Project 4: Monte Carlo Simulation with Threads

This assignment will extend your practice working with the POSIX threads API. Your task is to run a series of Monte Carlo simulations to determine the probability of different events.

Program Flow

The main thread will read the one command line argument (the name of a file containing any files needed for the simulation) and read the contents of those files into memory. Then, the main thread will create some number of threads and pass each thread information it needs to perform the simulation. After the main thread creates all of the other threads, it will wait for each thread to finish using pthread_join. Each worker thread will perform the actual simulation of the event. Once all threads have finished, the main thread will print the calculated probability of the event.

The interesting part of this assignment is that you will need to create your own architecture/API for running simulations of events (in a thread-safe way of course). This means coming up with data structures to hold onto the card lists, decks of cards, and the API needed to pass those structures to your worker threads.

The difference between simulating an event and calculating it…

Any of the events listed in this assignment can be calculated directly. Meaning, there is a (sometimes annoying or tedious) mathematical calculation you can perform that will tell you the EXACT probability of the event. If you WANT to figure out some of these calculations, you're more than welcome to. However, keep in mind that a direct calculation is different than a simulation. A calculation will tell you the true answer, whereas a Monte Carlo simulation is all about simulating an event a very large number of times to asymptotically approach the true probability of that event.

This assignment is asking you to do a simulation, NOT a calculation. **Therefore, if you use a direct calculation as an answer to one of the tests, even if Moodle shows 100/100, you will fail that test (meaning, the TAs will be manually inspecting everyone's work).** The point of this assignment isn't to give you problems that you will calculate directly. Rather, it's to present you with events whose probability are often annoying or difficult to calculate directly, and for you to simulate the event instead.

What POSIX Threads calls will be needed?

I STRONGLY SUGGEST YOU READ THESE SECTIONS OF THE TEXTBOOK: 27.1, 27.2
https://pages.cs.wisc.edu/~remzi/OSTEP/threads-api.pdf

There are only two thread API calls you should need to make:

> **pthread_create** -> Starts a new thread in the calling process, by invoking the function passed to it
> https://man7.org/linux/man-pages/man3/pthread_create.3.html
>
> **pthread_join** -> Waits for a specified thread to terminate
> https://man7.org/linux/man-pages/man3/pthread_join.3.html

You do not need any synchronization functions/primitives in this assignment as all of the threads will be incrementing their own success count, and sending it back to the main thread.

What are the input files, and what information do they contain?

There are three different types of input files:

- **SimSetup.txt**: This file specifies the paths of the card list and deck files needed for the simulation. It also picks which simulation event to run, and how many threads to use. The contents are listed in the following order:
    - Number of files to read in (the first file will always be the card list, subsequent files will be the decks)
    - A number to identify which event should be tested
    - A number to specify how many threads to use
    - The path of the Card List file
    - The path of each of the Deck List files

- **Card List File:** This file contains a single set of unique cards that will be used for a particular simulation. Since these cards are unique, think of them as having a unique ID based on their order in the file. For example, the first card would have an ID of 0, then the next 1, etc. The contents of the files are listed in the following order:
    - Number of unique cards total in the card set
    - Number of attributes per card
    - Each unique card, one per line, as a list of attributes

  Reminder: the cards will not have a LISTED ID in the card list file; they're implicit in the order.

- **Deck List Files:** These files contain the decks that will be used to simulate the events. Each deck contains only two fields: the number of cards in the deck, and the IDs of the cards in the deck (one card per line). The IDs correspond to the cards from the card list file.

EXAMPLE INPUT FILES:

- SimSetup.txt specifies that event number 7 will be simulated (first line), 3 files will be used for input (second line), and 4 threads will be used for this simulation (third line). The path for each file is listed as well.
- CardList.txt below has 52 cards (first line), with 4 attributes for each card (second line). The first 5 cards are shown; each card is on its own line with 4 integers representing the attributes of the card.
- Deck_1.txt has 6 cards (first line) as a list of IDs corresponding to the card list.
- Deck_2.txt contains 5000 cards (first line) (not all shown) as a list of IDs corresponding to the Card List.
- Notice that each deck can have duplicates of cards, but the cards in the card list are unique.

