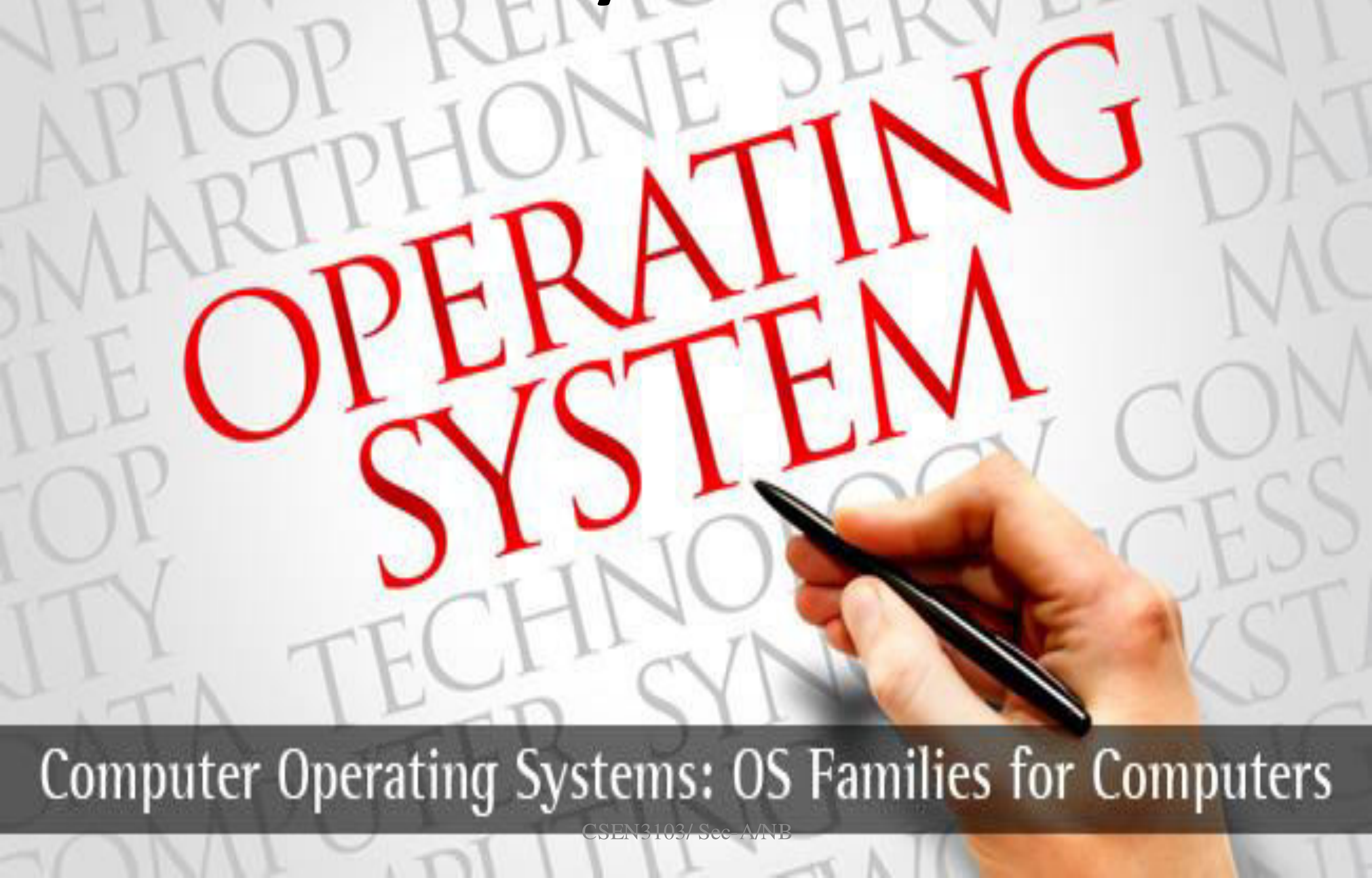
CS-3103 : Operating Systems : Sec-A (NB) :
Process Synchronization



Introduction

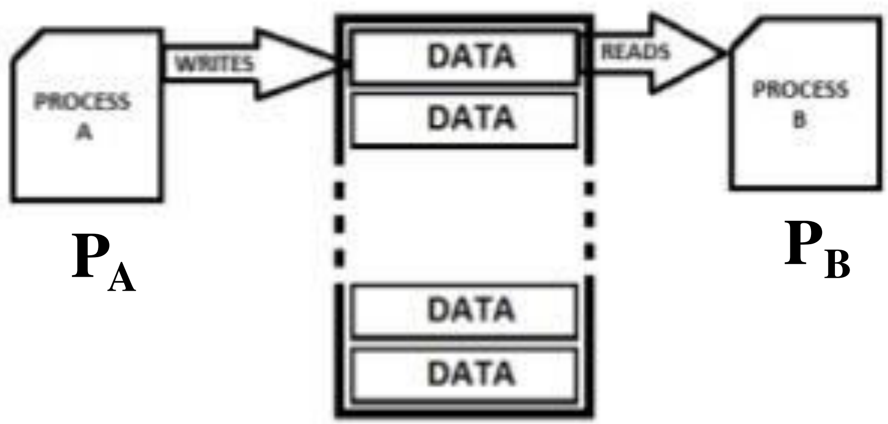
- What is Process Synchronization?
- Race Conditions
- Critical Sections
- Control Synchronization and Indivisible Operations
- Synchronization Approaches
- Structure of Concurrent Systems
- Classic Process Synchronization Problems
- Algorithmic Approach to Implementing Critical Sections
- Semaphores
- Monitors
- Case Studies of Process Synchronization

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



P_A P_B

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

Race Condition

John's balance=5000

At the same time two write operations are happening????
What would be John's balance????

1. Read balance
2. if enough balance
3. balance=balance-withdrawn
4. update balance

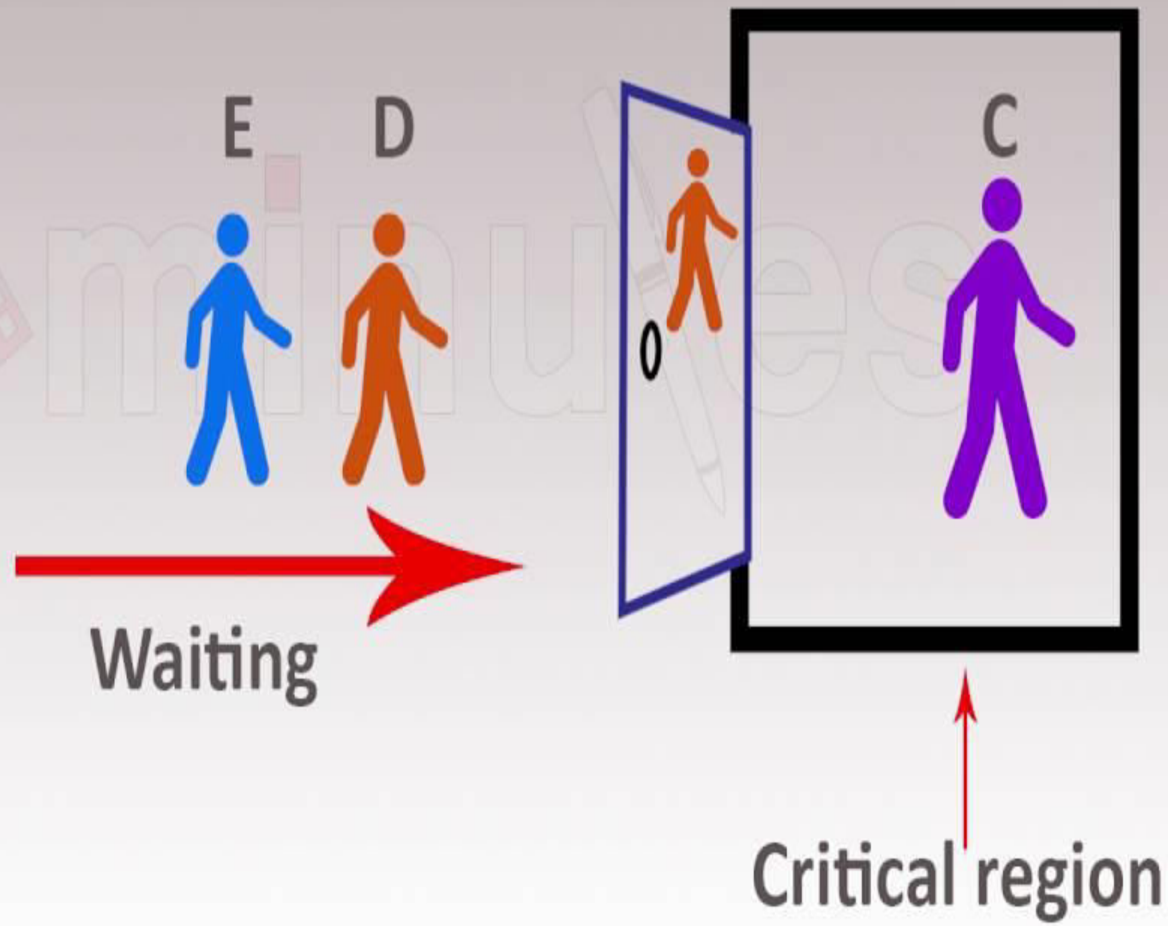
I am john's friend mac and I have to transfer amount to him.

