CS 5/7343 Fall 2021

Programming Homework 3

Deadlock Avoidance

Due Date: 4/19 (Tue) 11:59pm (NO late deadlines)

The goal of this program is to use thread programming to implement a deadlock avoidance algorithm (for single resources), using the resource allocation graph. You can use either C or C++ for this program, and you can reuse any code from program 1 and 2 if you want.

The application

We assume that there are N people (process 0, 1, … N-1) that want to obtain some (or all) of a set of k resources (labelled resource 1, 2, 3, … k)

Your program should read in a file that has the following format (you should pass the name of the file via argc/argv):

- The first line contains two numbers, N and k (in that order)
- Then there are N lines, each line represents a person
    - The first item on each line is a word that denote the person's name (for output purpose)
    - The second item is a number how many more numbers are there is on that line
    - Each of the rest of the line is a number either
        - From 1 to k : that means the person will request that resource
        - From -1 to -k: that means IF the person has that resource, it will release it (notice that it may want to acquire the same resource later).

For the base case, you should assume the resource are requested/released in the order of the number.

An example of the file is below

```
5 10
P1 4  4 2 1 3
P2 6 3 -3 10 8 -10 4
P3 6 7 4 3 -3 8 3
P4 5 2 4 8 10 6
P5 7 10 6 7 4 -6 -7 3
```

Base care (90 points)

For the base case, you are to implement the program that create a thread for each person and then wait for the threads to obtained and release the resources.

Notice that you MUST you the resource allocation graph to check for deadlocks and act accordingly.

You are welcomed to used any third-party graph implementation source code. However, you MUST have the source code for compilation, and you should also denote where you get the code from. You are welcomed to implement your own graph type/class, of course.

Main process

The main process of your program do the following

Read the input file
Do all necessary pre-processing
Create all thread, assign each person to a thread
Signal start (threads start to ask for resources)
Wait for all threads to finish

Threads

Each thread should do the following

Do all necessary pre-processing
Wait for the start signal from the main process
Repeat
    If the next item is requesting a resource
        Make the next request by calling the Request() function
        If the request is granted,
            sleep for (1 + q/100) seconds,
        Else
            sleep for (q/100) seconds,
        (q [in both case] is a random number between 0..99]
    Else (the next item is release a resource)
        Release the resource by calling the Release() function
Until all item is done
Sleep for (1 + q/100) seconds (q is a random number between 0..99])
Release all the resources that it is still holding
Exit

Note that q is to be generated every time it is needed using the rand() function. You should have a mutex to ensure only one thread is calling rand() every time.

Sleeping should be done via the nanosleep() function.

Also, if a request is denied, then it will be made again the next iteration (until the request is granted).

Request()

You need to have a function that handle request of resources by a person. It should take in (at least) the person that is making the request, and the resources to be obtained. Each call to Request() can only request one resource.

It should follow the avoidance algorithm for single resource mentioned and use a resource allocation graph. This can either be passed in as a (reference) parameter, or be set as a global object to be manipulated. If the request can be granted, then is should be granted and the graph updated. If it cannot be granted, then it should deny the request.

Release()

You need to have a function that release resource(s) and update the graph if necessary. It should take in (at least) the person that is releasing the resources and the list of resources that it is releasing. Notice that you can write the function such that it handles only one resource at a time.

Notice that a person may or may not request the same resource later after release, you should pass that information to the function (since it may affect how the function behaves).

Mutexes

You need to figure out where/whether you need mutexes for critical sections, and program accordingly.

Output

Your program should output (and only output) the following:

- The following should be printed inside the Request() function:
    - Whenever there is a request to obtain a resource being made, you should print: <Person name> requests resource <number>
    - Once the request is accepted or denied, you should print:: <Person name> requests resource <number> : accepted/denied
        - Accepted or denied is printed depend on the result of the request
- The following should be printed inside the Release() function:
    - Immediately after a person released a resource, you should print: <Person name> releases resource <number>
        - If you release more than one releases at one go, then list them in ascending order of resource number
- You should print the graph in the following instances:
    - Immediately before the main process signal the threads to start (but after all initialization is done)
    - Inside the Request() function
        - Every time before a request is processed (before you decide whether the request is denied or accepted
        - Every time after a request is accepted and the graph is updated
    - Inside the Release() function
        - Every time after the resource(s) are released (if multiple resources are releases inside a single release function, just print the graph after all of them are released).
    - All graph printing inside the Release() and Request() functions should be done after the corresponding print statements are printed
- How to print the graph:
    - The graph should be printed in 3 lines
        - The first line list all the edges from resources to threads.
            - It should be in the format (x, y), where x is the resource number and y is person's name
            - The edges should be listed in ascending order of x

