CENG 334 - Introduction to Operating Systems - Spring 2009

Synchronization

due date: 5 May 2009

1 Objectives

The aim of this assignment is to help you exercise with the basics of synchronization, using mutexes and semaphores. You will be implementing a number of everyday-life scenarios in an anonymous company.

2 Tasks

In the company, the computers of the employees are not connected to the Internet, lest they surf around and fritter away their working time. However, the company management realizes that the Internet will be necessary some time or other for the work, so there is a single room with three computers connected to the Internet in it. In a delicate balance where diplomacy is everything, you will be taking some precautions to guarantee the safety of the people using this room. You will model each "person" as a thread.

2.1 Task 1: Mutual Exclusion of Hierarchical Groups

There are two groups of employees in the company hierarchy: The managers and the workers. The managers, if not very busy, like reading some newspapers on the Internet, as well as doing some shopping. However, they prefer that there are no workers in the Internet room meanwhile, or there can fire up some unpleasant rumors around. Likewise, the workers, tired from trying to work all day, like to chat and read emails from time to time. Not surprisingly, they are also inclined to do these if there are no managers in the Internet room.

As your initial task, prepare a scenario where there can be $x <= 3$ managers or $y <= 3$ workers in the Internet room. There cannot be both managers and workers inside at the same time. If a manager comes to the room, and sees that the room is empty, he goes in directly. If the room is not empty, but there are only other managers inside, and moreover, there is an available computer, again he goes in. If the room is full, or if there are one or more workers inside, he waits outside until the conditions are available for him to go in. The same scenario goes for a worker, too.

2.2 Task 2: Mutual Exclusion with Everyone Happy

In the scenario above, there is a risk that one group will "starve to death" by not being able to surf all day. Consider that when a manager comes, there is a worker in the room already. The manager cannot go in, so he has to wait until the worker gets out. However, just before the worker gets out, another worker comes, and since this worker can safely go in, he does so immediately. Meanwhile, the first

worker goes out, but the manager cannot still get in, since there is still one worker inside. If it happens so that the workers keep coming, it is quite possible that the manager will wait all day long outside, and in the evening, become so angry that he might go tell the bosses to do something about the excessive abuse of the Internet room by those workers. The same thing will also happen if a worker waits all day long outside while managers keep coming. Although a worker will be less likely to complain to the bosses about the managers, we would not let them suffer just because of that.

Therefore, we must also implement a version of the above scenario, which prevents starvation of both managers and workers. It should be so that, if there are some workers in the room already, and a manager is waiting outside, the newcomer workers should not be allowed to the room, but should wait outside until first the manager goes in, and finally comes out. The opposite of this scenario should also be satisfied for a waiting worker.

(Note: In Task 1, do NOT try to eliminate starvation! Starvation will only be eliminated in this Task 2 scenario.)

2.3 Task 3: Three for a Quake

When there are no managers around, the workers can also decide to go for a quick Quake party. Assume that this scenario takes place on Saturdays and Sundays, and since the managers will not be working on these days, you need not deal with mutual exclusion of worker/manager groups. However, there is still some things to be careful about. The problem is that, there are three groups of workers which do not like each other very much: the engineers, the designers, and the marketeers. If a member of these three groups witnesses members of the other two groups playing Quake, but no one from his own group involved in the play, then he will surely blabber to the bosses about this. Therefore, to be on the safe side, there can be only four cases in which Quake can be played: (1) there are 3 engineers in the room, (2) there are 3 designers in the room, (3) there are 3 marketeers in the room, (4) there is one designer, one engineer, and one marketeer in the room.

Assume that in the weekends, the only solace of workers' lives is to play Quake. Therefore, until the necessary conditions are met, newcomers wait outside the room. If one of the above scenarios happens, for instance, there are 3 designers outside the room, they go inside. Once they are in, one of them signals everyone to start playing. When the game is over, they go outside.

3 Input/Output Specifications

1. The codes must be in C. No C++ codes are accepted.

2. Your programs will be compiled with gcc and run on the department Inek machines. No other platforms/gcc versions etc. will be accepted so check that your code works on Ineks before submitting it.

3. For the three tasks, your program will be run with one of the command line arguments, -t1, -t2, or -t3, specifying which scenario to run. In case of tasks 1 and 2, the program will also take as input the number of managers and the number of workers, as follows:

   ```
   ./sync -t 1 -m 2 -w 3
   ./sync -t 2 -m 5 -w 10
   ```

   If the task is the 3rd, then the program will take as input the number of engineers, designers and marketeers.

   ```
   ./sync -t 3 -e 5 -d 10 -m 15
   ```

The order of the command line arguments will be fixed as a courtesy ;)

4. The first thing your program should do is to create the necessary number of threads for each group of employees.

5. Each thread will read its own actions from an input file. The first thing a thread must do is open its own action file. The files will be named `manager_0`, `manager_1`, ..., `worker_0`, `worker_1`, ..., `engineer_0`, `engineer_1`, ..., `designer_0`, `designer_1`, ..., `marketeer_0`, `marketeer_1`, ... Take care of: (1) 0-based indexing, (2) that there is no ".txt" at the end of the file names.

