Lab 3:

2) Deadlock Detection

Implementation:

I have maintained a global 2 dimensional matrix of size (NPROC+NSEM) X (NPROC+NSEM). In waitd call, whenever a process enters a blocking state, it means that the process is waiting for that semaphore. I set the corresponding entry in my global matrix to 1. This indicates that the process is waiting for that semaphore. When it comes out of the blocking state, it means that the process has acquired the semaphore and hence I set the the corresponding entry in the matrix to 1. When a process calls signal, it means that it has completed using the semaphore and hence I set the value of 0 in the corresponding global matrix which indicates that the process has given up the semaphore.

I build my matrix in this way. So whenever a process enters a blocking state, I call my function cycle_detect which scans the global matrix and tells if a cycle is present. If a cycle is present, then it means that there is a deadlock in the system

Implemenatation of cycle_detect:- I have used the dfs approach to check if there is a cycle. I will take the first node of the graph, check the other nodes which are connected to this node. Put all the nodes that are connected in the stack. Check each node in the stack and check if it in turn is connected to the first node, which indicates that a cycle is present. The overhead is n^2, where n indicates the number of nodes in the graph. (for us n is NPROCXNSEM).

Concept of deadlock:

You have P1 which has acquired S1 and is waiting to acquire S2. But S2 has been acquired by P2 and P2 is waiting to acquire S1. This condition results in a deadlock. Since each process is waiting for the other semaphore.

Usually systems which provide the isolation/protection do not worry about deadlock detection and prevention. They ask the user to terminate one of the

processes when there is a deadlock. This is because deadlock does not pose any threat to the isolation/protection. This is the Ostrich approach.

Usually there is a overhead involved when we want to detect a deadlock has occurred or not. That is we have to check if there is a cycle in the process-resource graph. So each time whenever a process has to acquire a resource we check the graph to see if there is a cycle. This causes the overhead and makes the system slow.

In our implementation of waitd(), we have detected whether there is a cycle in the graph. So when a deadlock occurs, instead of killing the process, we can make the process with higher priority first acquire the semaphore and once when it finishes we can ask the lower priority process to use the semaphore.

In our example of 2 process, 2 semaphore, suppose P1 has higher priority than P2. Then we can make P2 to give up S2, so that it can be acquired by P1. Once P1 compltes it will give up both S1 and S2. And then P2 can use both of them and complete its task.

Test Cases:

2 process/2 semaphore scenario:

```
s1 = semcreate(1); s2 = semcreate(1);  s3 = semcreate(1);

resume(create(mydeadlocktest, 2048, 20, "d1", 2, s1, s2));
resume(create(mydeadlocktest, 2048, 20, "d2", 2, s2, s1));
```

many process/ many semaphore test case scenario:

```
resume(create(mydeadlocktest, 2048, 20, "d1", 2, s1, s2));
resume(create(mydeadlocktest, 2048, 20, "d2", 2, s2, s3));

resume(create(mydeadlocktest, 2048, 20, "d1", 2, s3, s1));
```

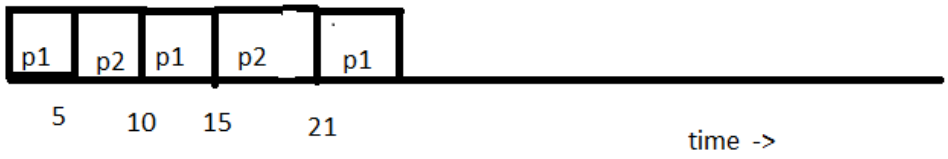
mydeadlocktest.c

```
void mydeadlocktest(sid32 sem1, sid32 sem2){

    waitd(sem1);
```

```
        kprintf("Acquired sem1:%d\n", sem1);


        sleepms(5);



        waitd(sem2);
        kprintf("Acquired sem2:%d\n", sem2);
        sleepms(5);
       signal(sem2);
       kprintf("Gave up sem2: %d\n", sem2);
       signal(sem1);
        kprintf("Gave up sem1: %d\n", sem1);
 }
```

Bonus Problem:

EDF – early deadline first algorithm always schedules the process which has the earliest deadline. For example if you have two processes P1 and P2 with deadlines being T1 = 10 and T2 = 22 respectively. Computation time of P1 and P2 being C1 = 5 and C2 = 11 respectively. The time graph will look like this:



So for EDF, we can design as follows in XINU:

Admission Control – first we check the admission control. Every time an RT process is created we calculate the sum of C_i / T_i . And we check if this sum is less than 1. Only if it is less than 1, we can allow it to continue or else we can raise a system error.

