CS 3500 – Operating Systems; July 2018 semester
Krishna Sivalingam
Lab 7: Threads Synchronization using Mutex and Semaphores
Due Date for Problem 1 (DP): Monday, **Sep. 24, 4.45PM**; via Moodle
Due Date for Problem 2 (Pizza): Sunday, **Sep. 30, 11.00PM**; via Moodle
NO LATE SUBMISSIONS

The objective of this project is to implement solutions to the following synchronization problem(s). The implementation will use multiple threads using the *Linux pthreads API*. The implementation will be done in Linux/C/C++ platform. An example use of pthread_mutex is available in the class Moodle site.

The first problem is worth 30% of Lab grade; the second problem is worth 70%.

1 Dining Philosophers Problem

Implement the solution with potential deadlock as described in the textbook.

Then, implement a deadlock-free solution to the dining philosopher's problem, using the following technique: Allow a philosopher to pick up her forks only if both forks are available simultaneously.

The command line is: `./dpp -N <integer>`

There are N philosophers (also, plates and forks). Each philosopher is represented using a thread (all threads created from a single parent process). Each philosopher sleeps for a random time between 50 and 100 milliseconds; and after getting both forks, eats for a random time between 100 and 200 milliseconds. Each philosopher thread will terminate after running for 100 rounds of thinking-eating cycles. The initial state is *thinking* for each philosopher.

The program will output to a file called `dp_out_thread_1_DL.txt` (and so on for other philosophers' threads), the following events along with the current time: round number, a philosopher entering thinking state, a philosopher entering hungry state, a philosopher entering eating state and a philosopher entering the leaving state.

If the process stops printing output after sometime due to deadlock, then you have to manually kill the process.

For the deadlock-free solution, the files will be named `dp_out_thread_1_DLF.txt` etc.

2 Pizzas and Students

Consider a pizza shop with the following specifications.

Pizza Makers:

- ▷ There are $M \geq 5$ pizza makers, who will make pizzas and place them on pizza shelves.
- ▷ There are a set of S shelves, each with capacity to hold up to D pizzas.
- ▷ The pizza maker is in two states: (i) making pizza batch and (ii) napping.

- ⋆ The pizza maker naps for a time Z milliseconds; and then
- ⋆ enters "making pizza batch" state, where he/she makes $B \geq 10$ pizzas per batch. It takes C milliseconds to make a pizza. After successfuly adding the B pizzas to one or more shelves, the maker goes back to napping state.

▷ Every pizza made by a maker will have an identifier that is unique to the pizza maker given by (Maker ID, Batch ID, Pizza ID). The Pizza ID gets set to zero for each batch.

▷ The initial state of each pizza maker is selected randomly (with equal probability).

System Constraints:

▷ Two pizza makers cannot add to the same shelf at the same time. When a pizza maker selects a shelf, but finds that it is in use by another maker, he/she waits until the shelf becomes free.

▷ When the first pizza of a maker's batch is ready, he/she will add it to a randomly selected pizza shelf.

It will continue to add to this shelf until either 10 pizzas have been added to this shelf in succession or the shelf is full. At this time, the pizza maker selects a different (random) shelf to add pizzas to, which might be the same as the last one.

▷ If there is no empty shelf when a pizza-maker has made a pizza, s/he waits until a shelf becomes available. It does not make a pizza while so waiting.

Hungry Students:

▷ There are $U \geq 7$ hungry undergraduate students who will remove and eat the pizzas (whether they pay for it or not is our concern) from the shelves.

▷ It takes E milliseconds for a student to eat a pizza.

▷ The student will be in two states: hungry and studying.

- ⋆ After entering "hungry" state, a student be in this state until he/she has consumed a certain number of pizzas (H).
- ⋆ Once a student's hunger is satisfied, they will spend R ms in "studying" state, before becoming hungry again.
- ⋆ Note: Students do not have "sleeping" state.

▷ The initial student state will be: "hungry", for all students.

▷ Every pizza eaten by a student will have an identifier that is unique to the student given by (Student ID, Pizza ID). The Pizza ID gets set to zero for each student at the start of the program.

Some constraints:

▷ A student can take a pizza from any available shelf. He/she can choose a shelf at random and if there is no pizza in that shelf, moves to check from two more randomly selected shelves. If there is no pizza in any of these three shelves, then the student waits till a pizza is available in the last selected shelf.

