COSC 3360/6310 - Operating Systems Fall 2017

Programming Assignment 2 - Deadlock Avoidance with SJF plus EDF-Tie-Breaker Scheduling and LLF plus SJF-Tie-Breaker Scheduling

Due Date: Wednesday, November 8, 2017, 11:59pm

In this assignment, you will implement a deadlock avoidance algorithm as part of the Process Manager to avoid deadlocks in a Unix/Linux system. Part of the assignment requires the manipulation of Unix/Linux processes and part of it consists of simulation. You will learn process synchronization with semaphores, process scheduling, and deadlock avoidance.

Both the deadlock-handling process and the processes requesting resources are real Unix/Linux processes created using fork(). However, you do not need to actually allocate any resource. The main process executes the Banker's algorithm. **You are required to use shared memory controlled by Unix/Linux semaphores to synchronize the resource-requesting processes and the deadlock-handling process.** This allows the resource-requesting processes to make requests to the deadlock-handling process.

The deadlock-handling process chooses the next process with a resource request having the shortest remaining execution time (Shortest Job First - SJF) to be serviced. Ties are broken in favor of the process with the nearest absolute deadline (Earliest Deadline First - EDF). After having one request serviced, a process has to allow another process to make a request before making another one, that is, another process with the shortest remaining execution time is chosen for service. A process can also release resources during its execution, and releases all resources held when it terminates.

Associated with each process is also a relative deadline (that is, the deadline from the current time instant). One time unit is needed for servicing each request (whether the resource is allocated or not), so the relative deadline for each process is decreased by 1 whether it is serviced or not. If the resource is allocated for a request, then the computation time of this process decreases by 1; otherwise, the computation time remains the same. If a process misses its deadline, keep servicing its requests but indicate the deadline miss in the output. A "release" instruction also needs 1 unit of computation time just like a request.

A "calculate" instruction does not use resources and its computation time is indicated in parentheses. A "useresources" instruction simulates the use of allocated resources and its computation time is indicated in parentheses.

The input format is as follows:

```
m        /* number of resources */
n        /* number of processes */

available[1] = number of instances of resource 1
   :
available[m] = number of instances of resource m

max[1,1] = maximum demand for resource 1 by process 1
   :
max[n,m] = maximum demand for resource m by process n
```

```
process_1:
deadline_1          /* an integer, must be >= computation time */
computation_time_1  /* an integer, equal to number of requests and releases */
  :                 /* plus the parenthesized values in the calculate and   */
  :                 /* useresources instructions.                           */
  :
calculate(2);          /* calculate without using resources */
calculate(1);
request(0,1,0,...,2);  /* request vector, m integers */
useresources(4);       /* use allocated resources */
  :
release(0,1,0,...,1);  /* release vector, m integers */
calculate(3);
  :
request(1,0,3,...,1);  /* request vector, m integers */
useresources(5);
  :
end.

  :

process_n:
deadline_n          /* an integer */
computation_time_n  /* an integer, equal to number of requests and releases */
  :                 /* plus the parenthesized values in the calculate and   */
  :                 /* useresources instructions.                           */
  :
calculate(3);          /* calculate without using resources */
  :
request(0,2,0,...,2);  /* request vector, m integers */
useresources(2);       /* use allocated resources */
useresources(5);
useresources(3);
  :
release(0,1,0,...,2);  /* release vector, m integers */
calculate(4);
calculate(5);
  :
request(1,0,3,...,1);  /* request vector, m integers */
useresources(8);
calculate(3);
  :
end.
```

For output, print the state of the system after servicing each request: the arrays available, allocation, need, and deadline misses, if any.

Next, let's try LLF (Least Laxity First) instead of SJF. The deadlock-handling process chooses the next process with the least laxity (slack) to be serviced. Laxity (l) is defined as deadline (d) minus remaining computation time (c) at a specific time (t), that is, l(t) = d(t) - c(t). Ties are again broken in favor of the process with the shortest remaining execution time (Shortest Job First - SJF). After having one request serviced, a process has to allow another process to make a request before making another one, that is, another process with the least laxity is chosen for service. Which scheduling technique yields fewer deadline misses, or they both meet the same number of deadlines?