OPERATING SYSTEM
WEEK 8 ASSIGNMENT

1.

Considering a system with five processes P0 through P4 and three resources of type A, B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time t0 following snapshot of the system has been taken. Is the system in a safe state? If yes, then what is the safe sequence?

Process	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Process	Need		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

Available Resources of A, B and C are 3, 3, and 2.

Now we check if each type of resource request is available for each process.

Step 1: For Process P1:

Need <= Available

7, 4, 3 <= 3, 3, 2 condition is **false**.

So, we examine another process, P2.

Step 2: For Process P2:

Need <= Available

1, 2, 2 <= 3, 3, 2 condition **true**

New available = available + Allocation

(3, 3, 2) + (2, 0, 0) => 5, 3, 2

Similarly, we examine another process P3.

Step 3: For Process P3:

P3 Need <= Available

6, 0, 0 < = 5, 3, 2 condition is **false**.

Similarly, we examine another process, P4.

Step 4: For Process P4:

P4 Need <= Available

0, 1, 1 <= 5, 3, 2 condition is **true**

New Available resource = Available + Allocation

5, 3, 2 + 2, 1, 1 => 7, 4, 3

Similarly, we examine another process P5.

Step 5: For Process P5:

P5 Need <= Available

4, 3, 1 <= 7, 4, 3 condition is **true**

New available resource = Available + Allocation

7, 4, 3 + 0, 0, 2 => 7, 4, 5

Now, we again examine each type of resource request for processes P1 and P3.

Step 6: For Process P1:

P1 Need <= Available

7, 4, 3 <= 7, 4, 5 condition is **true**

New Available Resource = Available + Allocation

7, 4, 5 + 0, 1, 0 => 7, 5, 5

So, we examine another process P2.

Step 7: For Process P3:

P3 Need <= Available

6, 0, 0 <= 7, 5, 5 condition is true

New Available Resource = Available + Allocation

7, 5, 5 + 3, 0, 2 => 10, 5, 7

Hence, we execute the banker's algorithm to find the safe state and the safe sequence like P2, P4, P5, P1 and P3

2. Explain deadlock prevention and avoidance.

If we simulate deadlock with a table which is standing on its four legs then we can also simulate four legs with the four conditions which when occurs simultaneously, cause the deadlock.

However, if we break one of the legs of the table then the table will fall definitely. The same happens with deadlock, if we can be able to violate one of the four necessary conditions and don't let them occur together then we can prevent the deadlock.

Mutual Exclusion

- At least one resource must be held in a nonsharable mode; that is, only one process at a time can use the resource.
- If another process requests that resource, the requesting process must be delayed until the resource has been released.

. **Deadlock With Mutex Locks**

- Let's see how deadlock can :occur in a multithreaded Pthread program using mutex locks.
- The pthread_mutex_init() function initializes an unlocked mutex.
- Mutex locks are acquired ;and released using pthread_mutex_lock() and pthread_mutex_unlock() respectively.
- If a thread attempts to acquire a locked niutex the call to X pthread_mutex_lock() blocks the thready until the owner of the rnutex lock invokes pthread_mutex_unlock().
- Two mutex lock are created uaing following code example.

/* create and initialize mutex locks */

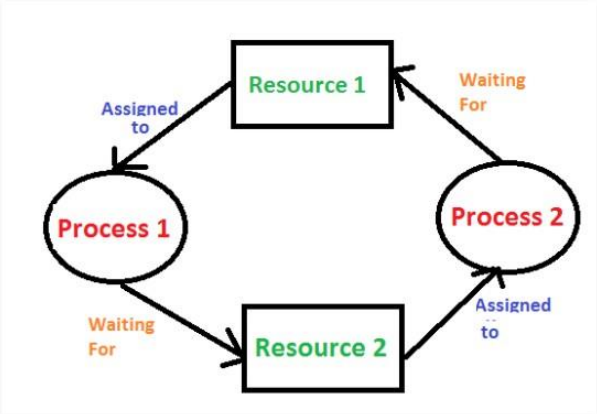
pthread_mutex_t first_mutex

pthread_mutex_t second_mutex

pthread_mutex_init(firxt_mutex,null)

pthread_mutex_init(second_mutex,null)

- Next, two threads—thread-one and thread-two are created, and both threads have access to both mutex locks, thread-one and thread-two run in the functions do_work_one() and do_work_two(), respectively as shown in Figure.



- In this example, thread one attempts to acquire the mutex locks in the order (1) first_mutex (2) second_mutex, while thread_two attempts to acquire the mutex locks: in the order (1) second _mutex (2) first_mutex. Deadlock is possible, If thread_one.

