LAB 3

Problem1:

RT scheduler implementation: Firstly, the processes perform admission control test to check whether it is possible to allocate all the processes at a same time. Therefore, $\Sigma_{i=0}$ rt_comp$_i$/rt_period$_i$ <0.7 if this test fails then we cannot allocate all the resources at the same time and it returns system error.
To implement RMS scheduler, the process with the highest rate should execute first. So, the process priority is defined by rate i.e 1/rt_period. Hence, the process that has smallest rt_period will execute first. If 3 Processes have rt_period and rt_comp times as (50,10), (200,40) and (100,15) respectively, then they will execute in order: P1>P3>P2. For RT processes I have a ready queue set at priority 60, it works as a default static scheduler of xinu. It runs in O(n) and uses insert_rt to insert rt processes and put in sorted order, it picks the highest priority process.
Output: When RT apps are made to run, then the process with highest priority/rate executes first.

```
arr[0] ->  39, arr[0][1]-->-49
arr[1] ->  94, arr[1][1]-->-105
.
.
.
.
arr[0] ->  108, arr[0][1]-->-123
arr[1] ->  208, arr[1][1]-->-223
.
.
.
arr[0] ->  53, arr[0][1]-->-94
arr[1] ->  254, arr[1][1]-->-295
.
.
.
```

We can observe from this output that, the process with highest rate i.e P1 executes first and then P3 and then P2. RT period is achieved when a process runs from period begin to period end. CPU requirements are met when RT comp is achieved. So, the following output can be seen as, P1 executes for 10ms(computation time) and then it sleeps till 94(~39+50) and again it executes for next 10ms. When a process sleeps then it context switches to another process. The CPU requirements met are always sandwiched between upper bound and lower bound of rt_periods.

Problem2:

To detect a deadlock, it is important to find whether there is a cycle or not in a resource allocation graph. If there is any cycle then deadlock is there. So, to detect deadlock in XINU, my algorithm performs depth first search traversal using adjacency matrix to find a cycle in graph. If a cycle is there it then returns system error else it runs the code for wait but add the edge in matrix when process tries to acquire semaphore and is waiting on that semaphore. The size of the matrix is NSEM + NPROC. Whenever a Process is waiting on a semaphore and process acquires a semaphore then it is put as 1 in matrix. When a process releases in signal then the value of process and semaphore index in matrix is put as 0 again.

LAB 3

Mydeadlockapp() has the test case to check deadlock. For testing, I made 2 processes and 2 semaphores: s1 and s2. To create a deadlock, I have conditions:
P1: wait(s1);
Wait(s2);
P2: wait(s2);
Wait(s1);

This detects that algorithm is there. As one process is waiting on the semaphore acquired by other process. It will result into a cycle and deadlock.
The overhead in my deadlock algorithm is O(n^2) as everytime a wait is called, it checks the cycle and cycle detection code take O(n^2).

Yes, if we are not killing a process then any other way to resolve deadlock is ostrich approach which is only valid if OS supports isolation and protection: in this approach OS doesn't care if there is a deadlock. Another way is to prevent deadlock which is to figure out if there will be deadlock in future and if there is any possibility then try to avoid that scenario. Try to avoid resource allocation whenever a cycle is possible. Just maintain a graph such as wait-for-graph and periodically search for cycles.

Problem 3:

EDF Scheduler is a dynamic scheduler where the highest priority processes are the ones which has earliest deadline. So, to implement this we will have one more variable called edf_deadline in process table which will keep value of deadlines of the EDF processes. When we are creating EDF processes then we do admission control test , instead of 0.7 we have <1; then we maintain a global variable which keep tracks of the CPU time a process has been allocated. Whenever a process starts the variable edf_clock maintain a counter of the amount of time a process took and then to calculate it's current deadline we do timer = edf_period-edf_clock. Everytime comp time is achieved we reset the timer value. Else After this we call resched which will place the particular process with new deadline as timer in the ready queue. We then pick the earliest deadline process from queue. Data structure will be maintained, we will just add more queues from 61-99 to keep track of edf processes.