Chapter 2

Interprocess Communication, Concurrency and Synchronization

LEARNING OBJECTIVES

- *Principles of concurrency*
- *Process interaction*
- *Mutual exclusion*
- *Semaphores*
- *Binary semaphore*
- *Mutual exclusion using semaphores*
- *Progress using semaphores*
- *Classical problems of synchronization*
- *Dining philosophers problem*
- *Monitors*
- *Message passing*
- *Indirect addressing*
- *Mutual exclusion using message passing*

Basic Concepts

Multiprogramming: It deals with the management of multiple processes within a uniprocessor system.

Multiprocessing: It deals with the management of multiple processes within a multiprocessor.

The fundamental operating system (OS) design is *concurrency*. Concurrency encompasses a host of design issues, including communication among processes, sharing of and competing for resources, synchronization of the activities of multiple processes and allocation of processor time to processes.

Principles of Concurrency

There are two examples for concurrent processing as follows:

1. In a single-processor multiprogramming system, processes are interleaved in time to yield the appearance of simultaneous execution.
2. In a multiprocessor system, it is possible not only to interleave the execution of multiple processes but also to overlap them.

There are two problems with these techniques:

1. Problem with sharing of global resources
2. Problem with allocation of resources optimally.
3. Problem with locating a programming error as results is not deterministic and reproducible.

Example:

```
void process( )
{
in = getchar( );
out = in;
putchar(out);
}
```

The procedure 'process' reads a character and prints it. Let us suppose that we have a uniprocessor system, with single user. Let the user running multiple applications and all applications use the procedure for reading and printing, that is, all the applications share common procedure for efficient and close interaction among them. But this sharing leads to problems. For example,

1. Let the process P_1 invokes 'process' and is interrupted immediately after 'getchar' returns its value and stores it in 'in'. Here the most recently entered character '*C*' is stored in variable 'in'.
2. Now, suppose the process P_2 is activated and it invokes 'process', which runs to conclusion, inputting and then displaying a single character, *D*, on the screen.
3. The process P_1 is resumed. By this time, the value '*C*' has been overwritten in 'in' and therefore lost. Instead 'in' contains '*D*', which is transferred to 'out' and displayed.

Here the problem is with sharing a global variable. To avoid these types of problems, we impose some rules like, only one process

at a time may enter 'process' and that once in 'process' the procedure must run to completion before it is available for another process.

This problem is also applicable to multiprocessor systems.

Race condition: A race condition occurs when multiple processes or threads read and write data items so that the final result depends on the order of execution of instructions in the multiple processes.

Example: Let P_1 and P_2 be two processes that share global variables a and b, with initial values $a = 0$, $b = 1$. At some point in its execution, P_1 executes $a = a + b$ and P_2 executes $b = a + b$.
If P_1 executes before P_2, then $a = 1$, $b = 2$.
If P_2 executes before P_1, then $b = 1$, $a = 1$.

OS Concerns for Concurrency

1. The OS must be able to keep track of various processes using PCBs.
2. The OS must allocate and deallocate various resources for each active process.
3. The OS must protect the data and physical resources of each process.
4. The functioning of a process and the output it produces must be independent of the speed at which its execution is carried out relative to the speed of other concurrent processes.

Process Interaction (IPC)

Process classification There are two types of processes as follows:

1. Independent/isolated
2. Cooperating

Independent Process: It cannot affect or be affected by the execution of another process.

Cooperating Process: It can affect or be affected by the execution of another process.

We can classify the ways in which processes interact on the basis of the degree to which they are aware of each other's existence. There are three types of process interaction as follows:

1. Process unaware of each other
2. Process indirectly aware of each other
3. Processes directly aware of each other

Competition Among Processes for Resources

1. This situation arises when processes unaware of each other.
2. There is no exchange of information between the competing processes.
3. But the execution of one process may affect the behaviour of competing processes.
4. With competing processes, there will be three control problems as follows:

Need for mutual exclusion

Example: Suppose two or more processes require access to a single non-sharable resource, such as a printer. Then that resource is referred as *critical resource* and the portion of the program that uses it is called *critical section* of the program. In the case of printer, only one process will have the control of printer while it prints an entire file.

Possibility of deadlock

Example: Two processes waiting for each other indefinitely for the release of resources.

Possibility of starvation

Example: One process is denied access to a particular resource which is required for the execution of that process.

Control of competition inevitably involves the OS, because it is the OS that allocates resources.

Cooperation Among Processes by Sharing

This situation arises when the processes are indirectly aware of each other. Processes may use and update the shared data without reference to other processes but know that other processes may have access to same data. So the control mechanism must ensure the integrity of the shared data. Problems with this type of sharing are

1. Mutual exclusion
2. Deadlock
3. Starvation
4. Data coherence

Data coherence

Suppose two items of data p and q are maintained in the relationship $p = q$, that is, any program that updates p and q values must maintain the relationship.

$$\begin{aligned} \text{Let } P_1 &: p = p + 1; \\ q &= q + 1; \\ P_2 &: p = p * 2; \\ q &= q * 2; \end{aligned}$$

Let initially the state is consistent, that is, $p = 2$, $q = 2$

Then the concurrent execution of P_1 and P_2 with mutual exclusion on p, q will be $p = p + 1$;

$$\begin{aligned} q &= q * 2; \\ q &= q + 1 \\ p &= p * 2; \end{aligned}$$

The final values of p and q will be $p = 6$, $q = 5$.

So the consistency is not maintained.

Cooperation Among Processes by Communication

This situation arises when processes are aware of each other directly. All the processes communicate with each other to synchronize or coordinate the various activities. Problems with this communication are as follows:

1. Deadlock
2. Starvation

Basic Definitions

Atomic operations: A sequence of one or more statements that appears to be indivisible, that is, no process will interrupt the operation.

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 A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.

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.

Critical Section

1. A section of code or set of operations, in which process may be changing shared variables, updating a common file or a table etc.
2. For the process that execute concurrently, it should ensure that execution of critical section should be made atomic. Atomic means that either an operation in the critical section should happen in its entirely or not at all.
3. Critical section of a process should not be executed concurrently with the critical section of another process.
4. To avoid Race Condition, we must have the following:

   'if one process is in critical section, other competing process must be excluded to enter their critical sections, that is, a process must enter the critical section in a mutually exclusive way'.

   This is called *problem of mutual exclusion*.

   Region of code that updates or uses shared data to provide a consistent view of objects need to make sure an update is not in progress when reading the data.
5. Need to provide mutual exclusion for a critical section.

Requirements for Critical Section Problem

Mutual exclusion: No two contending processes should be simultaneously executing inside their critical section.

Bounded waiting: No process should have to wait forever to enter its critical section.

Progress: If no process is executing in its critical section and there exists some processes that wish to enter their critical sections, then only those processes that are not executing in the critical section can participate in the decision of which will enter its critical section next and this selection cannot be postponed indefinitely.

