CSC 8980 Distributed Systems Name: Ishu Goyal

Assignment – 2

Question 1:

- There will be a deadlock in the bridge, with neither the WE nor the EW cars being able to proceed. To remove it, the monitor must regulate both EW and WE, so that if no cars are present at the bridge, the flow of traffic will be determined by the first automobile arriving at its respective WE/EW start point.
- If the automobiles only drive in one way, there will be starvation. To fix that, we can halt traffic flow after a k-car travelling in the same way if there is a car waiting in the opposite direction.
- If cars are travelling in the same direction, we must use the bridge. If cars are waiting on the opposite side, the number of cars will range from 1 to k. If not, traffic will continue to flow after k automobiles until another car from the opposite direction arrives.
- Cars travelling east will enter by the west gate (WE) and exit through the east gate (EW), and vice versa.
- Enter WE and Exit EW are functions for cars travelling from west to east through WE gate and exiting from EW gate, respectively. Westbound autos arriving from EW (process Enter EW) and exiting from WE gate (Exit WE) follow the same method.
- The bridge takes 15 minutes for each automobile to traverse. Assume a distance of t seconds can be bridged between two automobiles.

Pseudocode:

Monitor bridge

Functions: Enter_WE, Exit_EW, Enter_EW, Exit_WE

```
// N is to keep track of the cars.1 <= N <= k
int k,t,N=0


//for cars going to the east, entering from west gate
Enter_WE()
{
N=1
While cars are coming for East direction
do
{
If (no car from west direction) OR (there are k cars for westbound direction already passed) OR (k*(15 min + t sec) passed for east gate entry)

        Enter from East.
        If N<=k
                N = N + 1
```

```
        Continue
    If N==k
        Exit_EW()

}

}
//for cars going to east direction exiting east gate
Exit_EW()
{
    If (N==k for eastbound cars) OR (k*(15 min + t sec) passed for cars entering from West gate)
        Stop Eastbound cars from the west gate
    If there are car(s) going to the west
        Enter_EW()
    Else
        Enter_WE()

}


//for cars going to the west, entering from east gate
Enter_EW()
{
    N=1
    While cars are coming for West direction
    do
    {
        If (no car from East direction) OR (there are k cars for eastbound direction already passed or
k*(15 min + t sec) passed for west gate entry):
            Enter from West.
        If N is less than equal to k
            N = N + 1
            Continue
        if N is equal to k
            Exit_WE()

    }
}

//for cars going to west direction exiting west gate
Exit_WE()
{
    If N == k for westbound cars or k*(15 min + t sec) passed for cars entering from East gate
        Stop Westbound cars from East gate
    If there are car(s) going to East
        Enter_WE()
    Else
        Enter_EW()

}
```

// Calling functions through Main

```
Main()
{
        If cars are coming from WE gate
                If k*( 15 min + t sec) has passed for east going cars entering from WE gate and cars are
waiting at EW, halt cars at WE gate
                        Open EW gate
                        Enter_EW()
                Else
                        Enter_WE()
        If cars are coming from EW gate
                If k*( 15 min + t sec) has passed for west going cars entering from EW gate and cars are
waiting at WE, halt cars at EW gate
                        Open WE gate
                        Enter_WE()
                Else
                        Enter_EW()
}
```

Question 2:

To Prove: $0 \leq$ end_P[empty] – end_P[full] $\leq$ N

Solution:
The given problem describes two processes, the producer and the consumer, who share a common, buffer. The producer's job is to generate data, put it into the buffer, and start again. At the same time, the consumer is consuming the data from the buffer one at a time.
The producer will not add data into the buffer if it is full and that the consumer will not try to remove data from an empty buffer.

The given problem does not have any synchronization issue.

Number of times the consumer remove data from the buffer will be equal to number of times producer puts it in the buffer.

=> end_P[empty] = end_P[full]

=> end_P[empty] -end_P[full]=0

=> $0 \leq 0 \leq N$

Hence Proved.