Uncoordinated accesses to shared data (d_s) may affect consistency of data. In this example, d_s is John's balance that is getting affected by John himself and his friend Mac.

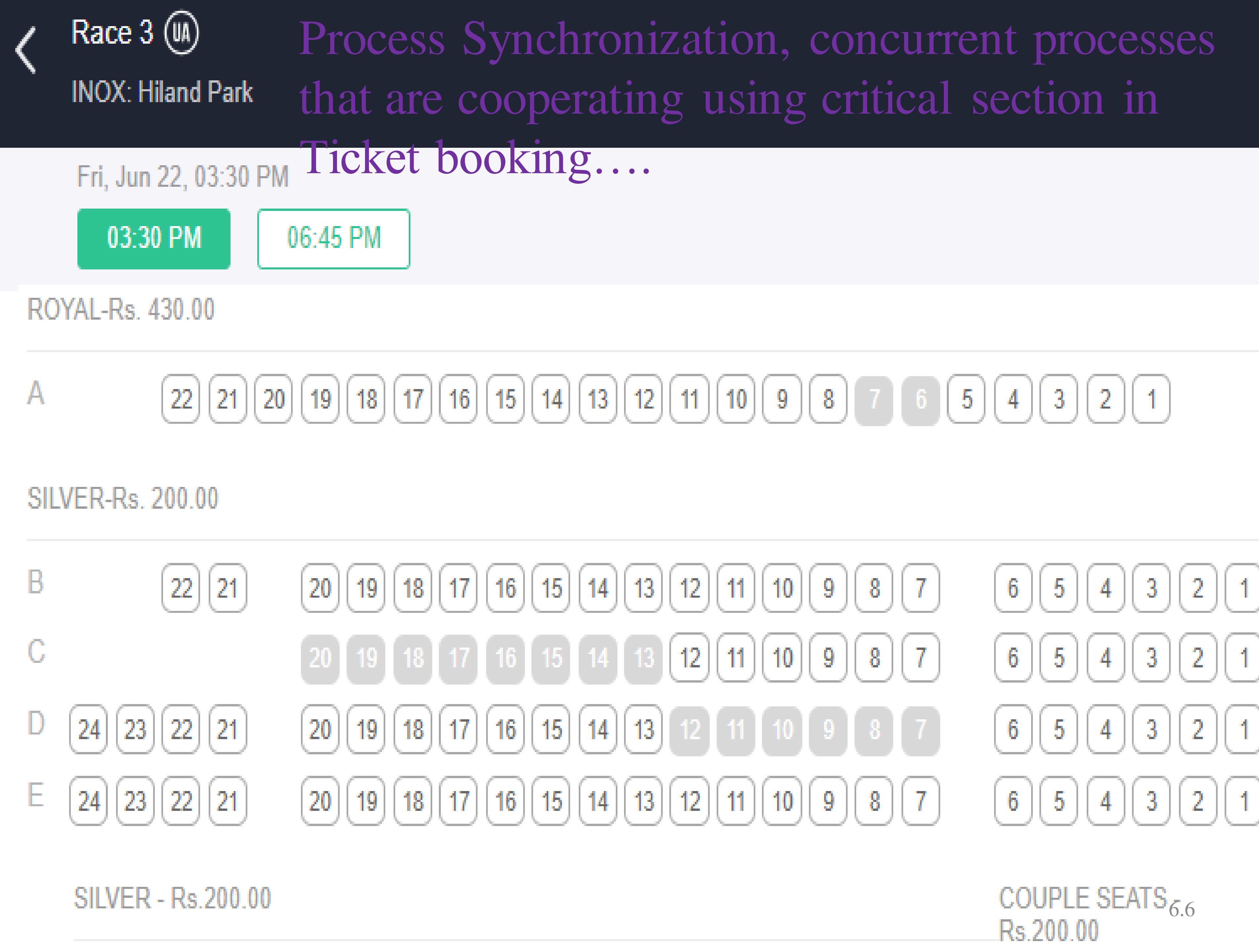
PROCESS SYNCHRONIZATION

THE CRITICAL REGION

When **C** is in Critical Region, C has got full control, like, you are holding a token at bank counter for cash withdrawal. As long as the token is with you, O, E and D has to ***wait***. When you ***release*** your token and ***signal*** that you are out of critical region, O, D (followed by E) would be able to gain access to CR one by one.



Process Synchronization, concurrent processes that are cooperating using critical section in Ticket booking….

