Ch#6

Process Synchronization

Definitions:-

- "Process synchronization" was introduced to handle problems that arose while multiple process executions.
- Process is categorised into two types:
  - Cooperative Process
  - Independent Process

Independent process:

- Two processes are said to be independent if the execution of one does not affect the other process.

Cooperative process:-

- Two process are called Cooperative process if the execution of one Process affects the execution of another process.
- These processes need to be synchronized so that the order of execution can be guaranteed.

Process Synchronization:

- It is the task phenomenon of coordinating the execution of processes in such a way that no two processes can have access to the same shared data and resources.
- it is a procedure that is involved in order to preserve the appropriate order of Cooperative processes.
- In order to synchronize the process, there are various mechanism to synchronize proc
- it is done to maintain data Consisten
- It is mainly needed in a multi-pro system.
- Data Consistency is maintained by usin variables or hardware so that one procen can make changes to the shared memory a time.

Need of Synchronization:

- Multiple processes execute at the same time and sharing the same resources.
- To avoid the inconsistent results.

What is critical Section?

- Critical section is a section of the program where a process access the shared resources during its execution.
- Only one process can access the critical section at a time or no two processes can access the critical section at the same time. otherwise race condition occurs.
- Suppose a system Containing n processes $[P_0, P_1, P_2, \ldots, P_n]$. Each process has a segment of code called a critical section in which the process may be changing common variables, updating a table, writing into files etc.
- if any other process also wants to execute its critical section, it must wait until the first one finishes.
- wait() function is used for handling the entry to the critical section.
- Signal() function is used for exit from the critical section.

```
do{
    [entry section]
        Critical Section
    [exit section]
        Remainder Section
} While (TRUE);
```

Entry Section:
- In this section mainly the process requests for its entry in the critical section.
- Controls the entry into critical section and gets a Lock on required resources.

Exit section:
- This section is followed by the critical section.
- it removes the Lock from the resources and let the others know that its critical section is over.

Race Condition

=> when more than two process at the same time is either executing the same code or accessing the same memory or any shared variable then there is a possibility that the output or the value of the shared variable is wrong so for that purpose all the processes are doing the race to say that my output is correct. This condition is commonly known as a race condition.

=> As several process access and process the manipulations on the same data in a concurrent manner and due to which the outcome depends on the particular order in which the access of data takes place.
- Several process compete with each other.
- Race condition is mainly occurs inside a critical section.
- It occurs in critical section when the result of multiple thread execution differs according to the order in which the thread executes.

What Criteria that the solution of critical section follow?

=> A solution of the critical section problem must satisfy the following three conditions:

(i) Mutual Exclusion: (mandatory criteria)
- out a group of cooperating process only one can be in its critical section at a given point of time.
- Only one process is executing in its critical section.

(ii) Progress:
- if no process is in its critical

section and some processes wish to enter in its critical section then only those processes can participate in that are not executing in their remainder section.

- Only those process should be competed or should be race ended in the critical section who actually wants to.
- It is a mandatory Criteria.

(3) Bounded wait:

- A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Critical Section problems:

=> The critical section problem is to design a protocol that the processes can use to synchronize their activity so as to cooperatively share data.

=> Each process must request permission to enter its critical section.

Two process Solution for critical Section:-

=> The following algorithms are used but first two has problem.

1) Turn variable
2) Flag variable
3) Peterson Solution or Dekkers algorithm

Turn variable: Algorithm 1:

P_0	P_1
While (1)	While (1)
{	{
while (turn != 0); → trap	while (turn != 1); → trap
Critical section;	Critical section;
turn = 1;	turn = 0;
Remainder section;	Remainder section;
}	}

=> it is a busy waiting solution.

Problems

- Progress requirement is not satisfied.
- it works in round robin fashion.

$P_0 \rightleftarrows P_1$

Flag variable: Algorithm 2:

P_0	P_1
while (1)	while (1)
{	{ flag [1] = T;
flag [0] = T;	while (flag [0]);
while (flag [1]);	C.S;
Critical section;	flag [1] = F;
flag [0] = F;	}
}	

if flag [1] == T then
do nothing;

Problem:

- Deadlock can occur if:
  flag [0] = flag [1] = T;

P_0	P_1
do	do
{	{
flag [0] = T;	flag [1] = T;
turn = 1;	turn = 0;
while (turn == 1 && flag [1] == T);	while (turn == 0 && flag [0] == T);
C.S;	C.S
flag [0] = F;	flag [1] = F;
}	}

=> This solution preserve all three conditions.
- Mutual exclusion
- Progress
- Bounded waiting

Peterson's algorithm

=> Widely used and software based solution to critical section problems.
=> Developed by peterson.
=> Combination of previous two

Bakery Algorithm multiple process Solution:-

=> This algorithm solves the critical section problem for n process.