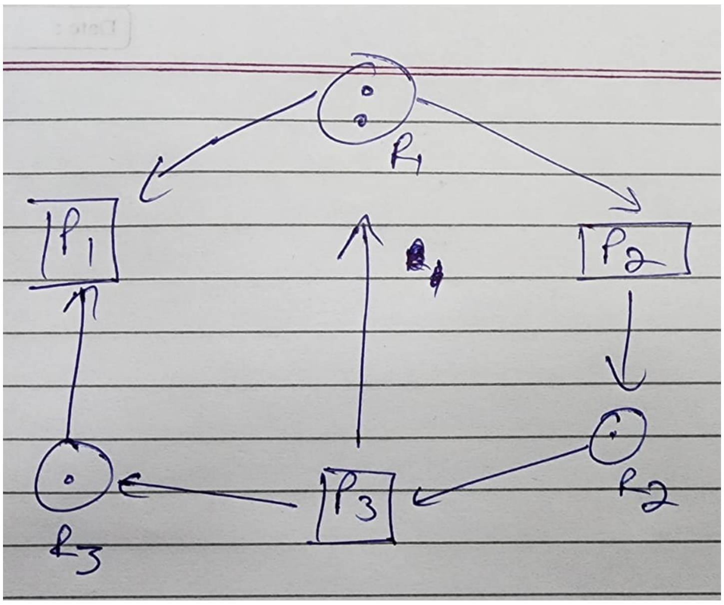
Question 3:

a) **Deadlock -> Cycle**
The statement is True.

All four conditions four deadlock to occur
i) Mutual exclusion
ii) Hold & wait
iii) No preemption
iv) Circular wait

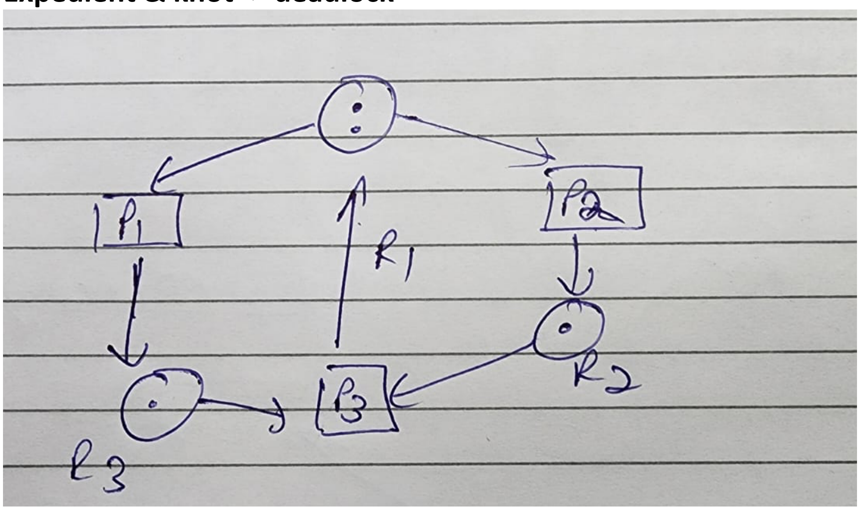
It means the fourth condition is necessary for existence of a cycle for deadlock

b) **Cycle -> deadlock**



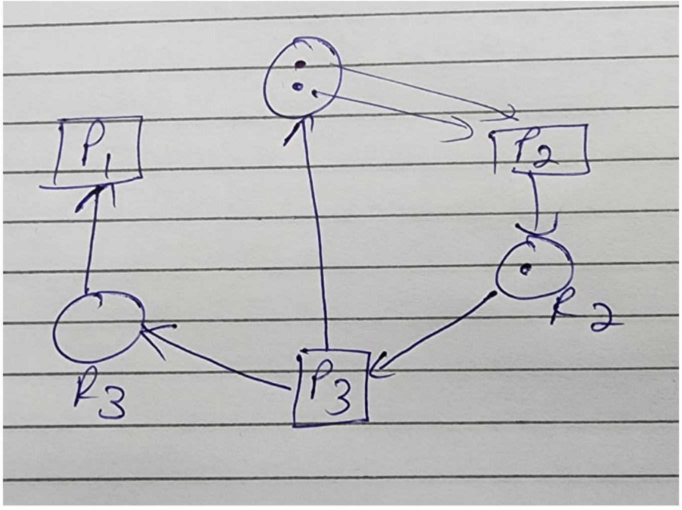
The statement is False because only the existence of a cycle in a resource allocation graph is not sufficient for deadlock to occur. It may be possible that resource graph contains cycle but there is no deadlock. This usually happens when there are more than one instance of a resource.

