Cookies to Code – A simulation with pthreads

In this project, you will develop a simulation that uses pthreads. The simulation is a variation on the producer-consumer problem.

Cookies to Code Simulation

This simulation will involve two sets of beings: elves – known for their great cookie making abilities, and cs3204 students – known for their great code writing abilities. In this simulation, elves produce cookies that cs3204 student consume in order to produce lines of code. The code that the students produce is part of an open source project to create a new operating system and all the students contribute lines of code to the same codebase. As you might expect, there are some constraints on how the elves can produce cookies and how the cs3204 students can produce code.

Elves harvest grain using tractors, bake cookies using ovens, and deliver cookies using trucks. Students eat cookies to gain "cookie energy" and use cookie energy to write lines of code using computers.

Actions

Elves can do the following actions:

Harvest
An elf can harvest grain in the fields. Every time an elf harvests, 1 unit of grain is produced and added to the elves' grain supply. The fields are magic fields that can supply an infinite amount of grain. However, the elves only have NUM_TRACTORS tractors, so only NUM_TRACTORS elves can be harvesting at the same time.

Bake
An elf can bake cookies in an oven. Every time an elf bakes, 1 unit of grain is used from the elves' gain supply and 1 unit of cookies are produced and added to the elves' cookie supply. The elves have NUM_OVENS ovens, so only NUM_OVENS elves can be baking at the same time.

Deliver
An elf can deliver cookies to the cs3204 students. Every time an elf delivers, 1 unit of cookies are transferred from the elves' cookie supply to the cookie supply for all the students in McBryde Hall. The elves only have NUM_TRUCKS trucks, so only NUM_TRUCKS elves can be making deliveries at the same time.

Sleep
An elf can sleep for 1 second of actual wall clock time (i.e. `sleep(1);`).

If an elf is trying to do an action, but cannot get the resources needed, it will block (not busy-wait) until the resources are available.

Students can do the following actions:

Code
A student can write lines of code. Every time a student codes, 50 lines of code are added to the students' combined codebase and 1 unit of cookie energy is subtracted from the student who was coding. A student cannot start coding if they have less than 1 unit of cookie energy. The students only have NUM_COMPUTERS computers, so only NUM_COMPUTERS students can be writing code at the same time. If a student has zero cookie energy, they cannot code – they will skip coding and go to the next activity on their list.

Eat
A student can eat cookies. Every time a student eats, 1 unit of cookies is consumed from the cookie supply for all the students and is converted into 1 unit of cookie energy for that student. A student can

only store up to 9 units of cookie energy – after that they are full and will not eat any more cookies until they are below 9 units of cookie energy (in other words, if a student tries to eat when they have 9 units of cookie energy, they will skip eating and go on to the next activity on their list).

Sleep
A student can sleep for 1 second of actual wall clock time (i.e. `sleep(1);`).

Except as noted in the descriptions of `code` and `eat`, if a student is trying to do an action, but cannot get the resources needed, it will block (not busy-wait) until the resources are available.

How to implement the simulation

You must implement a simulation of the processes described above using the pthreads library. The simulation should be named `cookies2code` and should read a configuration file with the following information in the format shown:

```
NUM_ELVES = [0-9]
NUM_STUDENTS = [0-9]
NUM_TRACTORS = [0-9]
NUM_OVENS = [0-9]
NUM_TRUCKS = [0-9]
NUM_COMPUTERS = [0-9]
START_ELVES_GRAIN = [0-99]
START_ELVES_COOKIES = [0-99]
START_STUDENTS_COOKIES = [0-99]
ELF_0 = <elf_command_filename0>
ELF_1 = <elf_command_filename1>
...
STUDENT_0 = <student_command_filename0>
STUDENT_1 = <student_command_filename1>
...
```

The configuration file specifies a number of elves (NUM_ELVES) and a number of students (NUM_STUDENTS) for the simulation run. It specifies the number of tractors (NUM_TRACTORS), ovens (NUM_OVENS), trucks (NUM_TRUCKS) and computers (NUM_COMPUTERS), and also specifies the starting values for the number of units of grain in the elves' supply (START_ELVES_GRAIN), the number of cookies in the elves' supply (START_ELVES_COOKIES), and the number of cookies in the student's supply (START_STUDENTS_COOKIES). All students start each simulation with a cookie energy level of zero. The configuration file also specifies a file of commands for each elf and each student to follow in the simulation.

