CMSC 330 Project #5 Fall 2012

Date due: Monday, December 10, 10:00:00 p.m.

1 Introduction

In this project you will develop a multithreaded Ruby program that can be used to simulate a subway system like the Washington Metro. You will also write code to process simulation output to display the state of the simulation, and (optionally) verify its feasibility.

2 End–of–semester TerpConnect account cleanup

Before the actual project is discussed, please note this information about accessing files and cleaning up your TerpConnect account after classes are over.

At some point after the semester ends you will lose permission to access the files in your extra disk space (probably including your projects), as well as permission to access the information we provided in the directory `330public` (examples from discussion, secret tests, etc.). At some point after that all these files will be automatically deleted by OIT. If you want to save any of your projects or other coursework, or any of the files in `330public`, it will be necessary to do so after finals are over. Sometimes students are asked for copies of class projects during job interviews, as part of scholarship applications, for internships, etc.; the instructional staff will not be able to provide these in the future, so you'll need to save copies yourself if you might ever need them later. Recall that the `-r` option to the `cp` command recursively copies a directory and all its contents, including subdirectories, so a command like `cp -r ~/330 ~/330.f12` would copy everything in your extra course disk space (which the symbolic link `330` points to) to a directory in your TerpConnect account home directory named `330.f12` (or use any other name if you like). Of course you have to have enough free disk space in your TerpConnect account to store the files; the `quota` command lets you know how much free space you have. Although you will lose your login permission to the Grace systems after the semester, you will still be able to log into your TerpConnect account on any TerpConnect machine (just log in to terpconnect.umd.edu or glue.umd.edu), since your TerpConnect account will remain as long as you're associated with the University.

You can also download files to your own computer; if you're using Windows a WinSCP client that will do this is WinSCP.

After copying files you want to save be sure to undo the changes that were made in the process of setting up your account before working on Project #1, since you may use the Grace systems for other courses in the future and the modifications made for this course may conflict with changes necessary for those courses:

1. Remove our directory from your path in your file `.path`, by editing it and removing the line beginning "`setenv PATH ...`" that you added.

2. Remove the symbolic link named `330` that you created from your home directory to your extra disk space for this course, as in `rm ~/330`. You could still reach the files in your extra disk space after that if you wanted to (until you lose access to them) just by using the full pathname, for example you could still use a command like:

   `cd /afs/glue/class/fall2012/cmsc/330/0101/student/`*loginID*

   instead of `cd ~/330`. For convenience though you probably want to wait to remove the symlink after you've copied any desired files from your extra disk space.

3. Also remove the symbolic link named `330public` that you created in your home directory, which points to the class public directory `/afs/glue/class/fall2012/cmsc/330/0101/public` (`rm ~/330public`).

At some point after the semester ends the class webpage will become inaccessible, so if there's any information you want from it that you don't already have, be sure to save it after your finals are over.

Lastly, after the final exam the class Piazza space will be disabled. If there's anything you need to discuss with the instructional staff past that time contact us individually.

3 Project description

3.1 Metro simulation rules

We will begin by describing how the metro simulation works. In the simulation there will be metro trains, each of which runs on a particular metro line, and passengers traveling on the metro, each with an itinerary. There are a number of rules governing how trains and passengers may move between metro stations.

3.1.1 Metro lines and stations

The metro consists of a number of metro lines. A metro line consists of a number of metro stations (stops). The same metro station may be on multiple metro lines (a transfer station).

3.1.2 Trains

At the beginning of the simulation trains enter the initial stop on their line (e.g., Greenbelt for the green line). Trains travel between the first and last station on their metro line, in order. Trains first move sequentially forward along the stations on their metro line, and when they reach the station at the end of the metro line that they are on they move sequentially backward along the line, until they reach the beginning of the metro line. The process repeats indefinitely, with trains continuing to move back and forth between the first and last station on the line, although trains may stop once all passengers have reached their destinations.

If there are no passengers in the entire simulation then each train must complete at least one round trip from the first to the last station, and back, on its metro line.

Multiple trains may travel on a metro line. Each train is assigned a number unique to that metro line, beginning with 1 and increasing by 1 at a time. Trains are named by the name of the metro line, followed by its number, separated by a space. For example, if the metro line "Red" has three trains, their names would be "Red 1", "Red 2", and "Red 3". Not very creative naming, but it will suffice for our purposes.