//SimSetup.txt	//CardList.txt	//Deck_1.txt	//Deck_2.txt
7	52	6	5000
3	4		
4		8	35
	1 0 0 0	8	35
CardList.txt	2 0 0 0	10	35
Deck_1.txt	3 0 0 0	11	35
Deck_2.txt	4 0 0 0	50	35
	5 0 0 0	50	35
	…		35
			…

How to run a simulation (high level)

Once you've loaded in the data needed to simulate a particular event, you must now run the Monte Carlo simulation:

- Create structures that will allow you to pass the relevant information to each thread
    - This will certainly involve allocating memory, sometimes per-thread
- Use pthread_create to invoke your threads, passing them the needed data
    - Remember to divide the work (more or less) evenly among all threads
- Each thread will simulate an event some number of times, and record the number of successes
- The main thread will use pthread_join to wait for each worker thread to finish
- The number of successes from each thread will be added together to determine the probability of the event
- And of course… clean up any resources, such as memory allocated per thread, etc

Your code will need to choose which unique event to simulate (based on the input file). This means you'll need to write code that is generic enough that adding new events doesn't entail writing LOTS of new code. To facilitate this, you might consider two approaches:

- Put a function pointer on your per-thread data, so that the threads know which unique event to simulate

- Put an index on your per-thread data, used to access an array of function pointers for each unique event

How to run a SPECIFIC simulation

Once each thread has its needed data, you'll need to use that data to simulate an event. For example, to simulate how often drawing 1 card from a standard deck of playing cards results in a red card, you'll need to:

- Shuffle the deck of cards
- Draw a card
- Determine if the card is red
- If red, record a success
- Repeat the steps above some enormous number of times (in the millions)

Other types of events will require more careful consideration and planning of how to use the data to simulate the event truthfully. For example, other events might involve drawing more than one card (or close to the whole deck!), or using more than one deck at a time. You're responsible for figuring out what this code looks like (but of course, feel free to ask the instructor and TAs for help).

Passing Information Around

You will not be using any global data in your program (meaning, data declared outside of your functions). All of the necessary data for the threads will be passed to them via the last argument of pthread_create. You will need to create a structure that contains all the information that a thread will need. The definition for this structure will be put in **mc-header.h**. You must figure out what this structure is on your own. Everyone doing this assignment may create a structure that looks slightly different.

Take special care when passing data to each thread. For example, **each thread will need its own copy of each deck**; if each thread is shuffling and drawing from the same deck, your simulation results will be off.

RNG, and shuffling the deck

To shuffle a deck of cards you'll need a random number generator. I have included a PRNG that you can use to generate random numbers that is thread safe. If you use rand/srand or the PRNG we used for the cache list assignment you will get incorrect results. Hence, the warning below:

WARNING: YOU MUST USE THE THREAD-SAFE PRNG WHEN YOUR SIMULATION RUNS ON THREADS, OTHERWISE YOU WILL GET INCORRECT RESULTS OR SEGFAULT!

The thread-safe PRNG is slightly more annoying to use. You must create a randData struct that is passed into the thead-safe functions. You'll need to create a new instance of this randData struct FOR EACH THREAD. They don't necessarily need to be PASSED to the threads, or live on the heap; as long as each thread creates its own randData struct, it can create/seed its own random number generator.

How many threads should I use?

In the previous project, one thread was responsible for a single number in the output matrix. This project is different; one of the fields in SimSetup.txt will determine how many threads you will use to split up the work of the simulation. Your solution should be able to handle an arbitrary number of threads without crashing. The maximum number of threads I will use when running tests on Moodle is 100.