Each <elf_command_filenameN> should be a text file that consists of actions for an elf to perform. For example:

```
sleep
bake
harvest
bake
deliver
```

Each <student_command_filenameN> should be a text file that consists of actions for a student to perform, each followed by a length of time to perform that action. For example:

```
sleep
eat
code
```

```
code
eat
```

Your program should create a separate thread for each elf and for each student. The thread for each elf and each student should read commands from the <command_filename> given in the configuration file and perform them in the sequence they are listed in the command file.

Your program should implement separate functions called `harvest()`, `bake()`, `deliver()`, `eat()`, and `code()` that take appropriate arguments and modify appropriate data structures. Each elf and student should be identified by a single digit identifier [0-9]. You may find it useful to pass this identifier into the `harvest()`, `bake()`, `deliver()`, `eat()`, and `code()` functions.

You will need to decide how to implement the simulation so that it keeps track of the elves' grain supply, the elves' cookie supply, the students' cookie supply, the students' codebase, and the individual students' cookie energy. There are several methods you could use to keep track of these simulation values and to insure they are maintained properly. Whatever method you choose, you should be sure that variable changes are properly protected and that deadlock and starvation cannot occur. Note that it is possible to give sequences of actions that will lead to elves and students being blocked – this is okay. Preventing deadlock and starvation in your simulation does not mean that there will always be enough harvested grain for baking and cookies for eating.

You must implement your simulation using the pthread library as installed on the cslab computers in McBryde 124. From the pthread library, you should only use the following functions in the code you write:

```
pthread_create()
pthread_exit()
pthread_mutex_init()
pthread_mutex_lock()
pthread_mutex_unlock()
```

Not all pthread libraries include built-in support for semaphores. However, there is a library of semaphore code for pthreads as part of the document, "Getting Started With POSIX Threads", by Tom Wagner and Don Towsley, Department of Computer Science, University of Massachusetts at Amherst, July 19, 1995. This document is available on the web. You must use the semaphore library provided in Appendix A of this document to provide your semaphores. Be sure to include comments in your code with a citation about the name, author, and where you obtained the semaphore library code. You may find it helpful to make a small modification to the `semaphore_init()` routine in the library to allow a starting value to be specified for the semaphore when it is initialized.

You will also need to implement two mechanisms in your parent process: 1) one that insures that your parent process does not exit before the threads have completed their execution and 2) one that allows you to terminate the parent process (which will in turn terminate all the threads) by typing "quit" at a prompt. The reason for this is that your simulation may encounter a situation in which all the elves and/or students are blocked and the user should be able to stop the simulation at any time after it starts.

Output and Example

Each elf and student thread should print output about its actions and the current status of the simulation. Imagine a simulation with 2 elves and 1 student with commands as shown below:

Elf 1 commands: `deliver`
Elf 2 commands: `harvest, bake`
Student 1 commands: `eat, code`

Your output should look something like the output shown below.

```
Starting cookies2code simulation with:
   Elves = 2
   Students = 1
   Tractors = 2
   Ovens = 1
   Trucks = 1
Begin simulation
   Status:  Grain = 0, Ecookies = 0, Scookies = 0, Scode = 0
            S1ce = 0
Elf 1 trying to deliver
Elf 2 trying to harvest
Student 1 trying to eat
Elf 2 got a tractor
Elf 2 finished harvesting
   Status:  Grain = 1, Ecookies = 0, Scookies = 0, Scode = 0
            S1ce = 0
Elf 2 trying to bake
Elf 2 got an oven
Elf 2 finished baking
   Status:  Grain = 0, Ecookies = 1, Scookies = 0, Scode = 0
            S1ce = 0
Elf 1 found enough cookies to deliver
Elf 1 finished delivering
   Status:  Grain = 0, Ecookies = 0, Scookies = 1, Scode = 0
            S1ce = 0
Student 1 found enough cookies to eat
Student 1 finished eating
   Status:  Grain = 0, Ecookies = 0, Scookies = 0, Scode = 0
            S1ce = 1
Student 1 trying to code
Student 1 found enough computers to code
Student 1 finished coding
   Status:  Grain = 0, Ecookies = 0, Scookies = 0, Scode = 50
            S1ce = 0
```

You may modify the format of the output slightly, but make sure that your status messages indicate the status of the variables at the time the action was finished, not at the time the message is printed. This means that you may need to use temporary variables to store the simulation status in the protected sections of code. Any time that an elf or student starts trying an action an informative message should be printed. Any time that an elf or student finishes an action an informative message should be printed and the status of the simulation should be printed.

