CMPUT 379 - Assignment #4 (10%)

Simulating Concurrent Jobs (first draft)

Due: Thursday, April 7, 2022, 09:00 PM
(electronic submission)

In this assignment, you are required to write a C/C++ program, called `a4w22`, that utilizes pthreads to simulate the concurrent execution of a set of jobs. The system has a number of resource types, and each resource type has a number of available units. All resource units in the system are **non-sharable** **non-preemptable** resources. The simulator reads the system parameters from an input file. The file has a number of lines formatted as follows:

- Lines starting with a `'#'` are comment lines. Empty lines are allowed.
- A line of the form

  ```
  resources  name1:value1  name2:value2  ···
  ```

  specifies the resource types available in the system. The line starts with the keyword `resources`, followed by one, or more, `name:value` pairs of a resource type name, and the number of available units of this resource type, respectively.
- A line of the form

  ```
  job  jobName  busyTime  idleTime  name1:value1  name2:value2  ···
  ```

  specifies a job in the system. The line has the following fields:

  - `job`: a keyword that specifies a job line
  - `jobName`: the job's name
  - `busyTime`: an integer specifying the real time (in millisec) spent by the job when executing
  - `idleTime`: an integer specifying the real time (in millisec) spent by the job after finishing execution and before it can be executed again
  - `name:value`: specifies the name of a resource type, and the number of units of this resource type needed for the job to execute

  Notes: A system may have up to `NRES_TYPES` = 10 resource types, and `NJOBS` = 25 jobs. Each string (a job or a resource type name) has at most 32 characters. Each white space between fields is composed of one, or more, space character(s). There is no white space around the `':'` field separator.

Example. The following data file corresponds to an instance of the Dining Philosophers Problem with 5 people, denoted `j1` to `j5`. The five chopsticks correspond to five resource types, denoted `A` to `E`. Each philosopher spends 50 millisec eating, followed by 100 millisec thinking, before getting hungry again.

```
# An instance of the Dining Philosophers Problem with 5 people
#
resources    A:1 B:1 C:1 D:1 E:1
job          j1 50 100   A:1 B:1
job          j2 50 100   B:1 C:1
job          j3 50 100   C:1 D:1
job          j4 50 100   D:1 E:1
job          j5 50 100   E:1 A:1
```

Use a single lock (either mutex lock or binary semaphore) for the entire resource pool.

The entire resource pool is represent by an unordered map (key: name, value: number of units

The resource pool is encapsulate inside ResourceManager class. It's private. (Need getter and setter method for outsider).

Before a thread start acquiring and release all needed resources, it first need to acquire a lock to access and use setter function to change resource pool.

After a thread get the lock and start accessing the resources the pool:
 if it can get all the needed resource, then make change in the resource pool, release the lock, go into busy time...
 if it cannot get all the needed resource, then release the lock, wait and retry again.

Lock placement decision: function call to get lock outside the getter and setter (in thread function) or inside getter and setter

No resource change during monitor print out something

Concern: will the value that keep track of the number of unit available in the resource pool

The Simulator

Critical section:

General idea : returning information from resource pool and making change in the resource pool.

Job thread : acquiring all the needed resource and releasing all the needed resource back to resource pool

Monitor : printing information about all job threads and the resource pool

The program is invoked using the command line

```
% a4w22  inputFile  monitorTime  NITER
```

where

- `inputFile`: is the input file describing the system, as detailed above,
- `monitorTime`: an integer (in millisec) that specifies how often a monitor thread (described below) runs, and
- `NITER`: the simulator finishes when each job in the system executes `NITER` times (a one time execution of a job is considered as one iteration of the job).

The program assigns one thread to simulate the execution of each job. Job threads run concurrently in an independent manner, except for synchronizing with each other when accessing shared resources. To simulate a single iteration of a job, the assigned thread must do the following

1. Acquire all resource units needed by the job
2. Spend (in real time) an interval of length `busyTime` millisec holding the needed resources
3. After this time interval, the thread releases all held resources. Next, the thread enters an idle period of length `idleTime` millisec.

After finishing one iteration, the thread is ready to proceed to another iteration.

> **Note:** Step (1) involves some waiting time until the thread acquires all resource units needed by a job. For simplicity, this step can be implemented using a loop where the thread tries to acquire the needed units. If failed, the thread tries again (perhaps after delaying a short period of time, say 10 millisec). You may choose to implement a different approach.

A thread simulating a job can be in one of the following states:

- **WAIT**: the thread is trying to acquire the needed resources.

- **RUN**: the thread has acquired the needed units, and is spending a (real time) interval of length `busyTime` millisec to simulate job execution.
- **IDLE**: the `busyTime` period has elapsed, and the thread is spending a (real time) interval of length `idleTime` millisec simulating the idle period.

Deadlock Prevention: For simplicity, the simulator uses a basic deadlock prevention scheme where each thread either holds all needed resources to execute its assigned job, or it does not hold any resource at all. You may choose to implement a different approach.

The Monitor Thread: In addition to running the main thread and job threads, the program runs a *monitoring* thread. This thread runs every `monitorTime` (specified on the command line) millisec. When run, the thread produces the output described below.

> **Note:** The implementation should not permit a job thread to change its state when the monitor is printing its output.

Termination: The main thread waits for each job thread to finish simulating its assigned job for the required number of `NITER` iterations. The main thread then terminates the monitor thread, and prints the information described below before exiting.

Concurrency: Your solution should strive to achieve a good level of concurrency among the executing threads. The program should not hold a synchronization object unnecessarily.

