Project 2: Process Synchronization

Due: Monday, July 6, 2020 @11:59pm
Late: Wednesday, July 8, 2020 @11:59pm with 10% reduction per late day

Table of Contents

CS/COE 1550 – Introduction to Operating Systems

Project Overview

Anytime we share data between two or more processes or threads, we run the risk of having race conditions and deadlocks. In race conditions, the shared data could become corrupted. In order to avoid these situations, we have discussed various mechanisms to ensure that one process's critical regions are guarded from another's.

One place that we might use parallelism is to simulate real-world situations that involve multiple independently acting entities, such as people. In this project, you will use the *condition variables and locks* (based on the semaphore implementation that you finished in Project 1) to model the **safe museum tour problem**, whereby visitors and museum tour guides synchronize so that:

- a visitor cannot tour a museum without a tour guide,
- a tour guide cannot open the museum without a visitor,
- a tour guide leaves when no more visitors are in the museum,
- a tour guide cannot leave until all visitors in the museum leave,
- at most **two** tour guides can be in the museum at a time, and
- each tour guide provides a tour for **at most ten** visitors.

Your job is to write a program that (a) always satisfies the above constraints and (b) under no conditions will cause a deadlock to occur. A deadlock happens, for example, when the museum is empty, a tour guide and a visitor arrive, and they stay waiting outside the museum forever.

The condition variables and locks have been implemented for you. The implementation files are at:

- /u/OSLab/original/condvar.h
- /u/OSLab/original/condvar.c

A sample program for using condition variables and locks is also provided at /u/OSLab/original/test.c.

The condition variables and locks are implemented using the semaphore implementation that you finished in Project 1. If you have not finished Project 1, you can find a modified kernel with semaphore implementation in the following files:

- /u/OSLab/original/bzImage
- /u/OSLab/original/System.map

Project Details

You are to write (a) the visitor process, (b) the tour guide process, and (c) **six** functions called by these processes: (c1) `visitorArrives`(), (c2) `tourMuseum`(), (c3) visitorLeaves(), (c4) `tourguideArrives`(), (c5) `openMuseum`(), and (c6) `tourguideLeaves`(). You will also write two processes, (d) one for simulating tour guides' arrival and (e) the other for simulating visitors' arrival.

`visitorArrives`() and `tourguideArrives`()

In order to open a museum for tours, there need to be at least one tour guide and one visitor. `visitorArrives`() is called by visitor processes and `tourguideArrives`() is called by tour guide processes. Both functions must **block** until a tour guide **and** a visitor both arrive.

An arriving visitor must wait if

- the museum is closed or
- the maximum number of visitors has been reached

An arriving tour guide must wait if

- the museum is closed, and no visitor is waiting outside or
- the museum is open, and two tour guides are inside the museum.

While only one tour guide is inside the museum,

- up to ten visitors may tour the museum (i.e., call `tourMusuem`()) and
- a second tour guide may open the museum (i.e., call `openMuseum`()).

When a tour guide arrives, the following message is printed to the screen:

```
Tour guide %d arrives at time %d.
```

When a visitor arrives, the following message is printed to the screen:

```
Visitor %d arrives at time %d.
```

`tourMuseum`() and `openMuseum`()

After `tourguideArrives`() returns, the tour guide immediately calls `openMuseum`(). After `visitorArrives`() returns, the visitor immediately calls `tourMuseum`(). Each visitor takes **2 seconds** to tour the museum (you can implement that by calling `nanosleep` or `sleep`). A visitor inside `tourMuseum`() must not block another visitor who is also inside `tourMuseum`().

When a tour guide opens the museum, the following message is printed to the screen:

```
Tour guide %d opens the museum for tours at time %d.
```

When a visitor tours the museum, the following message is printed to the screen:

```
Visitor %d tours the museum at time %d.
```

`visitorLeaves`() and `tourguideLeaves`()

Tour guides that are inside the museum cannot leave until all visitors inside the museum leave.