- ▷ Two students cannot take a pizza from the same shelf at the same time.
- ▷ A student can take a pizza from a shelf at the same time that a maker is adding a pizza to that shelf, providing that there is at least one pizza on the shelf when the student looked at it.

What to Do: Your task is to implement the synchronization mechanisms for this problem using **semaphores** and/or **locks/mutexes**, using threads. Each student and maker will be represented by a thread.

You will also implement driver code that will take as input as follows:

```
% ./dm -m M -s S -d D -z Z -c C -b B -u U -e E -h H  -r R -n N
```

Print an error message if the input values of variables do not meet the minimum constraints above and then exit.

The driver code will generate a random number of maker threads (uniform random between 5 and M), and a random number of student threads (uniform random between 7 and U). The maker IDs and student IDs will start from 1.

What to Output: There will be one output file per process, with output filename names: **out_maker_1.txt**, etc. for makers and **out_student_1.txt** for students. Any other output messages can go to the screen or to a separate file **out_other.txt**.

Each process will print its type (maker or student) and respective ID initially. It will also output a message when the following events take place: a pizza is produced, a pizza is put on a shelf, waiting for a shelf to be free, a pizza is removed and eaten, or a student is waiting for a pizza to be produced. Each event message will contain the corresponding pizza ID, if appropriate for that event.

2.1 Sample Session

Assume that you have created the necessary files and the corresponding executables in your LAB7 directory.

```
% cd LAB7
% make pzm
% ./pzm  -m 5 -s 4 -d 15 -z 25 -c 10 -B 12 -u 7 -e 5 -h 10 -r 30 -n 10000
.. Output goes to various files (and screen if so designed)
..
% cat out_maker_1.txt
..
% cat out_student_2.txt
..
```

3 What to Submit

Submit the following files, as a single tar-gzipped file (one on Sep. 24 and one on Sep. 30):

- ▷ Source File(s)
- ▷ Makefile: Typing command 'make' at the Linux command prompt **MUST** generate all the required executables.
- ▷ A Script file obtained by running UNIX command *script* which will record the way you have finally tested your program. The script file will contain at least TWO runs for each synchronization problem.
- ▷ a README file for the TA. The README should document known error cases and weaknesses with the program. You should also document if any code used in your submission has been obtained/modified from any other source, including those found on the web.
- ▷ a COMMENTS file which describes your experience with the project, suggestions for change, and anything else you may wish to say regarding this project. This is your opportunity for feedback, and will be very helpful.

4 Miscellaneous

1. WARNING ABOUT ACADEMIC DISHONESTY: Do not share or discuss your work with anyone else. The work YOU submit SHOULD be the result of YOUR efforts. The academic conduct code violation policy and penalties, as discussed in the class, will be applied.
2. You can use the functions *random* and *srandom* functions. Run `'man random'` for more information. The function *srandom* will be called once and initialized with the seed value (N), that will be obtained from the command line.
3. Note that you can simulate pizza-makers making pizzas and students eating pizzas by having them sleep using a suitable sleep function, such as the *nanosleep* call.
4. You can use *time, locatime, asctime, ctime* functions for printing the current time.

5 Extra Credit (35 Marks)

Implement a solution to the following synchronization problem (from William Stallings's Operating Systems book) and demonstrate that it works.

Stand Claus sleeps in his shop at the North Pole and can only be awakened by either (1) all nine reindeer being back from their vaca- tion in the South Pacific, or (2) some of the elves having difficulty making toys; to allow Santa to get some sleep, the elves can only wake him when three of them have problems.

When three elves are having their problems solved, any other elves wishing to visit Santa must wait for those elves to return.

If Santa wakes up to find three elves waiting at his shops door, along with the last reindeer having come back from the tropics, Santa has decided that the elves can wait until after Christmas, because it is more important to get his sleigh ready. (It is assumed that the reindeer do not want to leave the tropics, and therefore they stay there until the last possible moment.) The last reindeer to arrive must get Santa while the others wait in a warming hut before being harnessed to the sleigh.

Some additional specifications:

- ▷ After the ninth reindeer arrives, Santa must invoke PREPARESLEIGH, and then all nine reindeer must invoke GETHITCHED.
- ▷ After the third elf arrives, Santa must invoke HELPELVES. Concurrently, all three elves should invoke getHelp.
- ▷ All three elves must invoke GETHELP before any additional elves enter (increment the elf counter).
- ▷ Santa should run in a loop so he can help many sets of elves. We can assume that there are exactly 9 reindeer, but there may be any number of elves.