No Assumption: No assumption should be made about relative speed and properties of contenting processes.

Mutual Exclusion

1. Only one process at a time can be updating shared objects.
2. Successful use of concurrency among processes requires the ability to define critical sections and enforce mutual exclusion.
3. Mutual exclusion in the use of a shared resource is provided by making its access mutually exclusive among the processes that share the resources.

Any facility that provides mutual exclusion should meet the following requirements:

1. No assumption regarding the relative speed of the process.
2. A process is in its critical section for a finite time only.
3. Only one process allowed in the critical section.
4. Process requesting access to critical section should not wait indefinitely.
5. A process waiting to enter critical section cannot be blocking a process in critical section or any other process.

Mutual exclusion can be satisfied in one of the following ways:

1. ***Software approach:*** This approach leaves the mutual exclusion responsibility with the process that wish to execute concurrently. This approach is prone to high processing overhead and bugs.
2. ***Hardware support:*** Use special-purpose machine instructions. This approach involves less overhead.
3. ***Provide some level of support within the OS or a programming language:*** Such techniques are as follows:
   - Semaphores
   - Monitors
   - Message passing

Hardware support for mutual exclusion: Use one of the following techniques:

1. Interrupt disabling
2. Special machine instructions
   - Compare and swap instructions
   - Exchange instructions

Interrupt Disabling

To provide mutual exclusion, it is sufficient to prevent a process from being interrupted. A process can enforce mutual exclusion in the following way:

```
while(true)
{
/* disable interrupts */;
/* critical section */;
/* enable interrupts */;
/* remainder */;
}
```

Here, the critical section is not interrupted, so mutual exclusion is guaranteed.

Problems with Interrupt Disabling

The efficiency of execution could be noticeably degraded, because the processor is limited in its ability to interleave processes. Interrupt disabling does not work on in a multiprocessor architecture.

Special Machine Instructions

Two of the most commonly used special machine instructions are as follows:

1. Compare and swap
2. Exchange instruction

Compare and swap instructions It is defined as below:

```
int compare_and_swap(int *word, int
testval, int newval)
{
int oldval;
oldval = *word;
if(oldval = = testval)
*word = newval;
return oldval;
}
```

- 'Access to a memory location excludes any other access to that same location.' On the basis of this principle, special machine instructions provide mutual exclusion.
- The above compare and swap instructions will check a memory location (*word) against a test value. If current value is test value, it is replaced with new value. Always the old value is returned.
- The below code provides mutual exclusion using compare and swap instructions:

```
/* Mutual Exclusion */
const int n = /* number of processes */
int S;
void P(int i)
{
while (true)
{
while (compare_and_swap(S, 0, 1)==1)
/* do nothing */;
/* critical section */;
S = 0;
/* remainder */;
}
}
void main( )
{
S = 0;
begin (P(1), P(2) .... P(n));
}
```

Here, a shared variable '*S*' is initialized to '0'. The only process that may enter its critical section is one that finds '*S*' equal to 0.

All other processes at enter their critical sections go into a busy waiting mode.

Busy waiting or *spin waiting* is a technique in which a process can do nothing but continue to execute an instruction or set of instructions that tests the appropriate variable to gain entrance.

When a process leaves, its critical section sets '*S*' to 0. Then the one of the waiting process will get access to enter its critical section.

Exchange instructions The exchange instructions can be defined as follows:

```
void exchange (int reg, int mem)
{
int temp;
temp = mem;
mem = reg;
reg = temp;
}
```

Mutual Exclusion Using Exchange Instructions

```
/* Mutual Exclusion */;
int const n = /* number of processes */;
int S;
void P(int i)
{
int ki = 1;
```

```
while (true)
{
do
exchange(ki, S);
while(ki! = 0);
/* critical section */;
S = 0;
/* remainder */;
}
}
void main( )
{
S = 0;
begin (P(1), P(2), ... P(n));
}
```

A shared variable '*S*' is initialized to '0'. Each process uses a local variable *ki* that is initialized to 1. The only process that may enter its critical section is one that finds '*S*' equal to 0. It excludes all other processes from critical section by setting '*S*' to 1. When a process leaves its critical section, it resets '*S*' to 0 allowing another process to its critical section.

Advantages of using machine instruction approach

1. Applicable to any number of processes on either a single processor or multiple processors, sharing the main memory.
2. Simple and easy to verify.
3. Used to support multiple critical sections.

Disadvantages

1. Busy waiting
2. Starvation is possible
3. Deadlock is possible

Other Mechanisms for Mutual Exclusion

Let us discuss OS and programming language mechanisms that are used to provide concurrency.

Semaphores

Semaphore is an integer value used for signalling among processes.

There are two types of semaphores as follows:

1. Binary semaphore
2. Counting (or) general semaphore

Counting or general semaphore

Three operations may be performed on a semaphore all of which are atomic:

- Initialize
- Decrement
- Increment

The working of a counting semaphore with its operations is defined as below:

1. The semaphore may be initialized to a non-negative integer value.
2. The semwait operation decrements the semaphore value. If the value becomes negative, then the process executing the semwait is blocked, otherwise the process continues execution.
3. The semsignal operation increments the semaphore value. If the resulting value is less than or equal to zero, then one of the processes blocked by a semwait operation, if any, is unblocked.

Example: Let the semaphore value $S = 3$.
If the semaphore value is positive, then that value gives the number of processes that can issue a wait and immediately continue to execute.

Let five processes P_1, P_2, P_3, P_4, P_5 are going to execute a critical section code based on the semaphore value $S = 3$.

$S = \boxed{3} \leftarrow$ Initially

$\downarrow$ P_1 sem wait

$S = \boxed{2} \geq 0$, P_1 executes

$\downarrow$ P_2 sem wait

$S = \boxed{1} \geq 0$, P_2 executes

$\downarrow$ P_3 sem wait

$S = \boxed{0} \geq 0$, P_3 executes

$\downarrow$ P_4 sem wait

$S = \boxed{-1} \not\geq 0$, P_4 Blocked

$\downarrow$ P_5 sem wait

$S = \boxed{-2} \not\geq 0$, P_5 Blocked

Initially, the '*S*' value 3 means that at a time three processes can issue a 'wait' signal and continue execution.

Whenever S becomes 0, the next process which executes 'wait' operation will be blocked.

Here P_4 is blocked as it operates on the semaphore when $S = 0$.

If the semaphore value becomes negative, it specifies the number of processes waiting to be unblocked.

$S = -2$ means two processes are waiting to be unblocked.

