CS 4760 **Operating Systems**
Assignment # 5 Due Date: November 17, 2020

Resource Management

Purpose

The goal of this homework is to learn about resource management inside an operating system. You will work on the specified strategy to manage resources and take care of any possible starvation/deadlock issues.

Task

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 strategy, using maximum claims, to manage resources.

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 the 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. Thus, 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 children. Create descriptors for 20 resources, out of which about 20% should be sharable resources[1]. 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.

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.

`oss` also makes a decision based on the received requests whether the resources should be allocated to processes or not. It does so by running the deadlock detection algorithm with the current request from a process and grants the resources if there is no deadlock, updating all the data structures. If a process releases resources, it updates that as well, and may give resources to some waiting processes. If it cannot allocate resources, the process goes in a queue waiting for the resource requested and goes to sleep. It gets awakened when the resources become available, that is whenever the resources are released by a process.

User Processes

While the user processes are not actually doing anything, they will ask for resources at random times. You should have a parameter giving a bound B for when a process should request (or release) a resource. Each process, every time it is scheduled, should generate a random number in the range $[0, B]$ and when it occurs, it should try and either claim a new resource or release

[1] *about* implies that it should be $20 \pm 5\%$ and you should generate that number with a random number generator.

an already acquired resource. It should make the request by putting a request in shared memory. It will continue to loop and check to see if it is granted that resource.

Since we are simulating deadlock avoidance, each user process starts with declaring its maximum claims. The claims can be generated using a random number generator, taking into account the fact that no process should ask for more than the maximum number of resources in the system. You will do that by generating a random number between 0 and the number of instances in the resource descriptor for the resource that has already been set up by `oss`.

The user processes can ask for resources at random times. Make sure that the process does not ask for more than the maximum number of resource instances at any given time, the total for a process (request + allocation) should always be less than or equal to the maximum number of instances of a specified resources.

At random times (between 0 and 250ms), the process checks if it should terminate. If so, it should deallocate all the resources allocated to it by communicating to `oss` that it is releasing all those resources. Make sure to do this only after a process has run for at least 1 second. If the process is not to terminate, make the process request (or release) some resources. It will do so by putting a request in the shared memory. The request should never exceed the maximum claims minus whatever the process already has. Also update the system clock. The process may decide to give up resources instead of asking for them.

I want you to keep track of statistics during your runs. Keep track of how many requests have been granted.

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 `oss` gives someone a requested resource or when master sees that a user has finished with a resource. It should also log the time when a request is not granted and the process goes to sleep waiting for the resource. In addition, every 20 granted requests, output a table showing the current resources allocated to each process.

Invoking the solution

Execute `oss` with no parameters.You may add some parameters to modify simulation, such as number and instances of resources; if you do so, please document it in your `README`.

Log Output

An example of possible output might be:

```
Master has detected Process P0 requesting R2 at time xxx:xxx
Master running deadlock detection at time xxx:xxxx:
   Safe state after granting request
   Master granting P0 request R2 at time xxx:xxx
Master has acknowledged Process P0 releasing R2 at time xxx:xxx
   Resources released : R2:1
Current system resources
 ...
Master has detected Process P3 requesting R4 at time xxx:xxx
Master running deadlock detection at time xxx:xxxx:
   Processes P3, P4, P7 deadlocked
   Unsafe state after granting request; request not granted
   P3 added to wait queue, waiting for R4
Process P2 terminated
   Resources released: R1:1, R3:1, R4:5
Master has detected Process P7 requesting R3 at time xxx:xxxx
...
   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   ...
...
```

When verbose is off, it should only indicate what resources are requested and granted, and available resources.

Regardless of which option is set, keep track of how many times `oss` 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.

Suggested Implementation Steps

I'll suggest that you do the project incrementally. You are free and encouraged to reuse any functions from your proevious projects.

- Start by creating a `Makefile` that compiles and builds the two executables: `oss` and `user_proc`.
- Implement clock in shared memory; possibly reuse the one from last project.
- Have `oss` create resource descriptors in shared memory and populate them with instances.
- Create child processes; make them ask for resources and release acquired resources at random times.
- Use shared memory or message queues to communicate requests, allocation, and release of resources. Indicate the primitive used in your `README`.
- Implement deadlock avoidance and safety algorithms.
- Keep track of output statistics in log file.

Feel free to ask questions about clarifications.

Termination Criterion

`oss` should stop generating processes if it has already generated 40 children, ot if more than 5 real-time seconds have passed. If you stop adding new processes, the system should eventually have no children and then, it should terminate. Tune your parameters so that the systen is able to encounter an unsafe state and that is shown in log file.

Criteria for Success

Make sure that the code adheres to specifications. Document the code appropriately to show where the specs are implemented. *You must clean up after yourself.* That is, after the program terminates, whether normally or by force, there should be no shared memory, semaphore, or message queue that is left allocated to you.

Grading

1. *Overall submission: 30pts*. Program compiles and upon reading, seems to solve the assigned problem in the specified manner.
2. *README/Makefile: 10pts*. Ensure that they are present and work appropriately.
3. *Code readability: 10pts*. Code should be readable with appropriate comments. Author and date should be identified.
4. *Conformance to specifications: 50pts*. Algorithm is properly implemented and documented.

Submission

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:

```
% cd
% chmod 755 ~
% ~sanjiv/bin/handin cs4760 5
% chmod 700 ~
```