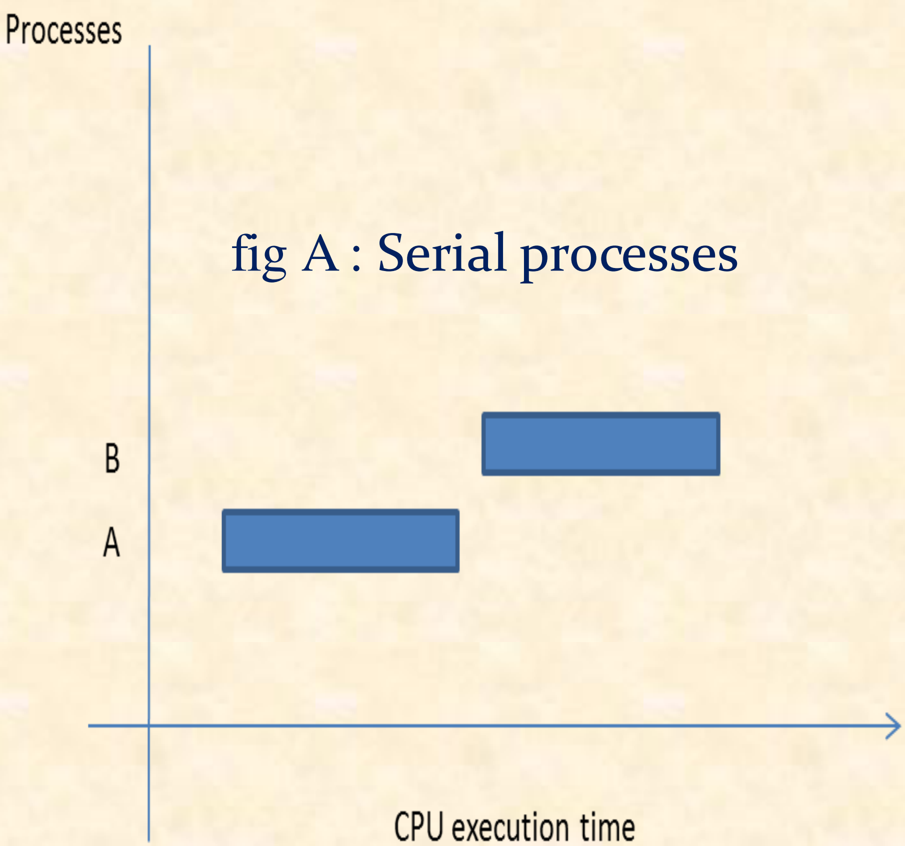


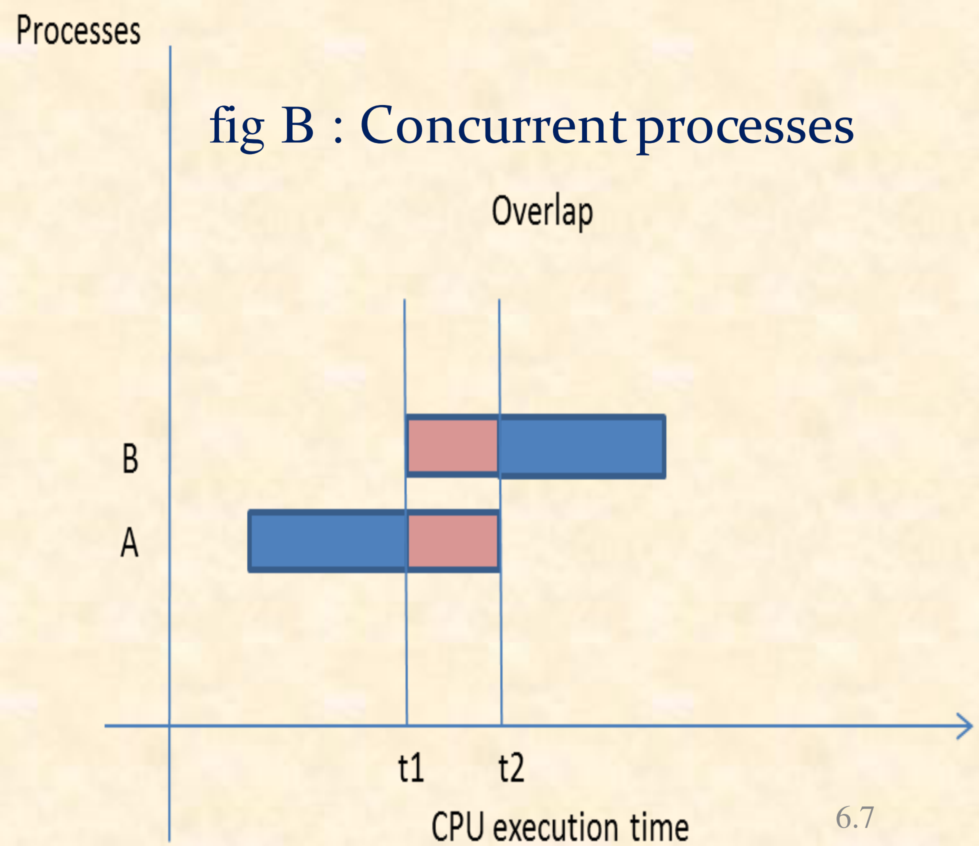
Concurrency

- 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.



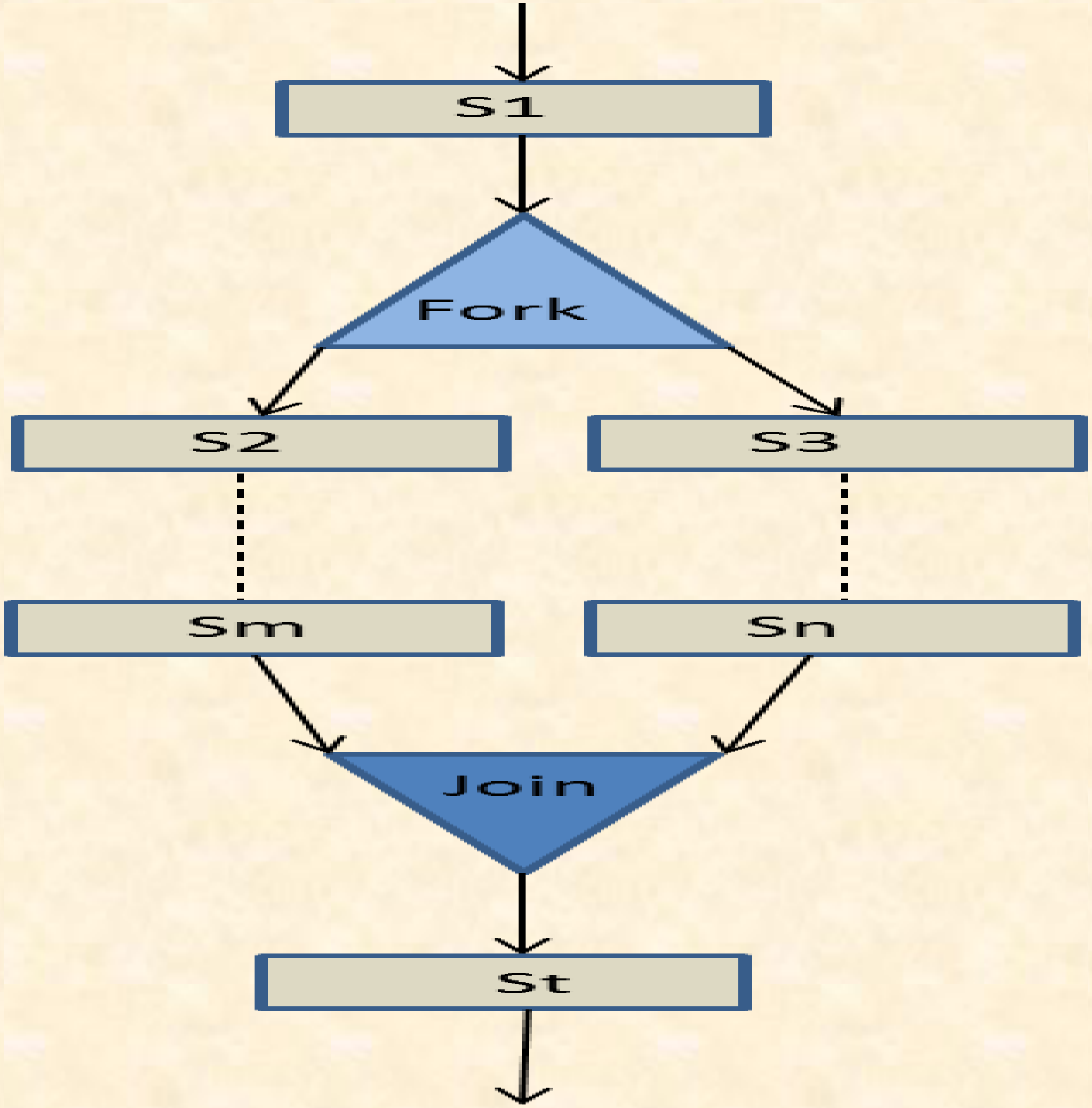


Categories of Concurrent Processes

- 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 interect with each other. Example: when an application program does not share data with other executing processes, its called independent process.
  - ***Cooperating processes*** - when execution of processes is affected by each other and they share data

Programming Language Constructs for Concurrency

Fork and Join Construct: This construct specifies concurrency in a program whereby the control flow is split into two or more flows that can be executed in parallel. Later on, the two parallel flows join back into a single flow.



After Statement S1, the control flow has forked into two independent statement flows, 'S2-Sm' and 'S3-Sn'. Both concurrent flows join at statement St.

Programming Language Constructs for Concurrency - *Parbegin– Parend Construct*

- This is a structured construct that specifies concurrency in a given program.
- The general format of this construct is given below:

parbegin

S1

S2

-

-

Sn

parend

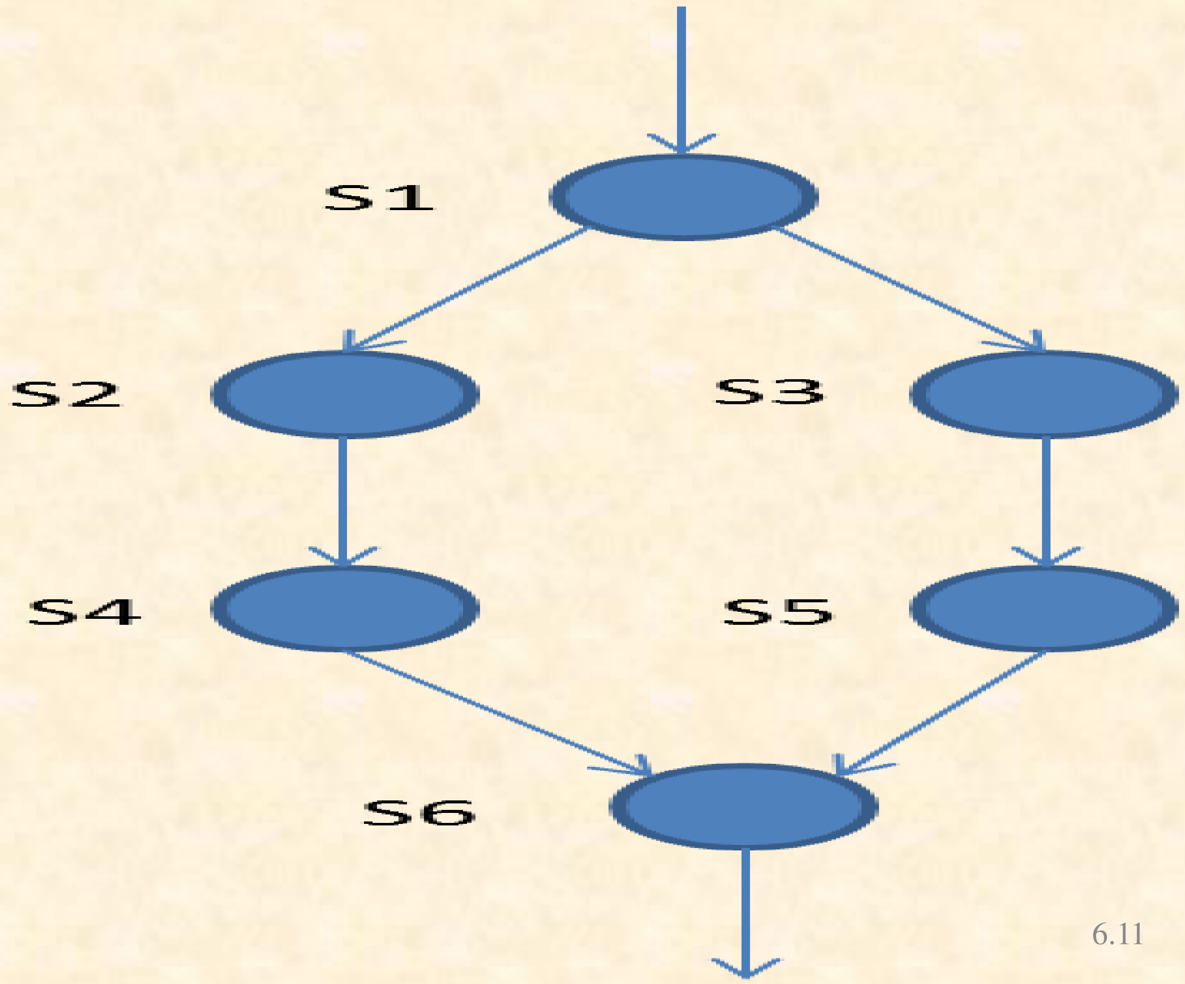
Here, parbegin and parend are keywords, and the statements S1, S2, …..Sn are concurrent statements. Statements enclosed within the normal begin-end pair are considered sequential statements.

Example of parbegin and parend construct and precedence graph

```
begin
   S1
   parbegin
         begin
               S2;
               S4;
         end
         begin
               S3;
               S5;
         end
   parend

   S6
end
```

Within the 'parbegin-parend' pair, there are two sequential instruction streams, S2-S4 and S3-S5, that execute concurrently.



Before we turn to communication between processes, mutual exclusion, critical section (critical region) ➔ learn how to synchronize a set of cooperating and/or competing processes, let's check ***Data Access & Control Synchronization & Race condition***

- The term ***process*** is a generic term for both a process and a thread.
- What are Interacting processes? Processes Pi and Pj are interacting if the write_set of one of the processes overlaps the write_set or read_set of the other.
- *Process synchronization* includes **Data Access Synchronization** and **Control Synchronization**.

- **Data Access Synchronization:** Let a_i and a_j be operations on ***shared data*** d_s performed by two processes P_i and P_j.
- $f_i(d_s)$ represents the value of d_s resulting from changes, if any, caused by operation a_i.
- **Race conditions** arise if processes access shared data in an uncoordinated manner, i.e., if the result of execution of a_i and a_j in processes P_i and P_j is other than $f_i(f_j(d_s))$ or $f_j(f_i(d_s))$.
- **Data access synchronization is used to access shared data in mutually exclusive manner.**
- **Control Synchronization:** This is needed if a process performs action a_i only after some other processes have executed a set of actions $\{a_j\}$.

Race Conditions

- Uncoordinated accesses to shared data may affect consistency of data
- Consider processes P_i and P_j that update the value of d_s through operations a_i and a_j , respectively:

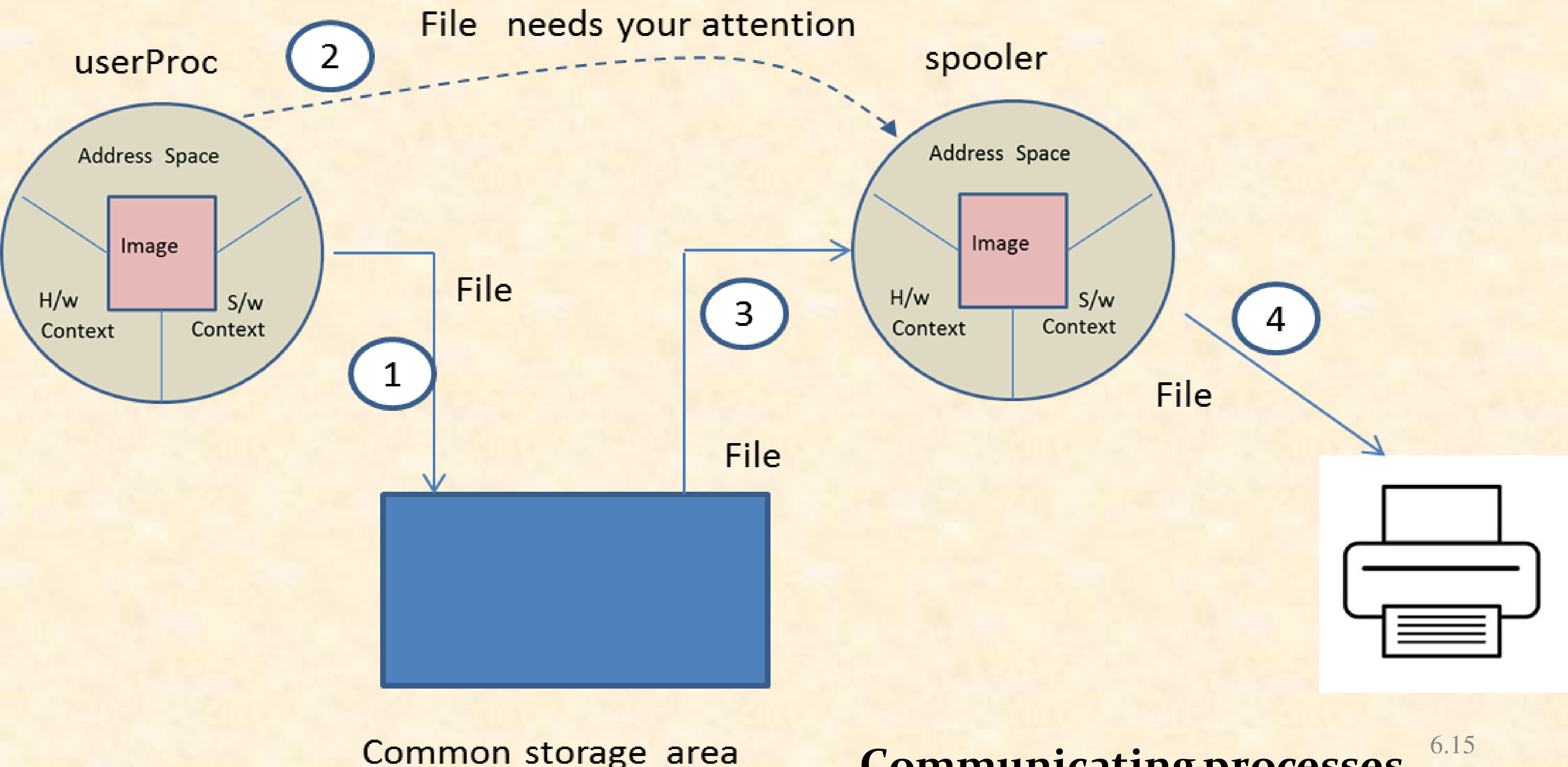
  Operation a_i : $d_s := d_s + 10$; Let $f_i(d_s)$ represent its result

  Operation a_j : $d_s := d_s + 5$; Let $f_j(d_s)$ represent its result

  – What situation could arise if they execute concurrently?
  – Say, d_s has initial value 5. Then, $f_i(d_s)$ is giving 15,and the concurrent operation, $f_j(d_s)$ generates output 10!!! d_s cannot have two different values for the same data item….. We have to synchronize the operation of a_i and a_j…..