Definition of semwait and semsignal operations

```
struct semaphore
{
int semvalue;
Queuetype Queue;
};
void semwait(semaphore S)
```

```
{
S.semvalue--;
if (S.semvalue < 0)
{
/* place the process in S. Queue */;
/* Block this process */;
}
}
void semsignal(semaphore S)
{
S.semvalue++;
if (S.semvalue < = 0)
{
/* remove a process from S. Queue */;
/* Place the process in ready queue */;
}
}
```

Advantages

1. Because the waiting processes will be permitted to enter their Critical Section in a FCFS order, so the requirement of bounded waiting is met.
2. CPU cycles are saved here as waiting process does not perform any busy waiting.

Disadvantages

1. Complex to implement, since it involves implementation of FCFS.
2. Context switching is more, so more overheads are involved.

Binary semaphore It is a semaphore that takes on only the values 0 and 1.

The operations performed on a binary semaphore are as follows:

1. A binary semaphore may be initialized to 0 or 1.
2. The semwaitB operation checks the semaphore value. If the value is 0, then the process executing the semwaitB is blocked. If the value is 1, then the value is changed to 0 and the process continues execution.
3. The semsignalB operation checks to see if any processes are blocked on this semaphore. If so, then a process blocked by a semwaitB operation is unblocked. If no processes are blocked, then the value of the semaphore is set to 1.

Definition of semwaitB and semsignalB

```
struct binary-semaphore
{
enum {zero, one} value;
Queue-type Queue;
};
void semwaitB(binary-semaphore S)
{
if (S.value = = one)
S.value = zero;
else
{
/* Place this process in S.Queue */;
/* Block this process */;
}
}
void semsignalB(binary-semaphore S)
{
if(S.Queue is empty)
S.value = one;
else
{
/* remove a process from S.Queue */;
/* Place process in ready list */;
}
}
```

Advantages

1. The implementation of binary semaphore is extremely simple.

Disadvantages

1. It does not meet the requirement of Bounded waiting.
2. A process, waiting to enter its Critical Section, will perform Busy waiting, thus wasting CPU cycles.

Notes:

1. Binary semaphores have the same expressive power as general semaphores.
2. *MUTEX*: It is similar to binary semaphore. The key difference between the two is that the process that locks the mutex must be the one to unlock it.
3. Both counting semaphores and binary semaphores use a queue to hold processes waiting on the semaphore. The order in which the processes removed from a Queue is FIFO, that is, the process that has been blocked the longest is released from Queue first.
4. A semaphore whose definition includes the order of removal of Blocked processes is referred as a strong semaphore otherwise it is a weak semaphore.
5. Strong semaphores guarantee freedom from starvation.

Mutual Exclusion Using Semaphores

```
const int n = /* number of processes */;
Semaphore S = 1;
void P(int i)
{
while (true)
{
Semwait(S);
/* critical section */;
Semsignal(S);
```

```
/* remainder */;
}
}
void main( )
{
begin (P(1), P(2),... P(n));
}
```

If no process is executing in Critical Section, then the semaphore values is 1. The first process that is executing 'wait' operation will decrement value to 0 and enter its critical section. The process which execute 'wait' operation while a cooperating process is executing in its critical section, will find the semaphore value to 0 and keep looping in the 'while-loop' of 'wait' operation. Spinning of a waiting process in the while-loop, the binary semaphores are also known as spin locks. When the process executing in the CS makes an exit (from CS), it will execute the 'signal' operation and increment the semaphore value to 1.

At a time, only one of the cooperating processes can enter critical section with the condition that the wait operation is executed automatically. Mutual Exclusion is satisfied.

Example: Consider the following figure (Figure 1) which shows the possible sequence of three processes using mutual exclusion with a semaphore, *S*. Processes *P*, *Q*, *R* accesses a shared resource protected by the semaphore *S*.

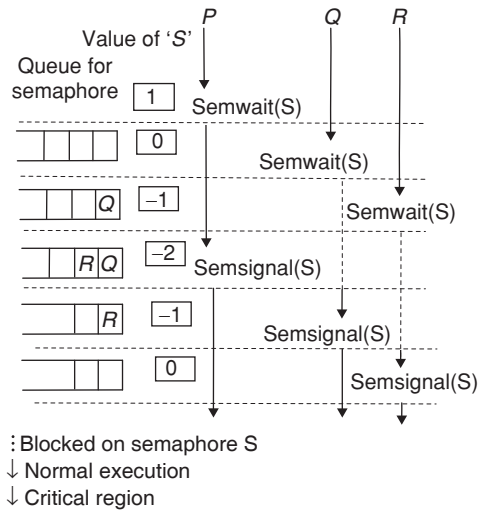


Figure 1 Mutual exclusion using semaphore.

Progress Using Semaphores

When no process is executing in the CS, the semaphore value will be 1. Then, one of the waiting process looping in the while loop of wait-operation will find the semaphore value of 1, exit from the while-loop, decrement the semaphore value to 0 and enter CS. Thus, if no process is executing in critical section and some are waiting to enter, then one of the waiting processes will enter its critical section immediately.

Bounded waiting using semaphores

Actually, one of the waiting processes will get entry into its CS when an operating process executing in its critical section exits. As this selection process is arbitrary, so a process waiting to enter its CS is likely to face starvation. So, the requirement of bounded waiting is not met.

Classical Problems of Synchronization

We will discuss three problems of synchronization:

1. Bounded buffer problem
2. Readers/writers problem
3. Dining philosophers problem

Producer–Consumer Problem

1. Producer inserts item in the buffer
2. Updates Insertion pointer
3. Consumer consumes items in the buffer
4. Updates removal pointer
5. Both update information about how full, how empty the buffer.
6. Prevents buffer overflow, prevents buffer underflow, proper synchronization.

Producer	Consumer
repeat	repeat
produce item *v*;	while (in < = out);
b[in] = *v*;	*w* = *b* [out];
in = in + 1;	out = out + 1;
forever;	consume *w*;
	forever;

Table 1 *Producer consumer problem solution using semaphores*

Producer	Consumer
repeat	repeat
produce item *v*;	while (in < = out);
SemwaitB(*S*);	SemsignalB(*S*);
b[in] = *v*;	*w* = *b*[out];
in = in + 1;	out = out + 1;
SemsignalB(S);	SemsignalB(S);
forever;	consume *w*;
	forever;

If producer is slow or late, then consumer will busy at the while statement.

Table 2 *Improved solution*

Producer	Consumer
repeat	repeat
produce item *v*;	Semwait(n);
Semwait(S);	Semwait(S);
b[in] = *v*;	*w* = *b*[out];
in = in + 1;	out = out + 1;
Semsignal(S);	Semsignal(S);
Semsignal(n);	consume *w*;
forever;	forever;

The initial value of n and S are $n = 0$, $S = 1$. (n is the number of items in the buffer).

Table 3 *Producer consumer bounded buffer problem*

Producer	Consumer
repeat	repeat
produce item v;	while(in == out)
while((in + 1) % n = = out)	no operation;
no operation;	$w = b$[out];
b[in] = v;	out = (out + 1) % n;
in = (in + 1) % n;	consume w;
forever;	forever;

The buffer size is enforced using another counting semaphore.

