- When several processes wish to use critical sections for a data item d_s, *a critical* section implementation must ensure that it grants entry into a critical section in accordance with the notions of correctness and fairness to all processes.
- We know (last slide) three essential properties a critical section implementation to satisfy these requirements.
- The *mutual exclusion property guarantees that* two or more processes will not be in critical sections for d_s *simultaneously.* ***Mutual exclusion ensures correctness of the implementation.***
- The second and third property (*Progress* and *Bounded Wait*) together guarantee that **no** process wishing to enter a critical section will be delayed indefinitely; i.e., starvation will not occur.
- The ***progress property*** *ensures that if some processes are interested in entering* critical sections for a data item *ds, one of them will be granted entry if no process* is currently inside any critical section for d_s*—that is, use of a CS cannot be* "reserved" for a process that is not interested in entering a critical section at present. However, this property alone cannot prevent starvation because a process might never gain entry to a CS if the critical section implementation always favours other processes for entry to the CS.
- The ***bounded wait property*** *ensures that this* does not happen by limiting the number of times other processes can gain entry to a critical section ahead of a requesting process *Pi* .
- *Thus the* ***progress*** and ***bounded wait*** properties ensure that every requesting process will gain entry to a critical section in finite time; however, these properties do not guarantee a specific limit to the delay in gaining entry to a CS.

if *nextseatno* $\leq$ *capacity*
then
    allotedno:=*nextseatno*;
    nextseatno:=*nextseatno*+1;
else
    display "*sorry, no seats available*";

Process P_i

if *nextseatno* $\leq$ *capacity*
then
    allotedno:=*nextseatno*;
    nextseatno:=*nextseatno*+1;
else
    display "*sorry, no seats available*";

Process P_j

Use of critical sections in an airline reservation system.

- Interacting processes need to coordinate their execution with respect to one another, to perform their actions in a desired order
  - A frequent requirement in process synchronization is that a process P_i should perform an action a_i only after process P_j has performed some action a_j.
  - ***This synchronization requirement is met using the technique of Signaling.***

attempt at signaling through boolean variables.

var

operation_aj_performed : boolean;
pi_blocked : boolean;

begin

operation_aj_performed := false;
pi_blocked := false;

Parbegin

```
. . .
if operation_aj_performed = false
then
    pi_blocked := true;
    block (P_i);
{perform operation a_i}
. . .
. . .
. . .
```

Process P_i

```
. . .
{perform operation a_j}
if pi_blocked = true
then
    pi_blocked := false;
    activate (P_i);
else
    operation_aj_performed := true
. . .
```

Process P_j

Parend;
end.

Synchronization Approaches

- ***Looping versus Blocking***
- Hardware Support for Process Synchronization
- Algorithmic Approaches, Synchronization Primitives, and Concurrent Programming Constructs

- ***Busy wait*** is *Not desired*…

while (some process is in a critical section on $\{d_s\}$ or
    is executing an indivisible operation using $\{d_s\}$)
  { do nothing }

Critical section or indivisible operation using $\{d_s\}$

- In the ***while*** loop, the process checks if some other process is in a CS for the same data item. If so, it keeps looping until the other process exits its CS.
- A ***busy wait*** is a situation in which a process repeatedly checks if a condition that would enable it to get past a synchronization point is satisfied. It ends only when the condition is satisfied. Thus, a busy wait keeps the CPU busy in executing a process even as the process does nothing! Lower priority processes are denied use of the CPU, so their response times suffer. System performance also suffers.

Hardware Support for Process Synchronization

- Indivisible instructions
  - Avoid race conditions on memory locations
- Used with a ***lock variable to implement CS*** and indivisible operations

```
entry_test:   if lock = closed
                  then goto entry_test;      Performed by
              lock := closed;                an indivisible
                                             instruction
              {Critical section or
              indivisible operation}
              lock := open;
```

Implementing a critical section or indivisible operation by using a lock variable.

- *entry_test* performed with indivisible instruction
  Test-and-set (TS) instruction, Swap instruction

Hardware Support for Process Synchronization

```
LOCK          DC    X'00'            Lock is initialized to open
ENTRY_TEST    TS    LOCK             Test-and-set lock
              BC    7, ENTRY_TEST    Loop if lock was closed

              ...                    { Critical section or
                                       indivisible operation }

              MVI   LOCK, X'00'      Open the lock(by moving 0s)
```

Implementing a critical section or indivisible operation by using test-and-set.

```
TEMP         DS    1               Reserve one byte for TEMP
LOCK         DC    X'00'           Lock is initialized to open
             MVI   TEMP, X'FF'     X'FF' is used to close the lock
ENTRY_TEST   SWAP  LOCK, TEMP
             COMP  TEMP, X'00'     Test old value of lock
             BC    7, ENTRY_TEST   Loop if lock was closed

             ...                   { Critical section or
                                     indivisible operation }

             MVI   LOCK, X'00'     Open the lock
```

Implementing a critical section or indivisible operation by using a swap instruction.