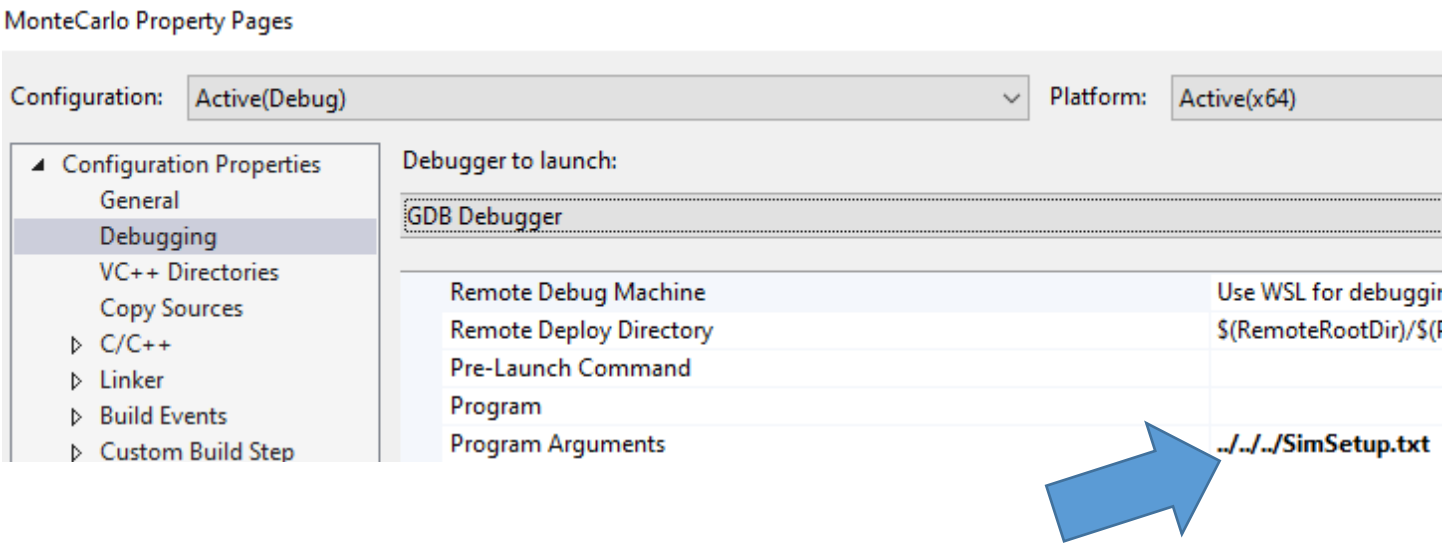
Running the Program with Command Line Arguments

To run the program, you will need to supply it with its own command line argument, like so:

```
./main-thread SimSetup.txt
```

Where the first parameter is the PATH to the text file that lists all other file paths.

To facilitate this in Visual Studio, open the project properties and go to the Debugging section of Configuration Properties. You can edit the command line arguments like so, **remember to replace them with your own, depending on the path to your local SimSetup.txt file**:

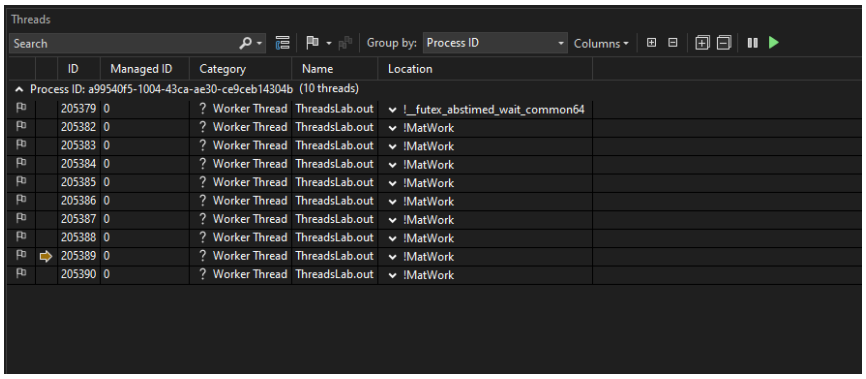


Compiling/Running your Code, and Memory Leaks (Valgrind)

The instructions from previous assignments still hold true. If you're having trouble, see the instructor or TAs for help.

Debugging

When debugging a multi-threaded application, breakpoints in Visual Studio won't work the same way; since threads are executing concurrently, program execution will appear to jump around, and you may hit a breakpoint more times than expected. Go to Debug->Windows->Threads, to see the threads associated with your process. Whenever you hit a breakpoint, you should be able to see which thread the debugger is currently stopped on. Below, you can see that there are 10 threads; the first thread is the main thread, waiting on the workers. The other 9 threads are the workers:



Project Files

File	Description
main.c	Source for the main thread, which will create the worker threads. This file contains **main**.
mc-thread.c	Source for the function that will handle the simulation. This will contain the function that the main thread will pass to pthread_create.
mc-header.h	Header that will contain the structure used to communicate info to the threads, and the declaration for the thread function that will handle the simulation.
ThreadSafe_PRNG.c/h	A thread-safe random number generator that will be used primarily for shuffling decks.
SimEvents (directory)	Input files including the sim setup, card lists, and deck lists

Deliverables/Submission

You must submit three files to the appropriate Moodle VPL assignment:

- **main.c**
- **mc-thread.c**
- **mc-header.h**

When submitting to Moodle, there will be an option to evaluate your submission. Press this button to see the results of the tests (the same tests included via the driver file). If your program passes all of the tests, you should see a 100/100 grade. YOU DO NOT NEED TO DO ANYTHING AFTER YOU SEE THIS GRADE. Your submission has been accepted and your grade was recorded on Moodle. If you do not pass all of the tests, VPL will show you the diff between the expected output and your program's output. You may submit an unlimited number of times before the due date.