When a tour guide leaves, the following message should be printed to the screen:

```
Tour guide %d leaves the museum at time %d
```

When a visitor leaves, the following message should be printed to the screen:

```
Visitor %d leaves the museum at time %d
```

Visitor Arrival Process

The visitor arrival process creates *m* visitor processes. The number of visitors, *m*, is read from a command-line argument (e.g., ./museumsim -m 10). Visitors arrive in bursts. When a visitor arrives, there is a *pv* (e.g., 70%) chance that another visitor is immediately arriving after her, but once no visitors arrive, there is a *dv* seconds (e.g., 20 seconds) delay before any new visitor arrives. The first visitor always arrives at time 0. The probability *pv* and the delay *dv* are to be read from the command-line (e.g., ./museumsim -pv 70 -dv 20).

Tour guide Arrival Process

The tour guide arrival process creates *k* tour guide processes. The number of tour guides, *k*, is read from a command-line argument (e.g., ./museumsim -k 10). Tour guides arrive in bursts. When a tour guide arrives, there is a *pg* (e.g., 30%) chance that another tour guide is immediately arriving after[2] her, but once no tour guides arrive, there is a *dg* second (e.g., 30 seconds) delay before any new tour guide arrives. The first tour guide always arrives at time 0. The probability *pg* and the delay *dg* are to be read from the command-line (e.g., ./museumsim -pg 30 -dg 30).

Requirements

To achieve process synchronization, your solution:

- must use condition variables and locks,
- must not use semaphores directly,
- must not use sleep() or nanosleep() for synchronization,
- must not use busy waiting,
- must be deadlock-free

Testing

Make sure to run various test cases against your solution; for instance, create k tour guides and m visitors (with various values of *k* and *m*, for example *k* > *m*, *m* > *k*, *k* == *m*), different values for the

[2] Recall that at most two tour guides can be in the museum at a time.

probabilities and delays, etc. Note that the exact order of the output messages can vary, within certain boundaries. These boundaries are checked by the autograder scripts.

Program and Output Specs

Create a program, `museumsim`, which runs the simulation. Your program should run as follows.

- Fork a process for visitor arrival and a process for tour guide arrival, each in turn forking visitor processes and tour guide processes, respectively, at the appropriate times as specified by the command-line arguments.
- To get an 80% chance of something, you can generate a random number modulo 100 and see if its value is less than 80. It's like flipping an unfair coin. You may refer to CS 449 materials for how to generate a random number.
- Have the following command-line arguments:
    - -m: number of visitors
    - -k: number of tour guides
    - -pv: probability of a visitor immediately following another visitor
    - -dv: delay in seconds when a visitor does not immediately follow another visitor
    - -sv: random seed for the visitor arrival process
    - -pg: probability of a tour guide immediately following another tour guide
    - -dg: delay in seconds when a tour guide does not immediately follow another tour guide
    - -sg: random seed for the tour guide arrival process
- Make sure that your **output** shows all the necessary events. You should sequentially number each visitor and tour guide starting from 0. Visitor numbers are independent of tour guide numbers. When the museum is empty, your program should display:

  ```
  The museum is now empty.
  ```

- Print out messages in the form:

```
The museum is now empty.

Tour guide %d arrives at time %d.

Visitor %d arrives at time %d.

Tour guide %d opens the museum for tours at time %d.
```

```
Visitor %d tours the museum at time %d.

Visitor %d leaves the museum at time %d.

Tour guide %d leaves the museum at time %d.
```

- The printed time is in **seconds since the start of the program**. You should use the function `gettimeofday` and use both the seconds and microseconds fields in `struct timeval` in your calculation of the number of elapsed seconds.

Shared Memory in our simulation

To declare our shared data and our locks and condition variables, what we need is for multiple processes to be able to share the same memory region. We can ask for N bytes of RAM from the OS directly by using the mmap() system call:

```
void *ptr = mmap(NULL, N, PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANONYMOUS, 0, 0);
```

The return value will be an address to the start of this page in RAM. We can then steal portions of that page to hold our variables much like the malloc() project from CS 0449. For example, if we wanted one lock and two condition variables to be stored in the shared memory page, we could do the following:

```
struct cs1550_lock *lock;
struct cs1550_condition *first_cond;
struct cs1550_condition *second_cond;
lock = (struct cs1550_lock *) ptr;
first_cond = (struct cs1550_condition *) (lock + 1);
second_cond = first_cond + 1;

cs1550_init_lock(lock, "TestKey1");
cs1550_init_condition(first_cond, lock, "TestKey2");
cs1550_init_condition(second_cond, lock, "TestKey3");
```

to allocate and initialize them. What should you use for the shared memory size N? Please check the supplied sample program at /u/OSLab/original/test.c.