6. The format of the files is as follows: Each file consists of actions to be done at each time step. At a time step, a thread may do nothing, request to go in the room, stay in the room (or play Quake, in the case of 3rd task), and leave the room. Here is a sample file, say `manager_0`'s action file:

```
-
-
-
i
s
s
s
s
o
-
-
-
-
-
i
s
o
-
```

   The files do not have an extra newline at the end.

   Here is the explanation: In the first 3 time steps, manager 0 does not want to do anything. At time step 4, it requests to go in. Note that, depending on the situation, it may or may not go in at this time step. If it can go in, it will. Otherwise, it will block until it is awakened by another thread. Once it is allowed, it will go in. It will stay inside for the next 4 time steps. Note that, if it is not allowed to go in until 20th time step, then it will stay inside at time steps 20 to 24. At the end of these 4 time steps, (when it executes "o") it will go outside. "Going outside" means awakening another thread who has blocked for going inside, if there is such a thread. Then it will do nothing for 5 time steps, and at the 6th, it will request to go in once more. It will stay inside for 1 time step, and go out again. It will do nothing at the last time step. It will exit after that.

7. A thread will exit when it has executed all the actions (or non-actions, in case of "-") in its action file. The program will exit when all the threads have exited.

8. If a thread has previously blocked, and another thread awakens it at a time step t, then the blocked thread must wake up at that time step t (that means immediately), and go in the room. We will count that thread in the room at that time step t also.

9. In the case of 3rd task, the action list will not include "s", but "q", meaning the thread wants to play Quake. You can assume a quake play takes a constant time, so all the threads in a specific

run will have the number of “q”s between “i” and “o”. Here is a sample file, say `engineer_0`’s action file:

```
-
-
-
i
q
q
q
q
q
o
-
-
-
```

10. In the case of the 3rd task, as mentioned above, once all of the players are in the room, one of them must give a notification to everybody else to start playing.

11. As mentioned, the actions will be executed in a time-step manner. The main thread will take care of timesteps. There will be time steps of 1 second. After the initialization of the employee threads, the main thread will initiate the cycle. At every second, it will “notify” the employee threads that a new cycle has begun. The threads will then take the next action in their action list. The possibilities are, a thread may do nothing in a time step, request to enter the room, stay inside/play Quake, or go outside the room. If it is going outside, it must also awaken a blocked thread if there is one. This awakening must be done at the same time step that it is going out. The awakened thread will be counted as inside the room beginning from the same time step, also.

12. The main thread will hold a data structure denoting the room. Once inside the room, a thread must write its id and type (manager, worker, etc.) in this data structure. When going outside, it must delete them. At the end of each time step (or possible just before starting the new time step by notifying the employee threads), the main thread must write the contents of this data structure to the standard output. This printing will be done in a strict manner. NO MATTER the order that information has been written to the data structure, the main thread will print FIRST the managers in the room, and THEN the workers in the room. In case of 3rd task, the printing order of the workers will be (1) engineers, (2) designers, (3) marketeers. The groups will also be printed IN ORDER themselves, i.e, manager i before manager i+k. Before printing, a line indicating the time step will be printed. If there is no one in the room at a time step, only the time information will be printed. Sample outputs might be:

```
time 0
time 1
time 2
manager 0
manager 1
manager 2
time 3
manager 0
manager 2
time 4
manager 2
```

```
time 5
worker 0
worker 1
worker 2
time 6
worker 0
worker 1
worker 2
time 7
```

or

```
time 0
time 1
engineer 0
engineer 1
engineer 2
time 2
engineer 3
designer 0
marketeer 0
time 3
```

Take care time steps begin from 0.

4 Tips

You are expected to use the pthread library (link with -lpthread). It offers a good API for creating and using threads, as well as mutexes and condition variables. However, you will need also semaphores for handling count limits. The Semaphore implementation, as well as wrappers for pthread library's mutex and condition variable implementations, from Allen B. Downey's The Little Book of Semaphores, will be very helpful. The code (copied rigorously) including the associated functions is also available on COW.

You can also find many other useful implementations of handy structures, as well as similar problems, in this book, so it is highly suggested that you would take a look at it. It is free for download on http://www.greenteapress.com/semaphores/

5 Submission

You will submit a single tar file (sync.tar or hw.tar, name not important) including a Makefile and your source file(s). The Makefile should create an executable called sync (yes, this name is important). The tar file should not contain any directories! The following command sequence (with the exception of the tar file name) is expected to run your program:

```
$ tar -xvf sync.tar
$ make
$ ./sync -t 1 -m 2 -w 3
```

Note that this is not possible if the tar file extracts into a directory ;)

May it be easy.
–Hande