Communication Between Processes

In the case of cooperating processes, inter-process communication becomes a necessity. For instance, consider a situation where the printing jobs in an operating system are handled by a 'spooler' process. Now, as soon as a user process (say, userProc) is ready to print a file on a printer, it must store the file as a common storage area from where the spooler can also load the file. Thereafter, the 'userProc' must communicate to the 'spooler' process that there is a file that needs its attention….



Communicating processes

Communication Between Processes

It may be noted that the concurrent processes communicate with each other through *shared variables* and *messages*. ***Shared variables*** are common variables or data items which are accessible to communicating processes. ***Messages*** are information or signals which are exchanged by communicating processes.

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

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 plotter cannot be shared among processes. Unless the job of one process is completed, the job of another process cannot be taken by the plotter. Similarly, two users cannot be allowed to write on the same file at the same time.

Key points regarding ➔ Mutual Exclusion and Synchronization

- A sharable (**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. AA.

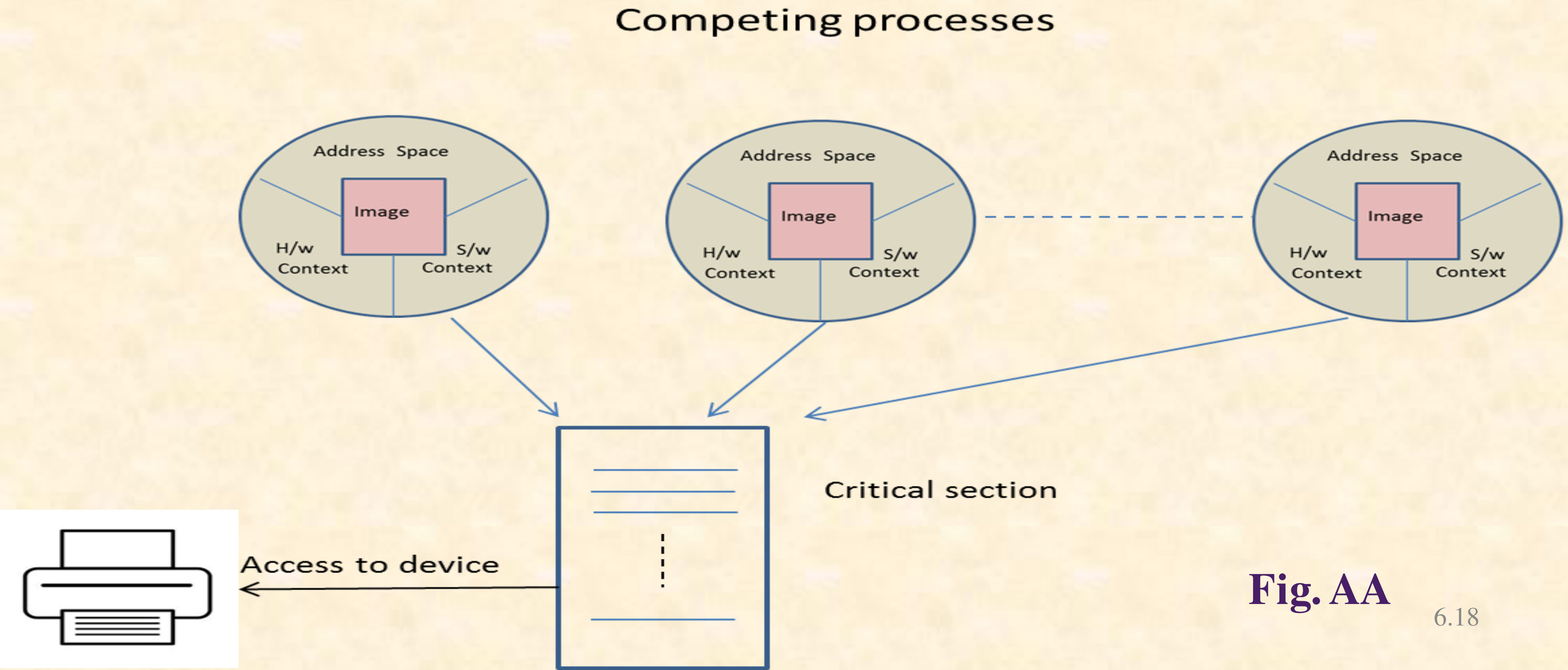


Fig. AA

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 t execute the CS, then only one is allowed to execute it, and the other is made to wait.
- The general format is given below

{

Non-critical section

<entry section>

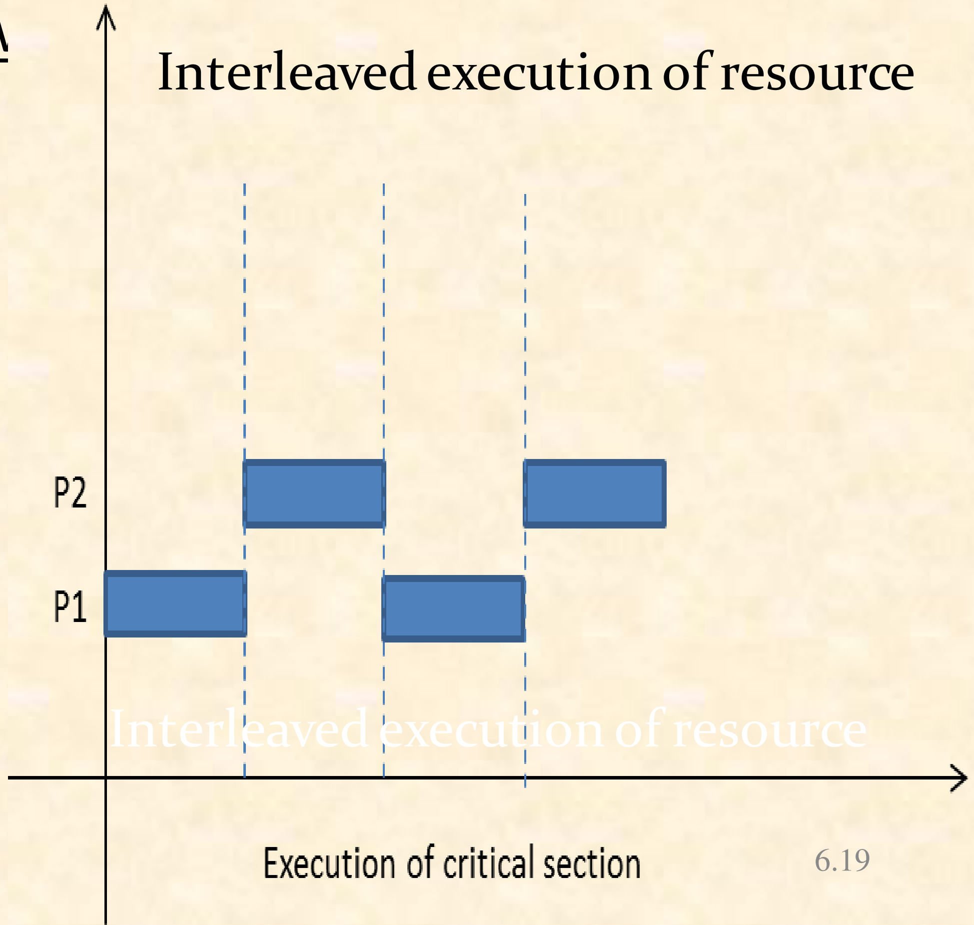
Critical Section

<exit section>

Non-critical section

}

The CS is guarded between two sections, the <entry> and <exit> sections.



Critical section

- ✓ A piece of code that only one task can execute at a time.
- ✓ If multiple tasks try to enter a critical section, only one can run and the others will sleep.

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 processes can gain entry to a CS for d_s ahead of P_i is bounded by a finite integer.

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 *ds, 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 *ds—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* ≤ *capacity*
then
allotedno:=nextseatno;
nextseatno:=nextseatno+1;
else
display "sorry, no seats available";

Process P_i

if *nextseatno* ≤ *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.