Table 4 *Producer consumer bounded buffer problem solution*

Producer	Consumers
Repeat	repeat
Produce item v;	Semwait(e);
Semwait(e);	Semwait(S);
Semwait(S);	$w = b$[out];
b[in] = v;	out = (out + 1) % n;
in = (in + 1) % n	Semsignal(S);
Semsignal(S);	Semsignal(e);
Semsignal(e);	consume w;
forever;	forever;

The initial value of buffer size, e is the size of the bounded buffer.

Observations on semaphores:

1. Semaphores are easy to use.
2. wait() and signal() are to be implemented as atomic operations.

Problems:

1. signal() and wait() may be exchanged by the programmer, this may result in deadlock or violation of mutual exclusion.

Readers/Writers Problem

1. A reader reads data.
2. A writer writes data.
3. Data is shared among a number of processes.
4. Multiple readers may read the data simultaneously, that is, concurrently.
5. Only one writer can write the data any time, that is, no reader should be present.
6. A reader and writer cannot access data simultaneously.
7. Locking table: Whether any two can be in the critical section simultaneously is shown in the table.

	Reader	Writer
Reader	OK	NO
Writer	NO	NO

Solution: Readers have priority; if a reader is in CS, any number of readers could enter irrespective of any writer waiting to enter critical section

Writer	Reader
`while(true)`	`while(true)`
`{`	`{`
`Semwait(S);`	`Semwait(x);`
`writeunit();`	`Num = Num + 1;`
`Semsignal(S);`	`if (Num = = 1)`
`}`	`Semwait(S);`
	`Semsignal(x);`
	`Readunit();`
	`Semwait(x);`
	`Num = Num - 1;`
	`if(Num == 0)`
	`Semsignal(S);`
	`Semsignal(x);`
	`}`

Semaphore 'S' is used to enforce mutual exclusion.

Semaphore 'x' is used to assure that 'Num' is updated properly.

Solution: If a writer wants critical section as soon as the critical section is available, writer enters it.

Dining Philosophers Problem

N philosophers are sitting around a dining table. There are N plates placed on the table such that each plate is in front of a philosopher and N forks placed between the plates. There is a bowl of Noodles placed at the centre of the table. Whenever a philosopher feels hungry, he tries to pick two forks which are shared with his nearest neighbour. If any of his neighbours happens to be eating at the time, the philosopher has to wait. Whenever a hungry philosopher gets two forks, he pours noodles into his plate. After he finishes, he places the chopsticks back onto the table and starts thinking. Now forks are available for neighbours.

Solution:

```
# define N 5 /* Number of philosophers*/
voidphilosopher(int i) /* philosopher number,
from 0 to 4*/
{
while (true)
{
think( ); /* philosopher is thinking*/
take_fork(i) ; /*take left fork*/
take_fork ((i+ 1)% N); /* take right fork; %
is modulo operator*/
eat( );
put _ fork ( ); /* put left back on the table*/
put _ fork ((i + 1) % N); /* put right fork
back on the table */
```

Notes:

1. This solution leads to deadlock.
2. Everyone picks the left fork and indefinitely wait for right fork causing starvation.

Monitors

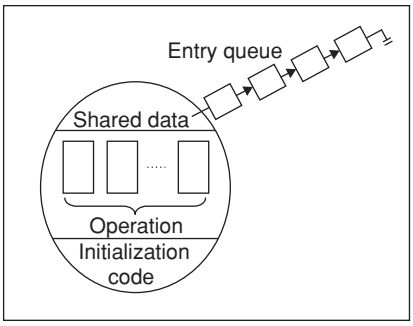
Monitor is a programming language construct that encapsulates variables, access procedures and initialization code within an abstract data type. The monitor's variable may only be accessed via its access procedures and only one process may be actively accessing the monitor at any one time. The access procedures are critical sections. A monitor may have a queue of processes that are waiting to access it.

1. If the data in a monitor represent some resource, then the monitor provides a mutual exclusion facility for accessing the resource.
2. A monitor supports synchronization by the use of condition variables that are contained within the monitor and accessible only within the monitor.
3. Operations on conditional variables:
   - cwait(c): Suspend execution of calling process on condition *c*. The monitor is now available for use by another process.
   - csignal(c): Resume execution of some process blocked after a wait on the same condition. If there are several such processes, choose one of them; if there is no such process, do nothing.

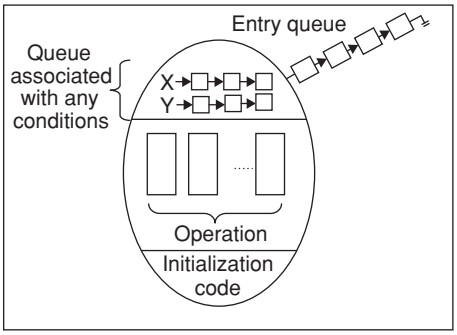
Monitor syntax is as follows:
Monitor monitor - name

```
{
shared variable declarations;
Procedure body P1 (...) {  }
Procedure body P2 (...) {  }
Procedure body Pn (...) {  }
initialization code {  }
}
```

Schematic view of a monitor:



Schematic View of a monitor with condition variables:



Message Passing

When processes interact with one another, two fundamental requirements must be satisfied:

1. Synchronization
2. Communication

One approach to provide both of these is *message passing*.

The primitive functions in message passing are send (destination, message) receive (source, message)

Design Characteristics of Message Systems for IPC and Synchronization

Synchronization There must be some synchronization existing between two processes to communicate with each other.

- ***Send:*** When a 'send' primitive is executed in a process, then the sender may
  - blocked or
  - non-blocked
  - until the message is received.
- ***Receive:*** When a process issues a 'Receive' primitive there are two possibilities:
  1. If a message has previously been sent, the message is received and execution continues.
  2. It there is no waiting message then either
     (i) The process is blocked until a message arrives or
     (ii) The process continues to execute, abandoning the attempt to receive.

Thus, both the sender and receiver may be in one of

1. Blocking send, blocking receive: Allows tight synchronization.
2. Non-blocking send, blocking receive:
   - Useful synchronization
   - Possibility of generating repeated messages
3. Non-blocking send, non-blocking receive: No need to wait

Addressing Two types of addressing methods:

1. Direct addressing
2. Indirect addressing

Direct Addressing

The send primitive includes a specific identifier of the destination process. The 'receive' primitive can be handled in one of two ways:

Explicit

The process must know ahead of time from which process a message is expected. Useful for cooperating concurrent processes.

