COSC 3360/6310 - Operating Systems Spring 2022

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

Due Date: April 1, 2022, 11:59pm CDT

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. Both the deadlock-handling process and the processes requesting resources are real Unix/Linux processes created using fork().

The input format is as follows. The first two lines contain integers m and n, followed by m integers and then by n * m integers. Next, the instructions for each of the n processes are given.

```
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   */
  :                 /* use_resources instructions.                           */
  :
calculate(2);          /* calculate without using resources */
calculate(1);
request(0,1,0,...,2);  /* request vector, m integers */
use_resources(4);      /* use allocated resources */
calculate(3);
use_resources(6);      /* use allocated resources */
print_resources_used;  /* print process number and current master string */
  :
release(0,1,0,...,1);  /* release vector, m integers */
calculate(3);
  :
request(1,0,3,...,1);  /* request vector, m integers */
use_resources(5);
  :
```

```
print_resources_used;  /* print process number and current master string */
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   */
  :                 /* use_resources instructions.                           */
  :
calculate(3);          /* calculate without using resources */
  :
request(0,2,0,...,2);  /* request vector, m integers */
use_resources(2);      /* use allocated resources */
use_resources(5);
print_resources_used;  /* print process number and current master string */
use_resources(3);
  :
release(0,1,0,...,2);  /* release vector, m integers */
calculate(4);
calculate(5);
  :
request(1,0,3,...,1);  /* request vector, m integers */
use_resources(8);
print_resources_used;  /* print process number and current master string */
calculate(3);
  :
print_resources_used;  /* print process number and current master string */
end.
```

Each resource consists of an English word (a string of characters) indicating an entity, item, food type, etc., followed by a list of resource instances (brands, actual item, etc.) consisting of English words to be read from a second input file or from standard input (stdin). For example, there are 5 resources (types): hotel, fruit, car, tool, and vegetable, with 6, 8, 7, 5, and 8 instances, respectively. Note that the names of the instances of each resource (type) are different, but they are instances of the same resource (type). When a process requests x instances of a resource, any x instances of this resource are acceptable. To ensure that resource instances are allocated mutually exclusively (one instance can only be allocated to one process), you will need Unix/Linux semaphores. The input file for this example is:

```
R1: hotel: Hilton, Marriott, Omni, Intercontinental, Westin, Sheraton
R2: fruit: orange, mango, pear, apple, lemon, banana, watermelon, guava
R3: car: Ford, Mercedes, BMW, Chrysler, Volvo, Porshe, Ferrari
R4: tool: screwdriver, drill, wrench, plier, hammer
R5: vegetable: lettuce, celery, broccoli, tomato, spinach, carroa, potato, onion
```

The main process executes the Banker's algorithm. The resource-requesting processes are required to make requests by communicating with the deadlock-handling process with Unix/Linux

shared memory controlled by Unix/Linux semaphores.

The deadlock-handling process chooses the next process with a resource request having the nearest absolute deadline to be serviced. Ties are broken in favor of the process with the longest remaining execution time (Longest Job First - LJF). 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 nearest absolute deadline 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 'use_resources' instruction "uses" the allocated resources by inserting the English words found within each resource (type) in the master string while maintaining the words alphabetically sorted. The master string in each process is initially empty. Duplicated words should be removed while indicating the number of each repeated word (making it plural) in English as in the first assignment. The names of the resources (types) also need to be sorted alphabetically, and the names of the instances are sorted within each resource (type). The computation time of this instruction is indicated in parentheses.

For output, print the state of the system after servicing each request: the arrays available, allocation, need, and deadline misses, if any. Also, print the process number and its current master string whenever the 'print_resources_used' instruction is executed, which takes 1 unit of computation time. Note that the 'print_resources_used' instruction may appear multiple times in a process code and not only at the end.

Next, let's try LLF (Least Laxity First) with shortest remaining execution time (Shortest Job First - SJF) tie breaker in the second version of your algorithm. thus the deadlock-handling process chooses the next process with the smallest laxity to be serviced. Ties are broken in favor of the process with the shortest remaining execution time. 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 highest priority according to LLF is chosen for service.

Therefore, this project has two runs corresponding to two different schedulers (EDF and LLF). Keep executing processes until they finish even if they have already missed their deadlines. Which scheduling technique yields fewer deadline misses?