To allocate and initialize two shared integers:

```
void *ptr = mmap(NULL, 2*sizeof(int), PROT_READ|PROT_WRITE, MAP_SHARED |
MAP_ANONYMOUS, 0, 0);

int *first, *second;
first = (int *) ptr;
second = first + 1;
*first = *second = 0;
```

At this point we have one process and some RAM that contains our variables. But we now need to share that memory region/variables with other processes. The good news is that a mmap'ed region (with the MAP_SHARED flag) remains accessible in the child process after a fork(). Therefore, do the mmap() in main before fork() and then use the variables in the appropriate way afterwards.

Building and Running museumsim

To build and run your program, please follow the following instructions:

Make sure that the following files are in the same folder:

- museumsim.c (your solution)
- condvar.h and condvar.c (lock and condition variable implementation)

Assuming that your linux kernel from Project 1 is at `/u/OSLab/USERNAME/linux-2.6.23.1,` to compile museumsim.c, run the following command inside the above folder on thoth:

```
gcc -g -m32 -o museumsim -I /u/OSLab/USERNAME/linux-2.6.23.1/include/
museumsim.c condvar.c
```

To run `museumsim`, you will have to use the QEMU virtual machine provided in Project 1. Make sure that you boot the virtual machine into the kernel that has semaphore implementation. This can be either your own modified kernel from Project 1 or the kernel supplied to you. Copy the `museumsim` file from thoth into the QEMU virtual machine. Then run `./museumsim` with the command-line arguments. You don't need to reboot the QEMU virtual machine if you had to recompile `museumsim`. Just copy the modified `museumsim` file into the QEMU virtual machine and run it right away.

Debugging

You can either use the debugging steps in Project 1 (with a special focus on the hints for debugging user programs) or use the following steps, which will allow you to run gdb inside the QEMU virtual machine.

Inside the QEMU VM:

```
scp PITT_ID@thoth.cs.pitt.edu:/u/OSLab/original/gdb/* .
mv gdb /bin/
mv libncurses.so.5 /lib
mv libtinfo.so.5 /lib

gdb <program name>
```

On thoth:
You will have to add the -g flag to the compilation command (typed on thoth) for the museumsim program.

To run valgrind inside the QEMU VM:

On QEMU:

```
scp -r PITT_ID@thoth.cs.pitt.edu:/u/OSLab/original/valgrind/* .
cd valgrind
mv bin/* /bin/
mv lib/* /lib/
echo export VALGRIND_LIB=/lib/valgrind >> ~/.bashrc
bash
valgrind ./<program name with command-line arguments>
```

Submission

You need to submit the following to Gradescope by the deadline:

- Your well-commented `museumsim.c` program's source,
- a brief, intuitive explanation of why (or why not) your solution is fair (maximum 10 visitors per tour guide), as well as deadlock- and starvation-free.

Grading Sheet/Rubrics

Item	Grade
Test cases on the autograder	80%
Comments and style	10%
Explanation report	10%

The following penalty points (among others) will be deducted using manual grading.

Penalty points

Item	Grade
Use of semaphores directly	-50%
Use of Busy waiting	-25%
No protection of shared variables access using a lock	-20%
Tenant and guide arrival delay not taken into consideration	-5%
First guide/visitor doesn't arrive immediately	-2%
Incorrect seeding of random numbers	-2%
Incorrect random number generation (generated 101 numbers (0-100))	-1%
Too many calls to mmap causing too much syscall overhead	-1%

Please note that the score that you get from the autograder is not your final score. We still do manual grading. We may discover bugs and mistakes that were not caught by the test scripts and take penalty points off. Please use the

autograder only as a tool to get immediate feedback and discover bugs in your program. Please note that certain bugs (e.g., deadlocks) in your program may or may not manifest when the test cases are run on your program. It may be the case that the same exact code fails in some tests then the same code passes the same tests when resubmitted. The fact that a test once fails should make you try to debug the issue not to resubmit and hope that the situation that caused the bug won't happen the next time around.