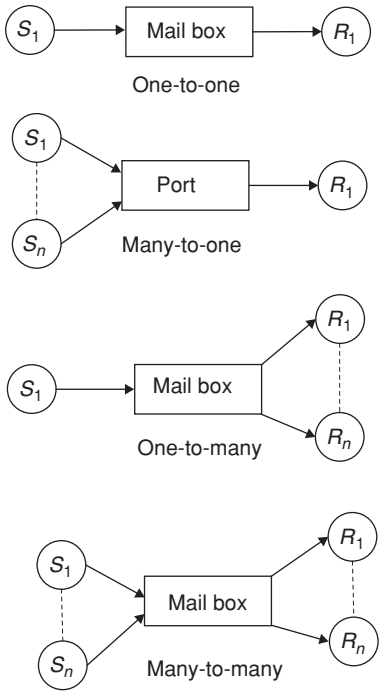
Implicit

The 'source' parameter of 'receive' primitive possesses a value returned when the receive operation has been performed. Example, Printer server.

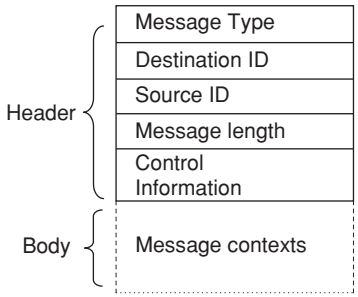
Indirect Addressing

Messages are not sent directly from sender to receiver but rather are sent to a shared data structure consisting of queues that can temporarily hold messages. These queues are referred to as mailboxes The relationship between the sender and receiver is

- one-to-one
- one-to-many
- many-to-one
- many-to-many



Message format The general message format will be



Queuing discipline The queuing discipline may be

- FIFO
- Priority

Mutual Exclusion using message passing:

```
const int n = /* number of processes */;
void P(int i)
{
message msg;
while (true)
{
receive (box, msg);
/* critical section */;
send (box, msg);
/* remainder */;
}
}
void main( )
{
create mailbox(box);
send(box, null);
begin (P(1), P(2) ... P(n));
}
```

1. Here a set of concurrent processes that share a mailbox 'box', which can be used by all processes to send and receive.
2. The mailbox is initialized to contain a single message with null content. A process wishing to enter its critical section first attempts to receive a message.
3. If the mail box is empty, then the process is blocked.
4. Once a process acquired the message, it performs its critical section and then places the message back into the mailbox.
5. Hence, the message functions as a token that is passed from process to process.

Exercises

Practice Problems I

Directions for questions 1 to 17: Select the correct alternative from the given choices.

1. The value of a counting semaphore is 7. Then 15 wait operations and 10 signal operations were completed on this semaphore. The resulting value of semaphore is

(A) 5 (B) 7
(C) 2 (D) 0

2. At a particular time of computation, the value of counting semaphore is 7. Then 20 wait operations and 'x' signal operations were completed on this semaphore. If the final value of the semaphore is 5, what is x?

(A) 18 (B) 13
(C) 5 (D) 0

3. A process using a semaphore has a start value of 1 for its semaphore. Since the start of execution of the program, 12 signal operations were completed. How many wait operations have been completed so far if the current value of semaphore is 6?

(A) 1 (B) 5
(C) 7 (D) 11

4. It is found that a program has multiple critical sections. Choose correct statements from below:

(i) Multiple semaphores are needed for handling the situation.

(ii) A single semaphore that uncompresses all the critical section is sufficient and is also more efficient.

(iii) To get better control of the code, monitors need to be implemented.

(A) (i) and (ii) (B) (ii) and (iii)
(C) (i) and (iii) (D) (i), (ii), (iii)

5. Consider the below psuedocode:

```
semaphore S = 1;
semaphore E = 1;
if(thread_count++ <100)
spawnnewthread( );
wait(E);

// critical section – begin
---------------
---------------
// critical section – end
signal(S);
```

Assume that above pseudocode gets called a hundred times, what is the count of semaphore *E*?

(A) 0 (B) 1
(C) –99 (D) –100

6. Consider the below code for a process *i*:

```
-------------
-------------
flag[i] = true;
if(turn == i and flag [i] == true)
/* critical section begin */
counter++;
/* critical section end */
turn = x;
-------------
-------------
```

If the value of a counter started, what would be the value of 'counter' count at the end of the program:

(A) Semaphore count
(B) Thread count
(C) Concurrency count
(D) Deadlock process count

7. Consider the below pseudocode:

```
function waitB(s)
{
if(s.value ==1)
s.value = 0;
else
place the process in the Queue;
}
function signalB(s)
{
s. value = 1;
}
```

What does the code most likely behave as

(A) general semaphore (B) weak semaphore
(C) binary semaphore (D) mutex

8. A shared variable *x*, initialized to 0 is operated on by four concurrent processes *P*, *Q*, *R*, *S* as follows:

```
P(x)                    Q(x)
{                       {
wait( );                wait( );
read(x);                read(x);
increment x by 1;       increment x by 1;
store(x);               store(x);
signal();               signal( );
}                       }
R(x)                    S(x)
{                       {
wait( );                wait( );
read (x);               read(x);
decrement x by 2;       decrement x by 2;
store (x);              store(x);
signal( );              signal( );
}                       }
```

A counting semaphore '*N*' is used by the processes whose value is initialized to 2. What is the maximum possible value of '*x*' after all processes complete execution?

(A) –2 (B) –1
(C) 1 (D) 2

9. Consider the following code:

```
Program concurrency;
Var x: Integer (: = 0);
    y: Integer (: = 0);
Procedure threadA( ) ;
begin
x = 1; /*S1*/
y = y + x; /*S2*/
end;
Procedure threadB( ) ;
begin
y = 4;      /*S3*/
x = x + 5; /*S4*/
end;
begin /*mainprogram*/
parbegin
threadA( );
threadB( );
parend;
   end.
```

Suppose a process has two concurrent threads: one thread executes statements S_1 and S_2, and the other thread executes statements S_3 and S_4. What are the maximum possible values of x and y when the code finishes execution? (All the statements S_1, S_2, S_3 and S_4 are atomic).

(A) $x = 6, y = 4$ (B) $x = 6, y = 5$
(C) $x = 1, y = 5$ (D) $x = 6, y = 10$

10. Consider the following program:

```
boolean lock[2];
int turn;
void P(int id)
{
while(true)
{
lock[id] = true;
while (turn! = id)
{
while (lock [1 – id])
/*do nothing*/
turn = id;
}
/*critical section*/
lock[id] = false;
/*reamainder*/
  }
}
void main( )
{
lock[0] = false;
lock[1] = false;
turn = 0;
parbegin (P(0), P(1));
}
```

Which of the following statements is correct for two processes executing this code?

(A) Given program provides mutual exclusion.
(B) Given program does not provide mutual exclusion.
(C) Given program provides mutual exclusion and also solves starvation problem.
(D) Given program provides mutual exclusion but does not prevent from starvation.

