University of Pittsburgh

CS/COE 1550 – Introduction to Operating Systems

Project 2: Process Synchronization

Due: Monday, October 19, 2020 @11:59pm
Late: Wednesday, October 21, 2020 @11:59pm with 10% reduction per late day

Table of Contents

Project Overview

Imagine that the ***CS1550 museum*** has been established to showcase old operating systems from all over the history of computing. Imagine also that a number of CS1550 students have volunteered to work as *tour guides* to answer the questions of the museum *visitors* and help them run the showcased operating systems. The museum is hosted in a room that can hold up to 20 visitors at a time (with COVID-19 physical-distancing restrictions in place). In order not to overwhelm the volunteer tour guides (who already have lots of work to do in the CS1550 class), each volunteer guide is limited to supervise at most ten visitors each day and is free to leave afterwards. However, being such nice students, if there are more than one guide inside the museum, they all wait for each other to leave together. To provide an adequate quality of service to the visitors, it should never be the case that the number of visitors inside the museum exceed ten times (10x) the number of tour guides who are inside the museum. The museum has an associated (physical or virtual) ticket booth. Each morning, a number of tickets is made available based on the number of tour guides that volunteer to show up on that day. The number of tickets available each day is **ten times** the number of tour guides on that day. Visitors claim the tickets in no particular order. Nevertheless, once there are no more tickets, no more visitors are admitted. In the CS1550 museum, it is possible that tour guides *wait* for visitors and visitors *wait* for tour guides. More details on the waiting conditions will follow shortly.

The CS1550 museum involves multiple independently acting entities (namely, the visitors and the guides). We will use our **process synchronization** skills to simulate the operation of the CS1550 museum. In particular, you will use semaphores (based on the semaphore implementation in Project 1) to simulate the **safe museum tour problem**, whereby visitors and guides are modeled as processes that need to synchronize in a such way that the following constraints are met.

- a visitor cannot tour the museum without a ticket; if there are no more tickets, the visitor leaves,
- a visitor cannot tour the museum without a guide,
- a guide cannot open the museum without at least one visitor,
- a guide cannot leave until all visitors inside the museum leave,
- guides in the museum leave together,
- at most **two** guides can be in the museum at a time, and
- each guide serves **at most ten** visitors.

To enforce the above conditions, you will need to use shared variables (e.g., the number of remaining tickets) among the visitor and guide processes. 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 deadlocks, the processes end up waiting indefinitely on each other. In order to avoid race conditions, we have discussed various mechanisms to ensure that critical regions are guarded.

Your job is to write a multi-process program that (a) always satisfies the above constraints, (b) under no conditions causes a deadlock to occur, and (c) under no condition causes a race condition to occur. A deadlock happens, for example, when a guide and a visitor arrive to an empty museum, and they stay waiting outside forever.

The semaphore that we will use in this project are based on the semaphore syscalls that you added to the Linux kernel 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
- /u/OSLab/original/vmlinux (needed only for debugging using gdb)

Project Details

You are to write C code for

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

`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** (i.e., wait) under the following conditions.

An arriving visitor must wait if

- the museum is closed or
- the museum capacity (10x the number of guides inside the museum) has been reached.

Note that an arriving visitor leaves immediately if there are no tickets available.

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.

When a tour guide arrives, the following message must be printed to the screen before any (potential) waiting:

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

When a visitor arrives, the following message must be printed to the screen before any (potential) waiting:

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

`tourMuseum`() and `openMuseum`()

After `tourguideArrives`() returns, the 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 any other visitor from touring the museum. Each visitor's tour is independent of any ongoing tours, that is, each visitor spends two seconds starting from the time the visitor enters the museum.

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 starts touring the museum, the following message is printed to the screen:

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

`visitorLeaves`() and `tourguideLeaves`()

Visitors leave once they are done touring the museum[1]. However, the leave process for guides is more involved.

(a) Tour guides that are inside the museum cannot leave until all visitors inside the museum leave.
(b) A guide cannot leave until they serve ten visitors (**the only exception is when there are no remaining tickets, in which case the guide can leave without serving ten visitors**).
(c) Guides that happen to be inside the museum must wait for each other and leave together.
(d) Meeting all previous conditions, guides must leave as soon as they can to get back to work on the CS1550 projects and labs.

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
```

[1] Recall that it is possible that a visitor leaves without touring (this happens when no tickets are available).

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[2] that another visitor is immediately arriving after her. With a (*100-pv)* (e.g., 30%) chance, there is a *dv* seconds (e.g., 20 seconds) delay before the next 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). **The initial number of tickets is *10*k***. 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 her. With a (*100-pg)* (e.g., 70%) chance, there is a *dg* seconds (e.g., 20 seconds) delay before the next 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 the semaphore syscalls of Project 1 (please check the test programs from Project 1 for examples of how to use the semaphores),
- must not use sleep() or nanosleep() for synchronization[3],
- 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 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

[2] 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 seed the random number generator and how to generate a random number.
[3] You can use these functions to simulate the museum touring duration only.

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

- It should start by forking a process for visitor arrival and a process for tour guide arrival, each in turn should fork visitor processes and tour guide processes, respectively, at the appropriate times as specified by the command-line arguments.
- It should accept the following command-line arguments in the following order:
    - -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 messages listed above.
- 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 following exact wording and punctuation without extra new lines:

```
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 may use the function `gettimeofday` and use both the seconds and microseconds fields in the `struct timeval` in your calculation of the number of elapsed seconds.
- Sample output and error cases can be found on the project page on Canvas.

Shared Memory in the simulation

To use shared data, what we need is for multiple processes to be able to share the same memory region. We can ask for N bytes of shared memory 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. We can then steal portions of that page to hold our variables. 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;
	(*first)++;[4]
```

At this point we have one process and some shared memory 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:

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 folder that contains museumsim.c on thoth:

```
gcc -g -m32 -o museumsim -I /u/OSLab/`whoami`/linux-2.6.23.1/include/ museumsim.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 the semaphore implementation. This can be either your own modified kernel from Project 1 or the kernel supplied by us. `scp` 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 have 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.

[4] A common mistake is to increment a variable using *first++; why?

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>
```

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 Instructions

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 sleep/nanosleep for synchronization	-25%
Use of Busy waiting	-25%
No protection of shared variables access	-20%
Visitor and guide arrival delay not taken into consideration	-5%
First guide/visitor doesn't arrive at time 0	-2%
Incorrect seeding of random numbers	-2%
Incorrect random number generation (e.g., 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.