Algorithmic Approaches, Synchronization Primitives, and Concurrent Programming Constructs

- Algorithmic Approaches
  - For implementing mutual exclusion
  - Independent of hardware or software platform
    - Busy waiting for synchronization
- **Synchronization Primitives**
  - **Implemented using indivisible instructions**
  - **E.g., *wait* and *signal* of *semaphores***
- Concurrent Programming Constructs
  - *Monitors*

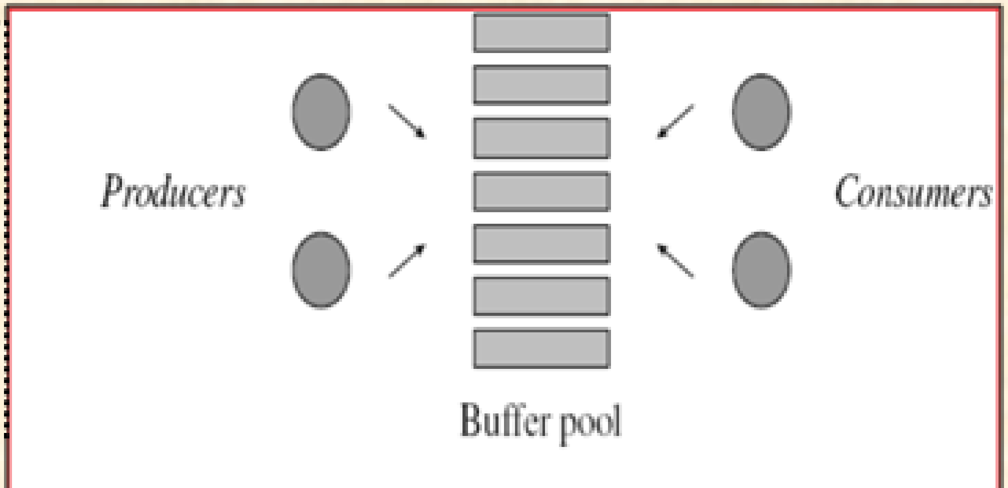
- **A solution to a process synchronization problem should meet three important criteria:**
  - **Correctness**
  - **Maximum concurrency**
  - **No busy waits**
- **Some classic problems:**
  - ***Producers-Consumers with Bounded Buffers***
  - ***Readers and Writers***
  - ***Dining Philosophers***


START

Producers-Consumers with Bounded Buffers

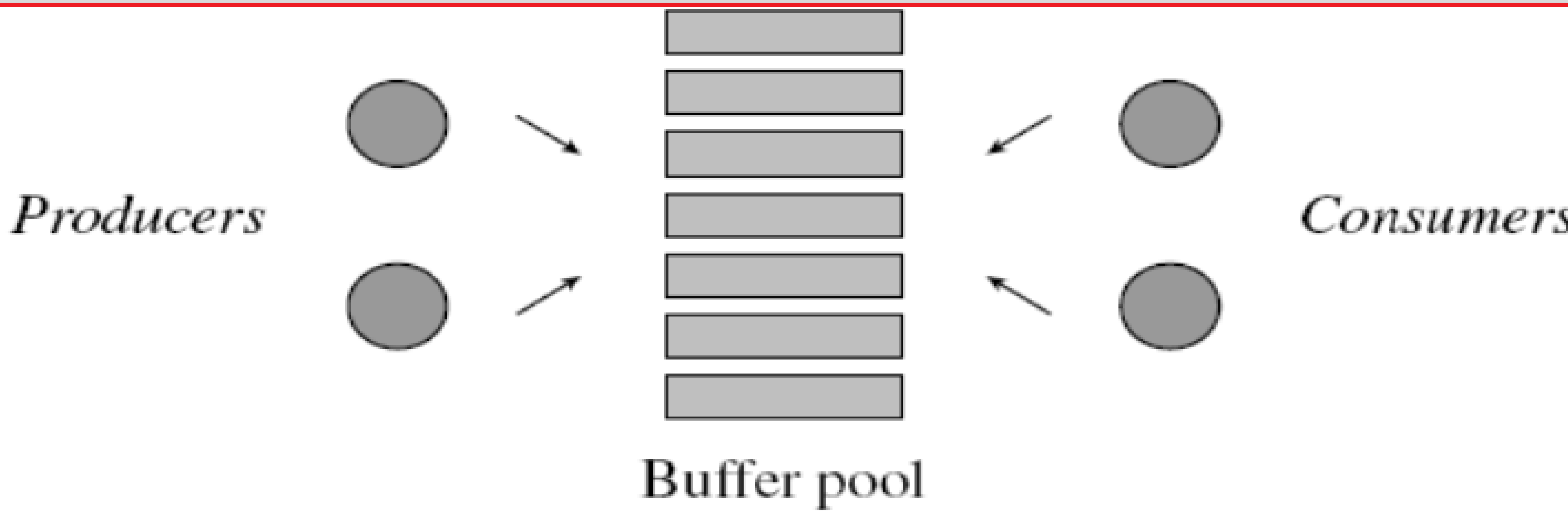
- A producers-consumers system with bounded buffers consists of an unspecified number of producer and consumer processes and a finite pool of buffers.
- Each buffer is capable of holding one record of information—it is said to become ***full*** when a producer writes into it, and ***empty*** when a consumer copies out a record contained in it; it is empty to start with.
- A producer process produces one record at a time and writes it into the buffer.
- A consumer process consumes information one record at a time.
- Example of producers-consumers process:

A print service. A fixed size queue of print requests is the bounded buffer. A process that adds a print request to the queue is a producer process.



A producers–consumers system with bounded buffers.

Producers-Consumers with Bounded Buffers



A producers–consumers system with bounded buffers.

- A solution must satisfy the following:
  1. ***A producer must not overwrite a full buffer***
  2. ***A consumer must not consume an empty buffer***
  3. ***Producers and consumers must access buffers in a mutually exclusive manner***
  4. (Optional) Information must be consumed in the same order in which it is put into the buffers, i.e., in FIFO order

Producers-Consumers with Bounded Buffers…

```
begin
Parbegin
   var produced : boolean;
   repeat
      produced := false
      while produced = false
         if an empty buffer exists
         then
            { Produce in a buffer }
            produced := true;
      { Remainder of the cycle }
   forever;
Parend;
end.
```

Producer

```
var consumed : boolean;
repeat
   consumed := false;
   while consumed = false
      if a full buffer exists
      then
         { Consume a buffer }
         consumed := true;
   { Remainder of the cycle }
forever;
```

Consumer

An outline for producers–consumers using critical sections.

- Suffers from two problems:
  - Poor concurrency and busy waits

- Producer and Consumer processes access a buffer inside a critical section.
- A producer enters its CS and checks to see whether an empty buffer exists. If so, it produces into the buffer, else it merely exits from its CS.
- This sequence is repeated until it finds an empty buffer.
- The boolean variable ***produced*** is used to break out of the ***while*** loop after the producer produces in the empty buffer.
- Consumer makes repeated checks until it finds a full buffer to consume from.
- The above design of producer-consumer problem suffers from busy-wait, because producer(consumer) keep on searching for empty(full) buffers.

- **Improved outline with Signaling**: Consider a producer-consumers system that consists of a single producer, a single consumer and a single buffer.
- The operation ***check_b_empty*** performed by the producer blocks it if the buffer is full, while the operation ***post_b_full*** sets ***buffer_full*** to ***true*** and activates the consumer if the consumer is blocked for the buffer to become full.
- Analogous operations ***check_b_full*** and ***post_b_empty*** are defined for use by consumer process.
- The boolean flags ***producer_blocked*** and ***consumer_blocked*** are used by these operations to note if the producer or consumer process is blocked at any moment.

Producers-Consumers with Bounded Buffers…

```
var
        buffer : . . . ;
        buffer_full : boolean;
        producer_blocked, consumer_blocked : boolean;
begin
        buffer_full := false;
        producer_blocked := false;
        consumer_blocked := false;
Parbegin
  repeat                                repeat
    check_b_empty;                        check_b_full;
    {Produce in the buffer}               {Consume from the buffer}
    post_b_full;                          post_b_empty;
    {Remainder of the cycle}              {Remainder of the cycle}
  forever;                              forever;
Parend;    Producer                              Consumer
end.
```

An improved outline for a single buffer producers–consumers system using signaling.

Producers-Consumers with Bounded Buffers…

```
procedure check_b_empty
begin
    if buffer_full = true
    then
        producer_blocked := true;
        block (producer);
end;

procedure post_b_full
begin
    buffer_full := true;
    if consumer_blocked = true
    then
        consumer_blocked := false;
        activate (consumer);
end;
```

Operations of producer

```
procedure check_b_full
begin
    if buffer_full = false
    then
        consumer_blocked := true;
        block (consumer);
end;

procedure post_b_empty
begin
    buffer_full := false;
    if producer_blocked = true
    then
        producer_blocked := false;
        activate (producer);
end;
```

Operations of consumer

Indivisible operations for the producers–consumers problem.

