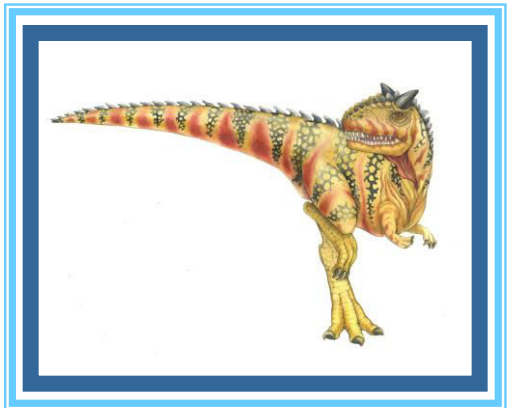
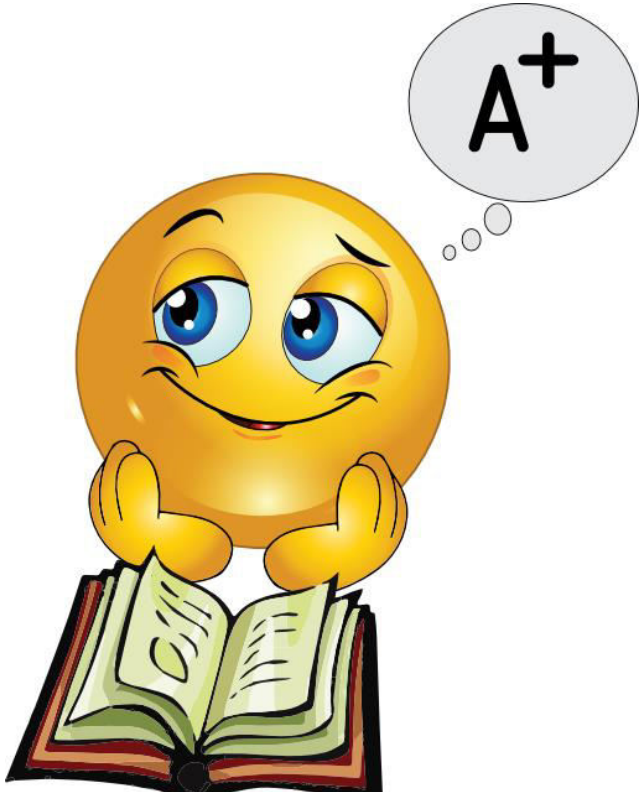
Process Synchronization



Modes of Execution

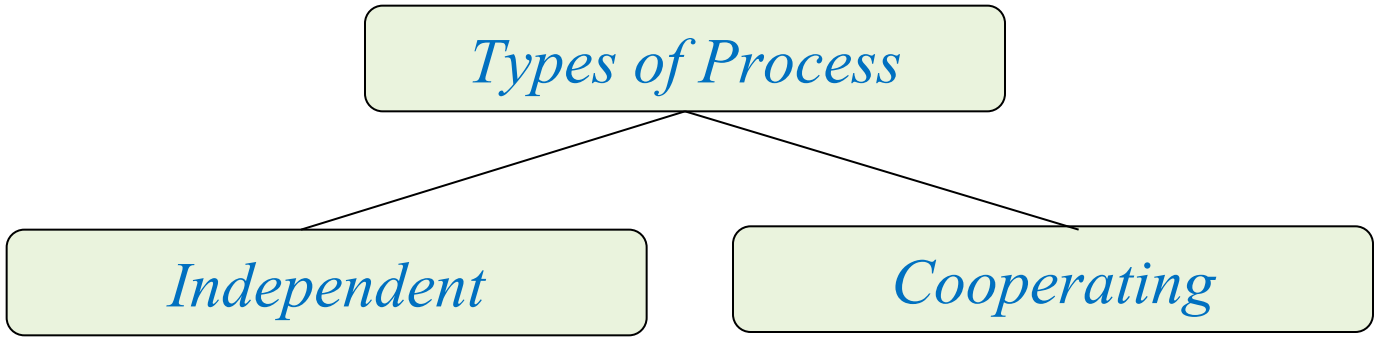
Processes can execute in either

- ***Synchronous/Serial Mode:***
- ***Concurrent/Asynchronous/Parallel Mode:***



Types of Processes

- *Processes are categorized as one of the following two types:*
- ***Independent Process*** *: Execution of one process does not affect the execution of other processes.*
- ***Cooperative Process :*** *Execution of one process affects the execution of other processes.*



Process Synchronization

***Problem:* Concurrent access to shared data may result in data inconsistency.**

- *When two or more process concurrently accessing to the shared data, their order of execution must be preserved otherwise there can be conflicts in their execution and inappropriate outputs can be produced (data inconsistency, integrity risk).*

Process Synchronization

- ***Process Synchronization*** *is a technique which is used to coordinate the process that use shared Data.*
- *Process synchronization problem arises in the case of* ***Cooperative process*** *also because resources are shared in Cooperative processes.*

Race Condition

During the concurrent execution of the processes, where several processes access and manipulate the same data concurrently and outcome of the data depends over particular order in which access takes place is called ***RACE condition.***

Or

When order of execution can change result is called Race Condition.