It is okay to use the semaphore library function `semaphore_value()` to output the value of a semaphore in the status messages if your implementation needs this. However, you should NOT use `semaphore_value()` in any type of decision making in the code – doing so would violate a fundamental principle of semaphore use.

Remember that things are happening "in parallel" – interpreting output from multiple threads can be confusing and can take some getting used to.

Documentation

In addition to writing good in-line documentation in your code, you must also include a file named "`readme`" that contains a brief description of the way you implemented the simulation, how you protected resources, and how you insured there would not be deadlock or starvation.

Test Cases

You must include with your implementation at least 3 sets of test cases that you used to satisfy yourself that your simulation works correctly. For each test case, you should include a configuration file and as many elf and student command files as are needed. For each test case, you should also include a `readme` file that explains the situations tested and why the output is correct. Name your test case files according to the following examples: "tc1-config", "tc1-elf1", "tc1-elf2", "tc1-student1", "tc2-config", etc.

Design and implementation requirements

There are some explicit requirements, in addition to those on the Programming Style page of the course website:

- Your simulation must be implemented in ISO-compliant C/C++ code. You may make use of any language-standard types and/or containers you find useful.
- You must decompose your implementation into separate source and header files, in some sensible manner that reflects the logical purpose of the various components of your design.
- You must document your implementation according to the *Programming Standards* page on the course website.
- You must properly allocate and de-allocate memory, as needed.

In general, you are expected to apply the design and implementation guidelines and skills covered in your previous computer science courses. You may implement your solution in pure C, or in C++.

Resources

You must use the library of semaphore code for pthreads in Appendix A of the document, "Getting Started With POSIX Threads", by Tom Wagner and Don Towsley, Department of Computer Science, University of Massachusetts at Amherst, July 19, 1995. This document is available on the web at:

```
http://dis.cs.umass.edu/~wagner/threads_html/tutorial.html
```

Note that to get the code in Appendix A to work with the LinuxThreads version of pthreads installed on the cslab machines, you will need to add the following `#define` statements:

```
#define pthread_attr_default NULL
#define pthread_mutexattr_default NULL
#define pthread_condattr_default NULL
```

There is a nice book available on-line that describes semaphores and techniques for using them. It is called "The Little Book of Semaphores" by Allen B. Downey. You may find this book helpful in understanding and using semaphores. It is available as a PDF file for free at the web site:

```
http://greenteapress.com/semaphores/
```

Evaluation

Remember that your implement will be tested in the McBryde 124 lab environment (GCC/Linux). No help will be given for any other development environments. It is your responsibility to make sure your implementation works correctly in the lab.

Your implementation will be evaluated for documentation and design, as well as for correctness of results. Note that the evaluation of your project may depend substantially on the quality of your code and documentation.

Your simulation should accept one argument, a configuration file:

```
bash> cookies2code <configuration-file>
```

We will run your simulation using a variety of configuration files to test its performance.

What to turn in and how

This assignment will be collected on the Curator system. The testing will be done under the McBryde 124 lab environment.

You may write your code in either C or C++ for this project.

Submit a single gzip'd tar file containing the C/C++ source and header files, the readme files, and the test case files for your implementation to the Curator system. Submit nothing else. Be sure that your header files only contain `include` directives for Standard C/C++ and UNIX header files; any other `include` directives will probably cause compilation errors. It must be possible to unpack the file you submit using the following command:

```
tar –zxf <name of your file>
```

The unpacked files will then be compiled, typically using the following command syntax:

```
[g++ | gcc] –o <name we give the executable> [*.cpp | *.c] -lpthread
```

The appropriate link for submitting to the Curator system will be posted on the course web site. You will be allowed to submit your project up to three times before the due date in order to fix errors you discover before your project is tested. Your latest submission will be the one that is tested.

Pledge

Your program submission must be pledged to conform to the Honor Code requirements for this course. Specifically, you must include the pledge statement provided on the course web site in one of your submitted files.