At most one train from each metro line may be at a station at a time (think of it as metro stations having only one platform per metro line). If there are multiple trains on a metro line, and two trains from the same metro line happen to arrive at the same time, one will have to wait until the station is clear. Two trains from the same line may not be at the same station even they are traveling in opposite directions. (Of course this is not the case for the real Metro, but it must be true in your simulation.) Trains from different metro lines may be at a station simultaneously if that station is a transfer station on those different lines. If there are multiple trains waiting to enter a station the order they enter is **unspecified** (again unlike the real Metro).

There may be multiple trains traveling between metro stations, and trains may pass each other between stations (think of it as stations being connected by multiple train tracks, for example a regular track and some express tracks between stations). Therefore the order that two trains arrive at a station does not affect the order they arrive at the next station.

3.1.3 Passengers

Each passenger has a itinerary, meaning a list of the stations that they plan to visit. Passengers travel on trains from station to station on their itinerary until they reach their final destination. At the start of the simulation passengers are located at the first station on their itinerary. Passengers wait until a train arrives at their station that will also stop at the next station on their itinerary. At that point they leave the station and board the train. A passenger may board any train traveling to the next station on their itinerary, unless the train is full (see below). Passengers may board trains traveling in either direction, regardless of which direction requires fewer stops. Similarly, if the current and destination stations are both on multiple metro lines, then any train on either line may be boarded.

When a train carrying a passenger arrives at the next station on the passenger's itinerary, the passenger disembarks (leaves the train and enters the station). Passengers continue traveling until they reach the final station on their itinerary.

It is possible that passengers at a station might miss a train as it passes through. In that case, the passengers remain in the station and wait for another opportunity to board a train. Similarly, if passengers on a train miss their stop, they remain on the train and wait for another opportunity to exit at the desired station. It may take a while, but the train will eventually return to that station.

Any line transfers will be explicit in the list of stations on a passenger's itinerary. For example, if there is no direct connection from College Park to Vienna you may assume passenger itineraries will not try to go directly from College Park to Vienna without using an intermediate station as a transfer point.

All trains in a simulation have a limit as to the maximum number of passengers who may be aboard. If a train is full, passengers may not board that train until other passengers leave. The same passenger limit applies to all trains (unlike the real Metro, which has four–car trains, six–car trains, and eight–car trains, all trains in the simulation are the same size), but since the passenger limit is part of your program's input, different limits may apply during different executions of your program.

3.2 Metro simulation outputs

A metro simulation may be described by a number of simulation events, and the order they occur. Four simulation events and their associated messages are:

Train *line number* entering *station–name*

Train *line number* leaving *station–name*

passenger–name boarding train *line number* at *station–name*

passenger–name leaving train *line number* at *station–name*

In these messages *line*, *number*, *station–name*, and *passenger–name* would be replaced by the line (color), train number, station name, and passenger name of the event that occurred. The simulator must output these simulation messages in the order they occur. These messages (and their order of occurrence) may then be analyzed and used to either display the state of the simulation, or to discover whether the simulation results are valid.

3.3 Metro simulation parameters and files

Each metro simulation is performed for a specific set of simulation parameters. These parameters are read from a simulation file, and include the following. Note that you will not have to write code to read these parameters; our code that is being provided to you does this.

Metro lines: This is the name of a metro line followed by a list of stations on that line. A simulation may have multiple lines.

Metro trains: This is the name of a metro line followed by the number of trains running on that line.

Passenger limit: This is the limit to the number of passengers that can be aboard any train at any time, during an execution of the program.

Passengers: This will be a sequence of passenger names, each followed by the list of stations in that passenger's itinerary.

Simulation output: A simulation file may contain an example output for a simulation performed with this set of simulation parameters.

If a simulation does not use any of these parameters they will be omitted from the simulation parameter file. Explanatory comments appear at the beginning of simulation files, describing what they are testing. Here is the first part of an example simulation file:

```
=== Lines ===
Red, Glenmont, Silver Spring, Union Station, Bethesda, Shady Grove
=== Trains ===
Red=1
=== Passenger Limit ===
limit=10
=== Passengers ===
Amy, Silver Spring, Bethesda
Ann, Glenmont, Bethesda
Art, Union Station, Silver Spring
Aaron, Bethesda, Glenmont
=== Output ===
Train Red 1 entering Glenmont
Ann boarding train Red 1 at Glenmont
Train Red 1 leaving Glenmont
Train Red 1 entering Silver Spring
Amy boarding train Red 1 at Silver Spring
Train Red 1 leaving Silver Spring
⋮
```

3.4 Simulation driver

In addition to the public test simulation files in the `proj5` directory of the public class directory you will also find the following files:

`simulate.rb`: This is a driver for the simulator.

