COMP2401 - Assignment #5

(Due: **Sun. Mar 29, 2020 @ 6pm**)

In this assignment, you will make a simulator for a Dispatch Center that dispatches taxis to different areas of a city to pick up customers and drop them off to other areas of the city. Your code will make use of threads to handle incoming requests and display. The code will also fork multiple processes to represent multiple taxis. You will then run separate processes as customers that connect to the dispatch center to be picked up and dropped off by available taxis.

To begin this assignment, you should download the following files:

- **simulator.h** – contains definitions and structs that will be used throughout your code
- **simulator.c** – contains a template for the code that will run the simulator server
- **dispatchCenter.c** – contains a template for the code for a process that will run the dispatch
- **display.c** – contains the window/drawing code that you will use to display the city and taxis
- **taxi.c** – contains a template for the code for a process that will run a single taxi
- **customer.c** – contains a template for the code for a process that runs one customer request
- **generator.c** – contains a template for the code that will continuously generate customers
- **stop.c** – a program used to shut down the server
- **makefile** – the file that you will use to compile everything

You will generate (and use) five executables in this assignment:

1. **simulator** – this will open a widow to display everything as well as set up the dispatch server
2. **stop** – this will shut down the server
3. **customer** – this will run a process that will simulate a single customer request
4. **generator** – this will continuously generate random customers for testing

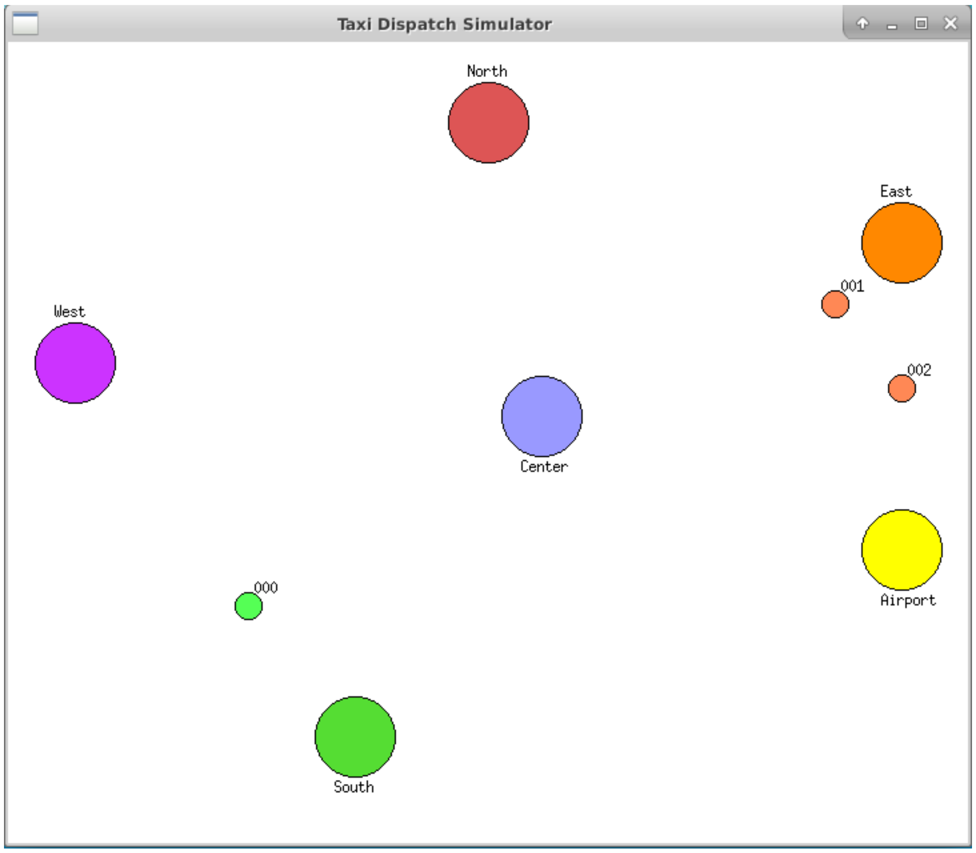
When compiling the files, you will need to include the `-lm -lpthread` and `-lX11` libraries. (but the `-lX11` is only needed for the **simulator** program since it uses a window and graphics.)

