

Resource Management

Purpose

The goal of this homework is to learn about resource management inside an operating system. In particular our project will be using deadlock prevention, by the processes only requesting resources in a particular ordering. You will work on the specified strategy to manage resources and take care of any possible starvation/deadlock issues.

Note that this project changes the flow from previous projects. In this project the workers loop continually and examine the clock, while oss loops and increments the clock while waiting for children to send it a message. The workers no longer wait for messages as we did in the previous projects.

In this project it will be necessary for the children to communicate back and forth with oss using a static array of 10 elements (indicating release or request of resources).

Task

In this part of the assignment, you will design and implement a resource management module for our Operating System Simulator oss. We will be using deadlock prevention to ensure a deadlock can never happen.

There is no scheduling in this project, but you will be using shared memory.

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 specific times as in previous projects. You will take in the same command line options as the previous project.

oss should be called as follows:

```
oss [-h] [-n proc] [-s simul] [-t timeLimitForChildren]
    [-i fractionOfSecondToLaunchChildren] [-f logfile]
```

While the first two parameters are similar to the previous project, the -t parameter is back to how it was in project 3 (if -t 3 is done, then a child should wait until 3 total seconds have passed in our simulated system and then terminate). The -i is as previous, indicating how often we decide to launch another one, as long as it does not violate our limits.

The -f parameter is for a log file, where you should write the output of oss (NOT the workers). The output of oss should both go to this log file, as well as the screen. This is for ease of use for this project, so yes I realize your output will be done in two places. Again, THE WORKER OUTPUT SHOULD NOT GO TO THIS FILE.

In the beginning, oss will allocate shared memory for our clock. In addition, you will need a process table as we had in previous projects, but possibly with different entries. You will also need to allocate a resource table. All the resources are static and should have a fixed number of instances defined in a header file. 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. We have 10 resources classes, with 5 instances of each resource class. We are using resource prevention in this project and the resources are ordered from R0 to R9, with requests only allowed from larger numbered resources if you already have lower numbered resources. Note that this is enforced by the child processes only making good requests and oss is not checking this. For grading and testing though, if you see that this is happening (or a deadlock happens) then your workers are doing something wrong!

After creating the structure, make sure to initialize the data structures. 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` initially start by creating a child process. Make sure that you never have more than 18 user processes in the system no matter what options are given, but other than that periodically try to add a process to the system as long as it does not violate our limits. If it would violate the limits, just do not launch it and wait until the next interval to try and launch it (but do not consider it as having launched as far as the total). The workers will run in a loop constantly till they decide to terminate.

`oss` should grant resources when asked as long as it can find sufficient quantity to allocate. If it cannot allocate resources, the process goes in a queue waiting for the resource requested and is not sent a message back, essentially leaving it in a blocked state. It gets awakened by a message when the resources become available, that is whenever the resources are released by a process. Make sure to release any resources claimed by a process when it is terminated.

Important: As `oss` is incrementing the clock continually while checking for messages, it should do a `msgrcv` using `IPC_NOWAIT` as the last parameter to the `msgrcv`. I have an example of this available under our `canvas` Modules.

Overview of the procedure

```
while (stillChildrenToLaunch or childrenInSystem) {

    increment the clock by some amount

    determine if we should launch a child

    check to see if we can grant any outstanding
    requests for resources by processes
    that didnt get them in the past. If so, grant them
    those resources and send them a message

    do a non-blocking message receive from any children

    if (msg from child) {
        if (request)
            grant it if we can by sending message to child
        if (release)
            release it by sending message to child
        if (terminate)
            release all resources it has
    }

    Every half a second, output the resource table and
    process table to the logfile and screen

}

Output statistics
```

User Processes

The user processes in this project do not require that they get sent any information. Instead, they will each make their own determination about what they do. In general, they will go into a loop, examining the system clock and then randomly deciding

to request or release resources or whether to terminate. This requires that they know how many resources exist in total in the system and how many they have in particular at any given time.

The user processes will ask or release resources at random intervals. In particular, each process, should generate a random time interval in the range $[0, 100ms]$ and when that time has passed in the system, it should try and either claim some resources or release already acquired resource. It should make the request or release by sending a message to oss. It should then wait to get a message back indicating it was granted or released before continuing on. In the case that it was blocked (ie: not given the resource), it would just be stuck waiting on a message.

