19PW13 ASSIGNMENT ON SYNCHRONISATION AND DEADLOCKS

**1.**

**Necessary conditions for a Deadlock:**

1. **Mutual Exclusion:**

There must exist at least one resource in the system that can be used by only one process at a time. Thus, The resources involved must be unshareable.

1. **Hold and wait:**

There must exist a process which holds some resource and waits for another resource held by some other process.

1. **No pre-emption:**

When one process is allocated a resource it cannot be pre-empted, it cannot be snatched and given to another process. The resource must be released by the process voluntarily

1. **Circular wait:**

The processes must wait for resources in a circular manner where the last process waits for resource held by the first one

The code can be fixed as,

|  |  |
| --- | --- |
| **Process P** | **Process Q** |
| acquire(L1) | acquire(L1) |
| acquire(L2) | acquire(L2) |
| release(L1) | release(L2) |
| release(L2) | release(L1) |

This change removed the Hold and wait condition

**2.**

**Difference between a process starting another copy of itself and starting another thread:**

1. Threads run in a shared memory space, while processes run in separate memory spaces.
2. Each process provides the resources needed to execute a program. Each process is started with a single thread, known as the primary thread. A process can have multiple threads in addition to the primary thread.
3. Threads have direct access to the data segment of its process but processes have their own copy of the data segment of the parent process.
4. Changes to the main thread may affect the behaviour of the other threads of the process while the changes made to the parent process do not affect child processes.
5. Processes are heavily dependent on system resources available while threads require minimal amounts of resource, so a process is considered as heavyweight while a thread is termed as a lightweight process.

The possible outputs of the program are:

|  |  |
| --- | --- |
| **Output** | **Order of execution** |
| 9 | [2 3 1] |
| 10 | [1 2 3] [3 2 1] [2 1 3] [3 1 2] |
| 11 | [1 3 2] |

A race condition is when it is possible to observe the order in which events in different processes occur, and that order is not constrained by synchronization.

**This process suffers from race condition** because the output can be different for different orderings of the program.

**3.**

**Deadlock Avoidance:**

1. Requires that the system has some additional apriori information available.
2. Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.
3. The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
4. Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes(future requests and release of resources of each process)
5. The system uses the above information and constructs an algorithm that ensures the system will never enter a deadlock state.

The **Deadlock Avoidance algorithm** dynamically checks the present state and ensures that there can never be any circular wait condition.

The Restaurant problem:

i)

|  |  |  |
| --- | --- | --- |
| **Need** | **Plates** | **Bowls** |
| **P1** | 5 | 4 |
| **P2** | 3 | 5 |
| **P3** | 1 | 1 |
| **P4** | 1 | 2 |

ii)

Work = Available Plates, Available Bowls Work = 2, 1

**Iteration - 1**

We can choose process P3 as Need of P3 is ≤ Work

Work = 2,1 + 0, 1 = 2, 2

Process P3 is completed

**Iteration - 2**

We can choose process P4 as Need of P3 is ≤ Work

Work = 2,2 + 1, 2 = 3, 4

Process P4 is completed

**Iteration - 3**

We can’t choose any process as Need of any process is not ≤ Work So there is **no** safe serving order.

iii)

|  |  |  |
| --- | --- | --- |
| **Need** | **Plates** | **Bowls** |
| **P1** | 5 | 4 |
| **P2** | 3 | 5 |
| **P3** | 1 | 1 |
| **P4** | 1 | 2 |
| **P5** | 3 | 3 |

**The restaurant must add two more plates for a safe serving sequence.**

The serving sequence is:

Work = Available Plates, Available Bowls

Work = 4, 1

**Iteration - 1**

Choosing P3 as Need of P3 ≤ Work

Work = 4,1 + 0, 1 = 4, 2

Process P3 is completed

**Iteration - 2**

Choosing P4 as Need of P4 ≤ Work

Work = 4,2 + 1, 2 = 5, 4

Process P4 is completed

**Iteration - 3**

Choosing P5 as Need of P5 ≤ Work

Work = 5,4 + 2, 0 = 7, 4 Process P5 is completed

**Iteration - 4**

Choosing P1 as Need of P1 ≤ Work

Work = 7,4 + 2, 3 = 9, 7

Process P1 is completed

**Iteration - 5**

Choosing P2 as Need of P2 ≤ Work

Work = 9,7 + 3, 5 = 12, 12 Process P2 is completed

All processes are completed. This is a safe sequence.

**4.**

**i**. A deadlock cannot occur in this case because pre-emption is allowed. If any process requests a resource which is “reserved” by any other “blocked” process, this “reserved” resource is transferred from the “blocked” process to the process that currently requested this resource.

Therefore, the process that requested the resource does not have to wait for the blocked process to execute hence, preventing a deadlock.

**ii.** Yes, Indefinite blocking can occur as the resources of one process is always transferred to other processes which results in that being always blocked from running.

**5.**

**Synchronization**:

**Process Synchronization** is the task of coordinating the execution of processes in a way that no two processes can have access to the same shared data and resources.