- Concurrent process which affect each other and share data are called:
  a. Cooperating processes b. Independent processes
  c. Serial processes d. None of the above

Ans. (a)

- Concurrent processes cooperate with each other:
  a. Information sharing b. Computation speed up
  c. Convenience d. All of the above

Ans. (d)

- Concurrent processes cooperate with each other:
  a. Information sharing b. Computation speed up
  c. Convenience d. All of the above

Ans. (d)

- The variables whose contents get updated during execution of a statement are called:
  a. Read variables b. Constants
  c. Write variables d. None of the above

Ans: c

- The concurrency of processes can be graphically represented by :
  a. Wait for graph b. Precedence graph
  c. Process state graph d. All of the above

Ans: b

- The resources that can be shared by multiple processes concurrently are called :

  Ans: a

  a. Shareable resources b. Non-shareable resources
  c. Consumable resources d. None of the above

- Which resources may be protected from simultaneous access?

  Ans: a

  a. Shareable resources b. Non-shareable resources
  c. Consumable resources d. None of the above

- The act of exchanging signals among processes for information sharing, is called:

  a. Critical section b. Mutual exclusion

  Ans: c

  c. Synchronization d. All of the above

- A critical section must satisfy:

  a. Correctness b. Progress

  Ans: d

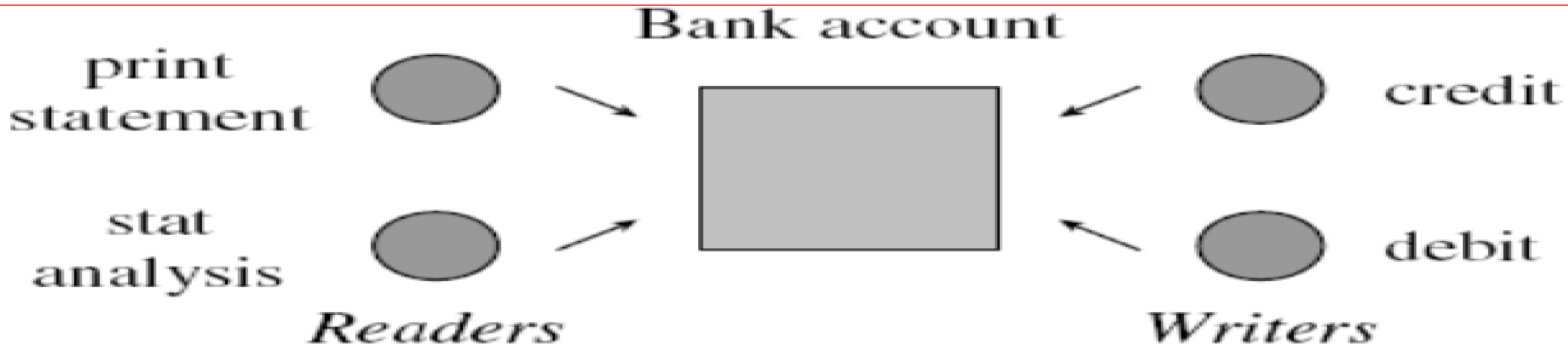
  c. Fairness d. All of the above


START

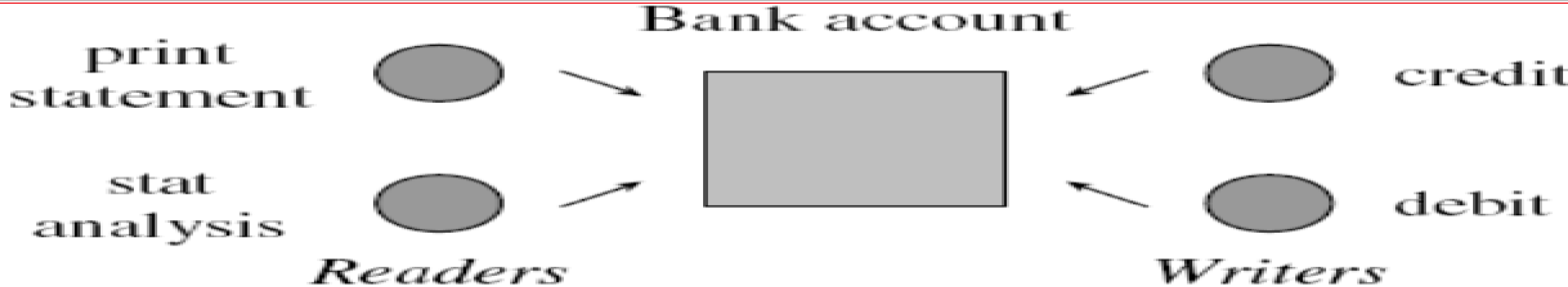
After Q-n-A session, let's come back to *Readers and Writers*

- A readers-writers system consists of a set of processes using some shared data.
- A process that only reads the data is a ***reader***; a process that modifies or updates it is a ***writer***.
- The correctness conditions for the readers-writers problem are :

1. Many readers can perform reading concurrently
2. Reading is prohibited while a writer is writing
3. Only one writer can perform writing at any time
4. (optional) A reader has a non-preemptive priority over writers
   – **Called *readers preferred readers–writers* system**



Readers and writers in a banking system.