(1) Examine the **simulator.h** file. It contains the following definitions and constants that represent the city that we will be testing:

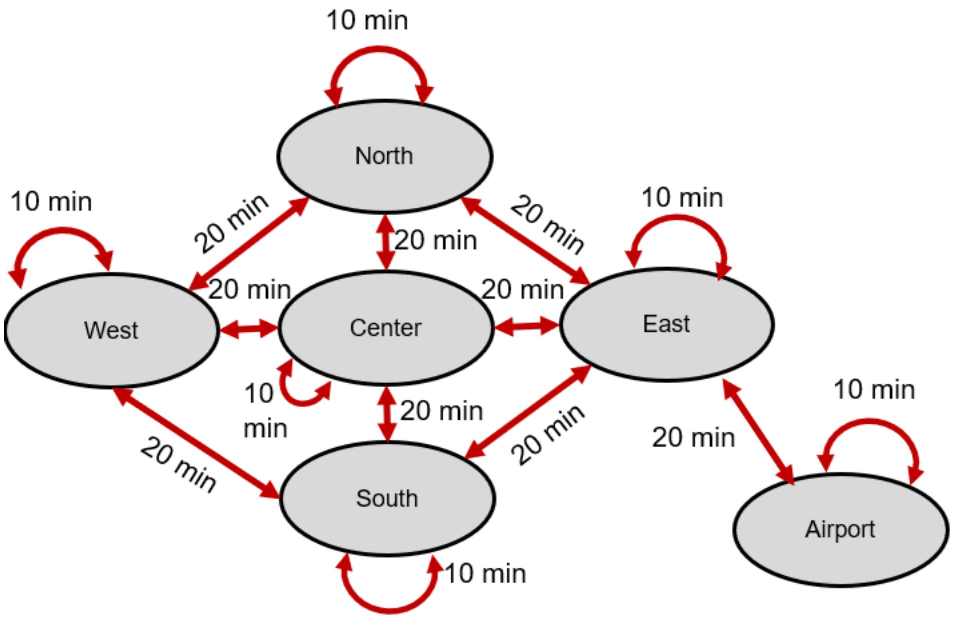
```
CITY_WIDTH              720  // The width & height of the city's display window
CITY_HEIGHT             600
NUM_CITY_AREAS            6  // Number of main areas of the city
UNKNOWN_AREA             -1  // An indicator that the city area is unknown

const char *AREA_NAMES[NUM_CITY_AREAS] = {"North", "East", "Airport",
                                          "South", "Center", "West"};
const short AREA_X_LOCATIONS[6] = {360, 670, 670, 260, 400,  50};
const short AREA_Y_LOCATIONS[6] = { 60, 150, 380, 520, 280, 240};
```

The next page shows a screenshot of what the city will look like when printed. Notice that each of the 6 city areas has a name as well as a fixed (x,y) location and is displayed as a large circle of a certain color. The smaller circles are moving taxis. A taxi will have the same color as a city area if it is on its way there to drop off a customer. A taxi will be black if it is picking up a customer and white if it is available and waiting for a customer. Each taxi has a 3-digit plate number as its ID. Each city area is actually just an index from 0 to 5. An index of -1 will represent an UNKNOWN_AREA.



Ideally, it would be nice to see a map of the city, but we kept it simple and just chose 6 circles to represent pickup and dropoff areas of the city.



As the program runs, the taxis will travel back and forth from one city area to another, as customers are picked up and dropped off. The travel time estimates from one area of the city to another area is defined by this figure. Note that taxis may pickup and drop off a customer in the same area, but this still takes 10 minutes. This diagram has been hard-coded into a constant 2D array which is also defined in the **simulator.h** file as follows. Make sure that you understand it.

```
    const char  TIME_ESTIMATES[NUM_CITY_AREAS][NUM_CITY_AREAS] =
      {{10, 20, 40, 40, 20, 20},
       {20, 10, 20, 20, 20, 40},
       {40, 20, 10, 40, 40, 60},
       {40, 20, 40, 10, 20, 20},
       {20, 20, 40, 20, 10, 20},
       {20, 40, 60, 20, 20, 10}};
```

A dispatch center is defined as follows:

```
typedef struct {
  char        online;                 // 0 = no, 1 = yes
  Taxi       *taxis[MAX_TAXIS];       // the taxis belonging to this dispatch center
  int         numTaxis;               // the number of taxis in the dispatch center
  Request     requests[MAX_REQUESTS]; // customer requests
  int         numRequests;            // number of customer requests in the system
} DispatchCenter;
```

Notice that it is basically two arrays … one for taxis and one for pending customer requests. Each of these is set to be a maximum of 100 in size, but the **numTaxis** and **numRequests** should indicate the actual number at any moment. **numTaxis** will remain constant in our program, whereas **numRequests** will vary over time.

