CS460 Operating Systems **Spring 2015**

Programming Assignment 4: Deadlock Detection

Instructor: Xinghui Zhao *Due Date: Apr. 24th, 11:59pm*

Program Description

This assignment should be programmed entirely in C and must compile and run on the lab's linux environment. Copying & pasting code is considered cheating and results in an automatic F.

Your goal for this assignment will be to design and implement a deadlock detection algorithm over a *Resource Allocation Graph (RAG)* for single-resource instances.

1. Your program is to be a pure simulation. There should be no need to call `fork()` or other related system calls.

2. Your program should input two parameters:

   (a) m is the number of unsharable resource types

   (b) n is the number of processes/threads we would like to simulate

3. Your program should input a file containing a thread/process' *(de)allocation sequence*. Each line in this file is a 3-tuple `<pid,req,rid>`, where `pid` is the requesting process ID, `req` is a request, which can be either `A` (allocation request) or `D` (deallocation request), and `rid` is the resource ID. For instance, assume we have $m = 3$ resource types and $n = 2$ threads. The following sequence in the file,

   ```
   1,A,1
   0,A,1
   0,D,1
   1,D,1
   ```

   means:

   (a) pid=1 wants to allocate rid=1

   (b) pid=0 wants to allocate rid=1

   (c) pid=0 wants to cancel allocation request for rid=1 (pid=0 was never allocated rid=1)

   (d) pid=1 wants to deallocate rid=1

4. To simplify your implementation, you should assume that `pids` run from 0 to $m-1$ and `rids` run from 0 to $n-1$ in your input file.

5. For each request line that is read from the input, your program should be updating an internal *Resource Allocation Graph (RAG)*. You may use an adjacency matrix or list to represent the RAG. (Which data structure would you choose out in the real-world, and why?)

6. Each line from the input file should generate one of the following output results:

   (a) `REQUEST`: an allocation-request edge has been generated, but not yet allocated (display `pid ---> rid`)

(b) `ALLOC`: an allocation-request edge has been transformed to an allocation edge (display `pid <--- rid`)

(c) `CANCEL`: an allocation-request edge has been removed (display `pid -/-> rid`)

(d) `DEALLOC`: an allocation edge has been removed (display `pid <-/- rid`)

(e) `DEADLOCK`: a deadlock has been detected (display the deadlocked nodes)

7. When a process p makes a request for a resource r, and r is free, then it is allocated to p in the same time instant. Otherwise, p waits in a queue for r. When r is later released, then r should be allocated to a waiting process in FCFS order in the same time instant (see sample interaction)

8. Your program should check for deadlocks on every request. On detecting a deadlock, your program should quit even if unfulfilled requests remain.

9. Your program should warn and ignore:

   (a) Redundant allocation requests. For instance, a process p already requested resource r, but requests for it again later.

   (b) Invalid deallocation or cancellation requests. For instance, a process p does not have a pending request edge nor an allocation edge to r, but requests for deallocation of r.

10. As always, your program should be robust in that it should not crash when given bad input. Your program should throw an error and exit when:

    (a) When the numbers of processes or resources are given to be less than one in the command prompt.

    (b) An unknown command (i.e., not `A` and not `D`) is given in the input file.

    (c) An invalid `pid` or `rid` is found in the input file. For instance, when $m = 3$ resources and $n = 3$ processes, then `3,A,0` is out of range because `pid=3` does not exist.

Sample Interaction

```
# ./Deadlock 1 0 < input_file2
Error: <num procs> must be a positive integer

# cat input_file
1,A,1
0,A,1
0,A,2
1,A,2
1,D,1

# ./Deadlock 3 2 < input_file
t=0     REQUEST         pid=1 ---> rid=1
t=0     ALLOC           pid=1 <--- rid=1
t=1     REQUEST         pid=0 ---> rid=1
t=2     REQUEST         pid=0 ---> rid=2
t=2     ALLOC           pid=0 <--- rid=2
t=3     REQUEST         pid=1 ---> rid=2
t=3     DEADLOCK        pid=0 rid=2 pid=1 rid=1

# cat input_file2
0,A,1
1,A,1
0,A,2
1,A,2
0,D,2
1,D,1
0,D,1

# ./Deadlock 2 2 < input_file2
t=0     REQUEST         pid=0 ---> rid=1
t=0     ALLOC           pid=0 <--- rid=1
t=1     REQUEST         pid=1 ---> rid=1
t=2     Error: "0,A,2" rid=2 out of range! Only allocated 2 resource(s)!

# ./Deadlock 3 2 < input_file2
t=0     REQUEST         pid=0 ---> rid=1
t=0     ALLOC           pid=0 <--- rid=1
t=1     REQUEST         pid=1 ---> rid=1
t=2     REQUEST         pid=0 ---> rid=2
t=2     ALLOC           pid=0 <--- rid=2
t=3     REQUEST         pid=1 ---> rid=2
t=4     DEALLOC         pid=0 <-/- rid=2
t=4     ALLOC           pid=1 <--- rid=2
t=5     CANCEL          pid=1 -/-> rid=1
t=6     DEALLOC         pid=0 <-/- rid=1
```

```
# cat input_file3
0,A,0
0,A,0
0,A,2
1,A,2
0,A,1
1,D,1

# ./Deadlock 3 2 < input_file3
t=0 REQUEST pid=0 ---> rid=0
t=0 ALLOC pid=0 <--- rid=0
t=1 Warn: "0,A,0" request is redundant!
t=2 REQUEST pid=0 ---> rid=2
t=2 ALLOC pid=0 <--- rid=2
t=3 REQUEST pid=1 ---> rid=2
t=4 REQUEST pid=0 ---> rid=1
t=4 ALLOC pid=0 <--- rid=1
t=5 Warn: "1,D,1" rid=1,pid=1 request/allocation edge non-existent!

# cat input_file4
0,A,3
1,A,2
2,A,1
3,A,0
1,A,3
2,A,2
3,A,1
0,A,2

./Deadlock 4 4 < input_file4
t=0 REQUEST pid=0 ---> rid=3
t=0 ALLOC pid=0 <--- rid=3
t=1 REQUEST pid=1 ---> rid=2
t=1 ALLOC pid=1 <--- rid=2
t=2 REQUEST pid=2 ---> rid=1
t=2 ALLOC pid=2 <--- rid=1
t=3 REQUEST pid=3 ---> rid=0
t=3 ALLOC pid=3 <--- rid=0
t=4 REQUEST pid=1 ---> rid=3
t=5 REQUEST pid=2 ---> rid=2
t=6 REQUEST pid=3 ---> rid=1
t=7 REQUEST pid=0 ---> rid=2
t=7 DEADLOCK pid=0 rid=3 pid=1 rid=2
```

What You Need to Submit

- You must create a `Makefile` to compile your source
- Zip up your source files and a README on how to compile and run your program.
- Submit your file(s) to Angel http://lms.wsu.edu before 23:59pm on the due date.