=> Before entering its critical section, process receives a number.

=> Holder of the smallest number enters the critical section.

=> if processes Pi and Pj receive the same number, if i<j, then Pi is served first, else Pj is served first.

The numbering scheme always generates numbers in increasing order of enumeration; 1, 2, 3, 3, 3, 4, 5, ...

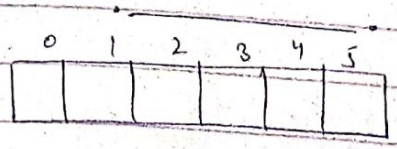
=> if the process receives the same number then alphabetically first will be served.

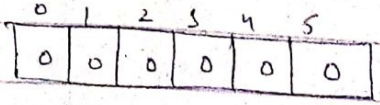
=> Process number starts from 1.

=> when a process finishes, it assigned 0 number.

=> Also known as Lesile Lampost Algorithm.

```
While (1)
{
    choosing[i] = true;
    number[i] = max(number[0], number[1], ..., number[n-1])
                +1;
    choosing[i] = false;
    for (j=0 ; j<n; j++)
    {
        while (choosing[j]);
        while( (number[j]!=0) && (number[j,j]<number[i,i]))
    }
    Critical section;
    number[i] = 0;
        remainder section;
}
```

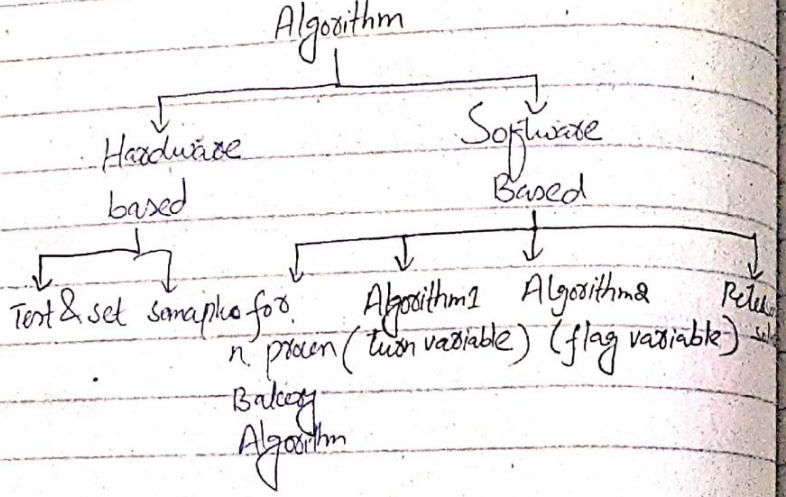




(a,b) < (c,d) = T
{a<c or if a=c and b< T }

Types of Algorithm to Solve Critical section problem:-

=> There are two main Category to Solve Critical section problems-



Hardware Solution to Synchronization

=> Hardware based Solution for the critical section problem introduces the hardware instructions that can be used to solve the critical section problem effectively.

=> This solution is not applicable in multi-processor system because disabling of interrupts is time-consuming.

=> it will ensures that no unpair modifications will be made to the shared variable.

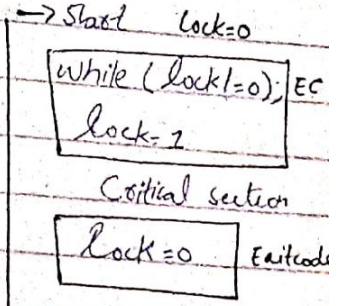
=> This method is often used by the systems with no preemptive kernels.

=> Hardware solution is based on simple variable lock.

=> The solution implies that before entering into the critical section the process must acquire lock and must release the lock when it exits its CS.

Lock variable:-

```
do
{
    acquire lock;
        CS
    release lock;
}
```



=> Execute in user mode.