Semaphore as synchronisation tool:

Semaphore is simply a variable that is non-negative and shared between threads. It is another algorithm or solution to the critical section problem. It is a signalling mechanism and a thread that is waiting on a semaphore, which can be signalled by another thread.

It uses two atomic operations, 1)wait, and 2) signal for the process synchronization.

wait():

The wait function will wait till the semaphore is free or 1 then decrements the semaphore or occupies it.

signal()

The signal function will increment the semaphore or release the resource.

Semaphores are of two types:

1. **Binary Semaphore –**   
   This is also known as Mutex lock. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problem with multiple processes.
2. **Counting Semaphore –**   
   Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

**Pseudo Code**

For each Intersection a semaphore is considered, so there are 5 semaphores

Sem\_1 = 0

Sem\_2 = 0

Sem\_3 = -2

Sem\_4 = -1

Sem\_5 = -2 **Process 1:**

void func() { *//execution* signal(Sem\_1) signal(Sem\_1) signal(Sem\_1)

}

**Process 2:**

void func() { wait(Sem\_1) *//execution* signal(Sem\_2) signal(Sem\_2)

}

**Process 3:**

void func() { wait(Sem\_2) *// execution*

|  |
| --- |
| signal(Sem\_5)  } |

**Process 4:**

void func() { wait(Sem\_2) *// execution* signal(Sem\_3)

}

**Process 5:**

void func() { wait(Sem\_1) *// execution* signal(Sem\_3) signal(Sem\_3)

}

**Process 6:**

void func() { wait(Sem\_1) *// execution* signal(Sem\_4)

}

**Process 7:**

void func() { wait(Sem\_3) *// execution* signal(Sem\_4)

}

**Process 8:**

void func() { wait(Sem\_4) *// execution* signal(Sem\_5)

}

**Process 9:**

void func() { wait(Sem\_3) *// execution* signal(Sem\_5)

}

**Process 10:**

void func() { wait(Sem\_5) *// execution*

}

**6.**

Here the processes are wait on semaphores ‘S’ at the same time so mutual exclusion is violated resulting in race conditions.

Problems that can occur:

1. When we use semaphores to block processes waiting for a limited resource , deadlocks can occur as one process is blocked while waiting for a resource that can only be free by one of the other blocked processes.
2. Another problem to consider is that of ***starvation***, in which one or more processes gets blocked forever, and never get a chance to take their turn in the critical section. For example, in the semaphores above, we did not specify the algorithms for adding processes to the waiting queue in the semaphore in the wait( ) call, or selecting one to be removed from the queue in the signal( ) call. If the method chosen is a FIFO queue, then every process will eventually get their turn, but if a LIFO queue is implemented instead, then the first process to start waiting could starve.

**7.**

The processes which are blocked are P1, P2, P3 and P5.

The processes which are in deadlock state are P2 and P5.

The Subgraph where deadlock occurs is

**8.**

Semaphores:

Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, wait and signal that are used for process synchronization.

i.

Exactly 3. Process 1 will execute its loop three times incrementing "signal(R)" each time through the loop. This will permit "wait(R)" to complete three times. For every "wait(R)" Process 2 executes, it also executes a "signal(R)" so there is no net change in the value of semaphore R caused by Process 2. Process 3 does decrement the value of semaphore R, typing out "D" each time it does so. So Process 3 will eventually loop as many times as Process 1.

ii.

None (0). If Process 3 is scheduled immediately after Process 1 executes "signal(R)", then Process 2 might continue being stalled at its "wait(R)" statement and hence never execute its "type" statements.

iii.

The output sequence CABABDDCABCABD cannot occur as process 3 cannot run before process 1 runs 2 times which increments the value of R

2 times

iv.

The output sequence CABACDBCABDD can occur when preemption is allowed.

**9.**

A semaphore is an integer variable that can be accessed via 2 atomic operations **wait()** and **signal().**

wait()

The wait function will wait till the semaphore is free or 1 then decrements the semaphore or occupies it.

signal()

The signal function will increment the semaphore or release the resource.

1. There are no initial values for which the given processes prints BUTBUTBUTBU but we can print the characters ‘BUT’ infinite times (when the value of semaphores are S1 = 0; S2 = 1; S3 = 0).
2. No process starts as the value of semaphores are zero and the minimum value required for starting a process is 1.

**10.**

The needs vector is [6,5,4,7]-[1,2,2,4]= [5,3,2,3]

There is 10 units of a single resource shared by 4 processes.

The system is said to be deadlock free if and only if , the sum of max needs< (m+n)

where m🡪no. of resources of same type shared ,

n🡪no. of processes sharing the resource

Therefore, this system is not deadlock free since 5+3+2+3= 13 !< 10

Since there are no processes in the system where the number of resources needed is not less than or equal to the available resources the processes cant be completed and is in safe state.

