CS 4760 **Operating Systems**
Assignment # 5 Due Date: November 12, 2018

Resource Management

In this part of the assignment, you will design and implement a resource management module for our Operating System Simulator `oss`. In this project, you will use the deadlock avoidance to manage resources, with processes being blocked on their requests until those requests are safe.

There is no scheduling in this project, but you will be using shared memory so be cognizant of possible race conditions.

Operating System Simulator

This will be your main program and serve as the master process. You will start the operating system simulator (call the executable `oss`) as one main process who will fork multiple children at random times. The randomness will be simulated by a logical clock that will be updated by `oss` as well as user processes. The logical clock resides in shared memory and is accessed as a critical resource using a semaphore. You should have two unsigned integers for the clock; one will show the time in seconds and the other will show the time in nanoseconds, offset from the beginning of a second.

In the beginning, `oss` will allocate shared memory for system data structures, including resource descriptors for each resource. All the resources are static but some of them may be shared. The resource descriptor is a fixed size structure and contains information on managing the resources within `oss`. Make sure that you allocate space to keep track of activities that affect the resources, such as request, allocation, and release. The resource descriptors will reside in shared memory and will be accessible to the child. Create descriptors for 20 resources. After creating the descriptors, make sure that they are populated with an initial number of resources; assign a number between 1 and 10 (inclusive) for the initial instances in each resource class. You may have to initialize another structure in the descriptor to indicate the allocation of specific instances of a resource to a process, as well as possible max claims possible from a process.

After the resources have been set up, `fork` a user process at random times (between 1 and 500 milliseconds of your logical clock). Make sure that you never have more than 18 user processes in the system. If you already have 18 processes, do not create any more until some process terminates. Your user processes execute concurrently and there is no scheduling performed. They run in a loop constantly till they have to terminate. These users will periodically request and release resources. As we are using deadlock avoidance, each user process needs a maximum on the amount of any particular resource it will claim. This should be determined in master, but the child will need to know this also, as it should never request a resource past one of its maximums. This should be done with a constant bound, and determined randomly. So if this constant was 3, then process0 would roll a random number from 1-3 to determine its max claim for all of the 20 resources. Note that as this upper bound increases, processes will have greater average max claims and so more resources will be denied by the algorithm. This is one parameter you can tune to increase the chance of resource denials (indicating processes using heavier resource loads).

While the user processes are looping `oss`, is also looping, incrementing the clock, checking to see if it should create another process and dealing with any messages that it might have received from the children (resource requests, releases). The clock should also be incremented slightly whenever the deadlock avoidance algorithm code is run and each time it grants or releases a resource (indicating that it had to do some work).

`oss` should grant resources when asked as long as doing so would not lead to an unsafe state. This will require `oss` to use the safety algorithm (banker's algorithm) to make sure an allocation will not lead to an unsafe state. If a process was not granted the resources that it requests, it should be put in a blocked queue. Whenever a process releases resources, this could unblock some process that is currently waiting, so then the system has to be checked to see if any processes can be woken up and granted their resources. This should be done in the order that they entered the blocked queue.

Your system should run for a maximum of 2 real life seconds, but the amount of simulated seconds maximum is up to you as long as processes are being blocked on invalid requests and I see that processes are then waking up from being blocked when sufficient resources become available.

User Processes

While the user processes are not actually doing anything, they will be asking for resources at random times.

You should have a parameter giving a bound for when a process should request/let go of a resource. Each process when it starts should roll a random number from 0 to that bound and when it occurs it should try and either claim a new resource or release a resource that it already has. It should make this request by putting a request in shared memory. It will continue looping and checking to see if it is granted that resource. Note that a user process should know enough to never try and request more than its max claims would allow.

Processes also need some chance to terminate. To do this, your system should have some probability of termination constant. This probability should be checked every time a process is granted a request. If this probability is met, it should deallocate all the resources allocated to it by communicating to master that it is releasing all those resources and then it should terminate. Make sure that this probability to terminate is low enough that churn happens in your system.

I want you to keep track of statistics during your runs. Keep track of how many requests have been granted, as well as the length of time a process was blocked and waiting on a resource. How many times the deadlock avoidance algorithm was run and the percentage of requests granted. These results should be displayed when your program terminates.

Make sure that you have signal handling to terminate all processes, if needed. In case of abnormal termination, make sure to remove shared memory and semaphores.

When writing to the log file, you should have two ways of doing this. One setting (verbose on) should indicate in the log file every time master gives someone a requested resource, a resource is released, or a process is blocked or unblocked. In addition, every 20 granted requests, output a table showing the current resources allocated to each process.

An example of possible output might be:

```
Master has detected Process P0 requesting R2 at time xxx:xxx
Master granting P0 request R2 at time xxx:xxx
Master has acknowledged Process P0 releasing R2 at time xxx:xxx
Current system resources
     R0  R1  R2  R3  ...
P0  2   1   3   4  ...
P1  0   1   1   0  ...
P2  3   1   0   0  ...
P3  7   0   1   1  ...
P4  0   0   3   2  ...
P7  1   2   0   5  ...
Master has detected Process P7 requesting R3 at time xxx:xxxx
Master blocking P7 for requesting R3 at time xxx:xxx
...
Master has acknowledged Process P0 releasing R2 at time xxx:xxx
Master unblocking P7 and granting it R3 at time xxx:xxxx
      R0  R1  R2  R3  ...
P0  2   1   2   4  ...
P1  0   1   1   0  ...
P2  3   1   0   0  ...
P3  7   0   1   1  ...
P4  0   0   3   2  ...
P7  1   2   0   6  ...
...
```

When verbose is off, it should only indicate when a process is blocked on a request and when one is woken up.

Regardless of which option is set, keep track of how many times master has written to the file. If you have done 100000 lines of output to the file, stop writing any output until you have finished the run.

Note: I give you broad leeway on this project to handle notifications to master and in what order to give resources to a process

once resources are available. Just make sure that you document what you are doing in your README.

Termination criteria comes down to showing churn in your system. I want to be able to see that your deadlock avoidance algorithm is working and that you are letting many processes start and then terminate. I do not want it running any longer than 2 real-life seconds. You can set a maximum on the number of processes launched if you would like (let us say 100) but the important part is that when I test it the log demonstrates deadlock avoidance. This might require tuning processes asking for more resources or increasing the rate at which new processes come into your system.

I suggest you implement these requirements in the following order:

1. Get a makefile that compiles two source files, have master allocate shared memory, use it, then deallocate it. Make sure to check all possible error returns.
2. Get Master to fork off and exec one child and have that child attach to shared memory and check the clock and verify it has correct resource limit. Master should wait for it to terminate. Then when the child terminates Master should verify to you that it terminated. Then test with more than one child.
3. Put in the signal handling to terminate after 2 seconds.Then set up a way for children to communicate with Master their resource requests. Test by having one child send requests for resources to master and master just always fake granting them
4. Set up necessary resource tables and start having the child request or release particular resources and get granted/released that resource if and only if it is available.
5. Get deadlock avoidance set up and figure out how you want to deal with processes that are blocked (could be a queue, or just looking through PCB for one every time)
6. Now set up processes going through the system

Deliverables

Handin an electronic copy of all the sources, `README`, Makefile(s), and results. Create your programs in a directory called *username*.5 where *username* is your login name on hoare. Once you are done with everything, *remove the executables and object files*, and issue the following commands:

```
chmod 700 username.5

cp -p -r username.5 /home/hauschild/cs4760/assignment5
```