=> multiprocessor solution

=> No mutual exclusion guarantee (Problem)

Test & Set instructions:-

```
While (1)
{
   while (TestAndset (lock));
            Critical section;
   lock = false;
            remainder section;
}
boolean  TestAndset (boolean &target)
   {
      boolean r = target;
      target = true;
      return r;
   }
```

Swap Algorithm

```
boolean lock=false
individual key=false
void Swap (boolean &a, boolean &b)
   {
      boolean temp = a;
         a=b;
         b=temp;
   }
while (1)
   key = true;
   while (key)
   { Swap (lock, key);
   }
   Critical section code;
   lock = false;
   Remainder section code;
}
while (key)
{
```

Working:

- Swaps the value of lock and key.
- key is set to true again in the second iteration of while(1)

Working of test and set:

- if lock = false, return false, lock = true
- if lock = true, return true, lock = true
- By default value of lock = F

Bounded wait Solution with TestAndTest:

```
While (1)
{
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = false;
```

entry section

```
    CS
    j = (i+1) % n;
    while ((j != i) && !waiting[j])
        j = (j+1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;
```

Exit section

```
}
```

Working:-

- Unlock and lock algorithm uses three boolean variables lock, key, and waiting[] for each process to regulate mutual exclusion.
- Unlock and lock algorithm maintains a ready queue for the waiting and incoming processes and the algorithm checks the queue for the next process j when the first process i comes out of the critical section.
- It ensures bounded waiting time.

lock

	P_0	P_1	P_2	. .	P_n
waiting[i]	F	F	F		F
key					

Mutual Exclusion:

- if (waiting[i]=T || key=F) then P_i enters to critical section.
- The value of key=F only if testAndSet instructions are executed.
- The value of waiting[i] can be false only if another process leaves its critical section.
- only one waiting[i] is false.

if (waiting[i] && key) → then
T T
if F && T — enter CS
if T && F — enter CS

Progress:

- A process that leaves its critical section either sets
  - lock = false
  - waiting[j] = false

Bounded waiting:

- A process that leaves its critical section, visits the array in a cycle and finds out any process with waiting[i] = true.
- it designate that process to enter its critical section.
- The selected process will also do so when leaving its critical section.

Semaphores

=> Semaphor is a Software based solution
=> Proposed by edsger dijkstra in 1965.
=> it is a technique to manage concurrent processes by using a simple integer value, which is known as Semaphor.
=> Semaphor is a variable which is non-negative and shared between threads.
=> Used to solve critical section problem and to achieve process synchronization in the multi-threading environment.
=> A Semaphor S ... apart from initialization, is accessed only through standard operations.
- Wait (P)
- Signal (V)

=> **Wait (P):**
- The wait operation decreases the value and blocks the process or thread if it becomes negative.
- P → from the Dutch word Proberen which means "to test".

Signal (V)
- V → from the Dutch word verhogen which means "to increment".

- Signal process/operation increases the value and potentially releases a waiting process or thread

Definition of wait (P):

```
P( Semaphore S)
{
    while( S<=0);
    S--;
}
```

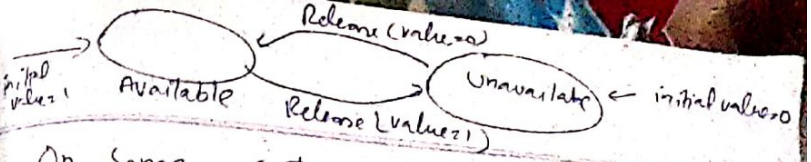
Definition of Signal (V):

```
V( Semaphore S)
{
    S++;
}
```

Types of Semaphores

(1) Binary Semaphore:

- The value of binary semaphore can range only between 0 and 1.

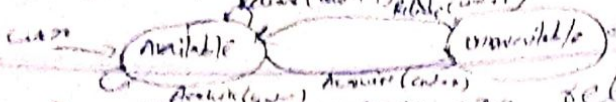


- On some systems, binary semaphores are known as muten locks, as they are locks that provide mutual enclusion.
- It is used for mutual enclusion, allowing only one process or thread to access the resource, it checks the semaphore value and waits if 0.
- once the resource become available, the semphor is set to 1, allowing the process or thread to proceed.
- when finished, the semaphor is reset to zero, allowing other process or threads to access the resource.

Counting semaphore:-

=> it manages accss to shared resources with a limited capacity.
=> it maintains the counter representing the number of available resource.
=> Processes or threads check the counter before accessing the resources, and if the resources are available, they decrement the counter and proceed.
=> if no resources may be available

It is low level synchronization mechanism.



⇒ when resources are released, the Counter is incremented, allowing waiting processes or threads to proceed.

