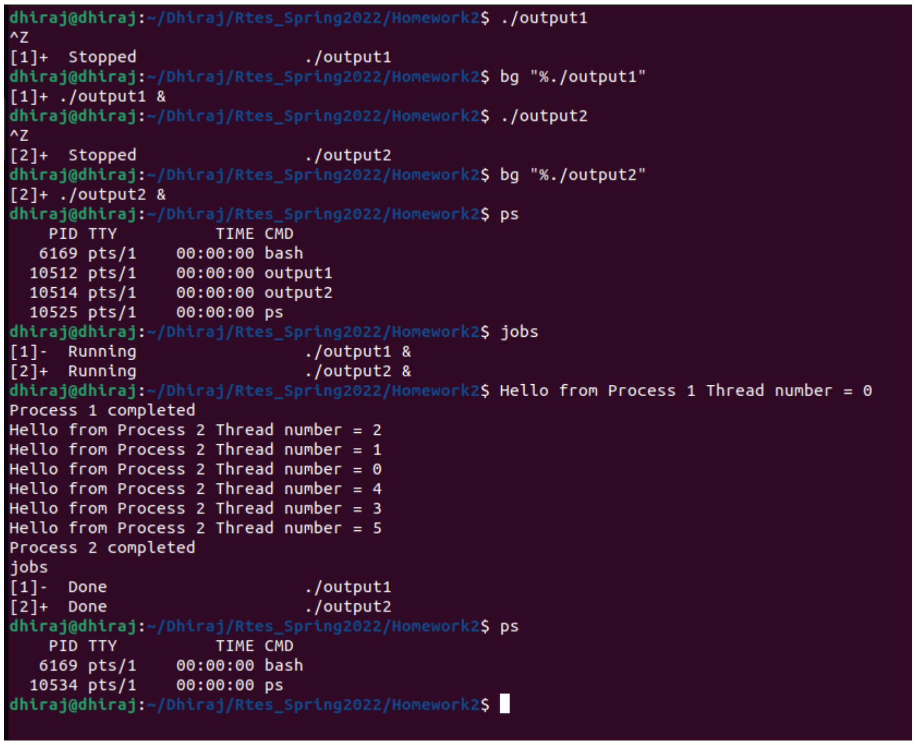
RTES Homework 2

Dhiraj Bennadi

February 20, 2022

Question 1



```
dhiraj@dhiraj:~/Dhiraj/Rtes_Spring2022/Homework2$ ./output1
^Z
[1]+  Stopped                 ./output1
dhiraj@dhiraj:~/Dhiraj/Rtes_Spring2022/Homework2$ bg "%./output1"
[1]+ ./output1 &
dhiraj@dhiraj:~/Dhiraj/Rtes_Spring2022/Homework2$ ./output2
^Z
[2]+  Stopped                 ./output2
dhiraj@dhiraj:~/Dhiraj/Rtes_Spring2022/Homework2$ bg "%./output2"
[2]+ ./output2 &
dhiraj@dhiraj:~/Dhiraj/Rtes_Spring2022/Homework2$ ps
    PID TTY          TIME CMD
   6169 pts/1    00:00:00 bash
  10512 pts/1    00:00:00 output1
  10514 pts/1    00:00:00 output2
  10525 pts/1    00:00:00 ps
dhiraj@dhiraj:~/Dhiraj/Rtes_Spring2022/Homework2$ jobs
[1]-  Running                 ./output1 &
[2]+  Running                 ./output2 &
dhiraj@dhiraj:~/Dhiraj/Rtes_Spring2022/Homework2$ Hello from Process 1 Thread number = 0
Process 1 completed
Hello from Process 2 Thread number = 2
Hello from Process 2 Thread number = 1
Hello from Process 2 Thread number = 0
Hello from Process 2 Thread number = 4
Hello from Process 2 Thread number = 3
Hello from Process 2 Thread number = 5
Process 2 completed
jobs
[1]-  Done                    ./output1
[2]+  Done                    ./output2
dhiraj@dhiraj:~/Dhiraj/Rtes_Spring2022/Homework2$ ps
    PID TTY          TIME CMD
   6169 pts/1    00:00:00 bash
  10534 pts/1    00:00:00 ps
dhiraj@dhiraj:~/Dhiraj/Rtes_Spring2022/Homework2$ 
```

Code Description

The system executes 2 processes named output1 and output2, both of which share a named semaphore.

Steps of Execution

1. Initially the first process ”output1” executes, which creates a thread and acquires the semaphore with the function call sem_wait()
   The thread also sleeps for 50 seconds through the function call sleep()

2. The thread execution is stopped and is made to run in the background

3. Now the process 2 "output2" executes. The process 2 creates 6 threads and all the threads try to acquire the same shared semaphore using the call sem_wait()
   However since the semaphore is with process 1, the 6 threads in process 2 keep waiting
4. Once the sleep time of thread 0 of the process 1 is over, it releases the semaphore using the function call sem_post()
5. Now the 6 threads try to acquire the semaphore and the order of acquiring is not guaranteed.
6. Among the 6 threads in process p2, whichever thread acquires the semaphore, sleeps for 5 seconds and releases the semaphore
7. The execution continues until all the semaphores get the semaphore.

Source code for Question 1

Question 2

EDF Policy
In EDF, the scheduler gives the highest priority to the service that has the soonest deadline whenever a dispatch decision is made.

Drawbacks of EDf Policy

1. Anytime an additional thread is placed on the ready queue, the EDF scheduler must reevaluate all dispatch priorities because the newly added thread may not have a deadline later than all the existing threads on the ready queue.
   For a system that has multiple threads and use forks, the EDF policy has an additional burden to handle this. This is of the order O(n), where n is the number of threads on the queue. This is comparatively a higher overhead if a fixed priority scheduling algorithm is used which has the order of O(1) or a constant time.
2. The EDF policy aims to achieve 100% of the CPU with no margin
   Even the slightest miscalculation on resources required can cause an overload. This requires the system to consider very pessimistic worst-case times for calculations.

EDF Overload

1. In EDF policy, overload leads to non-deterministic failure.
   It is very hard to predict exactly which and how many services will miss their deadlines in an overload. It depends upon the state of the relative priorities during the overload, which in turn depends upon the order of time to deadline times for all services ready to run.

2. An overrun by any service may cause all other services on the ready queue to miss their deadlines.

3. A new service added to the queue adjusts the priorities for all, but will not preempt the overrunning service.The overrunning service has a time to deadline that is negative because it has passed, so it continues to the the highest priority service and continues to cause others to wait and potentially miss deadlines.

4. In an overrun scenario, the common policy is to terminate the release of a service that has overrun. This causes service dropout. This requires CPU margin, hence other services will miss deadline.

5. All services queued while the overrunning service executes potentially miss their deadlines, and the next service is likely to overrun as well causing a cascading failure.

6. To be more deterministic, the best option would be to dump all services.

Question 3

CPU Frequency = 1 GHz = $1 * 10^9$
Number of instructions per frame = 2120
Cycles Per Instruction (CPI) = (CPU Frequency) / (Frames per second * Instruction Count Per frame)
= $1 * 10^9$ / $1 * 10^5$ * 2120
= **4.716**

Overlap Time = Intermediate I/O Time per frame / Time to execute 1 frame
= $4.5 * 10^-6$ / 10^-5
= **.45**

Overlap Percentage = **45%**

References

[1] Real-Time Embedded Components and Systems with Linux and RTOS
Sam Siewert and John Pratt