COMPSCI 340 & SOFTENG 370

Assignment 1 - Priority Inversion

Due date: 9:30pm Monday the 19th of August

Worth: 6% of total mark

The problem of priority inversion will be described in lectures but you should start this assignment immediately. This assignment demonstrates the problem using a simple simulation of a scheduler. You have to understand the scheduler and the resource controller, complete a little bit of scheduler code and then implement a solution to the priority inversion problem. The source code you start with is in the `part1.py` file

The scheduler

The `Scheduler` class implements a simple scheduler that runs on Linux (or any Unix). It is not a real scheduler in the sense that it is invoked automatically by the operating system when a new process needs to be scheduled. Instead, it is an emulation of a scheduler that runs once a second, making scheduling decisions.

The scheduler works by sending signals to the processes it schedules. It uses the same job control signals that the shell uses to suspend jobs and then resume them, e.g., by typing `ctrl-Z` in bash and then typing `fg`. These signals are known as `SIGSTOP` and `SIGCONT`. When a process receives the `SIGSTOP` signal it pauses, only to resume when it receives the `SIGCONT` signal. The code in the scheduler is:

```
# Suspends the currently running process by sending it a STOP signal.
def suspend(process):
    os.kill(process.pid, signal.SIGSTOP)

# Resumes a process by sending it a CONT signal.
def resume(process):
    if process.pid: # if the process has a pid it has started
        os.kill(process.pid, signal.SIGCONT)
    else:
        process.run()
```

The `resume` also starts processes running for the first time by calling `process.run`, a method in the `SimpleProcess` class.

Every second the scheduler checks to see which runnable process has the highest priority. If the current process still has the highest priority it continues. Otherwise the current process is suspended and the best priority process is resumed.

```
def run(self):
    current_process = None
    while True:
        next_process = self.select_process()
        if next_process == None: # no more processes
            controller_write.write('terminate\n')
            sys.exit()
        if next_process != current_process:
            if current_process:
                self.suspend(current_process)
            current_process = next_process
```

```
            self.resume(current_process)
        time.sleep(1)
        ...
```

If there are several processes with the same priority they are scheduled round-robin. e.g., If the ready queue looks like this

process 1:priority 4 - process 2:priority 4 - process 3:priority 2

Process 1 will be scheduled first; then process 2. These two processes will take turns until they are no longer runnable. Then process 3 can run.

Requesting the resource

In order to get priority inversion there must be a resource that is held by a lower priority process when it is needed by a higher priority process. There has to be a way to allocate the resource and control access to it. In this assignment the job is done by the `Controller` class.

A process requests a resource either by making a system call or by sending a message. In this assignment the processes will send messages via pipes that are set up for you. The Controller thread is actually the main thread of the program and is running in the same process as the Scheduler thread. The processes that are to be scheduled each run in their own independent processes.

When a process wants to use the resource it sends a `request` message to the controller. It then blocks until the controller sends a `reply` message back. After a process has finished with the resource it must notify the controller with a `release` message. This is demonstrated by the functions `low_func` and `high_func` which execute as SimpleProcesses in the `part1.py` file. e.g.,

```
def high_func(proc):
    pid = os.getpid() # who am I?
    print('High priority:', pid, '- just about to request resource')
    controller_write.write('{0}:request\n'.format(pid)) # request
    response = proc.read.readline() # wait for reply
    print('High priority:', pid, '- got resource')
    controller_write.write('{0}:release\n'.format(pid)) # release
    print('High priority:', pid, '- released resource')
```

The `request` and `release` messages include the process id (`pid`) of the process making the request.

When the resource is being used by one process, all other requesting processes are kept in a FIFO queue.

Part 1

You must complete the `Scheduler` class – `add_process`, `remove_process`, `select_process` in the `part1.py` program. You may also modify the scheduler's `__init__` method (or constructor). These methods add processes to the ready list, remove them from the ready list and select the process that should be running next. The ready list is the queue of processes that are ready to run (i.e. not waiting for any resource). The ready list must always be maintained in the order that processes should be run, i.e. element 0 must be the currently executing process and element 1 should be the next process to execute etc.

Unless you use a lock (or similar synchronisation mechanism) there will be a race condition in your code. Make sure you fix this. There are no race conditions within the controller because a single pipe is shared by all processes to send messages to the controller and pipes are guaranteed to preserve message integrity if the messages are small enough. The controller thread reads from this

one pipe. In order for the controller to send messages back to the correct process, each process has its own pipe to read from.

When you have completed the `Scheduler` class the `part1.py` program should demonstrate priority inversion when run.