Use of semaphor:

- Solve critical section problem for n processes.

```
while (1)
{
    wait (muten);
        CS;
    Signal (muten);
        RS;
}
```

Q. What is muten?

⇒ Muten lock is another name of binary semaphore.

- it can only have two possible values:
  - 0 and 1.
- Its value is set to 1 by default.

Applications of Semaphor:

- Solve critical section problem
- Order of execution of processes
- Resource management

Counting Semaphore Vs Binary Semaphore:

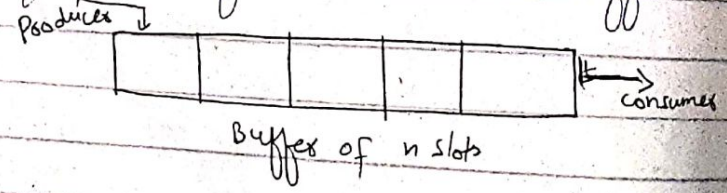
Counting semaphore	Binary semaphore
No mutual exclusion	mutual exclusion
Any integer value [0,n]	value only 0 & 1
More than one slot	only one slot
Provide a set of processes	it has a mutual exclusion mechanism

Pros	Cons
flexible management of resources.	it may lead to priority inversion
Machine independent and they should be run in machine-independent code of the microkernel.	signal & wait operations should be performed in correct order to avoid deadlock.
Allow more than one thread to access the cs.	At large scale its use is impractical, they lead to loss of modularity.
Mutual exclusion strictly	

Classic Problems of Synchronization

(i) Bounded-Buffer problem:-

⇒ It is also called producer Consumer problem.
⇒ It is one of the classic problem of the synchronization.
⇒ There are two processes running, namely producer and Consumer, which are operating on the buffer.
⇒ There is a buffer of n slots and each slot is capable of storing one unit of data.
⇒ Both producer and Consumer shared the same memory buffer of fixed size.
⇒ The producer tries to insert data into an empty slot of the buffer.
⇒ The Consumer tries to remove data from a filled slot in the buffer.



⇒ The problem is to allow producer and Consumers to access the buffer while ensuring the following:
(1) The producer must not insert data when the buffer is full.
(2) The Consumer must not remove data when the buffer is empty.
(3) The producer and Consumer should not insert and remove data simultaneously.
⇒ When condition (1) is dropped (the buffer can have infinite capacity) the problem is called unbounded-buffer problem.

Solution to the Bounded Buffer problem using Semaphore:

⇒ Three semaphore will make
(1) m (mutex), a binary semaphore which is used to acquire and release the lock.
(2) empty, a counting semaphore whose initial value is the number of slots in the buffer. Since, initially all slots are empty.
(3) full, A counting semaphore whose initial value is zero.

Producer

```
do
{
wait(empty); //wait until empty?? and then decrement empty.
wait(mutex); //acquire lock

/* add data to buffer
signal(mutex); //release lock
signal(full);
}while(TRUE)
```

Consumer

```
do
{
wait(full) // wait until full?? & then decrement "full".

wait(mutex) // acquire lock
/* remove data from buffer
signal(mutex) //release lock
wait(mutex); lock

signal(mutex); //release lock
signal(empty);
}while(TRUE);
```

(2)

Readers-Writers Problem:-

⇒ Readers writers problem is another example of a classic synchronization problem.

Problem Statement:-

⇒ There is a shared resource which should be accessed by multiple processes.

⇒ There are two types of processes in this context.

⇒ They are readers and writers.

Readers:

- Some processes extract information from the object are called readers.

Writers:

- Some processes change or insert information in the object are called writers.

⇒ **Bernstein state condition:**

- Any number of readers can read from the shared resources simultaneously.

⇒ But only one writer can write to shared resources.

=> When a [illegible] the resource, no other processes can access the resource.

=> A writer cannot write to the resource if there are non zero number of readers accessing the resource at that time.

Solution:

Reader

```
While (1)
{
// acquire lock
wait(mutex);
read_count++;
if(read_count==1)
    wait(w);
// release lock
signal(mutex)
/* Perform the reading operation
// acquire lock
wait(mutex);
read_count--;
if (read_count==0)
    signal(w);
// release lock
signal(mutex);
}
```

Writer

```
while(1)
{
wait(w)
/* Perform the write operation */
signal(w);
}
```

mutex and w are semaphore that are initialized to 1

=> The mutex semaphore ensure mutual exclusion and w handles the writing mechanism and is common to the reader and writer process code.

