CS/COE 1550 – Introduction to Operating Systems

Project 2: Process Synchronization

Due: Monday, March 4, 2019 @11:59pm
Late: Wednesday, March 6, 2019 @11:59pm with 10% reduction per late day

Table of Contents

Project Overview

Anytime we share data between two or more processes or threads, we run the risk of having a race condition where our data could become corrupted. In order to avoid these situations, we have discussed various mechanisms to ensure that one program'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 semaphore implementation that you finished in Project 1[1] to model the **safe apartment inspection problem**, whereby (potential) tenants and real-estate agents synchronize so that:

- a tenant cannot view an apartment without an agent,
- an agent cannot open the apartment without a tenant,
- an agent leaves when no more tenants are in the apartment,
- an agent cannot leave until all tenants in the apartment leave,
- once an agent opens the apartment, she can show the apartment to at most ten tenants, and
- at most one agent can open the apartment at a time.

Your job is to write a program that (a) always satisfies the above constraints and (b) where under no conditions will a deadlock occur. A deadlock happens, for example, when the apartment is empty, an agent and a tenant arrive, but cannot enter.

Project Details

You are to write (a) the tenant process, (b) the agent process, and (c) **six** functions called by these processes: (c1) tenantArrives(), (c2) viewApt(), (c3) tenantLeaves(), (c4) agentArrives(), (c5) openApt(), and (c6) agentLeaves(). You will also write two processes, (d) one for simulating agent arrival and (e) the other for simulating tenant arrival.

tenantArrives() and agentArrives()

In order to open an apartment for inspection, you need at least two people to be present simultaneously, the agent and the tenant(s) (each is represented by a process). tenantArrives() is called by a tenant process that wishes to view the apartment, and agentArrives() is called by an agent process that wishes to show the apartment. The functions must block until an agent and a tenant are both present.

When an agent arrives, the following message is printed to the screen:

```
Agent %d arrives at time %d.
```

[1] If you haven't finished Project 1, please let us know to provide a modified kernel for you to run your code on.

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

```
Tenant %d arrives at time %d.
```

viewApt() and openApt()

After an agent and at least one tenant are present simultaneously, the agent calls openApt() and the tenant calls viewApt(). Each tenant takes 2 seconds (by calling nanosleep or sleep) to view the apartment. When an agent opens the apartment, the following message is printed to the screen:

Commented [DM1]: Busy wait or spinning?

```
Agent %d opens the apartment for inspection at time %d.
```

When a tenant views the apartment, the following message is printed to the screen:

```
Tenant %d inspects the apartment at time %d.
```

tenantLeaves() and agentLeaves()

While the agent is still there, other tenants may arrive (i.e., call tenantArrives()), and as long as less than ten tenants have viewed the apartment while the agent is there, the tenants should immediately call viewApt() --- that is, multiple tenants can view the apartment concurrently.

If another agent arrives, s/he must wait until the tenants and agent currently in the apartment leave. Then s/he should wait until more tenants arrive.

When an agent leaves, the following message should be printed to the screen:

```
Agent %d leaves the apartment at time %d
```

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

```
Tenant %d leaves the apartment at time %d
```

Tenant Arrival Process

The tenant arrival process creates *m* tenant processes. The number of tenants, *m*, is read from a command-line argument (e.g., ./aptsim -m 10). Tenants arrive in bursts. When a tenant arrives, there is a *pt* (e.g., 70%) chance another tenant is immediately arriving after her, but once no tenant comes, there is a *dt* second delay before any new tenant will come. The probability *pt* and the delay *dt* are to be read from the command-line (e.g., ./aptsim -pt 70 -dt 20).

Agent Arrival Process

The agent arrival process creates *k* agent processes. The number of agents, *k*, is read from a command-line argument (e.g., ./aptsim -k 10). Agents arrive in bursts. When an agent arrives, there is a *pa* (e.g., 30%) chance another agent is immediately arriving after[2] her, but once no agent comes, there is a *da* second delay before any new agent will come. The probability *pa* and the delay *da* are to be read from the command-line (e.g., ./aptsim -pa 30 -da 30).

Requirements

Your solution must use binary semaphores, should not use busy waiting, and should be deadlock-free.

Testing

Make sure to run various test cases against your solutions to these problems; for instance, create k agents and m tenants (with various values of *k* and *m*, for example *k > m, m > k, k = m*, etc.), different values for the probabilities and delays, etc. Note that the output can vary, within certain boundaries.

Program and Output Specs

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

- Create a process for tenant arrival and a process for agent arrival, each creating tenants and agents, respectively, at the appropriate times.
- Create process for each tenant and agent.
- To get an 80% chance of something, you can generate a random number modulo 10, and see if its value is less than 8. It's like flipping an unfair coin. You may refer to CS 449 materials for how to generate a random number.
- Use the syscall nanosleep() or sleep() to pause your processes when needed (e.g., when the tenant is viewing the apartment for two seconds). **Extra credit: 2 points** if you're curious enough to come up with a loop that does some computations for about 2 seconds (without suspending the process) and compare your results with the process that uses the sleep or nanosleep syscall.
- Have the following command-line arguments:
    - -m: number of tenants
    - -k: number of agents
    - -pt: probability of a tenant immediately following another tenant

Commented [KS2]: I can go either way. It may be hard to come up with work that takes exactly 2 seconds on different machines.

[2] Subsequent agents will have to wait, as at most one agent can be in the apartment at a time.

- -dt: delay in seconds when a tenant does not immediately follow another tenant
- -pa: probability of an agent immediately following another agent
- -da: delay in seconds when an agent does not immediately follow another agent

Make sure that your output shows all of the necessary events. You can sequentially number each tenant and agent. Tenant numbers are independent of agent numbers. Display when the apartment is empty.

Print out messages in the form:

```
The apartment is now empty.

Agent %d arrives at time %d.

Tenant %d arrives at time %d.

Agent %d opens the apartment for inspection at time %d.

Tenant %d inspects the apartment at time %d.

Tenant %d arrives at time %d.

Tenant %d inspects the apartment at time %d.

Tenant %d leaves the apartment at time %d.

Tenant %d leaves the apartment at time %d.

Agent %d leaves the apartment at time %d
```

Shared Memory in our simulation

To make our shared data and our semaphores, 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 mmap():

```
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 449. For example, if we wanted two integers to be stored in the page, we could do the following:

```
        int *first_sem;
        int *second_sem;
        first = ptr;
        second_sem = first_sem + 1;
        *first_sem = 0;
```

```
	*second_sem = 0;
```

to allocate them and initialize them.

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 aptsim

Please use the instructions in Project 1 for building and running the test programs (named trafficsim in Project 1) to build and run aptsim.

Submission

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

- Your well-commented `aptsim.c` program's source
- The three, also well-commented, files that you modified in Project 1 from the kernel
- A brief, intuitive explanation of why your solution is fair (maximum 10 tenants per agent), as well as deadlock and starvation free.

Grading Sheet/Rubric

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