Hold and Wait

- A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

No Preemption

- Resources cannot be preempted. That is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

Circular Wait

- A set {P0, P1, ..., Pn} of waiting processes must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by Pi,……., P(i-1) is waiting for a resource held by Pn.
- We emphasize that all four conditions must hold for a deadlock to occur.
- The circular-wait condition implies the hold-and-wait condition, so the four conditions are not completely independent.

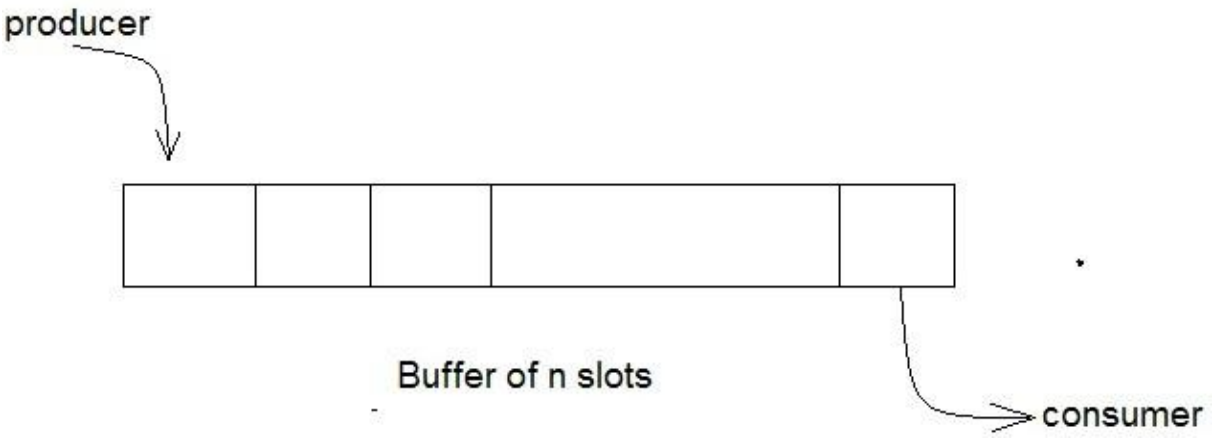
In deadlock avoidance, the request for any resource will be granted if the resulting state of the system doesn't cause deadlock in the system. The state of the system will continuously be checked for safe and unsafe states.

In order to avoid deadlocks, the process must tell OS, the maximum number of resources a process can request to complete its execution.

The simplest and most useful approach states that the process should declare the maximum number of resources of each type it may ever need. The Deadlock avoidance algorithm examines the resource allocations so that there can never be a circular wait condition.

3. With the help of a diagram, explain the bounded buffer problem.

- In Bounded Buffer Problem there are three entities storage buffer slots, consumer and producer. The producer tries to store data in the storage slots while the consumer tries to remove the data from the buffer storage.
- It is one of the most important process synchronizing problem let us understand more about the same.
- The bounded buffer problem uses Semaphore. Please read more about Semaphores here before proceeding with this post here. We need to make sure that the access to data buffer is only either to producer or consumer, i.e. when producer is placing the item in the buffer the consumer shouldn't consume.
- We do that via three entities –
- Mutex mutex – used to lock and release critical section
- empty – Keeps tab on number empty slots in the buffer at any given time
- Initialised as n as all slots are empty.
- full – Keeps tab on number of entities in buffer at any given time.
- Initialised as 0



Producer Buffer Solution

```
do
{
    // wait until empty > 0 and then decrement 'empty'
    // that is there must be atleast 1 empty slot
    wait(empty);
    // acquire the lock, so consumer can't enter
    wait(mutex);

    /* perform the insert operation in a slot */

    // release lock
```

```
    signal(mutex);
    // increment 'full'
    signal(full);
}
while(TRUE)
```

Consumer Buffer Solution

```
do
{
    // wait until full > 0 and then decrement 'full'
    // should be atleast 1 full slot in buffer
    wait(full);
    // acquire the lock
    wait(mutex);

    /* perform the remove operation in a slot */

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

4. What is the monitor? Justify with an example

It is a synchronization technique that enables threads to mutual exclusion and the **wait()** for a given condition to become true. It is an abstract data type. It has a shared variable and a collection of procedures executing on the shared variable. A process may not directly access the shared data variables, and procedures are required to allow several processes to access the shared data variables simultaneously.

Syntax :

```
monitor {

    //shared variable declarations
    data variables;
```

```
Procedure P1() { ... }
Procedure P2() { ... }
.
.
.
Procedure Pn() { ... }
Initialization Code() { ... }
```