=> The variable read_count denotes the number of readers accessing the objects

=> As soon as read_count becomes 1, wait operation is used on w.

=> After the read operation is performed, read_count is decremented.

=> When read_count becomes zero, signal operation is used on w. So a writer can access the object now.

Shared Data:

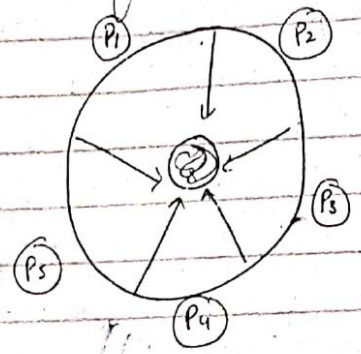
```
Typedef{ Semaphore mutex=1
         Semaphore w=1
         int rc=0;
}
```

(3) Dining philosphers problem:

Problem Statement:-

=> Consider there are five philsophers sitting around a circular dining table

=> The diping table have five chopsticks and a bowl of rice in the middle.



=> A philospher is either:
- eat
- Think

=> When a philosphes wants to eat he uses two chopsticks one from their left and one from their right.

=> When a philospher wants to think he keeps down both chopsticks at the original place.

Solution:

- Each philospher will pick left fork first then right fork. Each fork is represented as semaphore
- No two adjacent philospher pick two fork at the same time.

Shared data:

```
Semaphor chopstick[5];
```

Philospher i:

```
while (1)
{
  wait (chopstic[i]);
  wait (chopstic[(i+1)%5]);
  eat ();
  signal (chopstick[i]);
  signal (chopstick[(i+1)%5]);
  think ();
}
```

	left	right
P_0	0	1
P_1	1	2
P_2	2	3
P_3	3	4
P_4	0	4

=> chopstick[5] is an array of five Semaphores for each of the five chopsticks.

	0	1	2	3	4
chopstic[i]	1	1	1	1	1

=> When a philospher wants to eat the rice, he will wait for the chopstick at his left and picks up that chopstick

=> Then he waits for the right chopstick to be available, and then picks it too.
=> After eating, he puts both the chopsticks down.

Problem:-
=> if all five philosphers are hungry simultaneously, and each of them pickup one chopstick, then a "deadlock" situation occurs because they will be waiting for another process.

Several remedies to avoid deadlock

=> There are several possible ways to avoid deadlock:
(1) A philosphers must be allowed to pick up the chopsticks only if both left and right chopstids are available.
(2) Allow only four philosphers to sit - at the table. That way, if all the four philosphers pickup four chopsticks, there will be one chopstick left on the table. So, one philosphers can start eating.
(3) Use an asymmetric solution - an odd-numbered philosphers picks up her left chopstick and then her right and even-numbered philosphers picks up her right

(4)

Sleeping barber problem

Problem:-

- There is one barber, one barber chair and n waiting chairs.
- if there is no customer, barber sleeps in his own chair.
- When customer comes, that has to wake up the barber.
- When many customers come, they sit on waiting chairs if empty else leave (no chair empty).

Solution:

1) Semaphor customers - Counts waiting customers (excluding barber chair customers).
2) Semaphor barber - 0 if barber is idle. 1 if barber is active.
3) Semaphor mutex - Used for mutual exclusion.
4) int variable waiting - also counts no of waiting customers. (no way to read the current value of Semaphor customers).

chairs = n

Semaphore customers = 0 // no of customers waiting room

Semaphore barber = 0 // barber is idle

int waiting = 0 // no of waiting customers

Semaphore mutex = 1 // mutual exclusion

Customers

```
While (true)
{
    wait (mutex);
    if (waiting < chairs)
    {
        waiting = waiting + 1;
        signal (customers);
        signal (mutex);
        wait (barber);
        get_haircut();
    }
```

Barber

```
While (true)
{
    wait (customers);
    wait (mutex);
    waiting = waiting - 1;
    signal (barber);
    signal (mutex);
    cut_hair();
}
```

⇒ The solution ensures that the barber never cuts the hair of more than one customer at a time.