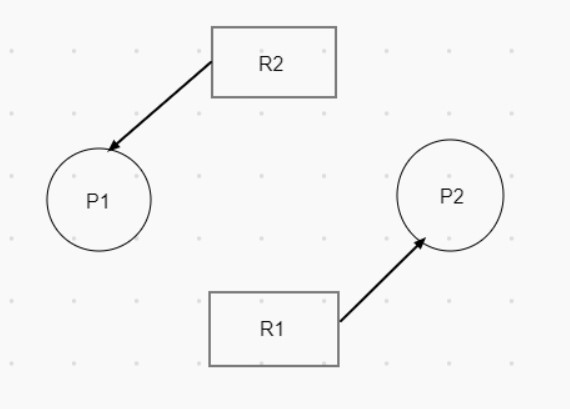
**11.**

i.

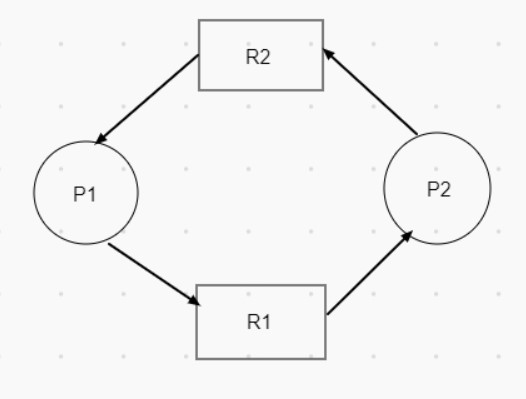
Area A indicates an impossible situation where both processes P1 and P2 have both resources R1 and R2 (deadlock inevitable).

ii.

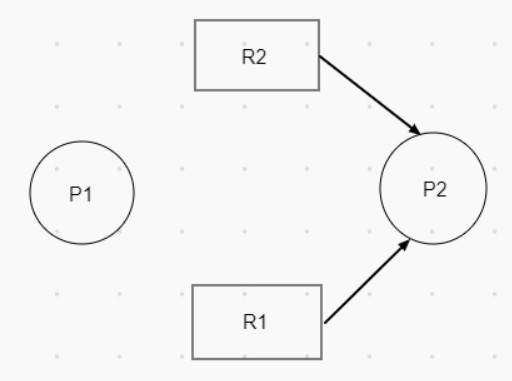
At B the resource allocation graph is



At C the resource allocation graph is



At D the resource allocation graph is



iii.

Different graphs are drawn for B and C as at the point B the process

P1 is using R2 and process P2 is using resource R2. But at Point C the Process P1 is requesting resource R1 and process P2 is requesting resource R2.

This tells us that a resource allocation graph can tell us whether there is a deadlock or not at a particular point of time but we can’t predict whether a deadlock occurs or not even a deadlock is inevitable.

iv.

The banker's algorithm is pessimistic as it assumes that all the processes request maximum required instances of all resources available.

v.

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

sem\_t mutex\_1, mutex\_2;

void\* thread\_1(void\* arg)

{

printf("Entered 1\n");

sem\_wait(&mutex\_1);

printf("Entered 1 Captured mutex 1\n");

sem\_wait(&mutex\_2);

printf("Entered 1 Captured mutex 2\n");

sleep(4);

sem\_post(&mutex\_1);

printf("Entered 1 Released mutex 1\n");

sem\_post(&mutex\_2);

printf("Entered 1 Released mutex 2\n");

}

void\* thread\_2(void\* arg)

{

printf("Entered 2\n");

sem\_wait(&mutex\_2);

printf("Entered 2 Captured mutex 2\n");

sem\_wait(&mutex\_1);

printf("Entered 2 Captured mutex 1\n");

sleep(4);

sem\_post(&mutex\_2);

printf("Entered 2 Released mutex 2\n");

sem\_post(&mutex\_1);

printf("Entered 2 Released mutex 1\n");

}

int main()

{

sem\_init(&mutex\_1, 0, 1);

sem\_init(&mutex\_2, 0, 1);

pthread\_t t1,t2;

pthread\_create(&t1,NULL,thread\_1,NULL); pthread\_create(&t2,NULL,thread\_2,NULL);

pthread\_join(t1,NULL);

pthread\_join(t2,NULL);

sem\_destroy(&mutex\_1);

sem\_destroy(&mutex\_2);

return 0;

}

This program will create a deadlock as there is circular wait occurs when the function thread\_1 executes and it calls wait on mutex\_1, thread\_2 executes and calls wait on mutex\_2 and they both call wait on mutex\_2 and mutex\_1 respectively.

vi.

As we know all the semaphores a program is going to use when thread\_1 executes wait on a semaphore we can make sure and block the wait on that semaphore in thread\_2 which will prevent the deadlock.

**12.**

There are 2 semaphores needed here Sem\_H, Sem\_O both initialized to 0

A variable count which is initialized to 0 is also needed.

|  |
| --- |
| hReady()  {  count++;  if(count%2 == 1) P(Sem\_H); else{ |
| V(Sem\_O);  P(Sem\_H);  } return;  } |

oReady()

{

P(Sem\_O); makeWater();

V(Sem\_H); V(Sem\_O);

return;

}