Readers and writers in a banking system.

The readers and writers share a bank account. The reader processes print statement and stat analysis read the data from the bank account. Hence they can execute concurrently.

Credit and debit modify the balance in the account. Clearly only one of them should be active at any moment and none of the readers should be active when they modify the data.

Readers and Writers……

Parbegin

Reader(s)

```
  repeat
      If a writer is writing
      then
          { wait };
      { read }
      If no other readers reading
      then
          if writer(s) waiting
          then
              activate one waiting writer;
  forever;
```

Writer(s)

```
repeat
    If reader(s) are reading, or a
        writer is writing
    then
        { wait };
    { write }  CS
    If reader(s) or writer(s) waiting
    then
        activate either one waiting
        writer or all waiting readers;
forever;
```

Parend;

end.

An outline for a readers–writers system. **Fig. AB**

Readers and Writers……

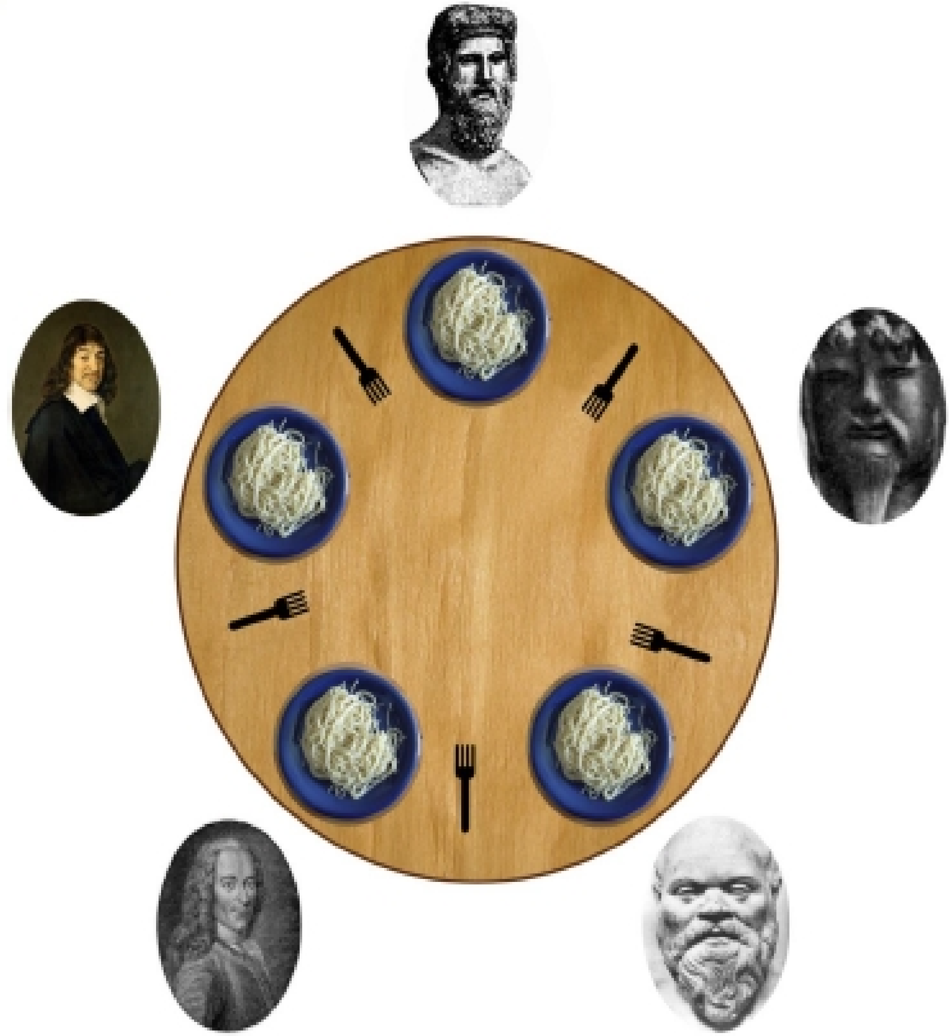
The synchronization requirements of readers-writers system is determined by analyzing its correctness conditions as follows:

1. Many readers can perform reading concurrently
2. Reading is prohibited while a writer is writing
3. Only one writer can perform writing at any time
4. (optional) A reader has a non-preemptive priority over writers
   – **Called *readers preferred readers—writers* system**

- ***Condition 3*** requires that a writer should perform writing in a critical section. When it finishes writing, it should activate one waiting writer or activate all waiting readers. This can be achieved using a signaling arrangement.
- From ***condition 1***, concurrent reading is permitted. We should maintain a count of readers reading concurrently. When the last reader finishes reading, it should activate a waiting writer.
- Based on Fig. AB, writing is performed in a CS. A CS is not used in a reader as that would prevent concurrency between readers. The outline in fig. AB, does not satisfy the bounded wait condition for both readers and writers, however, it provides maximum concurrency.