11. Consider two process P_0 and P_1 which share the following variables:

```
boolean flag [2]; /*initially false*/
int turn;
```

These two processes, $Pi(i = 0 \text{ or } 1)$, Pj $(j = 1 \text{ or } 0)$ execute the following code:

```
do
{
flag[i] = TRUE;
while(flag [j])
{
if (turn = = j)
{
flag[i] = false;
while (turn = = j);
flag [i] = TRUE;
}
}
// critical section
turn = j;
flag[i] = FALSE;
// remainder.
}
while(TRUE);
```

The code satisfies

(i) Mutual exclusion
(ii) Progress
(iii) Bounded waiting

(A) (i), (ii) only (B) (ii), (iii) only
(C) (i), (iii) only (D) (i), (ii), (iii)

12. Which of the following statement(s) is false?

(i) Spinlocks are not appropriate for single-processor systems.
(ii) Mailboxes may be used for synchronization.
(iii) Message passing and semaphores do not have equivalent functionality.

(A) (i) only (B) (iii) only
(C) (i), (iii) (D) (i), (ii), (iii)

13. Consider the following code:

```
signal (mutex);
.....
Critical section
.....
wait (mutex);
```

Here 'mutex' is a semaphore variable, which is initialized to 1. Then

(A) Mutual exclusion is provided
(B) Mutual exclusion violated, if several processes are simultaneously active in their critical section.
(C) Deadlock will occur
(D) Starvation is possible

14. Which of the following sequence of 'wait' and 'signal' operations leads to deadlock?
(Here 'mutex' is a semaphore variable initialized to 1.)

(A)
```
wait (mutex);
.....
Critical section
.....
Signal (mutex);
```

(B)
```
wait (mutex);
……..
Critical section
……..
Wait (mutex);
```
(C)
```
Signal (mutex);
……..
Critical section
……..
Wait (mutex);
```
(D)
```
signal (mutex);
……..
Critical section
……..
Signal (mutex);
```

15. Which of the following situation arises if a process omits the wait(S) or the signal(S) on a semaphore variable '*S*' (Initially $S = 1$).
(i) Mutual exclusion violated
(ii) Deadlock will occur
(A) (i) only (B) (ii) only
(C) Both (i) and (ii) (D) Neither (i) nor (ii)

16. consider the following shared data and code:
```
data:
int turn;
Boolean flag[2];
Code:
do
{
flag [i] = TRUE;
turn = j;
while (flag [j] && turn = = j);
//critical section
flag [i] = FALSE;
//remainder
}
while (TRUE);
```
Let two processes *Pi* ($i = 0$ or 1) and *Pj* ($j = 1$ or 0) use the shared data and executes the code. Then the code provides
(A) a solution to critical section problem
(B) mutual exclusion but not progress.
(C) progress but not mutual exclusion
(D) both mutual exclusion, progress but no bounded waiting.

17. Consider the following code that shows the structure of a process in an algorithm to solve the critical section problem for two processes.
var flag[2] of Boolean; /* initialized to false */
repeat
flag[*i*] = true;
while flag[*j*] do no – op;
//critical section
flag[*i*] = false;
// remainder
until false
Then which of the following statements is true?
(A) The algorithm satisfies all the requirements of critical section problem.
(B) The algorithm satisfies only mutual exclusion and progress.
(C) The algorithm only satisfies progress requirement.
(D) The algorithm does not satisfy critical section problem requirements.

Practice Problems 2

Directions for questions 1 to 15: Select the correct alternative from the given choices.

1. The result of a computation depends upon the speed of the processes involved, is said to be:
(A) Cycle stealing (B) Race condition
(C) A time lock (D) A deadlock

2. A relation between processes such that each has some part which must not be executed, while the critical section of another is being executed is known as
(A) Mutual exclusion (B) Semaphore
(C) Multi-tasking D) Mutli-programming

3. Producer–consumer problem can be solved using
(A) Semaphores (B) Event counters
(C) Monitors (D) All of the above

4. To avoid the race condition, the number of processes allowed in critical section is
(A) 0 (B) 1
(C) 2 (D) 3

5. Mutual exclusion problem occurs between
(A) Two disjoint processes that unaware of each other
(B) Processes that share resources
(C) Processes directly aware of each other.
(D) Both (A) and (B)

6. Semaphores are used to solve the problem of
(A) Race condition (B) Multitasking
(C) Mutual exclusion (D) Both (A) and (C)

7. At a particular time, the value of a counting semaphore is 10. It will become 7 after
(A) 3 signal operations
(B) 3 wait operations
(C) 5 signal operations and 2 wait operations
(D) None of the above

8. Critical region is
(A) A part of the OS which is not allowed to be accessed by any process
(B) A set of instructions that accesses common shared resource, which exclude one another in time

(C) The portion of main memory, which can be accessed only by one process at a time
(D) Both (A) and (C)

9. Concurrent processes are:
(A) Processes that don't overlap in time
(B) Processes that overlap in time
(C) Processes that are executed
(D) Processes that are executed by a processor at the same time

10. Semaphore operations are atomic because they are implemented within the_____.
(A) Kernel (B) Shell
(C) User process (D) Normal process space

11. The programming language construct that provides equivalent functionality of a semaphore and better control is
(A) Signal (B) Monitor
(C) Mutex (D) Critical section.

12. What is the ideal way of emptying the queue of a strong semaphore?
(A) Random (B) LIFO
(C) FIFO (D) binary

13. What are the disadvantages of machine instruction approach?
(i) While a process is waiting for entering a critical section, process still consumes resources.
(ii) There could be starvation
(iii) There could be deadlocks
(A) (i), (ii) only (B) (ii), (iii) only
(C) (iii), (i) only (D) (i), (ii), (iii)

14. Select from below the advantages of Machine Instruction approach?
(i) Applicable to any number of processes either uniprocessor or multi-processor system
(ii) Simple and easy to verify
(iii) Supports multiple critical sections
(A) (i), (ii) only (B) (ii), (iii) only
(C) (iii), (i) only (D) (i), (ii), (iii)

15. Which of the below are requirements for mutual exclusion?
(i) Only one process is allowed into critical section.
(ii) A process remains inside its critical section for finite time only.
(iii) It must be possible for a process accessing critical section to be delayed indefinitely.
(iv) A process halting in critical section must do so without interfering with other processes.
(A) (i), (ii), (iii) (B) (ii), (i), (iv)
(C) (i), (ii), (iv) (D) (i), (ii), (iii), (iv)

Previous Years' Questions

1. Consider these two functions and two statements S_1 and S_2 about them: **[2006]**

```
int work1 (int
    *a,int i, int j)
{
  int x=a[i+2];
  a[j]=x+1;
  return a[i+2]-3;
}
```

```
int work2 (int
    *a,int i, int j)
{
  int t1=i+2;
  int t2=a[t1];
  a[j]=t2+1;
  return t2 - 3;
}
```