> **Caveats:** Beware of the following race conditions. After the main thread creates a job thread (and passes an argument to its start function)
>
> 1. the main thread can execute its remaining code before the job thread is able to run
> 2. the job thread can run (perhaps to completion) before the call to `pthread_create` has returned to the main thread
> 3. in some cases, the job thread can run for sometime before it receives the intended value of the argument
>
> Points 1 and 2 can be addressed using synchronization objects and functions (e.g., semaphores, mutexes, pthread_join(), etc.). A possible workaround of point 3 is to make a thread select a job to do while avoiding the possibility of having two threads selecting the same job.

Simulator Output

Your program should print the following information. The output below concerns the input file of Example 1.

Job Threads: Each job thread is required to print one line following each successful iteration where the job is executed once. Each line should have the fields displayed in the following sample output.

```
...
job: j1 (tid= 140214559033088, iter= 3, time= 600 msec)
job: j4 (tid= 140214533854976, iter= 3, time= 620 msec)
job: j2 (tid= 140214550640384, iter= 3, time= 660 msec)
job: j5 (tid= 140214525462272, iter= 3, time= 660 msec)
job: j3 (tid= 140214542247680, iter= 3, time= 700 msec)
...
```

where `tid` is the thread's ID (of type `pthread_t`, defined as an unsigned long integer on lab machines), `iter` is the iteration number that a thread has just completed, and `time` is the time (in millisec) relative to the **start time** of the program when the iteration is finished.

The Monitor Thread: This thread runs every `monitorTime` millisec. When active, it prints the information displayed in the following sample output:

```
...
monitor: [WAIT]
         [RUN] j1 j4
         [IDLE] j2 j3 j5
...
monitor: [WAIT]
         [RUN] j2 j5
         [IDLE] j1 j3 j4
...
```

That is, each time the monitor thread runs, it prints all jobs in each of the three possible states: `WAIT`, `RUN`, and `IDLE`. Note that the system in Example 1 has enough resources to run `j1` and `j4` (or, `j2` and `j5`) concurrently. Note also that each thread appears exactly once (since the monitor thread does not allow a state change to happen during printing).

Termination: After all other threads finish, the main thread prints the information displayed in the following sample output:

1. Information on resource types:

   ```
   System Resources:
           A: (maxAvail=   1, held=   0)
           B: (maxAvail=   1, held=   0)
           C: (maxAvail=   1, held=   0)
           D: (maxAvail=   1, held=   0)
           E: (maxAvail=   1, held=   0)
   ```

   Note that no resource unit is being held at this stage.

2. Information on jobs (assuming `NITER` = 20):

```
System Jobs:
[0] j1 (IDLE, runTime= 50 msec, idleTime= 100 msec):
        (tid= 140233984644864)
        A: (needed=   1, held=   0)
        B: (needed=   1, held=   0)
        (RUN: 20 times, WAIT: 10 msec)

[1] j2 (IDLE, runTime= 50 msec, idleTime= 100 msec):
        (tid= 140233976252160)
        B: (needed=   1, held=   0)
        C: (needed=   1, held=   0)
        (RUN: 20 times, WAIT: 60 msec)
...
...
[4] j5 (IDLE, runTime= 50 msec, idleTime= 100 msec):
        (tid= 140233951074048)
        E: (needed=   1, held=   0)
        A: (needed=   1, held=   0)
        (RUN: 20 times, WAIT: 110 msec)
```

Note that

- No resource is being held by any job at this stage.
- Each job thread is `IDLE`.
- The `"RUN: x times"` field confirms that the program has successfully simulated the job for $x =$ `NITER` times.
- The `"WAIT: x msec"` field shows the sum (over all iterations) of the time intervals (in millisec) a thread spent trying to acquire the needed units.

3. The total running time of the simulator (in millisec):
   `Running time= 3110 msec`

Deliverables

1. All programs should compile and run on the lab machines (e.g., ug[00 to 34].cs.ualberta.ca) using only standard libraries (e.g., standard I/O library, math, and pthread libraries are allowed).
2. Make sure your programs compile and run in a fresh directory.
3. Your work (including a Makefile) should be combined into a single tar archive **'your_last_name-a4.tar'** or **'your_last_name-a4.tar.gz'**.
   (a) Executing 'make' or 'make a4w22' should produce the `a4w22` executable file.
   (b) Executing 'make clean' should remove unneeded files produced in compilation.
   (c) Executing 'make tar' should produce the above '.tar' or '.tar.gz' archive.

(d) Your code should include suitable internal documentation of the key functions. If you use code from the textbooks, or code posted on eclass, acknowledge the use of the code in the internal documentation. Make sure to place such acknowledgments in close proximity of the code used.

4. Typeset a project report (e.g., one to three pages either in HTML or PDF) with the following (minimal set of) sections:
   - **Objectives:** state the project objectives and value from your point of view (which may be different from the one mentioned above)
   - **Design Overview:** highlight in point-form the important features of your design
   - **Project Status:** describe the status of your project; mention difficulties encountered in the implementation
   - **Testing and Results:** comment on how you tested your implementation
   - **Acknowledgments:** acknowledge sources of assistance
5. Upload your tar archive using the **Assignment #4 submission/feedback** links on the course's web page. Late submission is available for 24 hours for a penalty of 10% of the points assigned to the phase.
6. It is strongly suggested that you **submit early and submit often**. Only your **last successful submission** will be used for grading.

Marking

Roughly speaking, the breakdown of marks is as follows:

20% : successful compilation of reasonably complete program that is: modular, logically organized, easy to read and understand, includes error checking after important function calls, and acknowledges code used from the textbooks or the posted lab material

05% : ease of managing the project using the makefile

65% : correctness of implementing the program and displaying the required information.

10% : quality of the information provided in the project report