Project 3: Duck Park

CS 415 - Operating Systems
Spring 2025 - Prof. Allen Malony
Due date: *11:59 pm, Friday June 6th*

Introduction

In this project you will build a discrete element distributed simulation based on the roller coaster synchronization problem. You'll simulate an amusement park where threads enter, explore the park, obtain tickets, queue for rides, board roller coasters, disembark, and return to the park environment.
In this project you will build a discrete element distributed simulation based on the roller coaster synchronization problem. You'll simulate an amusement park where threads enter, explore the park, obtain tickets, queue for rides, board roller coasters, disembark, and return to the park environment.

Project Details

Implement a solution to the roller coaster synchronization problem with the following parameters:

Suppose there are n passenger threads and a car thread. The passengers repeatedly wait to take rides in the car, which can hold C passengers, where $C < n$. The car can go around the tracks only when it is full.

Required synchronization operations:

- Passengers should invoke `board` and `unboard`.
- The car should invoke `load`, `run` and `unload`.

Synchronization constraints:

- Passengers cannot `board` until the car has invoked `load`.
- The car cannot depart until C passengers have `boarded`.

- Passengers cannot `unboard` until the car has invoked `unload`.

To generalize the solution to more than one car, we have to satisfy some additional constraints

- Only one car can be boarding at a time.
- Multiple cars can be on the track concurrently.
- Since cars can't pass each other, they have to unload in the same order they boarded.
- All the threads from one carload must disembark (`unload`) before any of the threads from subsequent carloads.