After admission control we can maintain 2 lists. One is a readylist where a real time process' computation time has not been met yet. Another list where a process' computation time has been met and is now waiting for its next deadline.

So first when we can create an RT process, it is put into the ready list. Based on its deadline we know when its deadline will start. Suppose simultaneously when we create other RT processes they are first put into the readylist. Out of all these processes in the readylist, we need to pick the one which has the next earliest deadline period and make it run. We can maintain a table which will keep track of all the next deadline periods of all the RT processes. Suppose when we are executing a process, from the table we know the process which has the next earliest deadline, we then context switch the process which was currently running. Suppose the process which was running, had not yet received its full computation time, it is again put into the reaylist itself and it has to wait for its turn to be executed based upon its next deadline.

Suppose a process gets its full computation time, and it still has some time before its next deadline, then we can put this process in another list called as maybe "comp_satisfied" list. And now its next deadline is reset in the table. A process remains in the comp_satisfied until its next deadline begins. When the next deadline begins for this process, it is moved from the comp_satisfied to the readylist.

So before picking a process to run the scheduler checks the readylist and picks a process which has the earliest deadline from the table.

So EDF can be designed in this way in XINU.

Problem 1:

An RT process is created when we call rt_create(). In this we first check for admission control. This is done by iterating through all the process and we will find the real time process and add its computation period by time period and use the formula given to check the admission control.
In our implementation, we have maintained readlist[60] and this is where we are storing all the ready real time process. This readylist[60] is sorted by the priority of the real time processes. The priority of the real time processes is given by 1/T. So, the process has a higher time period, it has less priority. So, whenever we call reschedule, it will check whether the old process is the real time process or a TS process. If it is a TS process, we again determine whether it is a IO bound process or a CPU bound process and change its priority accordingly. If the old process is a real time process, and is still running, it is still kept in the readylist[60] and it is context switched to other higher priority real time process. The higher priority process is found by scanning the multilevel queue that is the readylist. We scan the multilevel queue from the 60th index and always pick the higher priority process. Due to this the realtime process is always given a higher priority over other processes and they will only be executed 1st.

Test case:

```
resume(rt_create(40,10,myperiodicrtapp, 2048, 20, "r1", 2, 40, 10 ));
resume(rt_create(100,10,myperiodicrtapp, 2048, 20, "r2",2, 100, 10 ));
resume(rt_create(60,15,myperiodicrtapp, 2048, 20, "r2",2, 60, 15 ));
```

Output:

```
arr[1]= 74, arr[1]=-84
arr[2]= 114, arr[1]=-124
arr[3]= 154, arr[1]=-164
arr[4]= 199, arr[1]=-209
arr[5]= 239, arr[1]=-249
arr[6]= 279, arr[1]=-289
arr[7]= 334, arr[1]=-344
arr[8]= 374, arr[1]=-384
arr[9]= 414, arr[1]=-425
arr[10]= 470, arr[1]=-480
….

arr[0]= 55, arr[1]=-70
arr[1]= 124, arr[1]=-139
arr[2]= 184, arr[1]=-199
arr[3]= 259, arr[1]=-274
arr[4]= 319, arr[1]=-334
arr[5]= 384, arr[1]=-400
arr[6]= 444, arr[1]=-459
```

arr[7]= 504, arr[1]=-519

arr[8]= 580, arr[1]=-595

arr[9]= 640, arr[1]=-655

arr[10]= 705, arr[1]=-720

arr[11]= 775, arr[1]=-790

arr[12]= 840, arr[1]=-855

arr[13]= 900, arr[1]=-916

arr[14]= 966, arr[1]=-981

…..

arr[1]= 144, arr[1]=-154

arr[2]= 249, arr[1]=-259

arr[3]= 349, arr[1]=-360

arr[4]= 459, arr[1]=-470

arr[5]= 559, arr[1]=-570

arr[6]= 665, arr[1]=-676

arr[7]= 765, arr[1]=-775

arr[8]= 865, arr[1]=-875

arr[9]= 981, arr[1]=-991

arr[10]= 1081, arr[1]=-1091

arr[11]= 1203, arr[1]=-1213

arr[12]= 1303, arr[1]=-1313

arr[13]= 1403, arr[1]=-1414

arr[14]= 1509, arr[1]=-1519

…

As can be seen from the above output: The RT process which has a time period of 40 is executed first and then process with time period 60 is executed and then process with time period of 100 is executed.

All the computation time are being met within the deadlock time. And the successive differnces between two of the lines gives us time period i.e for example arr[10] – arr[9] = time period.