Each **Request** is just two small numbers indicating where the customer wants to be picked up and where he/she wants to be dropped off as follows:

```
typedef struct {
  char   pickupLocation;          // City area to be picked up in (0 to NUM_CTY_AREAS)
  char   dropoffLocation;         // City area to be dropped off in (0 to NUM_CTY_AREAS)
} Request;
```

Each Taxi has a unique **pID** (i.e., process id, since each will run as a separate process). A Taxi also has a unique plate number (0 to **numTaxis**) and their current (**x,y**) location in the city at any time. A taxi has a **status** of being either AVAILABLE, PICKUP_UP someone or DROPPING_OFF someone. The **currentArea** is the area that the taxi is currently in. The **pickupArea** is the area that the taxi is on the way to in order to get the customer and the **dropoffArea** is the area that the customer is to be dropped off at. Each of these areas is a number from 0 to NUM_CITY_AREAS or set to UNKNOWN_AREA. Finally, the **eta** is the estimated time of arrival to the pickup or destination area:

```
typedef struct {
  int      pID;                   // process ID for this taxi
  int      x;                     // x location of the taxi
  int      y;                     // y location of the taxi
  short    plateNumber;           // unique plate id for taxi
  char     status;                // AVAILABLE, PICKING_UP, or DROPPING_OFF
  char     currentArea;           // index of city area currently in
  char     pickupArea;            // index of city area to pick up person in
  char     dropoffArea;           // index of city area to drop off person in
  short    eta;                   // time left until reaching destination
} Taxi;
```

(2) The **simulator.c** file contains the code for the main application. You need to add code so that the function does the following:

- It should dynamically-allocate a single Taxi and store it in the dispatch center. The new taxi should be placed in a random area of the city (i.e., in the center of one of the city area circles) and have plate number 0. The **currentArea** of the taxi should be set to that area, while the pickup/dropoff areas should be UNKNOWN_AREA. New taxis should start off being available for customers.

- It must use **fork()** to start a new process for that taxi … calling the **runTaxi()** function in the **taxi.c** file … passing newly created taxi as its parameter. Upon returning from the **runTaxi()** function, the taxi process should simply exit.

- It should spawn a thread to handle incoming requests. This thread should call the **handleIncomingRequests()** function in the **dispatchCenter.c** file and pass in a pointer to the **ottawaDispatch** dispatch center.

- It should spawn a thread to handle the displaying of the city. This thread should call the **showSimulation()** function in the **display.c** file and pass in a pointer to the **ottawaDispatch** dispatch center.

- The code should then wait for the request-handling thread to complete and it should then kill all running taxi processes.

- The code should free all allocated memory so that **valgrind** shows no leaks.

The **showSimulation()** function (in the display.c file) has been completed for you. You MUST NOT alter any code in the **display.c** file. Use the **makefile** that was given to you to compile everything. Once you have this step completed, run the simulator in the background (i.e., use **&**). Make sure that you see the screen snapshot (on the previous page) appear … showing the city areas and a single taxi in one of the areas. Once you see this, you can be sure that the display thread is working properly. Type **ps** in the terminal window. You should see two processes labelled simulator. That is because you have a taxi process running with the same name. When you close the window, you should see some XIO error (don't worry about this). After closing the window, you can type **ps** again … you should see that one simulator process is no longer running, but you will likely see the taxi process (also called simulator) still running. You should kill this process in the terminal using **kill** followed by the process id.

(3) Write code in the **handleIncomingRequests()** function (from dispatchCenter.c) so that it first starts up a server and then goes into an infinite loop that repeatedly waits for incoming user requests and handles them. The idea is that taxis will communicate with the server to ask for a customer to pickup. When the server receives a **REQUEST_CUSTOMER** command from a taxi, it should check to see if there is a customer request available in its requests buffer. If not, the taxi should be told that no request is available. Otherwise, the taxi should be given the pickup and dropoff location for the customer … and this customer request should be removed from the request buffer so that no other taxis get that request. While on route to pickup or drop off the customer … the taxi will send an **UPDATE** command to the server along with its plate number, current x and y location, status and the dropoffArea. These should be recorded at the dispatch center so that the dispatch center always has each taxi's latest information. Customers will send a **REQUEST_TAXI** command to the server which will contain the requested pickup and dropoff locations as city area indices. If the maximum number of requests has been reached, the customer should be told that the request cannot be handled. Otherwise, it should be told that all is ok and the request should be added to the buffer. Requests should always be handled on a first-come-first-served basis. Finally, a **SHUTDOWN** command should cause the server to go off line (gracefully) when it is received. Make sure the code compiles, but you cannot test it until you do the next step.

(4) Complete the code in the **stop.c** file so that it attempts to shut down the dispatch center server by sending the **SHUTDOWN** command to it. Test your code by running the **simulator** server in the

background and then running the **stop** program. If all works well, the simulator window should close and should shut down gracefully. Use **ps** to make sure that the simulator has indeed shut down properly as well as the **stop** program. Make sure that you don't have any segmentation faults.