What kinds of events must be simulated?

Listed below are twelve (12) different kinds of events, across three different types of "cards". You must run a Monte Carlo simulation to determine the probability of each event. To facilitate the running of different events, a number contained in the initial input file will be used to pick which event to simulate. For example, to run Event 5, the number 5 would be specified in the first line of SimSetup.txt.

The SimSetup.txt file has all of the events and their corresponding setups. To run a particular event, just copy and paste its information above the line denoted in the file.

Standard Playing Cards -> 1_PlayingCards.txt

Represents the standard set of 52 playing cards. The attributes of these cards in order are:

- **Value:** The number value of the card, with aces as 1, jacks as 11, queens as 12, and kings as 13.
- **Color:** either 0 or 1 to represent red or black; choice doesn't matter as long as you're consistent.
- **Suit:** spades, clubs, hearts, diamonds. How you pair each suit number (0 - 3) with the card list doesn't matter as long as your choices are consistent with color (spades and clubs are black, hearts and diamonds are red).
- **Face card:** 0 means this is NOT a face card, 1 means it is. Jacks, queens, and kings are face cards.

Standard Playing Cards EVENTS

Event 1

Draw five cards from the deck. How often will you encounter a royal flush? (Yes, ace is high)

Event 2

Draw five cards from the deck. How often will you encounter 4 of a kind? (Four cards of the same value).

Event 3

Draw seven cards from the deck. How often will you encounter all four suits OR two pairs of face cards?

Event 4

Looking through the entire shuffled deck, how often will you encounter at least 30 switches between red and black? If the color of a card is different from the color of the previous card that constitutes a switch.

Pokémon Party -> 2_Pokemon.txt

This list represents the first 151 Pokémon in the Kanto region. The attributes of these "cards" in order are:

- **Family:** Number denoting which family this Pokémon belongs to.
  Example: Charmander, Charmeleon, and Charizard are all in the same family.

- **Type 1:** Value between 0 and 17 representing Pokémon Type.
- **Type 2** A second value between 0 and 17. The value of this attribute is -1 if a Pokémon has only one type.

The types are as follows (note that there aren't any Dark-type Kanto Pokémon, so you won't see any with a type of 16):

Normal	Fighting	Flying	Poison	Ground	Rock	Bug	Ghost	Steel	Fire	Water	Grass	Electric	Psychic	Ice	Dragon	Dark	Fairy
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Example: Blastoise is a Water type, so its Type 1 attribute has a value of 10, and its Type 2 attribute has a value of -1. Charizard is a Fire/Flying type, so its Type 1 attribute has a value of 9, and its Type 2 attribute has a value of 2.

- **Evolution Stage** (0 -> first stage, 1 -> second stage, 2 -> third stage)
  Example: Charmander -> 0 Charmeleon -> 1, Charizard ->2
  Pikachu -> 0, Raichu -> 1 (Pikachu evolves into Raichu)
  Snorlax -> 0 (Snorlax doesn't evolve into anything, nothing evolves into Snorlax)

To calculate a type matchup, you can feed the types of the attacking and defending Pokémon into the function called TypeChartXAttackYFull. This function will return a float, specifying the effectiveness of one Pokémon against another.

Pokémon Party TESTS

Event 5

Draw three cards from each deck. How often will one of the players draw only evolution stage 0 Pokémon, one of the players draws only stage 1, and one of the players draws only stage 2?

Event 6

Draw two cards from each deck. How often will one of the Pokémon be unable to attack one of the Pokémon drawn by the other two players, due to type immunity?
(Example: A Pokémon with Electric as its only type can't attack a Ground type.)

Hint: to calculate type matchup, feed the attacking/defending types into TypeChartXAttackYFull, which will return a float (0 -> Immune, 0.25 or 0.5 -> Resistant, 1 -> Standard Effectiveness, 2 or 4 -> Super Effective)

Event 7

A starting hand of 7 cards is drawn from each deck. What is the probability that outcomes 1 and 2 (below) happen simultaneously?