The output from the program should look very much like this:

```
$ python3 part1.py
Low priority: 99517 - just about to request resource
owner pid: 99517
Low priority: 99517 - got resource
High priority: 99528 - just about to request resource
owner pid: 99517
Mid priority: 1
Mid priority: 2
Mid priority: 3
Mid priority: 4
Mid priority: 5
Mid priority: 6
Mid priority: 7
Mid priority: 8
Mid priority: 9
Mid priority: 10
remove process 99551 from ready list
Low priority: 99517 - released resource
owner pid: 99528
High priority: 99528 - got resource
High priority: 99528 - released resource
owner pid: None
remove process 99528 from ready list
Low priority: 99517 - finished
remove process 99517 from ready list
finished
```

The high priority process is blocked because the low priority process is holding the resource and the low priority process cannot release the resource until the middle priority process has completed. Even after the middle priority process is finished the high priority process has to wait but that isn't priority inversion it is just normal waiting for a resource to be released.

Part 2

Once you have that working, make a copy of your `part1.py` program and rename it `part2.py`. Then modify the `Controller` class to prevent priority inversion.

Also ensure that if more that one process is waiting for the resource, the process with the highest priority will get the resource when it is released. If there are several waiting processes with the same priority the resource is allocated to the one that has been waiting the longest.

The output from the `part2.py` program should be very much like this:

```
$ python3 part2.py
Low priority: 181 - just about to request resource
owner pid: 181
Low priority: 181 - got resource
High priority: 191 - just about to request resource
owner pid: 181
Low priority: 181 - released resource
owner pid: 191
High priority: 191 - got resource
High priority: 191 - released resource
owner pid: None
```

```
remove process 191 from ready list
Mid priority: 1
Mid priority: 2
Mid priority: 3
Mid priority: 4
Mid priority: 5
Mid priority: 6
Mid priority: 7
Mid priority: 8
Mid priority: 9
Mid priority: 10
remove process 302 from ready list
Low priority: 181 - finished
remove process 181 from ready list
finished
```

In this case the low priority process has its priority boosted to that of the high priority process when the high priority process waits for the resource. This means that the low priority process releases the resource quickly, allowing the high priority process to continue before the middle priority process completes.

Your `part1.py` and `part2.py` programs will be tested as is. Then the markers will add extra functions at the top of your `part2.py` program and create processes from them and add these to the scheduler after the call to `time.sleep` near the bottom of the file to create a more complicated test. The more complicated test will use different priority values (a higher number means a better priority) and have a number of different processes running and accessing the resource. Several processes may have the same priority and some processes may request the resource more than once. If a process requests the resource while it already has access to it, your solution must count the number of allocations and not free the resource until the process has released the resource the same number of times it was allocated (similar to the way a Java thread can get repeated access to an object lock).

Questions

1. Explain where the race condition could occur in `part1.py`, show and explain the code you inserted to stop this happening.

2. In the Schedulers `run` method there is the following statement:

   ```
   os.waitpid(current_process.pid, os.WNOHANG) != (0, 0)
   ```

   What does this code do, and why is it necessary?

3. The Scheduler code is different from a true scheduler in very important ways – how the scheduler code is invoked and how the processes are suspended and resumed. Briefly describe how these things are done in a real scheduler.

4. The implementation of the Controller class depends on writing and reading from pipes. Describe the purpose of each line in the SimpleProcess `run` method. Also explain what would happen if the `os._exit(0)` call was not made, and why.

5. Extra for experts. There is at least one other possible race condition in the code (not counting the one you fixed in Part 1. Describe what it is, how it could occur and what the consequences would be in that case.

6. Extra for SOFTENG 370 students. The `processes` dictionary (hash map) variable has a memory leak (or something similar to it). Describe how the *leak* occurs and give a solution so that values which are no longer needed by the program can be released and eventually garbage collected. You don't need to provide the code to do this although some code would probably help in explaining your solution.

Submitting the assignment

Make sure your name and upi is included in every file you submit. You may not get the marks for these files if this is not done.

Use the assignment drop box `https://adb.auckland.ac.nz` to submit your `part1.py`, and `part2.py` files.

Submit your answers to the questions as a single text file, called `a1Answers.txt`.

Any work you submit must be your work and your work alone – see the assignment page on the Operating System web site http://www.cs.auckland.ac.nz/compsci340s2c/assignments/.

Marking guide

Proper use of synchronization when required. (2 marks)

`part1.py` works correctly (i.e. it shows priority inversion). (2 marks)

`part2.py` works correctly (i.e. solves the problem of priority inversion). (4 marks)

`part2.py` works correctly with more complicated schedules of processes and resource requests. (4 marks)

Questions

1. 2 marks
2. 2 marks
3. 2 marks
4. 2 marks
5. 1 mark (but you can't go over 20 or 22 marks)
6. 2 marks

Total: 20 marks for COMPSCI 340, 22 marks for SOFTENG 370

Penalty for late submission: 1 mark per day up until 4 days late.