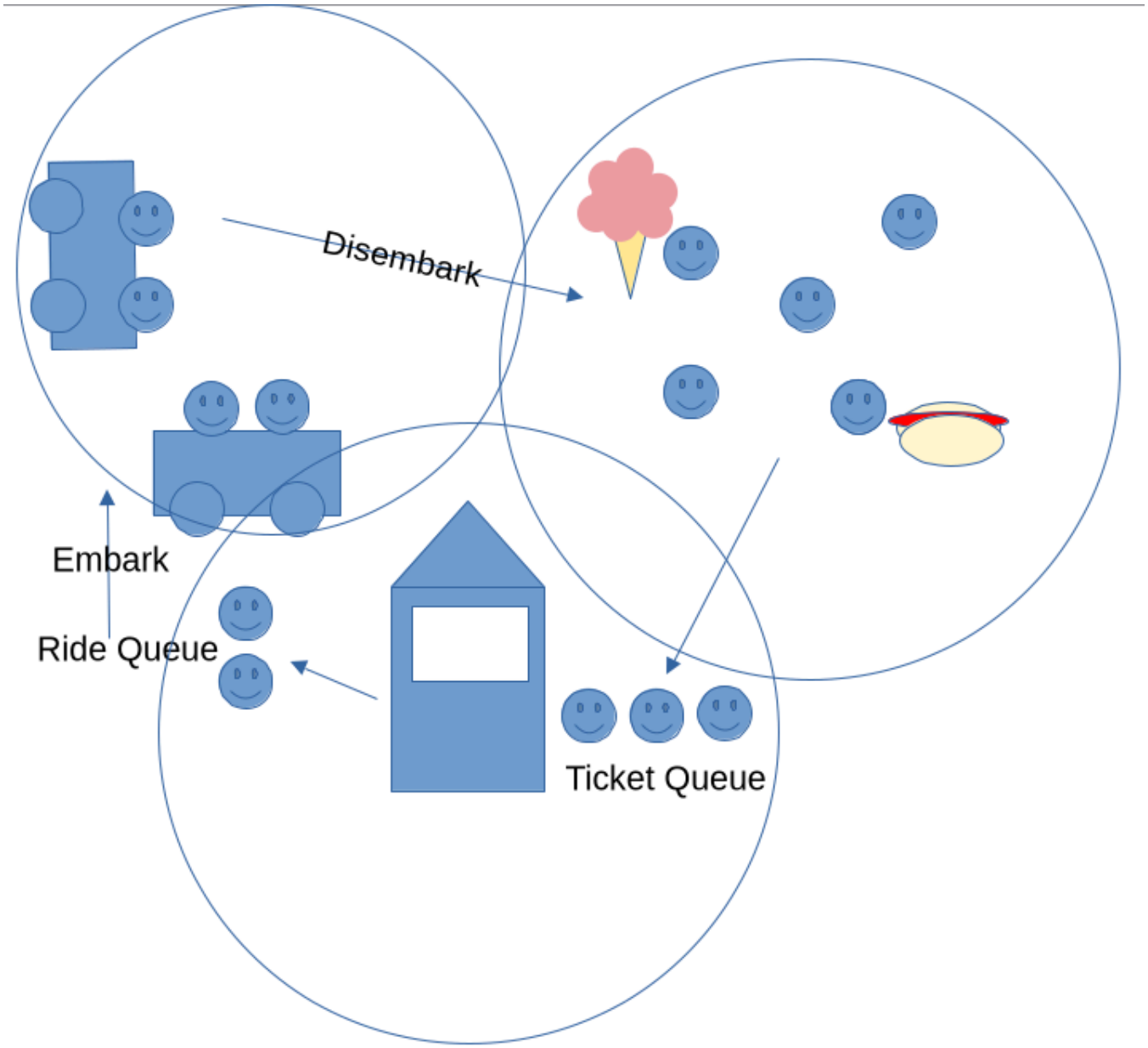
Build the amusement park simulation

- Threads enter the park and sleep for some time before wanting to ride the coaster (maybe they are playing games, eating funnel cake, etc.).
- Threads need to get a ticket from a ticket booth. Only one thread can get a ticket at a time.
- Threads can load on their own (instantaneous).
- Threads will wait in the car until the car is running

Implement these additional features:

- If only one thread is in the park and ready to ride, the car should depart without waiting.
- Add a timer to make partially-filled cars depart after a set waiting period.
- Use time intervals for:
    - Park exploration (sleeping)
    - Car waiting period
    - Ride duration
- Use random time intervals for:
    - Threads waiting in the park (1-10 sec)

Illustration:



The figure displays three circles encapsluating the three states of the project: park, ticket, ride. In the first circle, there are four smiley faces representing threads that are hanging out in the park. From this group of smiley faces, and arrow points from them to the second conjoined circle where there is a queue of smiley faces at a ticket booth. This is labeled ticket queue. From the booth there is an arrow pointing to the third "ride" state that is labeled "embark". Here, we see two roller coasters waiting and one smiley face in the first car waiting. **note** The ride queue represents the processes that are waiting to get *in* the car. There is also a full roller coaster depicted to be running on the track. There is a final arrow form the ride element back to the park labeled "disembark".

Function & lib requirements:

Library

- <pthread.h>

Functions you need

- `pthread_create()`
- `pthread_exit()`
- `pthread_join()`
- `pthread_mutex_lock()` and/or `sem_wait()`
- `pthread_mutex_unlock()` and/or `sem_post()`

Part 1: Single Threaded Solution

Develop and test each component with a single thread to verify basic functionality.

Part 2: Multi-threaded Solution

Test your implementation with multiple threads entering and exiting at varying times to ensure correct synchronization.

Part 3: Monitor

Create a monitoring system that:

- Reports on the state of all queues (ticket and ride)
- Regularly samples and displays system state
- Implements inter-process communication using pipes or mmap

Deliverables

Code that utilizes Pthreads to implement the discrete element distributed system.

Terminal output that reflects the state of the system.
With monitoring, output will show the states of the queues.

Your output does **not** have to match exactly, but it will include the **time** and be neat and descriptive. It should demonstrate that your simulation working seamlessly with parallelism and synchronicity.

Report: Write a 1-2 page report on your project using the sample report collection format given. Your report could also discuss your synchronization approach, potential deadlocks and how your avoided them, and how you tested edge cases. Explain design decisions and trade-offs made. Feel free to go over the limit if you wish. Report format and content suggestions are given in the report collection template. If you are not able to complete all 3 parts, state in your report which part you finished, so partial credit can be given.