Race Condition (Example 1)

P1(){	P2(){
Read(a)	Read(a)
a=a+1	a=a+1
write (a)	write(a)
}	}

- *If a=10 and p1 , p2 executes serially then final value of a will be 12.*
- *If process P1 context switch after Read(a) and then P2 executes (concurrent fashion), inconsistent result (a=11) may occur.*

Race Condition (Example 2)

int z = 10;

P1{	p2(){
int x = z	int y = z
x = x+1	y = y-1
z = x	z=y
}	}

If process P1 and P2 executes serially then result is 10.

Race Condition (Example 2)

int z = 10;

P1{	p2(){
int x = z	int y = z
x = x+1	y = y-1
z = x	z=y
}	}

If process P1 and P2 executes above code in concurrent fashion (p1 context switch after first instruction or p2 context switch after first instruction) inconsistent results (11/9) may come.

Race Condition (Producer Consumer)

Producer Process:

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

Consumer Process:

```
while (true)
{
     while (counter == 0)
        ; /* do nothing */
     nextConsumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     counter--;
     /* consume the item in nextConsumed */
}
```

Race Condition (Producer Consumer)

Producer:

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

Consumer:

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

Interleaving:

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

Race Condition (Producer Consumer)

- *Unfortunately we have now introduced a new problem, because both the producer and the consumer are adjusting the value of the variable counter, which can lead to a condition known as a* ***race condition****.*
- *In this condition a piece of code may or may not work correctly, depending on which of two simultaneous processes executes first, and more importantly if one of the processes gets interrupted such that the other process runs between important steps of the first process.*

Critical Section Problem

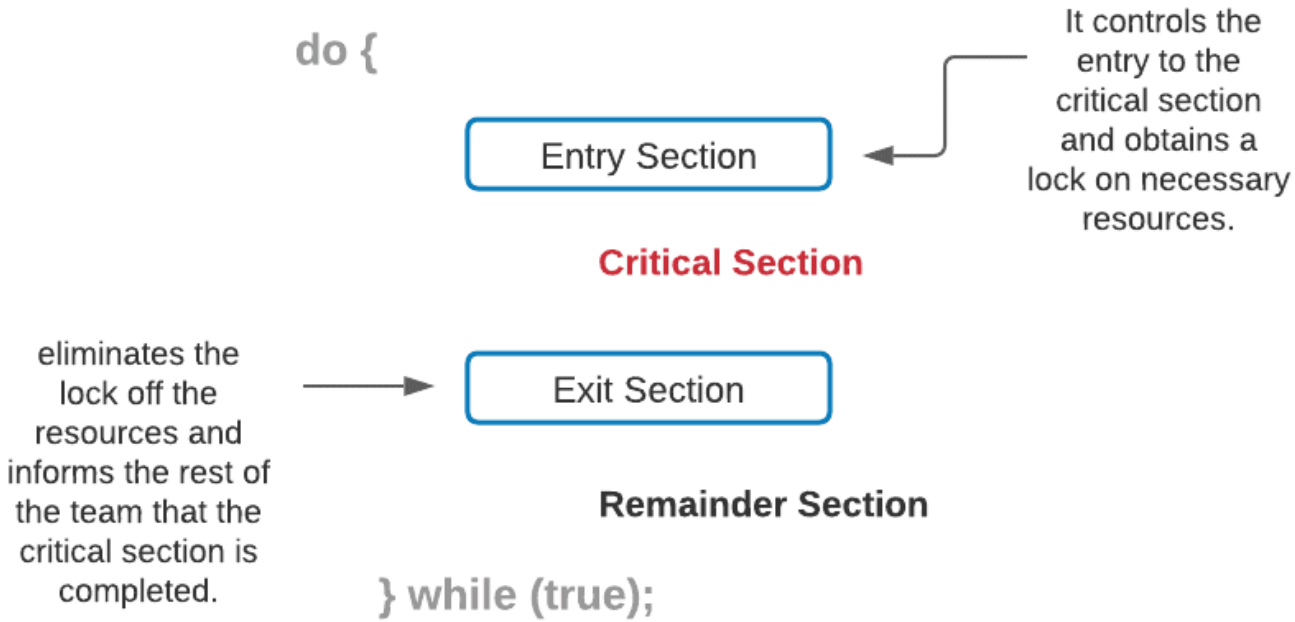
- *A critical section refers to a portion of a computer program or code that must be executed by only one process or thread at a time.*
- *The purpose of a critical section is to ensure that shared resources or data are accessed and modified in a way that avoids conflicts and maintains data integrity.*