(5) Complete the **runTaxi()** function in the **taxi.c** file so that it runs in an infinite loop. If the taxi is available, it should contact the dispatch center to get a request. If it is busy picking up or dropping off a customer, it should send updates to the dispatch center. A delay has been added to the loop so that it does not overwhelm the server with many requests. Here is how to do these two steps (you may want them done as helper functions):

a. To request a customer from the dispatch center, the taxi should first attempt to connect to the dispatch center and then if successful, it should send the REQUEST_CUSTOMER command. If the server responds with a YES, then the taxi should handle the customer request as follows:
    i. If the pickup area is the same as the area the taxi is currently in, assume that the pickup is immediate. Then the taxi just needs to head to the dropoff area. The taxi's <u>currentArea</u>, <u>pickupArea</u> and <u>dropoffArea</u> should all be set properly and the taxi should be in DROPPING_OFF mode. The estimated time of arrival (eta) should be set accordingly based on the 2D TIME_ESTIMATES array from the <u>pickupArea</u> to the <u>dropoffArea</u>.
    ii. If the pickup area is different from the area that the taxi is currently in, then the taxi should go into PICKING_UP mode. Make sure to set the <u>pickupArea</u> and <u>dropoffArea</u> accordingly. The eta should be set according to the time it takes to get from the <u>currentArea</u> to the <u>pickupArea</u>.
b. To send an update to the dispatch center, the taxi should first attempt to connect to the dispatch center and then if successful, it should send the UPDATE command along with its (x,y) location, plate number, status and dropoffArea.

You will also need to make the taxi move ahead a little bit each time in the loop. You may want to make a helper function for this as well. To do this … follow this logic:

```
IF (taxi is PICKING_UP) {
   deltaX = (x of pickup area – x of current area) / (time estimate from current area to pickup area)
   deltaY= (y of pickup area – y of current area) / (time estimate from current area to pickup area)
}
ELSE {
   deltaX = (x of dropoff area – x of pickup area) / (time estimate from pickup area to dropoff area)
   deltaY= (y of dropoff area – y of pickup area) / (time estimate from pickup area to dropoff area)
}
New taxi position = (x + deltaX, y + dektaY)
New eta should now be one second less.
```

You will need to check for when the taxi arrives at its pickup or destination locations. If it is DROPPING_OFF a customer and the eta has decreased to 0, then the taxi's (x,y) location should be set to the dropoff area's center, the taxi should be AVAILABLE now, the currentArea should be the dropoffArea and the pickup and dropoff areas should be unknown now. If the taxi is PICKING_UP a customer and reaches the pickup location (i.e., eta down to 0), the taxi's (x,y) location should be the pickup area's center, the taxi should go into DROPPING_OFF mode, the current Area should be unknown now (as it is travelling) and the eta should be set to the TIME_ESTIMATE from the pickup to dropoff areas.

(6) Complete the **customer.c** program so that it takes two command line arguments. The first being the pickup area index and the 2nd being the dropoff area index. It should connect to the dispatch center server and send a single request according to the command line arguments. There is no loop … just a single request should be sent and the response received. If the request was denied, a message should be printed to the screen to indicate this.

(7) As a final step, complete the **generator.c** program so that it keeps generating customer requests at random pickup and dropoff areas in the city by using the **system()** command. Once you get it working… run the **simulator** in the background. Then run the **generator** in the background as well. You should see the taxis being busy picking up and dropping off clients. Finally, stop the generator (you can use **fg** to get it back into the foreground and then use CNTRL+C to stop it). You should see the requests stop generating and eventually all the taxis will lie still at one of the city area centers. Use **ps** to make sure that all customer processes are completed, with the exception of the simulator. Run the generator again for a bit. Then stop it again … and while there are still some customers being shown … run the **stop** program. It should shut down the server gracefully. Run **ps** again. All processes should be stopped and there should be no simulator nor customer processes running.

(8) Modify the **simulator.c** code so that 10 taxis are running. Make sure that the code still works.

IMPORTANT SUBMISSION INSTRUCTIONS:

Submit all of your **c source code** files (and **makefile**) as a single **tar** file containing:

1. A **Readme** text file containing
   - your name and studentNumber
   - a list of source files submitted
   - any specific instructions for compiling and/or running your code
2. All of your **.c source** files and all other files needed for testing/running your programs.
3. Any output files required, if there are any.

The code **MUST** compile and run on the course VM.

- If your internet connection at home is down or does not work, we will not accept this as a reason for handing in an assignment late ... so make sure to submit the assignment WELL BEFORE it is due !

- You WILL lose marks on this assignment if any of your files are missing. So, make sure that you hand in the correct files and version of your assignment. You will also lose marks if your code is not **written neatly with proper indentation and containing a reasonable number of comments.** See course notes for examples of what is proper indentation, writing style and reasonable commenting).