Structure Requirements

For a project to be accepted, the project must contain the following files and meet the following requirements: (You must use the C programming language with the pthread library for this assignment. No projects written in another programming language will be accepted.)

park.c: this is the main program

Makefile: Your project must include a standard make file. It must produce the executable with the following names: **park**

How to run:
USAGE:
./park [OPTIONS]

OPTIONS:
-n, -INT Number of passenger threads
-c, INT Number of cars
-p, INT Capacity per car
-w, INT Car waiting period in seconds
-r, INT Ride duration in seconds
-h, --help Display a help message

EXAMPLE:
./park -n 30 -c 4 -p 6 -w 8 -r 10

Use the `getopt()` function in main. See appendix for details.

Include **default options** that will work without defining the flags.

What to output: Terminal output showing your simulation running in real time.

Report: in pdf format

Note: Additionally, you are allowed to add any other *.h and *.c files you wish. However, when we run your code we will only be running the server file. Make sure your code runs in the VM before submission.

Example output:

```
===== DUCK PARK SIMULATION =====
[Monitor] Simulation started with parameters:
- Number of passenger threads: 20
- Number of cars: 3
- Capacity per car: 5
- Park exploration time: 2-5 seconds
- Car waiting period: 10 seconds
- Ride duration: 8 seconds

[Time: 00:00:01] Passenger 1 entered the park
[Time: 00:00:01] Passenger 2 entered the park
[Time: 00:00:02] Passenger 3 entered the park
[Time: 00:00:02] Passenger 4 entered the park
[Time: 00:00:02] Passenger 1 is exploring the park...
[Time: 00:00:03] Passenger 5 entered the park
[Time: 00:00:03] Passenger 2 is exploring the park...

[Monitor] System State at 00:00:05
Ticket Queue: []
Ride Queue: []
Car 1 Status: WAITING (0/5 passengers)
Car 2 Status: WAITING (0/5 passengers)
Car 3 Status: WAITING (0/5 passengers)
Passengers in park: 5 (5 exploring, 0 in queues, 0 on rides)

[Time: 00:00:06] Passenger 1 finished exploring, heading to ticket booth
[Time: 00:00:06] Passenger 1 waiting in ticket queue
[Time: 00:00:06] Passenger 1 acquired a ticket
[Time: 00:00:06] Passenger 1 joined the ride queue
[Time: 00:00:07] Passenger 3 is exploring the park...
[Time: 00:00:08] Passenger 2 finished exploring, heading to ticket booth
[Time: 00:00:08] Passenger 2 waiting in ticket queue
[Time: 00:00:08] Passenger 2 acquired a ticket
[Time: 00:00:08] Passenger 2 joined the ride queue
[Time: 00:00:08] Passenger 4 is exploring the park...

[Time: 00:00:10] Car 1 invoked load()
[Time: 00:00:10] Passenger 1 boarded Car 1
[Time: 00:00:10] Passenger 2 boarded Car 1
```

```
[Time: 00:00:10] Passenger 5 is exploring the park...

[Monitor] System State at 00:00:10
Ticket Queue: []
Ride Queue: [Passenger 1, Passenger 2]
Car 1 Status: LOADING (2/5 passengers)
Car 2 Status: WAITING (0/5 passengers)
Car 3 Status: WAITING (0/5 passengers)
Passengers in park: 5 (3 exploring, 0 in queues, 2 on rides)

[Time: 00:00:11] Passenger 6 entered the park
[Time: 00:00:12] Passenger 7 entered the park
[Time: 00:00:13] Passenger 3 finished exploring, heading to ticket booth
[Time: 00:00:13] Passenger 3 waiting in ticket queue
[Time: 00:00:13] Passenger 3 acquired a ticket
[Time: 00:00:13] Passenger 3 joined the ride queue
[Time: 00:00:13] Passenger 3 boarded Car 1
[Time: 00:00:14] Passenger 4 finished exploring, heading to ticket booth