As we are doing deadlock prevention by avoiding circular wait, when processes want to request resources, they should always request one normally and decide the resource they want randomly. Then, before actually requesting it, they should check to see if granting that resource breaks resource ordering. If so, they should request from oss to release the amount of necessary resources to be able to grant that request and then, after oss sends a message back granting the release, the process should request those resources and the new resource all at once. For example, suppose a child already had 2 of R0, 3 of R2 and 4 of R4. It then wanted to request 1 of R3. It would first request to release 2 of R0 and 3 of R2. Once oss sent a message back, it would then immediately request 2 of R0, 3 of R2 and 1 of R3 all at once.

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 resource. This requires that each process keep track of how many resources they have of each particular type and for them to know the maximums but they DO NOT need to know what any other process wants or needs. Just to emphasize this point, a process could over time end up requesting all the resources of the system and in fact would be granted this if it was the only process in the system.

As each process could request or release resources, we should prefer that processes request resources more than they release them. This should be a parameter in your system for this. You should tune this so we sometimes see processes trying to break the ordering, which would require that they do a mass release and reacquire. I would suggest starting at about 60% request and 40% release and tune from there.

If processes notice that their time has passed in the system, they should terminate by sending a message to oss that they are releasing all their resources and in the same message also indicate that they are terminating.

Ending report

oss should do an end report with some statistics. Have it output the total number of resources requested and the amount of times a mass release and reacquire was done. Also the percentage of requests that were granted immediately versus the amount of total requests.

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 message queues.

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 oss 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.

Log Output

An example of possible output might be:

```
OSS has detected Process P0 requesting R2 at time xxx:xxx
OSS granting P0 request R2 at time xxx:xxx
OSS has acknowledged Process P0 releasing resources at time xxx:xxx
    Resources released : R2:1
Current system resources
...
OSS has detected Process P3 requesting R4 at time xxx:xxx
```

```

OSS: no instances of R4 available, P3 added to block queue at time xxx:xxx
OSS has detected Process P2 releasing resources at time xxx:xxx
    Resources released: R1:1, R3:1, R4:5
OSS has detected Process P2 requesting resources at time xxx:xxx
    Resources released: R1:1, R3:1, R4:5, R5:1
OSS 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 10000 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 `oss` and how you resolve the deadlock. Just make sure that you document what you are doing in your README.

Suggested Implementation Steps

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

- Start by creating a `Makefile` that compiles and builds the two executables: `oss` and `user_proc`. [1 day]
- Implement clock in shared memory; possibly reuse the one from last project. [1 day]
- Have `oss` create resource descriptors and populate them with instances. [2 days]
- Create child processes; make them ask for resources and release acquired resources at random times. [4 days]
- Use message queues to communicate requests, allocation, and release of resources. Indicate the primitive used in your README. [2 days]
- Implement processes requesting and releasing according to deadlock prevention.
- 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 up to `n` processes, or if more than 5 real-time seconds have passed.

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.

README

It is required that you submit a README file. This file should start with the following:

```
Name: Yourname
Date: When project started
Environment: What environment you used (linux, vi)
How to compile the project:
    Type 'make'
Example of how to run the project:
    ./oss -n 3 -s 2 -t 4 -i 0.6 -f log.txt
```

In addition, if you use generative AI, you must include a section describing your use of generative AI in detail. This should include a bare minimum of what generative AI was used and some list of prompts that you used. I also want at least a few sentence description of in general how useful you found it.

If you are not using generative AI, please explain why not. This is not a problem of course.

For example, at a bare minimum something like:

Generative AI used : chatgpt

Prompts:

```
Please give me a C++ command line program to play tic-tac-toe between two people
Change the previous code to add the capability to play multiple games
Add the capability to keep track of the previous record of players.
Explain to me why dogs like fire hydrants.
Given the previous code, please add a command line option
to allow one player to cheat
etc
```

```
Summary: The initial game generated worked well, but it had one bug
that I had to fix.
It did not work to generate the cheating option as chatgpt
claimed I was unethical.
```

Submission

Handin an electronic copy of all the sources, README, Makefile(s), and results. Create your programs in a directory called *username.6* where *username* is your login name on opsys. 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/hauschildm/cs4760/assignment5
```