CPSC 359 – Winter 2022
Assignment 2
Concurrency & POSIX Threads
Due: February 14th @ 11:59PM
Weight: 8% of your final mark
Maximum points: 100

Objective: In this assignment, you will create C programs with POSIX threads.

This Assignment Lead TA: Mackenzie Bowal <mackenzie.bowal@ucalgary.ca>

Important Note: The assignment must be developed and tested on the university compute servers. Use the load balancer linux.cpsc.ucalgary.ca. Do not use arm.cpsc.ucalgary.ca. It is your responsibility to make sure that your code runs on linux.cpsc.ucalgary.ca.

Background:
Use the POSIX threads library in order to create two C programs:

1. The first program creates three threads: a main thread and two player threads. The player threads take turns playing a Number Scrabble game. The main thread creates the board, displays the board between player moves, decides the result of the game, and exits the program.

   Use a 3x3 board for this game. Player threads take turns randomly choosing a number between 1 and 9 and placing it on the board. A drawn number cannot be used twice by the same player. The player that completes a row, column, or diagonal that adds to 15 wins the game. See https://en.wikipedia.org/wiki/Number_Scrabble for details.

2. The second program creates a main thread and *n* "bidder" threads; the maximum value for *n* is 100. The program simulates secret bidding on an auctioned object. The main thread announces the winner of the bid.

Details:

- **Number Scrabble**: All three threads share a structure object, which contains the game board and a *turn* variable. The *turn* variable is used for synchronization between the three threads. The game board is a 3x3 array of integers. Initially all cells have the value 0. Player threads randomly enter their number in the board. The *turn* variable can have the values 0, 1, or 2. When *turn = 0* only the main thread can access the game board. Otherwise, when *turn* = 1 or 2, only player 1 or 2 can access the board. More specifically:

A global variable of type shared is needed (you can add more elements if you wish):

```
struct shared {
      char *board;
      int turn;
};
```

Main thread:

Initialize the game board // define a function
Set *turn* to 0 // no players can enter moves yet
Create two player threads with ids 1 and 2
Repeat the following // infinite loop (`while(1){ …}`)
`while (turn != 0) ;` // wait until *turn* becomes 0
Display the game board on the standard output // define a function
If (there is a winner || the game board is full)
game_on = false;
Cancel player threads // `thread_cancel()`
Determine and announce winner/draw // define a function
Exit the game
End If
Else
Set *turn* to the next player
End Repeat

Player thread (id): // this id is not of type `pthread_t`; it is an `int`
Repeat (infinite loop)
`while (turn != id);` // wait for my turn
Randomly generate an integer between 1 and 9
If the integer is used before by id thread id, try again
Randomly generate an integer between 0 and 8 //cell for my move
If that cell is taken, try again until the generated random number
corresponds to an empty cell
// Define a function *decideMove* for the previous two steps
Enter my move to the shared board
Set *turn* to 0 // Main thread's turn
End Repeat

Note that the thread id is passed through the create function, where `playGame` is a user-defined function that defines your top-level function for the thread:

```
pthread_create(&tid1,&attr,playGame,"1");
pthread_create(&tid2,&attr,playGame,"2");
```

For the termination of a thread, use: `pthread_cancel(tid1);`

- **Auction**: In the Number Scrabble game, the main thread cancels the player threads. In the Auction program, the main thread waits until all bidders enter their bids. The bidder threads keep on changing their mind about their bids. That is, they enter several bids before deciding on the final bid. Here too, we need a shared object that is also a structure.
  - The structure has at least four elements: an array to keep track of the bidding action: *bid[i] == j* when bidder i has j as a bid. The bid j is a random number.
  - A Boolean array: committed[i] is true when the main thread can accept the bid of bidder i (signals that bidder i has entered their final bid)
  - A Boolean variable *auctionFinished* signaling the end of the auction. The auction ends when all bidders commit their bids (committed is true for all bidders.)
  - The fourth element is the array *bidTime* that records the time a bidder commits to their bid, where *bidTime[i] == -1* if bidder i has not finished bidding yet (did not commit); it has a time value otherwise. You can use the `time_t` type of time.h. The C `time` function returns a value of type `time_t`.