[Time: 00:00:14] Passenger 4 waiting in ticket queue
[Time: 00:00:14] Passenger 4 acquired a ticket
[Time: 00:00:14] Passenger 4 joined the ride queue
[Time: 00:00:14] Passenger 4 boarded Car 1
[Time: 00:00:15] Passenger 5 finished exploring, heading to ticket booth
[Time: 00:00:15] Passenger 5 waiting in ticket queue
[Time: 00:00:15] Passenger 5 acquired a ticket
[Time: 00:00:15] Passenger 5 joined the ride queue
[Time: 00:00:15] Passenger 5 boarded Car 1

[Time: 00:00:15] Car 1 is full with 5 passengers
[Time: 00:00:15] Car 1 departed for its run

[Monitor] System State at 00:00:15
Ticket Queue: []
Ride Queue: []
Car 1 Status: RUNNING (5/5 passengers)
Car 2 Status: WAITING (0/5 passengers)
Car 3 Status: WAITING (0/5 passengers)
Passengers in park: 7 (2 exploring, 0 in queues, 5 on rides)
```

```
[Time: 00:00:16] Passenger 6 is exploring the park...
[Time: 00:00:17] Passenger 7 is exploring the park...
[Time: 00:00:18] Passenger 8 entered the park
[Time: 00:00:19] Passenger 9 entered the park
[Time: 00:00:20] Car 2 invoked load()

[Monitor] System State at 00:00:20
Ticket Queue: []
Ride Queue: []
Car 1 Status: RUNNING (5/5 passengers)
Car 2 Status: LOADING (0/5 passengers)
Car 3 Status: WAITING (0/5 passengers)
Passengers in park: 9 (4 exploring, 0 in queues, 5 on rides)

[Time: 00:00:21] Passenger 6 finished exploring, heading to ticket booth
[Time: 00:00:21] Passenger 6 waiting in ticket queue
[Time: 00:00:21] Passenger 6 acquired a ticket
[Time: 00:00:21] Passenger 6 joined the ride queue
[Time: 00:00:21] Passenger 6 boarded Car 2
[Time: 00:00:22] Passenger 7 finished exploring, heading to ticket booth
[Time: 00:00:22] Passenger 7 waiting in ticket queue
[Time: 00:00:22] Passenger 7 acquired a ticket
[Time: 00:00:22] Passenger 7 joined the ride queue
[Time: 00:00:22] Passenger 7 boarded Car 2
[Time: 00:00:23] Car 1 finished its run
[Time: 00:00:23] Car 1 invoked unload()
[Time: 00:00:23] Passenger 1 unboarded Car 1
[Time: 00:00:23] Passenger 2 unboarded Car 1
[Time: 00:00:23] Passenger 3 unboarded Car 1
[Time: 00:00:23] Passenger 4 unboarded Car 1
[Time: 00:00:23] Passenger 5 unboarded Car 1
[Time: 00:00:23] Passenger 1 is exploring the park...
[Time: 00:00:23] Passenger 2 is exploring the park...

[Time: 00:00:25] Passenger 8 is exploring the park...

[Monitor] System State at 00:00:25
Ticket Queue: []
Ride Queue: [Passenger 6, Passenger 7]
```

```
Car 1 Status: WAITING (0/5 passengers)
Car 2 Status: LOADING (2/5 passengers)
Car 3 Status: WAITING (0/5 passengers)
Passengers in park: 9 (7 exploring, 0 in queues, 2 on rides)

*** Simulation continues... ***

[Monitor] FINAL STATISTICS:
Total simulation time: 00:05:00
Total passengers served: 42
Total rides completed: 12
Average wait time in ticket queue: 1.2 seconds
Average wait time in ride queue: 3.5 seconds
Average car utilization: 78% (3.9/5 passengers per ride)
```

Submission Requirements

Once your project is done, do the following:
Your executable should be able to run with the following command "./park [options]

1. Open a terminal and navigate to the project folder. Compile your code in the VM with the -g, -pthread, and - lpthread flag.
2. Run your code and take screenshots of the output as necessary (of each part).
3. Create valgrind logs of each respective part:
   a. “valgrind --leak-check=full --tool=memcheck ./a.out > log*.txt 2>&1 ”
4. Tar/zip the project folder which should contain the following directories and content. Your project should have part1, part2 and part3 folders.