Dining-Philosophers Problem ☺

- Philosophers eat or think
- Eating needs 2 forks
- Pick one fork at a time
- How to prevent deadlock



Plato
Confucius
Socrates
Voltaire
Descartes

- Shared data
  - Bowl of rice (data set)
- Semaphore needed

Dining Philosophers

- Five philosophers sit around a table thinking philosophical issues.
- A plate of noodles is kept in front of each philosopher, and a fork is placed between each pair of philosophers (Fig. AA).
- To eat, a philosopher must pick up the two forks placed between him and his immediate neighbors on either side, one at a time.
- ***The problem is to design processes to represent the philosophers such that each philosopher can eat when hungry and none dies of hunger.***

- **The correctness condition in the dining philosophers system is that a *hungry philosopher should not face indefinite wait when he decides to eat*.**
- The challenge is to design a solution that does not suffer from either ***deadlocks***, where processes become blocked waiting for each other, or ***livelocks***, where processes are not blocked but defer to each other indefinitely.
- Consider the outline of a ***philosopher process Pi*** shown in Fig. A1, where we do not have the details of process synchronization.
- .... Next slide.... With fig. A1....

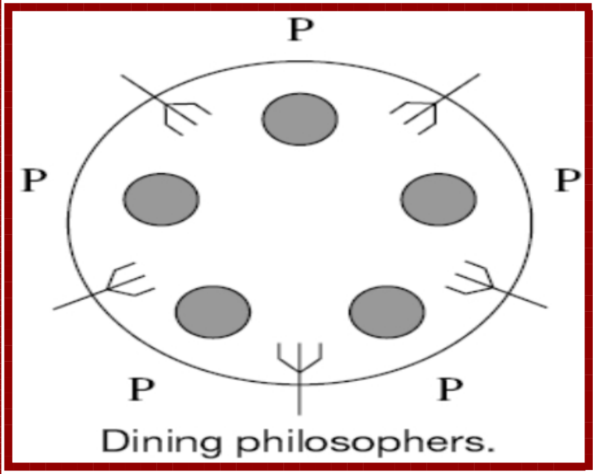
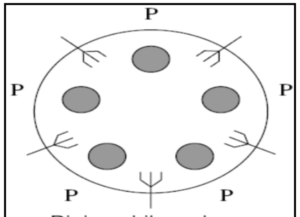


Fig. AA

Dining Philosophers….

repeat
    if left fork is not available
    then
        block (P_i);
    lift left fork;
    if right fork is not available
    then
        block (P_i);
    lift right fork;
    { eat }
    put down both forks
    if left neighbor is waiting for his right fork
    then
        activate (left neighbor);
    if right neighbor is waiting for his left fork
    then
        activate (right neighbor);
    { think }
forever

Outline of a philosopher process P_i.



Dining philosophers.

Fig. A1

- This solution is prone to deadlock, because if all philosophers simultaneously lift their left forks, none will be able to lift the right fork!!!
- It also contains race conditions because neighbors might fight over a shared fork.
- We can avoid deadlocks by modifying the philosopher process so that if the right fork is not available, the philosopher would defer to his left neighbor by putting down the left fork and repeating the attempt to take the forks sometime later.
- This approach suffers from livelocks because the same situation may recur.

Dining Philosophers….

Fig. A2

```
var     successful : boolean;
repeat
    successful := false;
    while (not successful)
        if both forks are available then
            lift the forks one at a time;
            successful := true;
        if successful = false
        then
            block (P_i);
    { eat }
    put down both forks;
    if left neighbor is waiting for his right fork
    then
        activate (left neighbor);
    if right neighbor is waiting for his left fork
    then
        activate (right neighbor);
    { think }
forever
```

An improved outline of a philosopher process.

- A philosopher checks availability of forks in a CS and also picks up the forks in the CS. Hence race conditions cannot arise.
- This arrangement ensures that at least some philosopher(s) can eat at any time and deadlocks cannot arise.
- A philosopher who cannot get both forks at the same time blocks himself.
- However, it does not avoid busy waits because the philosopher gets activated when any of his neighbors puts down a shared fork, hence he has to check for availability of forks once again. This is the purpose of the ***while*** loop.