- 1. Player of the first deck has either all 3 evolution stages of a single family, OR has two individual stage 0 Pokémon of any family (duplicates are fine)
- 2. Player of the second deck does NOT fulfill the conditions in 1

Event 8

Draw 100 cards. What is the probability that you will see 8 Pokémon with evolution stage 1 or 2 that also have Water as their ONLY type and that you simultaneously see 0 Pokémon that are Water AND another type?
Water as only type → (Type1 = 10, Type2 = -1), Water and another type → (Type1 or Type2 = 10, Type2 != -1)

Dungeon Tiles -> DungeonTiles.txt

- **Tile:** Represents a single "card" in the deck of tiles.
- **Square:** one location on a tile. A tile is a 3x3 grid of squares.

One tile, with 9 squares:

[ ] [ ] [ ]
[ ] [ ] [ ]
[ ] [ ] [ ]

Each tile in the "deck of tiles" has 9 attributes, with each attribute corresponding to a single square on the tile itself:

[Attribute 0] [Attribute 1] [Attribute 2]
[Attribute 3] [Attribute 4] [Attribute 5]
[Attribute 6] [Attribute 7] [Attribute 8]

These attributes can have a value of 0 or 1:

- **0 ->** This space is blank/empty
- **1 ->** This space is a wall

For example, the tile to the right has a wall in the top left corner, a wall in the middle row, and a wall in the bottom row, with the rest of the spaces empty:

[1] [0] [0]
[0] [1] [0]
[0] [1] [0]

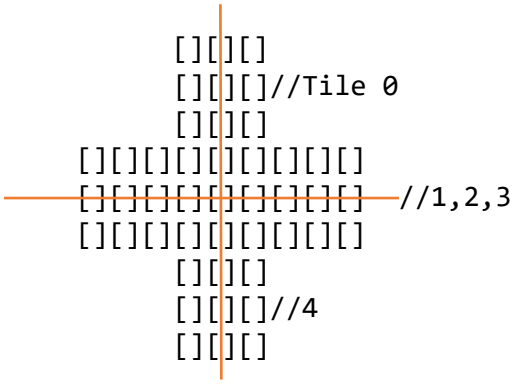
This entry/row in the "card list" file would appear like so:

1 0 0 0 1 0 0 1 0

Dungeon Tiles Tests

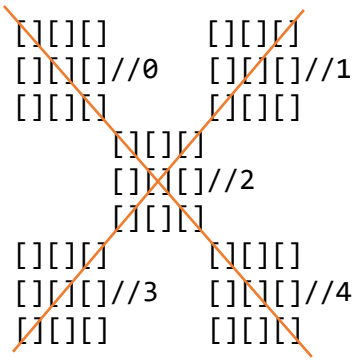
Event 9

Draw 5 tiles, and arrange them like the picture on the right. Tiles are ordered top to bottom, left to right. What is the probability that a cross shape can be drawn through the middle row/column of squares? (NOTE: All spaces used for this shape must be blank (meaning 0)).



Event 10

Same test as above, except an X shape like the picture on the right:



Event 11

Lay out all of the tiles in the deck horizontally. What is the probability that there is at least one continuous horizontal line of squares at least 100 squares in length ANYWHERE in the chain of tiles? (The example below is looking at the middle row, but the 100 squares could happen in ANY of the 3 rows.)

EXAMPLE:

```
[0][1][1] [0][1][0] [0][0][1] [1][0][0] [0][0][1] [1][0][0]      Longest line so far: 7
[0][0][0] [1][0][0] [0][0][0] [0][0][1] [1][1][1] [0][1][0]      If a line of length 100
[1][1][1] [0][0][1] [0][0][0] [0][0][1] [1][1][0] [0][1][0]…     is found, success!
```

Event 12

Draw 4 tiles from the first deck, and four tiles from the second deck arranging them like the picture below. What is the probability that at least one square in C2 of the first deck is open, and the corresponding square in Column 0 of the second deck is also open? The example below is successful because at least one OPEN was found.

```
//First deck       //Second deck
       C2               C0
[0][1][1]   Closed [0][1][1]
[0][0][0]   Open   [0][1][0]
[1][1][1]   Closed [1][1][1]

[0][0][0]   Closed [1][1][1]
[0][0][0]   Closed [1][0][0]
[1][0][0]   Closed [1][1][1]

[0][1][0]   Open   [0][1][1]
[0][0][1]   Closed [0][0][0]
[1][1][0]   Open   [0][1][1]

[0][0][1]   Closed [0][1][1]
[0][0][1]   Closed [0][0][0]
[0][0][1]   Closed [1][1][1]
```