Control Synchronization and Indivisible Operations

- 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.***

Control Synchronization and Indivisible Operations….

attempt at signaling through boolean variables.

var
    operation_aj_performed : boolean;
    pi_blocked : boolean;
begin
    operation_aj_performed := false;
    pi_blocked := false;
Parbegin

Process P_i	Process P_j
. . .	. . .
if *operation_aj_performed = false*	{*perform operation* a_j}
then	**if** *pi_blocked = true*
pi_blocked := true;	**then**
block (P_i);	*pi_blocked := false;*
{*perform operation* a_i}	*activate* (P_i);
. . .	**else**
. . .	*operation_aj_performed := true*
. . .	. . .

Parend;
end.

Control Synchronization and Indivisible Operations…

Indivisible Operation: An operation on a set of data items that cannot be executed concurrently either with itself or with any other operation on a data item included in the set.

```
procedure check_aj
begin
    if operation_aj_performed=false
     then
        pi_blocked:=true;
        block (Pi)
end;

procedure post_aj
begin
    if pi_blocked=true
     then
        pi_blocked:=false;
        activate(Pj)
    else
         operation_aj_performed:=true;
end;
```

Indivisible operations $check\_a_j$ and $post\_a_j$ for signaling

Synchronization Approaches

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

Looping versus Blocking

❑ ***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.

Looping versus Blocking (continued)

- To avoid busy waits, a process waiting for entry to a CS is put in ***blocked* state**
  - Changed to ***ready state*** only when it can enter the CS

if (some process is in a critical section on $\{d_s\}$ or
is executing an indivisible operation using $\{d_s\}$)
then *make a system call to block itself;*

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

- Process decides to loop or block
  - The above decision is subject to race conditions. Avoided through 1) *Algorithmic approach, 2)* Use of computer hardware features

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*;
lock := *closed*;

Performed by 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

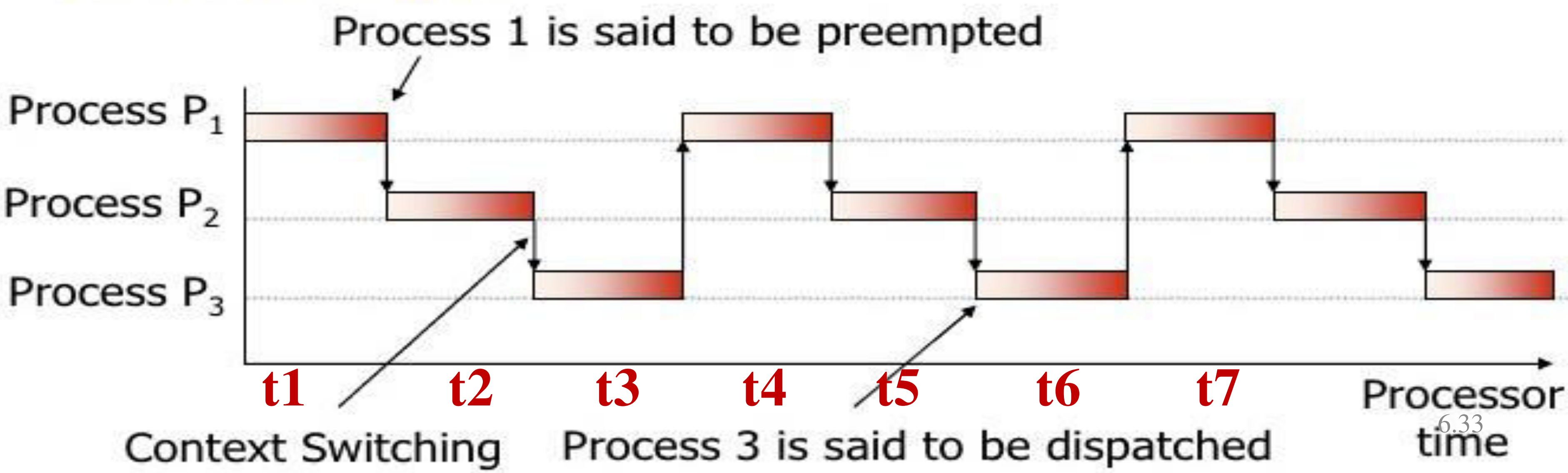
- Algorithmic Approaches
  - For implementing mutual exclusion
  - Independent of hardware or software platform
    - Busy waiting for synchronization
- **Synchronization Primitives**
  - **Implemented using indivisible instructions**
  - **E.g., *wait* and *signal* of *semaphores***
- Concurrent Programming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

Structure of Concurrent Systems

Processes entering the system following the order → P1 – P2 – P3. At time instant t1, CPU is allocated to P1, P2 and P3 are in ready queue. Then, like RR scheduling, P2 occupies CPU, and P1 is waiting (blocked), P3 ready state. This way, as time flows in x-axis, processes proceed with CPU and complete the execution.

– **Interacting processes: A *snapshot* of a concurrent system is a view of the system at a specific time instant**

- Here several processes share the same CPU and other common resources.



Classic Process Synchronization Problems

- **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***

CS must satisfy the following *correctness conditions*:

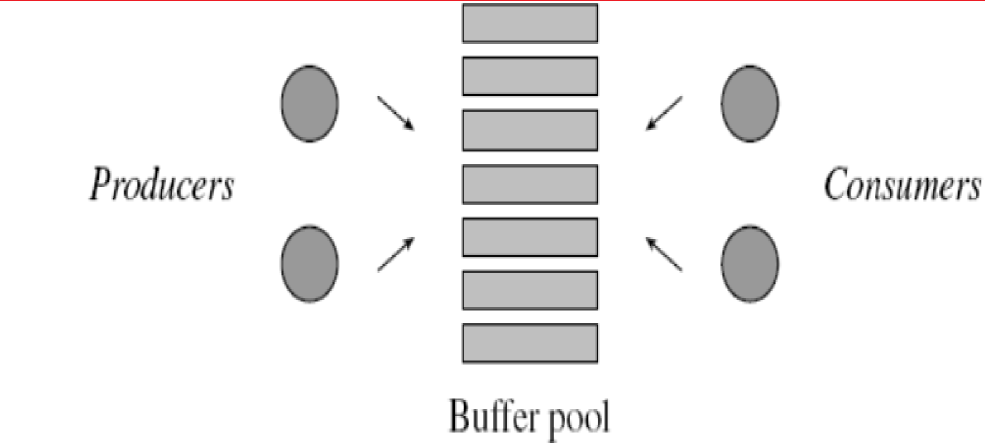
- **Mutual exclusion**: Only one process is allowed to enter into the critical section that guards a shared resource. At any moment, at MOST one process may execute a CS for a data item d_s.
- **Progress:** If critical section is available for execution then only the processes can participate in the 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 processes can gain entry to a CS for d_s ahead of P_i is bounded by a finite integer.

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*.

Producers-Consumers with Bounded Buffers

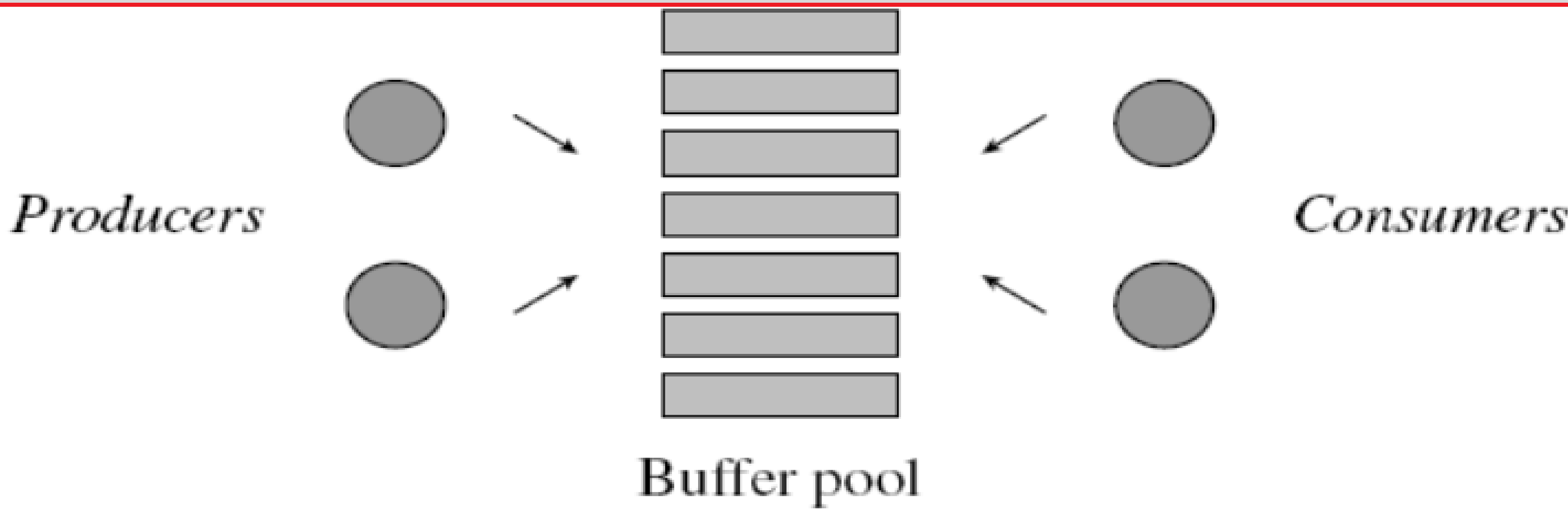
- A producers-consumers system with bounded buffers consists of an unspecified number of producer and consumer processes and a finite pool of buffers.
- Each buffer is capable of holding one record of information—it is said to become ***full*** when a producer writes into it, and ***empty*** when a consumer 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;              var consumed : boolean;
  repeat                               repeat
     produced := false                    consumed := false;
     while produced = false               while consumed = false
        if an empty buffer exists            if a full buffer exists
        then                                 then
           { Produce in a buffer }              { Consume a buffer }
           produced := true;                    consumed := true;
     { Remainder of the cycle }           { Remainder of the cycle }
  forever;                             forever;
Parend;
end.
          Producer                             Consumer
```

An outline for producers–consumers using critical sections.

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

Producers-Consumers with Bounded Buffers…

- 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;
```