Software Solutions for Critical Section ➔ *Dekker's Algorithm for Two-process CS Problem*

- Dekker proposed that in case of race condition let ***turn*** be used to break the tie.
- This means that if a process (say P2) desires to enter the CS and finds that the other process (P1) is also interested in entering, and it is the turn of the other process (P1), then the previous process (P2) forgoes its claim and sets its state to false, enabling the other process (P1) to enter the CS.
- Dekker's algorithm is shown in Fig. A3

Dekker's Algorithm CS()

stateP1 = false, stateP2 = false, turn =1

Fig. A3

```
// process P1
While (true)
  {
        stateP1 = true;
     while (stateP2 == true)
        {  if (turn = = 2)
                stateP1 = false;
                while  ( turn == 2) {....  }   // do nothing
                stateP1= true;
        }
        { enter CS }
        stateP1 = false;
        turn =2;
                perform rest of the work
  }
```

```
// process P2
While (true)
  {
        stateP2 = true;
     while (stateP1 == true)
        {  if (turn = = 1)
                stateP2 = false;
                while  ( turn == 1) {....  }   // do nothing
                stateP2 = true;
        }
        { enter CS }
        stateP1 = false;
        turn = 1;
                perform rest of the work
  }
```

Peterson's Algorithm for Two-process CS Problem

The Dekker's solution is correct and complete. Let us consider the following scenario: Assume that ***turn = 1***.

1. P1 shows its interest in entering the CS. It sets 'stateP1 = true'.
2. P1 is preempted and a context switch takes place with P2.
3. P2 shows its interest in entering the CS. It sets 'stateP2 = true'.
4. P2 tries to enter the CS but finds that 'stateP1 = true' and also 'turn == 1', It sets 'stateP2 = false'.
5. P2 is preempted and a context switch takes place with P1.
6. P1 tries to enter into CS and it finds that 'stateP2 == false' and therefore it enters the CS.

7.In Peterson's algorithm, a process allows the other process too get into the CS by giving the turn to other process. The algo. is given in Fig. A4

Peterson Algorithm CS ()

stateP1 = false; stateP2 = false;

Fig. A4

```
// process P1
While (true)
  {
      stateP1 = true;
      turn =2;   // give the turn to other  process
       while (stateP2 == true  &&  turn == 2)
              {  ....  }        // do nothing
      { enter CS }
      stateP1 = false;
              perform rest of the work
  }
```

```
// process P2
While (true)
  {
      stateP2= true;
      turn =1;        // give the turn to other  process
      while (stateP1 == true  &&  turn == 1)
              {  ....  }        // do nothing
      { enter CS }
      stateP2 = false;
              perform rest of the work
  }
```

```
parbegin
   P1;
   P2;
parend
```

Bakery's Algorithm – Solution to the N-process CS Problem

- Let us assume there are ***N*** processes.
- Every process is allotted a number from ***1..N***.
- If a process desires to enter the CS, it obtains a token from the system.
- The system allots a pair of values (***token[i], i***) to the process, where ***token[i]*** is the token number of ***i***th process, and ***i*** is the process number.
- The token number is so assigned that its value is one more than the highest token assigned thus far.
- The token is not protected by the system and therefore there are chances that two processes, under race conditions, may get the same token number.
- However, the process of obtaining the token is controlled by a ***boolean variable called choosing***.
- This algorithm uses a ***Boolean array*** called ***choosing[]*** and an ***integer array*** called ***token[]***.
- All entries of the array ***choosing[]*** are initially set to ***false*** indicating that no process is choosing the token.
- Similarly, all entries of the ***array token[]*** are initially set to ***0*** indicating that all processes are out of CS.