  Threads simulate the bidding process by sleeping for a random number of seconds, then making a bid. The bidder randomly decides to commit its bid. The main thread periodically displays the progress of each bidder as an arrow (clear your screen from within your C program each time you display the progress to get the illusion that these arrows are moving!):

```
bidder0: ------->
bidder1: --->
bidder2: ------*
…
```

  In the above example *bidTime[0] == -1* and *bidTime[1] == -1*, but bidder2 has committed the bid and *bidTime[2]* should contain the time when it committed the bid.
  The auction winner is the bidder with the highest bid value, with ties broken by smallest bid time.

  More specifically:
  Main thread:
  Initialize the shared object
  Initialize an upper bound on temporary bids (a thread must commit a bid before reaching this bound) // users choose a value appropriate for their screen
  Create *bidder* threads
  while (!*auctionFinished)*
  Display the bidding progress // define a function
  If there are bidders who committed their bids, then rank them
  auctionFinished = true when all bids are committed
  End while
  Join all bidder threads

Display the final ranking in descending order starting with the winner
Exit

Bidder thread(id):
While (i < bound on bids – 1 and !committed[id])
Sleep for random seconds // `sleep(seconds)`
bid[id]= random bid// enter a temporary bid
Randomly decide if this bid is committed (committed[id] = true)
End - While
bidTime[id] = `time()`;
Exit

Discussion:
In this Auction implementation, there is what we call a *race condition*, which takes place when two or more threads access the same object and at least one of them changes it. Specifically, the main thread can be reading *bid[id]* or *bidTime[id]* when thread *id* is writing its value there. Race conditions can be dangerous in general. However, in this case, they are benign. It is either that the main thread will see an up-to-date value or not. If it does not see it, then it is equivalent to the situation that the write happened immediately after the main thread finished its read. This will not affect your game logic.
However, in the Number Scrabble game, a race condition would be very serious and can affect the game logic. For example, if both players check the game board at the same time and choose the same empty cell, they can both enter their moves in that same cell, which should not be allowed. Note how we disallowed in our design two threads to access the game board at the same time by using the *turn* variable. Hence, we eliminated race conditions when accessing the game board. The variable *turn* can have only one value, allowing only one thread to access the game board. However, sometimes, it is possible for *turn* to have more than one value! For example, this is possible if the underlying architecture allows the different threads to have their own (cached) copies of *turn*. Due to performance reasons, these copies may not be kept consistent all the time. So, it may be possible for player1's copy of *turn* to be 1 and player2's copy to be 2, allowing both to think it is their turn. Luckily, for this course, you do not need to worry about such "weird" behaviors!

Deliverables:
Submit two C programs and two typescripts, one for the Number Scrabble game and the second for the Auction. Submit four separate unzipped files. Name your programs *game.c* and *race.c*. Name your type scripts *game.script* and *auction.script*.

Marking Guide:
Number Scrabble

Code compiles	5
Code runs (no RT errors)	5

Three threads	15
Shared object	5
Synchronization	5
Random numbers	5
Game logic	5
Modularity & documentation	5
Auction	
Code compiles	5
Code runs (no RT errors)	5
n+1 threads	15
Shared object	5
Random sleeping/bidding	5
Progress Display	5
Display bid rankings	5
Modularity & documentation	5
Total	**100**

Teams: You are advised to work with another student in class in order to complete the assignment, but you are not required to do so. Peer evaluation in teams may be conducted.

Submission: Submit your game.c, auction.c, game.script, and auction.script files to the appropriate dropbox on D2L. Do not combine the files into a ZIP folder. Further submission requirements/instructions may be imposed by the lead TA.

Late Submission Policy
Late submissions are penalized as follows:

- 12.5% are deducted for each late day or portion of a day

Hence no submissions are accepted 8 days after the deadline

Late Submission Policy: Late submissions will be penalized as follows:
-12.5% for each late day or portion of a day.
Hence, no submissions will be accepted after 8 days (including weekend days) of the announced deadline

Academic Misconduct: Any similarities between submissions will be further investigated for academic misconduct. Your final submission must be your team's original work. While you are encouraged to discuss the assignment with colleagues outside your team, this must be limited to conceptual and design decisions. Code sharing outside your team by any means is prohibited, including looking at someone else's paper or screen. Any re-used code of excess of 5 lines in C and MAL (5 MAL instructions) must be cited and have its source acknowledged. Failure to credit the source will also result in a misconduct investigation.

D2L Marks
Marks posted on D2L are subject to change (up or down).