S1: The transformation from work1 to work2 is valid, that is,for any program state and input arguments, work 2 will compute the same output and have the same effect on program state as work 1
S2: All the transformations applied to work 1 to get work 2 will always improve the performance (i.e., reduce CPU time) of work 2 compared to work 1
(A) S_1 is false and S_2 is false
(B) S_1 is false and S_2 is true
(C) S_1 is true and S_2 is false
(D) S_1 is true and S_2 is true

2. The atomic fetch-and-set x, y instructions unconditionally sets the memory location x to 1 and fetches the old value of the of x in y without allowing any intervening access to the memory location x. Consider the following implementation of P and V functions on binary semaphore S.

```
void P (binary-semaphore *s) {
    unsigned y;
    unsigned *x = & (s → value);
    do {
        fetch-and-set x, y ;
        } while (y) ;
    }
void V (binary-semaphore *s) {
s → value = 0;
}
Which one of the following is true?
```

[2006]
(A) The implementation may not work if context switching is disabled in P
(B) Instead of using fetch-and-set, a pair of normal load/store can be used
(C) The implementation of V is wrong
(D) The code does not implement a binary semaphore

3. The P and V operations on counting semaphores, where s is a counting semaphore, are defined as follows:

$P(s)$: $s = s - 1$;
if $s < 0$ then wait;
$V(s)$: $s = s + 1$;
if $s <= 0$ then wake up a process waiting on s;

Assume that P_b and V_b, the wait and signal operations on binary semaphores, are provided. Two binary semaphores x_b and y_b are used to implement the semaphore operations $P(s)$ and $V(s)$ as follows:

```
P(s) :          P_b (x_b) ;
    s = s - 1;
    if (s < 0) {
       V_b (x_b) ;
       P_b (y_b) ;
    }
    else V_b (x_b) ;
```

```
V(s) :          P_b (x_b) ;
    s = s + 1;
    if (s<=0) V_b (y_b) ;
       V_b (x_b) ;
```

The initial values of x_b and y_b are respectively **[2008]**

(A) 0 and 0 (B) 0 and 1
(C) 1 and 0 (D) 1 and 1

4. Consider a system with four types of resources R_1 (3 units), R_2 (2 units), R_3 (3 units), R_4 (2 units). A non-pre-emptive resource allocation policy is used. At any given instance, a request is not entertained if it cannot be completely satisfied. Three processes P_1, P_2, P_3 request the resources as follows if executed independently. **[2009]**

Process P_1:	Process P_2:	Process P_3:
$t = 0$: requests 2 units of R_2	$t = 0$: requests 2 units of R_3	$t = 0$: requests 1 unit of R_4
$t = 1$: requests 1 unit of R_3	$t = 2$: requests 1 unit of R_4	$t = 2$: requests 2 units of R_1
$t = 3$: requests 2 units of R_1	$t = 4$: requests 1 unit of R_1	$t = 5$: releases 2 units of R_1
$t = 5$: releases 1 unit of R_2 and 1 unit of R_1.	$t = 6$: releases 1 unit of R_3	$t = 7$: requests 1 unit of R_2
$t = 7$: releases 1 unit of R_3	$t = 8$: Finishes	$t = 8$: requests 1 unit of R_3
$t = 8$: requests 2 units of R_4		$t = 9$: Finishes
$t = 10$: Finishes		

Which one of the following statements is true if all three processes run concurrently starting at time $t = 0$?

(A) All processes will finish without any deadlock
(B) Only P_1 and P_2 will be in deadlock.
(C) Only P_1 and P_3 will be in a deadlock.
(D) All three processes will be in deadlock.

5. The enter_CS() and leave_CS() functions to implement critical section of a process are realized using test-and-set instruction as follows: **[2009]**

```
void enter_CS(X)
{
    while (test-and-set(X)) ;
}
void (leave_CS(X))
{
    X=0;
}
```

In the above solution, X is a memory location associated with the *CS* and is initialized to 0. Now consider the following statements:

(i) The above solution to *CS* problem is deadlock-free.
(ii) The solution is starvation free.
(iii) The processes enter *CS* in FIFO order.
(iv) More than one process can enter *CS* at the same time.

Which of the above statements is true?

(A) (i) only
(B) (i) and (ii)
(C) (ii) and (iii)
(D) (iv) only

6. Consider the methods used by processes P_1 and P_2 for accessing their critical sections whenever needed, as given below. The initial values of shared Boolean variables S_1 and S_2 are randomly assigned. **[2010]**

Method used by P_1	Method used by P_2
while (S_1 = = S_2);	while (S_1 != S_2);
Critical section	Critical section
$S_1 = S_2$;	S_2 = not (S_1);

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

7. The following program consists of three concurrent processes and three binary semaphores. The semaphores are initialized as $S_0 = 1$, $S_1 = 0$, $S_2 = 0$.

Process P_0	Process P_1	Process P_2
while (true) {	wait (S1);	wait (S2);
wait (S0);	Release (S0);	release (S0);
print '0'		
release (S1);		
release (S2);		
}		

How many times will process P_0 print '0'? **[2010]**

(A) At least twice (B) Exactly twice
(C) Exactly thrice (D) Exactly once

8. Fetch _ And _Add(X, i) is an atomic Read- Modify-Write instruction that reads the value of memory location X, increments it by the value i, and returns the old value of X. It is used in the pseudocode shown below to implement a busy wait lock. L is an unsigned integer shared variable initialized to 0. The value of 0 corresponds to lock being available, while any non-zero value corresponds to the lock being not available. **[2012]**

```
AcquireLock(L) {
while(Fetch_And_Add(L, 1))
L = 1;
}
ReleaseLock (L) {
L = 0;
}
```

This implementation

(A) fails as L can overflow
(B) fails as L can take on a non-zero value when the lock is actually available
(C) works correctly but may starve some processes
(D) works correctly without starvation

9. A shared variable x, initialized to 0, is operated on by four concurrent processes W, X, Y, Z as follows. Each of the processes W and X reads x from memory, increments by one, stores it to memory, and then terminates. Each of the processes Y and Z reads x from memory, decrements by two, stores it to memory, and then terminates. Each process before reading x invokes the P operation (i.e., wait) on a counting semaphore S and invokes the V operation (i.e., signal) on the semaphore S after storing x to memory. Semaphore S is initialized to 2. What is the maximum possible value of x after all processes complete execution? **[2013]**

(A) –2 (B) –1
(C) 1 (D) 2

10. A certain computation generates two arrays 'a' and 'b' such that $a[i] = f(i)$ for $0 \le i < n$ and $b[i] = g(a[i])$ for $0 \le i < n$. Suppose this computation is decomposed into two concurrent processes X and Y such that X computes the array 'a' and Y computes the array 'b'. The processes employ two binary semaphores R and S, both initialized to zero. The array 'a' is shared by the two processes. The structures of the processes are shown below.