c) **Expedient & knot -> deadlock**



The statement is True because the presence of a knot is itself a sufficient condition for deadlock to occur.

d) **Deadlock -> Knot**



The statement is False because Occurrence of deadlock doesn't mean that the presence of a knot is guaranteed. Other conditions like Mutual exclusion, Hold & wait, No preemption and Circular wait need to be satisfied for deadlock to occur

Question 4:

Pseudocode:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

void fill_list(int *);
void empty_list(int *);
void show_list();

NODE * headptr;
NODE head;

pthread_mutex_t list_mutex;

/* semaphores for synchronizing fill_list and empty_list threads */
unsigned int threads_fill_done;
sem_t done_filling_list;
sem_t filling_list;

int main()
{
   int i;

   /* initialize list */
   headptr = &head;
```

```
headptr->next = NULL;

/* initialize mutex */
pthread_mutex_init(&list_mutex, NULL);

/* initialize semaphores */
int res = sem_init(&done_filling_list,  /* pointer to semaphore */
               0 ,                /* 0 if shared between threads, 1 if shared between processes */
               0);                /* initial value for semaphore (0 is locked) */
if (res < 0)
{
   perror("Semaphore initialization failed");
   exit(0);
}
if (sem_init(&filling_list, 0, 1)) /* initially unlocked */
{
   perror("Semaphore initialization failed");
   exit(0);
}
threads_fill_done = 0;


pthread_t threads[11];
int param[5] = {0, 1, 2, 3, 4};

for (i = 0; i < 5; i++)
{
   /* creating 5 threads. Each thread enters one number (0-4) in the list */
   pthread_create(&threads[i],
              NULL,
              (void )fill_list,
              (void )&param[i]);
}

for (i = 5; i < 10; i++)
{
   pthread_create(&threads[i],
              NULL,
              (void *)empty_list,
              (void *)&param[i-5]);
}

for (i = 0; i < 10; i++)
   pthread_join(threads[i], NULL);


pthread_mutex_destroy(&list_mutex);
sem_destroy(&filling_list);
sem_destroy(&done_filling_list);

printf("All threads completed. List:\n");
Traverse(headptr);
```

```
    return 0;
}

void fill_list(int *value)
{
    int i;

    /* using mutex before entering critical section */
    pthread_mutex_lock(&amp;amp;list_mutex);
    printf("Thread is inserting number %d in list\n", *value);
    InsertOrdered(headptr,value);// i); / critical section */
    pthread_mutex_unlock(&amp;amp;list_mutex);

    sem_wait(&filling_list); // blocks is semaphore 0. If semaphore nonzero,
                         // it decrements semaphore and proceeds
    if (threads_fill_done == 4)
    {
        printf(&amp;quot;Done filling list. Lifting barrier for 5 empty_list threads.\n&amp;quot;);
 for (i = 0; i < 5; i++)
            sem_post(&done_filling_list); // sem_post increments semaphore. Incrementing it to 5
    }
    else
    {
        threads_fill_done++;
        sem_post(&filling_list);
    }
}

void empty_list(int *value)
{
    /* waiting for list to be filled up */
    printf("Thread is waiting for semaphore to be released to remove %d from list.\n",*value);
    sem_wait(&done_filling_list);

    /* list filled. Removing elements */
    pthread_mutex_lock(&list_mutex);
    printf("Thread is removing number %d from list\n", *value);
    Delete(headptr,*value);
    pthread_mutex_unlock(&list_mutex);
}


void show_list(int *thread_num)
{
    pthread_mutex_lock(&list_mutex);
    printf("Current list from thread %i:\n",*thread_num);
    Traverse(headptr);
    pthread_mutex_unlock(&list_mutex);
}
```