Monitor

- It is a high level abstraction that provides a convenient and effective mechanism for process synchronization.
- A monitor is a module that contains:
  - Shared data
  - Procedures that operate on the shared data

Syntax:

```
monitor monitor_name
{
    condition variables;
    Shared variable declarations
    Procedure P1 (...)
    {
        ...
    }
    Procedure P2 (...)
    {
        ...
    }
    :
    Procedure Pn (...)
    { ... }
    initialization code (...)
    { ... }
}
```

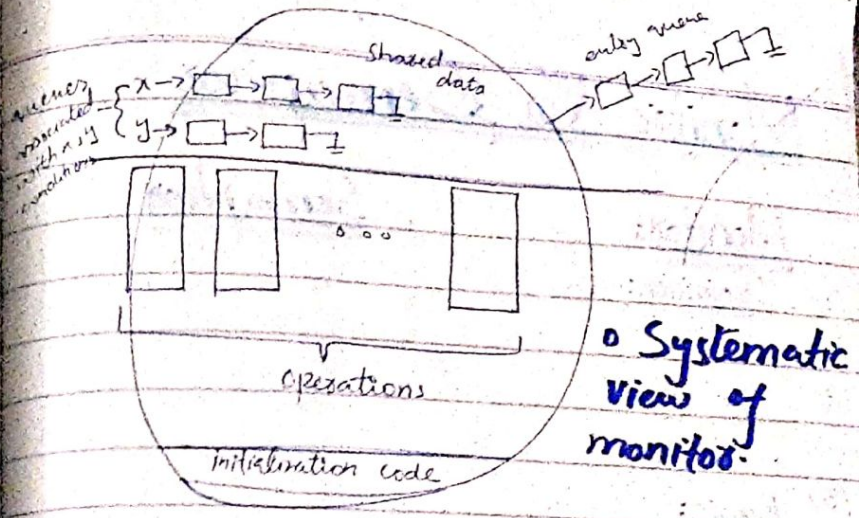
- A procedure defined within a monitor can access only those variables declared locally within the monitor and its formal parameters.
- Similarly, the local variables of a monitor can be accessed by only the local procedures.
- The monitor construct ensures that only one process at a time can be active within the monitor.

Condition variables:-

- A condition variable is just a data structure (or class) consisting of the following:
  - A Boolean value
  - A queue of delayed process
- The only operations that can be invoked on a condition variable are wait() and signal().

Condition x, y;

- The operation x.wait(); means that the process invoking this operation is suspended until another process invokes x.signal().
- The x.signal() operation resumes exactly one suspended process.



- **Systematic view of monitor.**

Advantages

- Mutual exclusion is automatic in monitors.
- Less difficult to implement than Semaphores.
- Overcome the timing errors that occur when semaphores are used.
- The implementation of a monitor can be changed without affecting the application.

Disadvantages:-

- It must be implemented into programming language.
- Compilers should generate code for them.
- Additional burden of knowing what OS features is available for controlling access to crucial sections in concurrent processes.

Monitor Vs Semaphore

Monitor:	Semaphore
(1) Definition:	
(2) Syntax:	
(3) Basic:	
○ Abstract data type	○ Integer variable
(4) Accen:	
○ When a process used shared resources, it calls the wait() method on S, and when it releases them, it uses the signal method on S.	○ When a process uses shared resources in the monitor, it has to accen them via procedures.
(5) Action:	
• The monitor type includes shared variables as well as a set of procedures that operate on them.	○ The semaphore value shows the number of shared resources available in the system.
(6) Condition variable:-	
○ No condition variables.	○ It has condition variables.

Past Papers Questions:

Q. Differentiate between progress and bounded-waiting with example. **(repeat)**

Q. Why peterson solution not updating bounded-wait?

Q. What is semaphores? **(Repeat)**

Q. What is process synchronization?

Q. What is critical section? **(Repeat)**.

Q. What is busy waiting?

Q. Define monitor? what does it consist of? **(Repeat)**

Q. What is the use of semaphore?

Q. What is race condition?

Q. What are different types of semaphore?

Q. What kinds of operations can be performed on semaphore?

Long.

Q. Write down Solution of producer-Consumer problem for bounded buffer using semaphore? **(Repeat)**

Q. Discuss the critical section problem. Solve the dinning philospher problem

Q. Explain the monitors?

Q. What is critical section problem. Explain three requirements of CS?

Q Write down Solution for Readers/Writers problem using semaphores with Readers having priority. (repeat)

Q. Write down the solution of dining philosphers problem using semaphore? (repeat)

Q. What is meant by race condition? How it is important during inter-proc Communication? Explain your answer in conte of critical sections.

Q. What is semaphore? How semapho works for synchronization of the processes? (repeat)

Q. How peterson's solution preserves 3 properties for critical section problem