`metro.rb`: This is the simulator (write all your code in this file).

`display-example.rb`: This is an example showing how to use `display_state()` (see below).

`run-public-tests.rb`: This is a simple Ruby script to run the public tests.

The simulation driver in `simulate.rb` will read in a simulation file containing simulation parameters and put the data in a number of Ruby hashes for use by your code. For instance, the simulation parameters and simulation output in the simulation file shown above will be processed to produce the following:

```
lines = {
  "Red" => ["Glenmont", "Silver Spring", "Union Station", "Bethesda",
            "Shady Grove"]
}

numTrains = {
  "Red" => 1
}

passengers = {
  "Amy" => ["Silver Spring", "Bethesda"],
  "Ann" => ["Glenmont", "Bethesda"],
  "Art" => ["Union Station", "Silver Spring"],
  "Aaron" => ["Bethesda", "Glenmont"]
}
```

```
sim_output = [
  "Train Red 1 entering Glenmont",
  "Ann boarding train Red 1 at Glenmont",
  "Train Red 1 leaving Glenmont",
  "Train Red 1 entering Silver Spring",
  "Amy boarding train Red 1 at Silver Spring",
  "Train Red 1 leaving Silver Spring",
  ⋮
]
```

The data structures are organized as follows:

- The hash for metro lines contains the metro line "Red", which begins at the station Glenmont and ends at the station Shady Grove.
- The hash for the number of trains contains an integer value for each metro line, specifying the number of trains on that line.
- The hash for passengers contains a list of stations names (forming the itinerary) for each passenger. The first station is the passenger starting point, the last station is the final passenger destination.
- Any simulation output will be put into an array of strings, with each string corresponding to one simulation event message.

`simulate.rb` will also take a command–line parameter specifying whether the program should perform a simulation or simply display or verify the feasibility of the simulation output. It can be invoked as:

`simulate.rb` (`simulate`|`display`|`verify`) *simulate–filename*

(I.e., one of the three options `simulate`, `display`, or `verify` should appear following the program name and before the name of the simulation input file.) So running the command `simulate.rb simulate public01.input` would execute a simulation (calling your simulation code) using the simulation parameters that are in the file `public01.input` (ignoring the simulation output in `public01.input`), while running `simulate.rb verify public01.input` would perform an analysis of the simulation output in `public01.input` to determine whether it is feasible (again calling your code).

Note that `simulate.rb` outputs simulation parameters before simulation messages, so that its output if directed to a file may be passed directly to the simulation display/verify routines.

The simulation files include the simulation parameters and one possible output, but some different possible outputs are also provided for the tests that perform simulation (see below).

4 Project implementation

You must implement three major methods: `display()`, `verify()`, and `simulate()`. The three parts may be implemented independently, though `display()` and `verify()` are similar.

4.1 Part 1: Simulation display

A multithreaded simulation can clearly have many different behaviors, depending on the thread scheduler. One way to help determine whether a simulation is proceeding correctly (i.e., avoiding race conditions) is to model the state of the simulation by processing the simulation outputs. The model can then be used to display the state of the simulation, or determine its validity.

The first part of your project is to implement a model of the simulation (by processing simulation event messages) sufficiently detailed to display the following:

- the trains at each station,
- the passengers at each station, and
- the passengers on board each train.

Your code should display the initial state of the simulation. Then it should list each simulation event message in order, followed by a display of the state of the simulation after each event. For instance, for the simulation parameters shown above, your code should display the initial state of the simulation as follows:

```
Red
                    Glenmont       Ann
               Silver Spring       Amy
               Union Station       Art
                    Bethesda     Aaron
                 Shady Grove
```

Then your code should process the simulation event messages in the simulation output, displaying the message and the resulting state. For instance, after processing the message `Amy boarding train Red 1 at Silver Spring` in the simulation output, your model should contain enough information to display the following:

```
Amy boarding train Red 1 at Silver Spring
Red
                    Glenmont
               Silver Spring             [Red 1 Amy Ann]
               Union Station       Art
                    Bethesda     Aaron
                 Shady Grove
```

For the simulation display part of the project you may assume that the sample simulation output is valid. `simulate.rb` will invoke your verifier for this part of the project as:

```
display(lines, passengers, sim_out)
```

where `lines`, `passengers`, and `sim_out` are the hashes and array produced in `simulate.rb` from the simulation file. The submit server will be doing the same thing when your code is run there.