- a. part1
    - park.c
    - ii. Any additional header file and their corresponding ".c" file
    - iii. Makefile
    - iv. Output (directory)
    - v. valgrind log
- b. part2

        - i. park.c
        - ii. Any additional header file and their corresponding ".c" file
        - iii. Makefile
        - iv. Output (directory)
        - v. valgrind log
- c. part3
        - i. park.c
        - ii. Any additional header file and their corresponding ".c" file
        - iii. Makefile
        - iv. Output (directory)
        - v. valgrind log

Valgrind can help you spot memory leaks in your code. As a general rule any time you allocate memory you must free it. Points will be deducted in both the labs and the project for memory leaks so it is important that you learn how to use and read Valgrind's output. See (https://valgrind.org/) for more details.

The naming convention of the zip/tar file while uploading to canvas
Name_ProjectX (an example is given below)
• firstname: abie
• lastname: safdie
• Submission for: Project3
• So the name of the zip/tar file should be:
abie_safdie_Project3.zip

Rubric

Part	Points	Description
1	30	Simulation runs correctly with a single thread, All critical sections are well protected with locks, Correct usage of pthread_create, Correctly return a value using pthread_join
2	30	Simulation runs correctly with any number of threads.
3	20	Correct Usage of Pipes for IPC. Accurately displays live states of

Part	Points	Description
		queues.
Valgrind	10	No memory leak/errors. 1 point each until all 10 points are gone.
Report	10	1 - 2 page report

Note: Some sections may have more detail points than the total points, meaning there are more than 1 way you can get a 0 in that section.

1. 0/100 if your program does not compile.
2. 10 points deduction if your makefile does not work
3. 30 points deduction if pthread synchronization functions used but did not contribute to the actual functionality
4. Missing functionality caused by chain effects will not receive credit. (correctly implemented but does not work due to other mistakes)

Late Policy:

- 5% penalty (1 day late)
- 10% penalty (2 days late)
- 20% penalty (3 days late)
- 100% penalty (>3 days late) (i.e. no points will be given to projects received after 3 days)

Appendix

`getopt()` usage from tutorialspoint

The getopt() is one of the built-in C function that are used for taking the command line options. The syntax of this function is like below –

getopt(int argc, char *const argv[], const char *optstring)

The opstring is a list of characters. Each of them representing a single character option.

This function returns many values. These are like below

- If the option takes a value, then that value will be pointed by optarg.
- It will return -1, when no more options to proces
- Returns ‘?’ to show that this is an unrecognized option, it stores it to optopt.
- Sometimes some options need some value, If the option is present but the values are not there, then also it will return ‘?’. We can use ‘:’ as the first character of the optstring, so in that time, it will return ‘:’ instead of ‘?’ if no value is given.

EXAMPLE: test.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

main(int argc, char *argv[]) {
   int option;
   int n;
   int num_cars;
   // put ':' at the starting of the string so compiler can distinguish between '?' and '
   while((option = getopt(argc, argv, ":n:c:")) != -1){ //get option from the getopt() me
      switch(option){
         case 'n':
            int n = atoi(optarg);
            break;
         case 'c':
            int num_cars = atoi(optarg);
            break;
         case ':':
            printf("option needs a value\n");
            break;
         case '?': //used for some unknown options
            printf("unknown option: %c\n", optopt);
            break;
      }
   }
   for(; optind < argc; optind++){ //when some extra arguments are passed
      printf("Given extra arguments: %s\n", argv[optind]);
   }
}
```

./test -n 5 -c 5