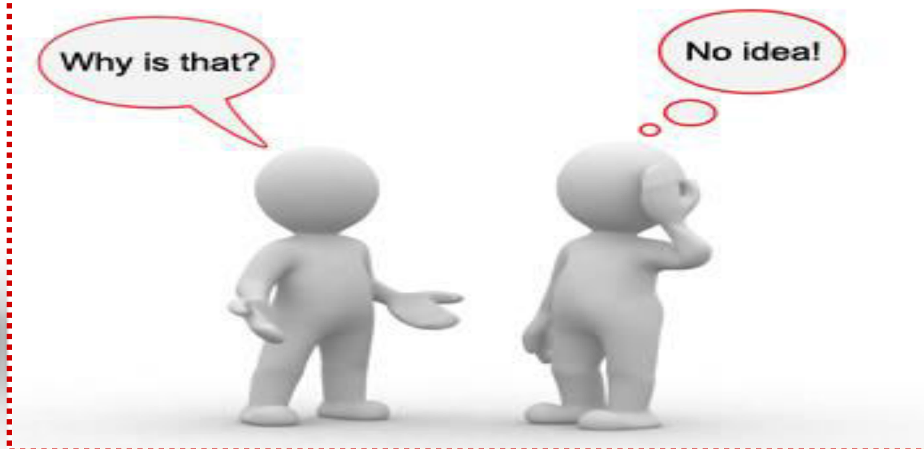
Operations of producer

```
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 :

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

Ans: a

❑ Which resources may be protected from simultaneous access?

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

Ans: b

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

a. Critical section
b. Mutual exclusion
c. Synchronization
d. All of the above

Ans: c

- The condition that allows only one process to enter the critical section is called:

Ans: b

a. Live-lock
b. Mutual exclusion
c. Synchronization
d. All of the above

- A critical section must satisfy:

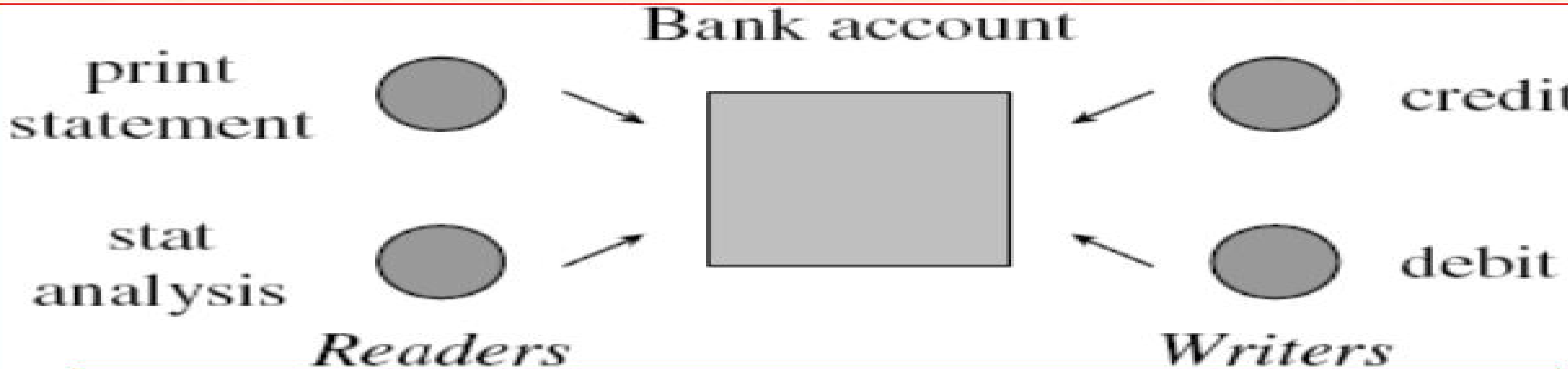
a. Correctness
b. Progress
c. Fairness
d. All of the above

Ans: d

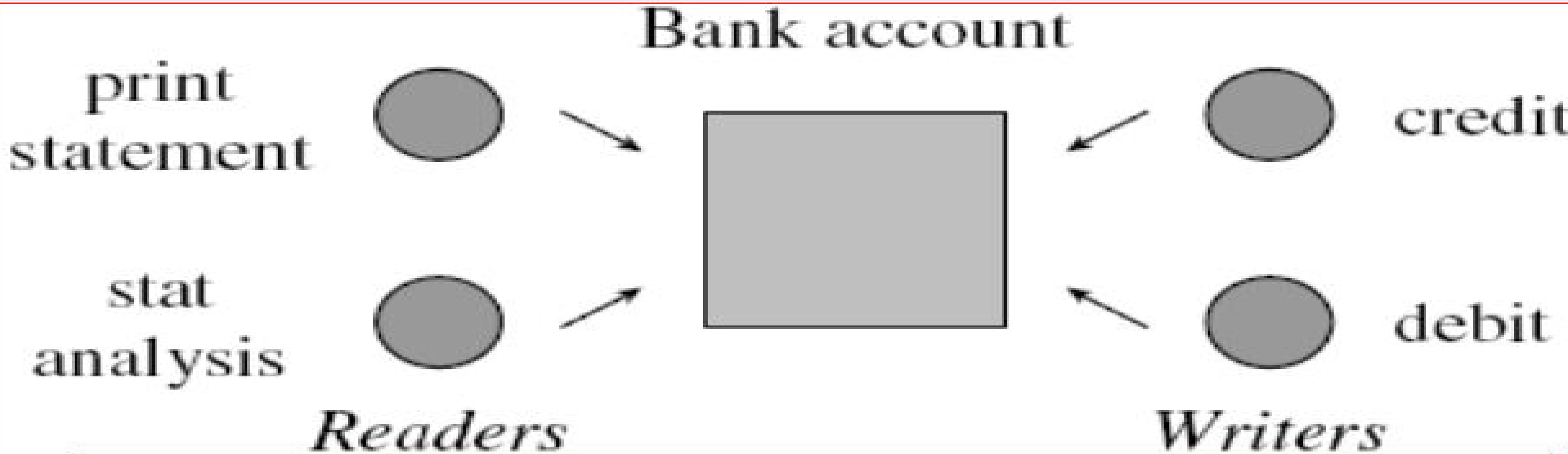
After Q-n-A session, let's come back to *Readers and Writers*

- 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......

Reader(s)

```
Parbegin
  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;
Parend;
end.
```

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;
```