        - If there is no such edge, just print (-1, -1)
    - The second line should print all the claim edges from threads to resources
        - It should be in the format (y, x), where y is person's name and x is the resource number
        - The edges should be sorted in ascending order of the line number (in the file) that the person is (do NOT print the line number). If there are multiple edges coming out of y, those should be listed in ascending order of x
        - If there is no such edge, just print (-1, -1)
    - The third line should print all other edges from threads to resources
        - The same instructions for claim edges applies here.
    - Each pair of edges on a line should be separated by a semi-colon (:)

You should have a print mutex to ensure only one thread is printing at any given time.

Notice that printing a graph should be an atomic process and should not be interrupted by other print statements.

Stage 1 (15 points)

In this part your input file format will change slightly

- The first line now has 3 numbers, the first number is A (the total number of persons), the second is N (the number of threads that is concurrently running at the same time), the third number is k (the number of resources)
- Then each of the following A lines contains information about each thread (same format as the base case).
- You should check whether A >= N, if not you should set A to N.

The main program will start by creating N threads, and start assigning each thread to one of the first N persons.

Whenever a person has finished all the steps, then the corresponding thread should inform the main process that it is done, and then the main process should assign the next person on the list to that "thread". Notice that this does **NOT** mean that you should destroy the current thread and create a new thread, Instead, you should assign the next person to be run to the thread that the person is just finished and let the current thread continue to run. Notice there may be initialization needed when the new person is assigned. You need to decide where to the initialization and whether mutex(es) are needed.

Notice that other threads should continue running as usual.

Extra output for stage 1

In additional to the required output in the base case you are also to print the following:

- When a person is done with all its steps, you should print the following: <person name> finishes.
- When a new person is assigned, the program should print: <person name> assigned to a thread.
    - You should also print the updated graph after the initialization is done.

Stage 2 (20 points)

In this part, we enable a certain degree of pre-emption to occurs.

If a person requests for the same resource 3 times and is denied, then the Request() function should pick a resource that is held by another person to be released (by calling the Release() function) so that the Request() can be satisfied with the avoidance condition still holds. Notice that the person that have the resource pre=empted will need to requested by putting the request at the end of the list of operations. Also any operation for the pre-empted person that release that specific resource will be ignored and skip over.

Notice that it may happen that a person may request a resource while holding it. In the case, the request is granted but there is no need to update the graph.

For example, suppose the list of a person P3 is 1 2 3 4 -3 6 -1 3 7. Now assume P3 has done the first three steps and holds resource 1, 2, 3. Now suppose a request is made by another process which end up forcing P3 to release resource 3. In that case P3 will release resource 3, and a request for resource 3 will be put to the end (so the remaining request is now 4 -3 6 -1 3 7 3). Notice that in this case the -3 operation is ignored (and jumped over), and the second request for resource 3 is deemed successful if at that time P3 still holds resource 3.

Extra output for bonus 2

Here are the additional output required.

- If a request is denied, you should print: <Person name> requests resource <number> : denied <q> times (where q is the number of times the request is denied). Notice that you should print this even at the third time of denial before preempting.
- After the preemption is done, you should print: <Person name> preempted <Resource name> from <Person name>
    - You should also print the updated graph immediately after the print statement

Comment bonus (5 points)

You will get a maximum of 5 points for providing good comments for the program. This includes:

- For each function, put comment to describe each parameter, and what is the expected output
- For each mutex/semaphore used, denote what is it used for
- For each critical section, denote when the start and end of that critical section is (you can name your critical section in various ways to distinguish among them (if necessary))
- For loops and conditional statements that do non-trivial work (you decide what is non-trivial) put comment before it describes what it does.

What to hand in

You should put all you source code into a zip file and upload the zip file to Canvas. For each stage you should have a separate program. You should have a separate program for each stage (base case, stage 1 and stage 2). You can reuse the code from one part to the other.

Also, you are allowed to use any publicly available third-party code for a linked list/FIFO queue. However, you need to include where you obtain the code in your comments.

Full marks

Full marks for 5000-level students is 95, For 7000-level students is 110.