COMP2401 - Assignment #5

(Due: **Thursday, March 31st, 2022 @ 6pm**)

In this assignment, you will make a simulator for a Fair that allows Guests to go onto Rides. Your code will make use of threads to handle incoming requests, run the rides and display the fair information. You will then run separate processes as guests that connect to the fair server so that the guest may go onto the rides until he/she runs out of tickets.

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

- **makefile** – the file that you will use to compile everything.
- **fair.h** – contains definitions and structs that will be used throughout your code.
- **display.h** – contains a few display definitions that you will use for display.
- **fairApp.c** – a program that will run the fair simulator.
- **requestHelper.c** – contains code for a thread that will handle incoming guest requests.
- **display.c** – contains the window/drawing code that you will use to display the ride information.
- **ride.c** – contains code for a thread that will run a single ride.
- **guest.c** – a program that represents a guest process.
- **generator.c** – a program that will generate 100 guests.
- **stop.c** – a program used to shut down the server.

You will generate (and use) 4 executables in this assignment:

1. **fairApp** – opens a widow to display everything as well as set up the server threads. It is the main program simulator.
2. **stop** – shuts down the server.
3. **guest** – runs a process that will simulate a single guest at the fair.
4. **generator** – generates random guests 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 **fairApp** program since it uses a window and graphics.)

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

```
#define   NUM_RIDES            10  // # of rides at the fair
#define   RIDE_NAME_MAX_CHARS  20  // Max # of characters in a ride's name, including '\0'
#define   MAX_GUESTS          300  // Max # of guests allowed at the fair
#define   MAX_LINEUP           50  // Max #of guests that can wait in a lineup

// Various modes that a Ride may be in at any time
#define   OFF_LINE     0
#define   STOPPED      1
#define   LOADING      2
#define   RUNNING      3
#define   UNLOADING    4

// Server Responses
#define   NO           0
#define   YES          1

// Message requests that a Guest makes to a Fair
#define   SHUTDOWN           0
#define   ADMIT              1
```

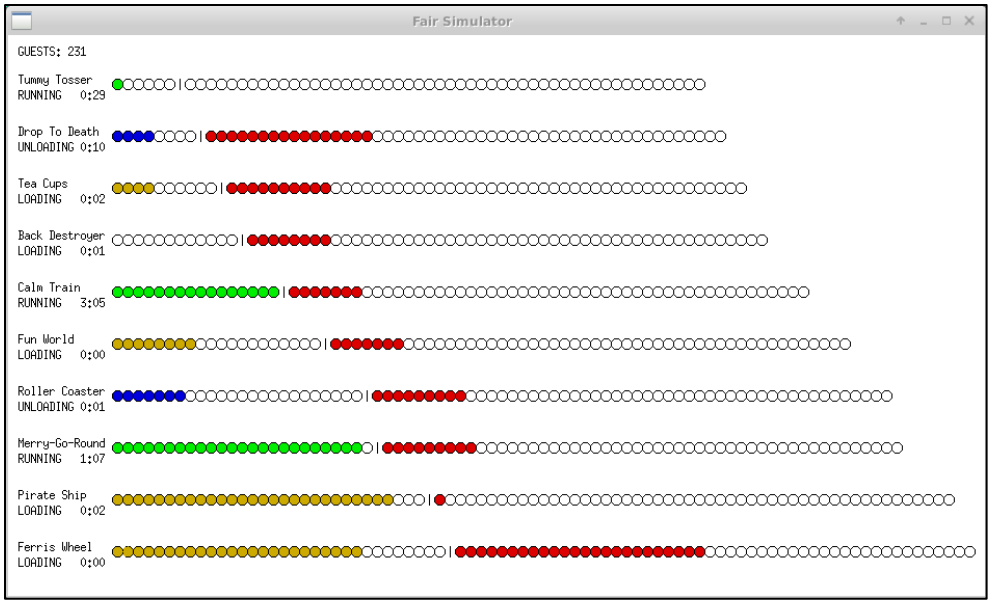
```
#define   GET_WAIT_ESTIMATE   2
#define   GET_IN_LINE         3
#define   LEAVE_FAIR          4

// Settings for Fair Server setup
#define SERVER_IP     "127.0.0.1"   // IP address of simulator server
#define SERVER_PORT          6000   // PORT of the simulator server


// This struct represents a Ride that the guests can go on
typedef struct  {
  int              pid;                    // Process ID for this ride
  char            *name;                   // Name of the ride
  unsigned char    status;                 // Status of the ride as either OFF_LINE, STOPPED,
                                           // LOADING, RUNNING or UNLOADING
  unsigned char    ticketsRequired;        // Between 1 to 5
  unsigned char    capacity;               // Number of guests that can ride at the same time
  unsigned char    onOffTime;              // Number of seconds that it takes for a single rider
                                           // to get on/off the ride
  unsigned short   rideTime;               // Number of seconds that the ride runs for each time
  unsigned short   waitTime;               // Number of seconds that the ride waits before
                                           // running, if not at full capacity
  unsigned int     waitingLine[MAX_LINEUP]; // Lineup of guests for this ride (keeps process IDs)
  unsigned char    lineupSize;             // Number of guests waiting in line
  unsigned int    *riders;                 // Guests that are on the ride (keeps process IDs)
  unsigned char    numRiders;              // Number of guests currently on the ride
  unsigned short   countdownTimer;         // Number of seconds to count down until
                                           // loading/unloading/running/waiting is done
} Ride;


// This struct represents a Fair with rides and guests
typedef struct {
    Ride          rides[NUM_RIDES];
    unsigned int  guestIDs[MAX_GUESTS];    // guest Process IDs ... needed to stop the guest
                                           // processes after a shutdown
    unsigned int  numGuests;               // Number of admitted guests
} Fair;
```