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

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*

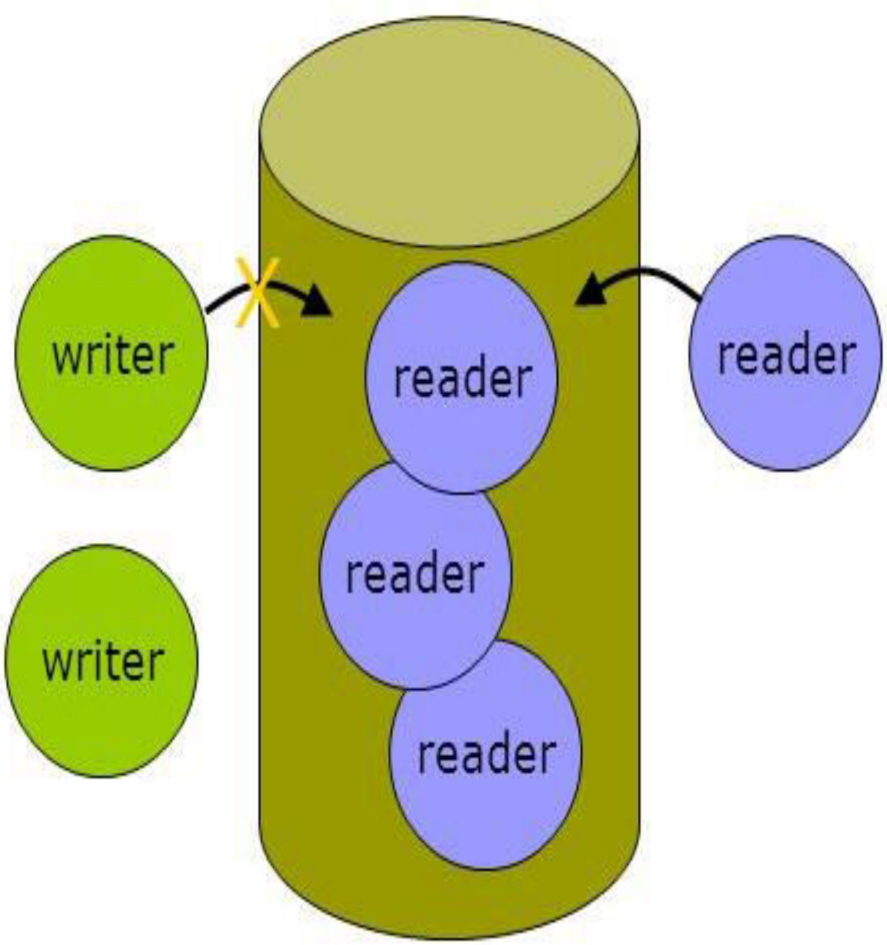
❑ ***Condition 3*** requires that a writer should perform writing in a critical section. When it finishes writing, it should activate one waiting writer or activate all waiting readers. This can be achieved using a signaling arrangement.

❑ 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.

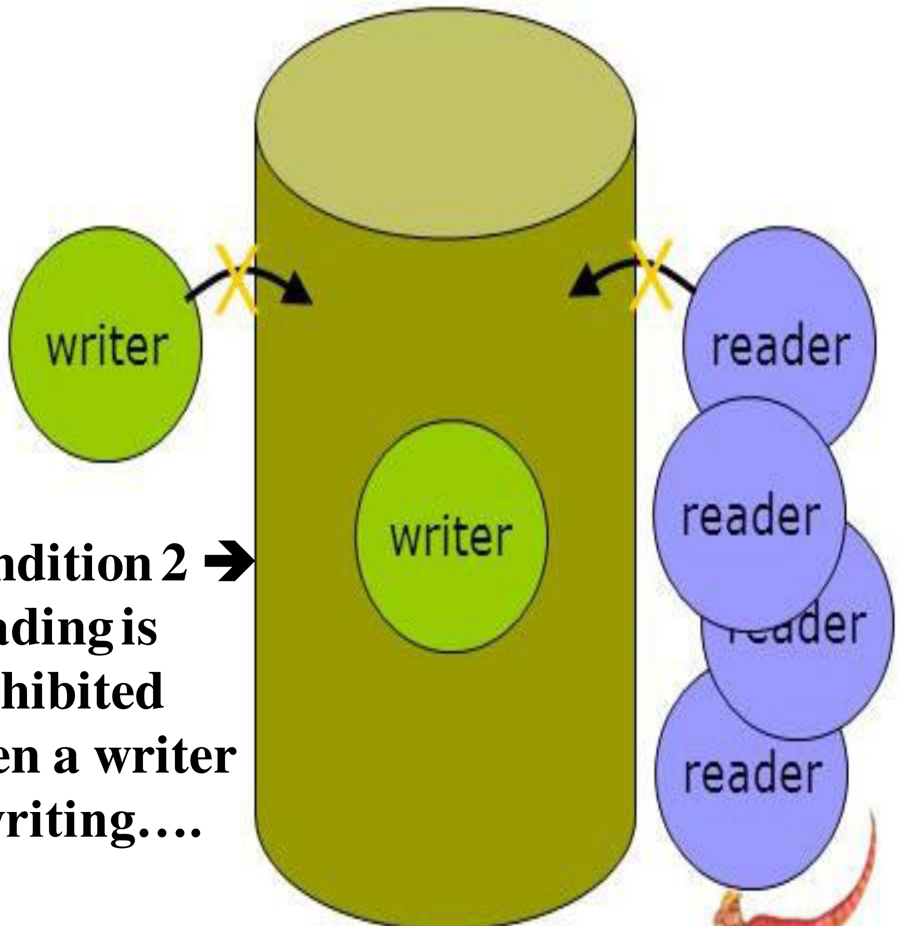
❑ Based on Fig. AB, writing is performed in a CS. A CS is not used in a reader as that would prevent concurrency between readers. The outline in fig. AB, does not satisfy the bounded wait condition for both readers and writers, however, it provides maximum concurrency.

The Readers-Writers Problem

- Multiple readers or a single writer can use DB.



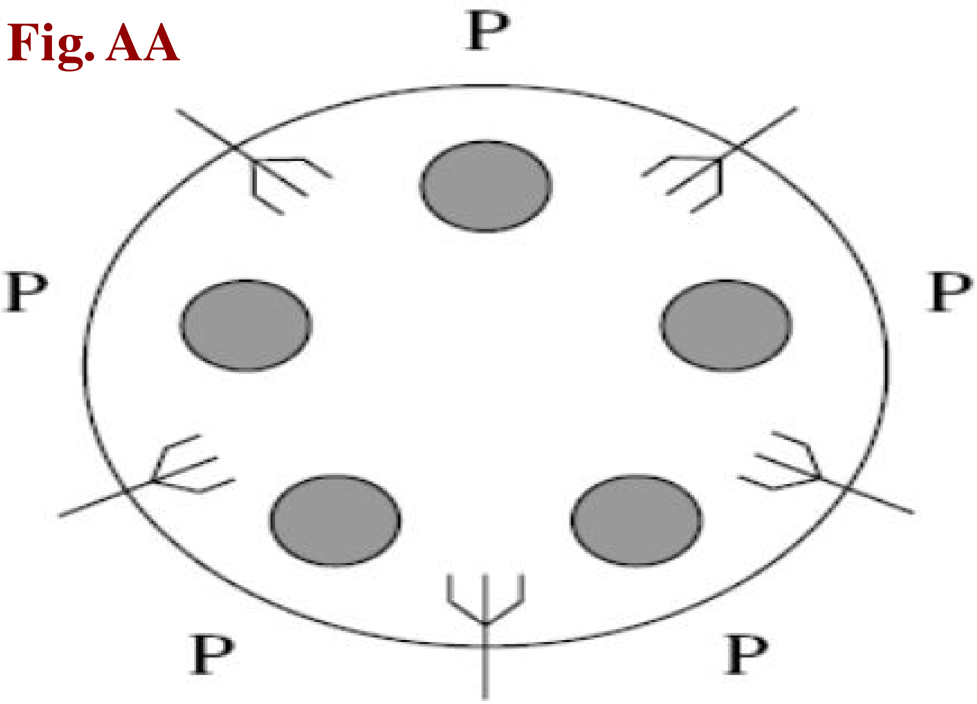
Condition 2 ➔ Reading is prohibited when a writer is writing….




Lets Get Started

Dining Philosophers

Fig. AA



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….

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
```

Fig. A1

Outline of a philosopher process P_i.

- 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….

```
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
```

Fig. A2

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 (slide 6.56).**

Dekker's Algorithm for Two-process CS Problem

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's Algorithm for Two-process CS Problem

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.

Goodbye.

{for now}