A function `display_state(lines, stations, trains)` is provided for actually printing out the state of the simulation. If you want to use it you will simply need to collect enough information in your model to provide two hashes `stations` and `trains`, which should contain information on trains and passengers at each station, and passengers on each train, as follows:

`stations[station name]`	hash for metro line
`stations[station name][line name]`	hash for trains at station (for line)
`stations[station name][line name][train num]`	where train num is "1", "2", etc...
`stations[station name]["passenger"]`	hash for passengers at station
`stations[station name]["passenger"][passenger name]`	exists when passenger is at the station
`trains[train name]`	hash for passengers on train
`trains[train name][passenger name]`	exists when passenger is on the train

The double–quoted string `"passenger"` above means that literal string is used as a hash key.

See the example `display-example.rb` in the project starter files for more detailed illustration of how to use `display_state()`.

Notice that this part of the project does not actually use concurrency or threads at all.

4.2 Part 2: Metro simulation

For part 2 (possibly reusing your data structures from part 1) you will write a Ruby program that actually performs a multithreaded simulation using the simulation parameters supplied. Your simulation should be implemented as follows:

- Each train and passenger in the simulation should be represented by its own thread, so if you are simulating m trains and n people, there should be $m + n + 1$ threads in the system (m train threads, n passenger threads, and one thread for the main program.
- The initial state of the simulation should be as described in the metro simulation rules (i.e., all passengers are at the first station in their itinerary, and all trains poised to enter the first station on their metro line).
- Trains must never carry more passengers at any time than the value read from the simulation parameter file.
- You **must** use synchronization (i.e., Ruby monitors) to avoid race conditions and ensure your simulation is valid. You should use the monitor associated with each metro line to prevent all race conditions for that metro line. So a simulation with n metro lines could have up to n train or passenger threads executing concurrently.
- You must use condition variables to ensure your simulation uses synchronization efficiently. Use at least two condition variables per monitor, one for trains and one for passengers. Trains should wake up passengers when entering stations, and wake up other trains when exiting stations.
- Use enough locks to permit concurrent execution and avoid race conditions, but not so many locks that concurrency is reduced.
- Use Ruby's `wait` condition methods and `broadcast` to avoid busy waiting.
- Each train should sleep for 0.01 seconds after entering a station before exiting.
- Your threads **may not use busy–waiting** (for example trains waiting for passengers or other trains, or passengers waiting for trains); you may lose substantial credit on the project if your threads do this. Other than trains sleeping for exactly 0.01 seconds after entering a station before exiting, your threads also may **not** just sleep while waiting (for passengers or trains). Instead use the `wait` condition methods as mentioned to wait until the desired event occurs, at which time the waiting thread should be woken up by a `broadcast` from another thread.
- The TAs will be looking at your submitted code to check that you are using synchronization correctly, and have not used disallowed operations as described above.
- Notice that the list of stations in the itinerary for each passenger does not tell you what metro line they want to take. You'll need to compute that based on the current station and next destination. If there is more than one valid metro line, the passenger can board a train from any valid line.
- The simulation ends when all people have arrived at their final stops. To achieve this, in the main thread you'll probably need to do a join on all the threads representing the people. Notice that it is legal if trains continue running for a while even after all passengers have arrived.
- If there are no passengers, each train must complete at least one round trip from the first to the last station (and back) on its metro line before the simulation ends.
- You should set `Thread.abort_on_exception = true` in your code, to avoid hiding errors in threads.
- In order to see what's going on during your simulation, your program must print out various lines of text as simulation events occur. These lines of text are described in Section 3.2 above. For the output to make sense you must do the following:
  - Only print out a message while you are holding a lock.
  - Immediately after printing, and before you release the lock, call `$stdout.flush()` to flush standard output to the screen.

Following the two rules above should ensure that if you build the simulation correctly, your simulation output will be valid. Otherwise, you might get strange interleavings of output messages that look incorrect even if your simulation code is actually correct.