Here is a screenshot of what the **fairApp** may look like while running. It indicates the number of guests at the fair (in the top left corner). Then for each of the 10 rides, it shows the **name**, **status** and **countdownTimer** along the left side of the window. For each ride it also shows the guests on the ride (tan, green, blue) and those waiting in line (red). Those boarding the ride are shown as tan-colored. When the ride is running, the riding guests are shown in green. When the ride is done, those getting off the ride are shown as blue. All the displaying work has been done for you, so you do not need to worry about that.



(2) The **fairApp.c** file contains the code for the main application. A Fair has been created and stored as **ottawaFair** and all rides have been initialized. You need to add code so that the function does the following:

- It should spawn a **thread** for each ride. This thread should call the **runRide()** function in the **ride.c** file and pass in a pointer to the **Ride** from the fair that it is running.

- It should spawn a thread to handle incoming requests from guests. This thread should call the **handleIncomingRequests()** function in the **requestHandler.c** file and pass in a pointer to the **ottawaFair**.

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

- The code should then wait for the request-handling thread to complete and it should then shut down all running ride threads and free up the riders arrays. For any guest processes that are still running, they should be shut down.

- 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 **fairApp** in the background (i.e., use **&**). Make sure that you see the screen snapshot (on the previous page) appear … showing the fair. However, you should see 0 guests, all stopped rides, and empty black circles representing guest locations … but there are no guests yet. Once you see this, you can be sure that the display thread is working properly. Type **ps** in the terminal window. You should see a processes labelled **fairApp**. 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 the **fairApp** process is no longer running. If you ever see that a **fairApp** process is still running (perhaps from an old test), you should kill this process in the terminal using **kill** followed by the process id.

(3) Write code in the **handleIncomingRequests()** function (from **requestHandler.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 guests will communicate with the server to ask for information (i.e., admit to the fair or ask how for a ride wait time estimate) or to indicate what he/she will do (e.g., get on a ride or leave the fair).

- When the server receives an **ADMIT** command from a guest, it should make sure that the fair is not full with MAX_GUESTS already. If so, the guest cannot be admitted. Otherwise, the guest can be admitted. The fair will need to know that guest's process ID for later (in the fair's **guests** array), in case it shuts down and needs to inform that guest. Therefore, it must receive this information along with this **ADMIT** request. If the guest is admitted to the fair, the server should send back a lost of rides names along with the number of tickets required for each ride. This list should be in the same order as they are stored in the fair's **rides** array.

- When the server receives a **GET_WAIT_ESTIMATE** command from a guest, it will need to know which ride is being asked about. Assuming that the ride is a valid ride, it should then

send back a wait time (in seconds) that the guest is expected to wait for (assuming that he/she was to get in line at that time). The estimated wait time is calculated as follows:

$$\frac{\text{(\# guests waiting in line)}}{\text{(ride's capacity)}} \times \text{(ride's combined running/loading/unloading time)}$$

- When the server receives a **GET_IN_LINE** command from a guest, it will need to know the ride number as well as the guest's process ID. If the ride number is invalid or the ride lineup has reached MAX_LINEUP already, then the guest may not get in line. Otherwise, the guest should be added to that ride's lineup.

- When the server receives a **LEAVE_FAIR** command from a guest, it will need to know the process ID as well. It should find that guest in the fair's guests array and remove it.

- Finally, a **SHUTDOWN** command may come in. When this is received, it must cause the server to go off line (gracefully) and shut down. Make sure the code compiles, but you cannot test it until you do the next step.

(4) Write code in the **stop.c** file so that it attempts to shut down the fair server by sending the **SHUTDOWN** command to it. Test your code by running the **fairApp** in the background and then running the **stop** program. If all works well, the **fairApp** window should close and should shut down gracefully. Use **ps** to make sure that the **fairApp** process has indeed shut down properly as well as the **stop** program. Make sure that you don't have any segmentation faults.

(5) Complete the **runRide()** function in the **ride.c** file so that it begins with a status of **STOPPED** and has a countdown time of its **waitTime** upon start. Then, as long as the fair is still running, the ride should go into an infinite loop to operate the ride. The loop has a delay in it, which is 1/100th of a second. Do not change this. However, we will pretend that each 1/100th of a second is actually 1 second in reality. So, we will be running at a 100 times the speed of normal time … otherwise it would be painfully slow to test! Therefore, each time through the loop, you can assume that one second has passed in real life. You should use the **countdownTimer** to count down to zero. You will set it at various times to the **waitTime**, **rideTime** or **onOffTime**, depending on the ride's status .. and then simply count down until it gets to zero.