Critical Section Problem

- *Every process has a reserved segment of code which is known as Critical Section. In this section, process can change* ***common*** *variables, update tables, write files, etc. The key point to note about critical section is that when one process is executing in its critical section, no other process can execute in its critical section.*
- *No two process can execute in their critical section at same time.*
- *Critical Section Problem: CSP is to design a protocol that the processes can use to cooperate.*

Critical Section

- ***Entry Section:*** *Each process must request for permission before entering into its critical section and the section of a code implementing this request is the Entry Section.*
- ***Exit Section:*** *Other processes waiting in the Entry Section are able to enter the Critical Sections through the Exit Section*



Critical Section Properties

Any solution to the critical section problem must satisfy three requirements:

- ***Mutual Exclusion*** *: If a process is executing in its critical section, then* ***no other process is allowed to execute in the critical section****.*

Mutual Exclusion

Progress

Bounded Waiting

Critical Section Properties

- ***Progress*** *: If no process is executing in the critical section and other processes are waiting outside the critical section, then only those processes that are not executing in their remainder section can participate in deciding which will enter in the critical section next, and the selection can not be postponed indefinitely.*

Or

If a process is not using the critical section, then it should not stop any other process from accessing it.

Critical Section Problem

- ***Bounded Waiting*** : *Bounded waiting means that each process must have a limited waiting time. It should not wait endlessly to access the critical section.*

CSP solution using Turn Variable (Wrong)

P0

```
while(true)
{
    While (Turn != 0);

    Critical Section

    Turn = 1;

    Remainder Section

}
```

P1

```
while(true)
{
    While (Turn != 1);

    Critical Section

    Turn = 0;

    Remainder Section

}
```

Mutual exclusion is there but progress is not there.
Strict alternation is there between processes.

Critical Section problem solution using flag

P0

```
while(true)
{
    flag [0] = T;
    while (flag[1]);

    Critical Section

    flag[0]=F;
}
```

P1

```
while(true)
{
    flag [1] = T;
    while (flag[0]);

    Critical Section

    flag[1]=F;
}
```

Mutual exclusion is there but progress is not there system will go in deadlock if context switch happen after first statement flag[0] and p1 and p2 both are interested to get enter into critical section

Peterson's Solution (software based solution)

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

Figure 6.2 The structure of process P_i in Peterson's solution.

Peterson's Solution (software based solution)

P0

```
while(true)
{
    flag [0] = T;
    turn = 1;
    while (turn==1 && flag[1]
    ==T);

        Critical Section

    flag[0]=F;

}
```

P1

```
while(true)
{
    flag [1] = T;
    turn = 0;
    while (turn==0 && flag[0]
    ==T);

        Critical Section

    flag[1] = F;

}
```

Peterson's Solution (Limitations)

1. **Peterson's solution** works only for two processes, but this solution is best scheme in user mode for critical section.
2. Software-based solutions (**Peterson's solution**) are not guaranteed to work on modern computer architectures.

Problem 1

Consider the methods used by processes P1 and P2 for accessing their critical sections whenever needed, as given below. The initial values of shared boolean variables S1 and S2 are randomly assigned

P1	P2
while (S1 == S2) ;	***while (S1 != S2) ;***
Critica1 Section	Critica1 Section
S1 = S2;	***S2 = not (S1);***

Which one of the following statements describes the properties achieved?
(A) *Mutual exclusion but not progress*
(B) *Progress but not mutual exclusion*
(C) *Neither mutual exclusion nor progress*
(D) *Both mutual exclusion and progress*

Problem 2

Two processes X and Y need to access a critical section. Consider the following synchronization construct used by both the processes.

Process *X*

```
/* other code for process X */
while (true)
{
      varP = true;
      while (varQ = = true)
      {
      /* Critical Section */
            varP = false;
      }
}
/* other code for process X */
```

Process *Y*

```
/* other code for process Y */
while (true)
{
      varQ = true;
      while (varP = = true)
      {
            /* Critical Section */
                  varQ = false;
      }
}
/* other code for process Y */
```

Problem 2

Two processes X and Y need to access a critical section. Consider the following synchronization construct used by both the processes.

Process *X*

```
/* other code for process X */
while (true)
{
    varP = true;
    while (varQ = = true)
    {
    /* Critical Section */
        varP = false;
    }
}
/* other code for process X */
```

Process *Y*

```
/* other code for process Y */
while (true)
{
    varQ = true;
    while (varP = = true)
    {
        /* Critical Section */
            varQ = false;
    }
}
/* other code for process Y */
```

Here, varP and varQ are shared variables and both are initialized to false. Then no mutual exclusion

Problem 3

Consider Peterson's algorithm for mutual exclusion between two concurrent processes i and j.

The program executed by process is shown below.

```
repeat
    flag [i] = true;
    turn = j;
    while ( P ) do no-op;
    Enter critical section, perform actions, then exit critical
    section
    flag [ i ] = false;
    Perform other non-critical section actions.
until false;
```

For the program to guarantee mutual exclusion, the predicate P in the while loop should be.

(A) flag[j] = true and turn = i

(B) flag[j] = true and turn = j ⟵