Process *X*:

```
private i;
for (i = 0; i<n; i++) {
    a[i] = f(i);
    ExitX(R, S);
}
```

Process *Y*:

```
private i;
for (i = 0; i<n; i++) {
EntryY(R, S);
    b[i] = g(a[i]);
}
```

Which one of the following represents the correct implementations of Exit$X$ and Entry$Y$? **[2013]**

(A)
```
ExitX(R, S) {
    P(R);
    V(S);
}
    EntryY(R,S){
    P(S);
    V(R);
}
```

(B)
```
ExitX(R, S) {
    V(R);
    V(S);
}
    EntryY(R,S){
    P(R);
    P(S);
}
```

(C)
```
ExitX (R, S){
    P(S);
    V(R);
}
    EntryY(R,S){
    V(S);
    P(R);
}
```

(D)
```
ExitX (R, S){
    V(R);
    P(S);
}
    EntryY(R,S){
    V(S);
    P(R);
}
```

11. Consider the procedure below for the *producer-consumer* problem which uses semaphores; **[2014]**

```
semaphore n = 0;
semaphore s = 1;
void producer ()
{
while (true)
{
produce ( )
semWait (s);
addToBuffer ();
semSignal (s);
semSignal (n);
}
}
void consumer ()
{
while (true)
{
semWait (s);
semWait (n);
remove FromBuffer ();
semSignal(s);
consume ( ) ;
}
}
```

Which one of the following is true?

(A) The producer will be able to add an item to the buffer, but the consumer can never consume it.
(B) The consumer will remove no more than one item from the buffer.
(C) Deadlock occurs if the consumer succeeds in acquiring semaphore s when the buffer is empty.
(D) The starting value for the semaphore n must be 1 and not 0 for deadlock free operation.

12. The following two function *P*1 and *P*2 that share a variable *B* with an initial value of 2 execute concurrently. **[2015]**

```
P1 ( ) {                P2 ( ) {
  C = B – 1;              D = 2 * B;
  B = 2 * C;              B = D – 1;
}                         }
```

The number of distinct values that B can possibly take after the execution is _____

13. Two processes *X* and *Y* need to access a critical section. Consider the following synchronization construct used by both the processes **[2015]**

Process X	**Process Y**

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

Here, var*P* and var*Q* are shared variables and both are initialized to false. Which one of the following statements is true?

(A) The proposed solution prevents deadlock but fails to guarantee mutual exclusion.
(B) The proposed solution guarantees mutual exclusion but fails to prevent deadlock.
(C) The proposed solution guarantees mutual exclusion and prevents deadlock.
(D) The proposed solution fails to prevent deadlock and fails to guarantee mutual exclusion.

14. Consider the following proposed solution for the critical section problem. There are n process: $P_0 \ldots P_{n-1}$. In the code, function pmax returns an integer not smaller than any of its arguments. For all i, $t[\,i\,]$ is initialized to zero. **[2016]**

```
do {
c[ i ] = 1; t [ i ] = pmax (t[ i ], ……,
t[n – 1]) + 1; c[ i ] = 0;
for every j ≠ i in (0, …., n – 1) {
while (c [ j ]);
while (t[ j ] ! = 0 && t[ j ] < = t[ i ]);
}
Critical Section;
t[ i ] = 0;
Remainder Section;
} while (true);
```

Which one of the following is TRUE about the above solution?

(A) At most one process can be in the critical section at any time.
(B) The bounded wait condition is satisfied.
(C) The progress condition is satisfied.
(D) It cannot cause a deadlock.

15. Consider the following two - process synchronization Solution.

Process 0	Process 1
Entry: loop while (turn = = 1); (Critical section) Exit: turn = 1;	Entry: loop while (turn = = 0); (Critical section) Exist: turn = 0;

The shared variable *turn* is initialized to zero. Which one of the following is **TRUE**? **[2016]**

(A) This is a correct two - process synchronization Solution.
(B) This Solution violates mutual exclusion requirement.
(C) This Solution violates progress requirement.
(D) This Solution violates bounded wait requirement.

16. Consider a non-negative counting semaphore *S*. The operation *P*(*S*) decrements *S*, and *V*(*S*) increments *S*. During an execution, 20 *P*(*S*) operations and 12 *V*(*S*) operations are issued in some order. The largest initial value of *S* for which at least one *P*(*S*) operation will remain blocked is _____ . **[2016]**

17. A multithreaded program P executes with *x* number of threads and uses *y* number of locks for ensuring mutual exclusion while operating on shared memory locations. All locks in the program are *non-reentrant*, i.e., if a thread holds a lock *l*, then it cannot re-acquire lock *l* without releasing it. If a thread is unable to acquire a lock, it blocks until the lock becomes available. The *minimum* value of *x* and the *minimum* value of *y* together for which execution of P can result in a deadlock are: **[2017]**

(A) $x = 1, y = 2$ (B) $x = 2, y = 1$
(C) $x = 2, y = 2$ (D) $x = 1, y = 1$

18. Consider the following solution to the producer-consumer synchronization problem. The shared buffer size is N. Three semaphores empty, full and mutex are defined with respective initial values of 0, *N* and 1. Semaphore empty denotes the number of available slots in the buffer, for the consumer to read from. Semaphore full denotes the number of available slots in the buffer, for the producer to write to. The placeholder variables, denoted by *P*, *Q*, *R*, and *S*, in the code below can be assigned either empty or full. The valid semaphore operations are: wait () and signal ().

Producer	Consumer

do {
wait(P);
wait (mutex);
//Add item to buffer
signal (mutex);
signal (Q);
} while (1);
	do {
	wait(R);
	wait (mutex);
	//Consume item from buffer
	signal (mutex);
	signal (S);
	} while (1);

Which one of the following assignments to P, Q, R and S will yield the correct solution? **[2018]**

(A) P: full, Q: full, R: empty, S: empty
(B) P: empty, Q: empty, R: full, S: full
(C) P: full, Q: empty, R: empty, S: full
(D) P: empty, Q: full, R: full, S: empty

Answer Keys

Practice Problems 1

1. C	**2.** A	**3.** C	**4.** D	**5.** C	**6.** C	**7.** C	**8.** D	**9.** D	**10.** B
11. D	**12.** B	**13.** B	**14.** B	**15.** C	**16.** A	**17.** D			

Practice Problems 2

1. B	**2.** A	**3.** D	**4.** B	**5.** D	**6.** C	**7.** B	**8.** B	**9.** B	**10.** A
11. B	**12.** C	**13.** D	**14.** D	**15.** C					

Previous Years' Questions

1. D	**2.** A	**3.** C	**4.** A	**5.** A	**6.** A	**7.** A	**8.** B	**9.** D	**10.** C
11. C	**12.** 3	**13.** A	**14.** A	**15.** C	**16.** 7	**17.** D	**18.** C		