A ride should work as follows:

- If the ride is in a **STOPPED** state, it should determine if there is a guest in line. If there is one, it should go into **LOADING** state with a **countDown** time equal to the **onOffTime**. However, if there is at least one person who boarded the ride … and the maximum time for waiting has elapsed (i.e., the **countdownTimer** has reached **0**), then the ride should begin **RUNNING** with the **countdownTimer** set to the **rideTime**. If there is nobody in line and nobody on the ride waiting to ride, then there is nothing else to do in this state.

- If the ride is in a **LOADING** state, it should wait until the **onOffTime** has elapsed, at which point we can assume that the guest has boarded the ride. Once the time has elapsed, it should **signal** (with a **SIGUSR1**) the first guest in line (assuming that there is still a guest in line … who may have left for any possible reason) that he/she may begin boarding. It should then remove the guest from the **waitingLine** and put him/her into the list of **riders** for the ride. If the ride has reached its capacity of riders, the status should change to **RUNNING** with a timer count of **rideTime**, otherwise it should go back to a **STOPPED** state with a timer count of **waitTime** to wait for the next guest.

- If the ride is in a **RUNNING** state, it should wait until the **rideTime** has elapsed, at which point it should go into an **UNLOADING** state with an **onOffTime** to wait.
- If the ride is in an **UNLOADING** state, it should wait until the **onOffTime** has elapsed, at which point we can assume that the guest has gotten off the ride. Once the time has elapsed, it should **signal** (with a **SIGUSR2**) one of the riding guests that he/she may begin deboarding. It should then reset the timer so that another **onOffTime** seconds has elapsed before unloading the next guest. Once all guests have been unloaded, the ride should go into the **STOPPED** state with a **waitTime** on the timer.

Since the **requestHandler** and the **ride** threads are both manipulating the ride data at potentially the same time, you will want to make sure that you use a semaphore when there is a need to modify the shared information. You will need to go back to the **requestHandler** code as well to insert the appropriate semaphore code. At this point you can run the **fairApp** again and watch to ensure that the rides all count down their time to 0 and stay there. We cannot fully test what you wrote yet until we create the guest processes.

(6) Complete the **guest.c** program so that it takes two command line arguments. The first should be the number of tickets that the guest has (in the range from 5 to 40). The second should be the maximum time (in seconds) that the guest is willing to wait in line for a ride (in the range from 600 - 1200). The third should be the ride that this guest wants to go on first (in the range of 0 to NUM_RIDES).

The guest should then request admittance to the fair with the **ADMIT** command. If it is not allowed to be admitted, then the process should exit. Otherwise, it should store the ride **names** and **ticketsRequired** information that was sent back from the server.

The guest should then go into a loop, continuously getting onto rides until the guest has no more tickets, a which point it should send a **LEAVE_FAIR** request to the fair server.

The first ride to get on should be the one requested in the command line argument. All subsequent ride requests should be a randomly chosen ride that the guest is able to get on (based on the number of tickets he/she has). A ride should be chosen repeatedly, until one is found that the guest has enough tickets for … there will always be one such ride, since there is a ride that requires only 1 ticket. The guest should then send a **GET_WAIT_ESTIMATE** request to the server to find out how long of a wait there is for that ride. If the wait is unreasonable (i.e., more than the maximum time the guest is willing to wait (from command line arg) then the guest should try to choose another ride. Otherwise, it should request to **GET_IN_LINE** for that ride. If not allowed to get in ride for that ride, then it should choose another ride. Otherwise, the guest is assumed to be in line for that ride. Once in line, the guest simply waits for a SIGUSR1 signal indicating that they can board the ride. Then, the guest should simply wait for the SIGUSR2 signal indicating that the ride is over. At this point, the guest should choose another ride to ride on, assuming that he/she still has tickets remaining.

Test your program by running the **fairApp** in the background. Then run a single guest process with these parameters: `./guest 40 1200 0`. You should see the guest attempt to get on ride 0 (i.e., Tummy Tosser). Likely, the guest will not wait long in line and will board quickly, then ride, then unboard, then go onto another ride. If all goes well, the guest should keep riding until he/she runs out of tickets, at which point it should leave the fair.

(7) As a final step, complete the **generator.c** program so that it generates 100 guest processes with random tickets, willing wait times and preferred rides (see ranges in step 6) by using the **system()**

command. Once you get it working… run the **fairApp** in the background. Then run the **generator** in the background as well. You should see the guests being added to the lineups right away … as well as the rides loading, running and unloading. You should see the guest count increasing in the top left of the window. Eventually, all guests will run out of tickets and the rides should all be STOPPED. You can stop the program and use **ps** to make sure that all guest processes are gone. If you find that there are a lot of guest processes still running, then you will need to stop them all before re-testing again. You can use **pkill -f guest** to kill all the processes at once. Run the generator program 4 times in a row … you should see that some guests will not be admitted (since the fair's capacity is 300 guests).

IMPORTANT SUBMISSION INSTRUCTIONS:

Submit the following:

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).