Bakery's Algorithm – Solution to the N-process CS Problem

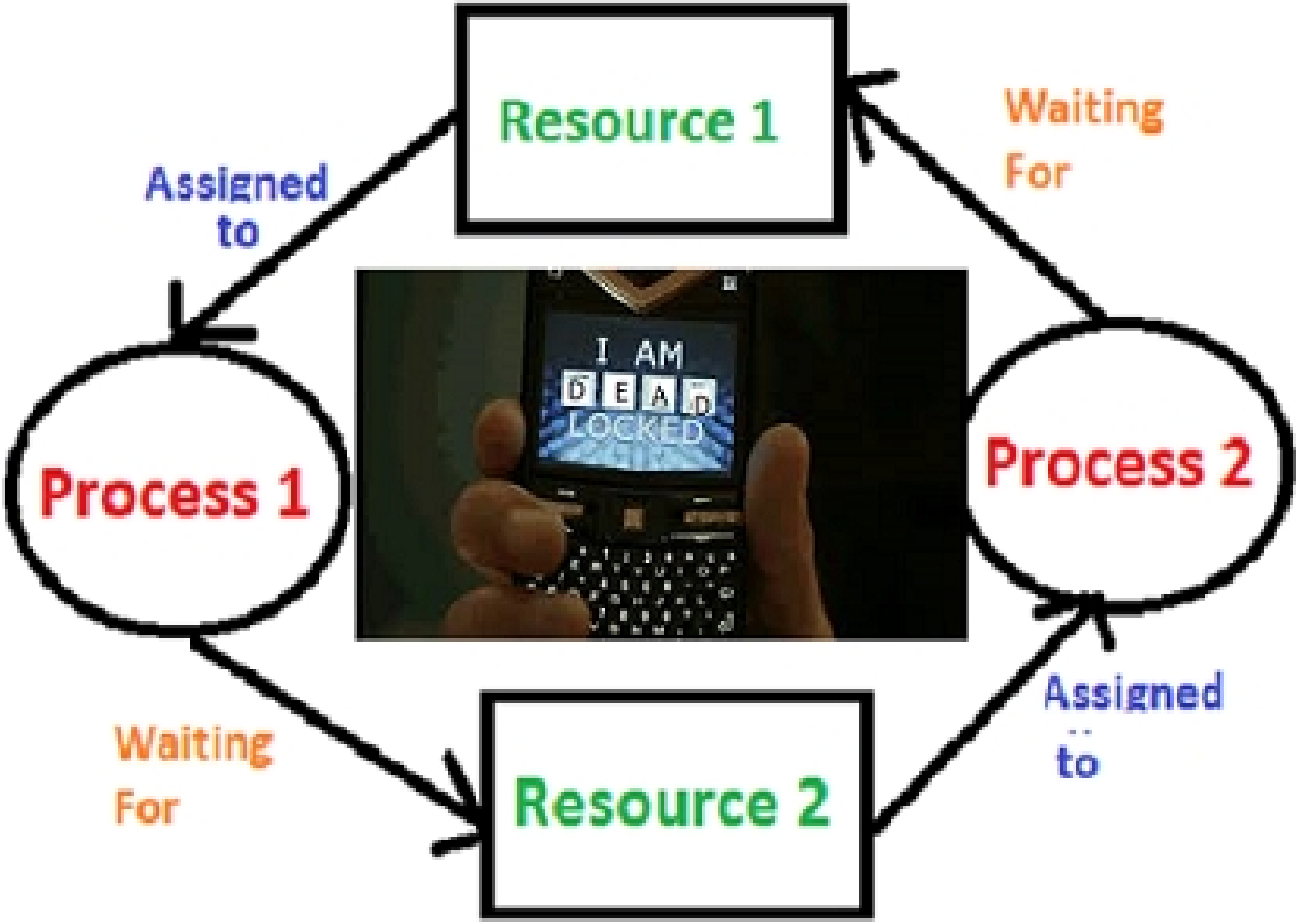
Fig. A5

```
Algorithm Bakery()
 {
   choosing[] = { false, false, …, false};
    token[] = { 0,0, …, 0};
   while ( true)
      {
            choosing[i] = true;
             for ( j= 0; j < N; j++) {
                        if (token [i] <= token[j])
                                  token[i] = token[j] +1;
              }
            choosing[i] = false;

            for ( j =0; j < N; j++)
              {
                        while ( choosing[j])  { …}  //  do nothing  as jth process is obtaining the token
                        while ( j != I  && token [j] != 0 && (token [j],j  < token[i],i) )  {…} // do nothing


              }
                        enter CS
                        token [i] = 0;
                        perform rest of the work;
      }
                        parbegin
                        P0, P1, …..PN-1
                        parend
```

- Now, if a process ***Pi*** desires to enter the CS, it sets ***choosing[i] = true*** indicating that it is obtaining the token.
- After obtaining the token it sets '***choosing[i] = false***'.
- Once the token is obtained by the process, it tries to access the CS in the order of its token number.
- Before entering the CS the process ensures that either the other processes have token numbers greater than its token value, or their token value is equal to 0, indicating that they are out of the CS.

✓ Thus, this algorithm satisfies the following:

- Correctness: Only one process enters the CS at a time.
- Progress: If multiple processes are trying to enter the CS, one of them will eventually enter the CS.
- Fairness: No process will wait indefinitely. A time will come for each process, when it becomes eligible and eventually enters the CS.
- Generality: It works for N processes.


Resource 1
Assigned to
Waiting For
I AM DEAD LOCKED
Process 1
Process 2
Waiting For
Assigned to
Resource 2