`simulate.rb` (and the submit server) will invoke your code for this part of the project as:

```
simulate(lines, num_trains, passengers, sim_monitor, limit)
```

where `lines`, `num_trains`, and `passengers` are the hashes produced in `simulate.rb` from the simulation file. `sim_monitor` is a hash containing a monitor variable for each metro line. You must use these monitors (and condition variables derived from them) for synchronization in your code.

4.3 Optional simulation verifier

It should be clear that a multithreaded simulation may have many different behaviors, depending on the thread scheduler. However, there are certain restrictions on the simulation output, for example, people can board trains only when those trains are at the station where those people are. If you want to you may write a Ruby method that examines your simulation outputs and checks whether they are valid (i.e., follow all the metro simulation rules in the project description).

The list of possible errors in simulation output is nearly endless, so you can only check some common errors associated with race conditions resulting from incorrect synchronization. Some conditions you can look for in the simulation are:

- Trains start by entering the initial station on their line.
- Trains move forward and backward along the stations on their line, and visit all stations without skipping any.
- Trains enter a station before they leave it.
- Two trains from the same metro line are not at the same station at the same time.
- Each passenger follows their itinerary.
- Passengers only board and leave a train while it is at a station (i.e., after that train has entered the station, but before it has left the station).
- All passengers reach their final destination by the end of the simulation.
- If there are no passengers in the simulation, each train must complete at least one round trip from the first to the last station (and back) on its line.
- Trains never have more than the maximum number of passengers on board.

We are providing some test cases for your verifier if you want to test it, but they are not set up on the submit server because they are not directly worth any points (of course writing the verifier may help you get a higher score on the tests that are worth points, by finding problems in your code). `simulate.rb` will invoke your verifier as:

```
result= verify(lines, num_rains, passengers, sim_out, limit)
```

where `lines`, `numTrains`, `passengers`, `sim_out` are the hashes and array produced in `simulate.rb` from the simulation file. The function verify should return true if the simulation is valid, and false otherwise.

Notice that this part of the project does not use concurrency or threads either.

5 Project requirements and submitting your project

1. Your methods will be called by `simulate.rb` so they will not read any input. You may call our method `display_state()` (or print output directly if you prefer to do that for some reason), but all output should be written to the program's standard output.

2. You may use any Ruby language features in your project that you would like, regardless of whether they were covered in class or not, so long as your program works successfully using the version of Ruby installed on the OIT Grace Cluster and on the CMSC project submission server, and as long as you use threads, and synchronization, as described above.

3. Your submitted methods **must** be the file named `metro.rb`, otherwise our `simulate.rb` running on the submit server will not find them, consequently your program's score will be zero. Although Ruby programs can be broken up into multiple source files (not shown in class), all your code should be in this one source file. Your program would work otherwise, but this is just to make it easy for the TAs to check your code. They will only be looking at the code in your `metro.rb` source file, so if you have code that's not in that file you will lose credit because it will not be seen.

4. To check that your output matches the expected output use the verify option of `simulate.rb` as described above. It will print "`VALID.`" if your `verify()` method says that the simulation is valid, and "`INVALID.`" otherwise.

5. Your `metro.rb` source file **must have** a comment near the top that contains your name, TerpConnect login ID (which is your directory ID), your university ID number, and your section number.

   The Campus Senate has adopted a policy asking students to include the following statement on each major assignment in every course: "I pledge on my honor that I have not given or received any unauthorized assistance on this assignment." Consequently you're requested to include this pledge in a comment near the top of your program source file. See the next section for important information regarding academic integrity.

6. Note that although you are not being graded on your source code or style (see the separate project grading handout), if the TAs cannot read or understand your code they cannot help should you have to come to office hours, until you return with a well–written and clear program.

7. As before, to submit your program just type the single command `submit` from the `proj5` directory, where your source file should be.

6 Academic integrity

Please **carefully read** the academic honesty section of the course syllabus. **Any evidence** of impermissible cooperation on projects, use of disallowed materials or resources, or unauthorized use of computer accounts, **will be submitted** to the Student Honor Council, which could result in an XF for the course, or suspension or expulsion from the University. Be sure you understand what you are and what you are not permitted to do in regards to academic integrity when it comes to project assignments. These policies apply to all students, and the Student Honor Council does not consider lack of knowledge of the policies to be a defense for violating them. Full information is found in the course syllabus– please review it at this time.