CS 4323 - Design and Implementation of Operation Systems (Fall 2020)

Final Group Project
Final Report

Due Date
December 1, 2020

By
Final Group D

Members
Collin Thornton (CWID: A11773881)
Tyler Krebs (CWID: A11784550)
Robert Cook (CWID: A10073520)

Table of Contents

Work Distribution

Collin Thornton

Deadlock Detection - Part 2 (c)

Overview

The risk of deadlock is inherent in multiple portions of the simulator. Many processes are attempting to access a few communal resources; it is likely that near-simultaneous requests will occur. If left unhandled, such requests can lead to deadlock, during which processes wait indefinitely for a resource to become available. In this section, I am to develop an algorithm for the detection of deadlock scenarios.

Scope of Work

The client processes must obtain weights from a communal resource pool in the gym. This process is abstracted through a resource manager API, which handles file I/O and data handling for the clients. Deadlock occurs when multiple clients, which already have weights allocated from the resource manager, make additional requests for weights held by other clients. This section must detect the occurrence of any deadlock and report it through standard output.

As this section is focused on the detection of deadlock, I must ensure that my algorithms are likely to generate a deadlock event during execution. I complete this by presetting weights the initiate a deadlock. These weights do not cause deadlock every execution; it is dependent upon the relative number of clients, sets, and weights in play at any given time. This said, it does trigger deadlock more often than not. Once deadlock occurs, code from this section must detect it in a timely manner. Finally, code from this section must gather relevant information from the deadlocked process(es), including

Tasks and Design

- Create a resource manager API to handle file I/O for resource management between multiple processes
  - Layout the framework for weights.txt.
    - 1st section - master list of weights possessed by gym
    - 2nd section - currently allocated weights, by client
    - 3rd section - current I'm requests, by client
    - "---" delimits a section. ";" delimits a field. "," separates plate weight from number of plates
  - Design API to access weights.txt
    - Functions access by the section
    - Separate functions for reading and writing
    - Function calls are locked with a mutual exclusion semaphore to mitigate race conditions
- Time-bound deadlock detection
  - All weight requests made by clients must be completed through the resource manager
  - Resource manager publishes requests and allocations to weights.txt
  - Parent process periodically checks weights.txt to see if any circular requests exist
  - If exists, all affected process ids are reported

Deadlock Recovery - Part 2 (d)

Overview

Detection of deadlock scenarios is only half of the problem. Deadlock recovery requires that a process be preempted such that releases acquired resources for other processes to utilize. The process must then be reset. In this assignment, we will implement a partial-rollback recovery scheme. Partial-rollback requires that the process maintains its current state such that it may be reset and fast-forwarded to the previous state.

Scope of Work

This section has an equivalent scope to that of Part 2 (c). It must be implemented on all processes which access weights from the gym, including all client processes. Utilizing partial-rollback, this section will reset deadlocked processes detected in Part 2 (c).

Tasks and Design

- Identification of victim process
    - Parent process gets list of deadlock clients from Part 2 ©
    - Iterates through list, choses client with least number of completed sets
    - Sets a flag in shared memory
- Reset the client
    - Upon obtaining a total weight from the trainer, the client initiates a request from the resource manager
    - If the request is immediately granted, there is no risk of deadlock. Otherwise, the client enters a loop until
        - The request is granted
        - It is notified through shared memory of its selection as deadlock victim
- Rollback the process
    - If the second occurs, the client exits the loop, replaces all allocated weights, and resets the set
- See Appendix A for UML Diagrams

Tyler Krebs

Record Book Implementation and Mutex Synchronization - Part 3 (e)

Overview

I have been given the role of creating a record book for trainers to keep track of the clients' total weights used during their workout. To keep from multiple trainers accessing the record book at once it is my job to utilize mutual exclusion synchronization.

<u>Scope of Work</u>

I am to create a record book file with a couple key functions. Add - to append the client info and weight to the record book text file, Display - to print the information from the record book text file to the console, Clear - to clear the record book text file, and Init - to initialize the record book.

<u>Tasks and Design</u>

- Create a recordbook.c file
    - Add - to append the client info and weight to the record book text file
    - Display - to print the information from the record book text file to the console
    - Clear - to clear the record book text file
    - Init - to initialize the record book
- Create a test file to test the functionality of the record book functions
    - Needs to initialize the record book
    - Needs to be able to add clients name, id, and total weight
    - Needs to read and display all information from the record book text file
    - Needs to clear the record book text file of all contents

Robert Cook

Boundary Cases Simulator - Part 1 (a)

<u>Overview</u>

There are three boundary cases when it comes to synchronization and semaphores: deadlocks, starvation, and first-come-first-served policy violations. For this part of our group project, we needed to create a simulator which would demonstrate these particular boundary conditions as well as provide our group a way to demonstrate the proper corrective actions/remediations for these issues.

<u>Scope of Work</u>

Create a simulator to run three scenarios, one for each of the three boundary conditions. The simulator for this section of the project should NOT utilize semaphores or any other method of object synchronization to ensure boundary conditions are met. It should also leverage multiple processes to account for multi-process situations.

<u>Tasks and Design</u>

- Create a simulator which will run through each of the three scenarios. The simulator should have the following:
    - Call a customer and trainer “arrival” method
    - Fork() as needed to spawn processes for each customer and trainer object
    - Manage child processes
    - Accept a numeric parameter allowing the simulator to be ran for a specific scenario, 0 would mean run all scenarios
    - Manage a gym structure which will maintain basic data about the gym that will be shared through shared memory among all the customer/trainer objects
- Create a customer object, which will have the following:
    - A start or arrival method that can be accessed by the simulator which should perform the following workflow:
        - “Arrive” at the gym (setting a state) and check for trainers
        - “Travel” to the waiting room (via a 3 second sleep to account for walking)
        - Look for an available couch and “sit” if able, otherwise leave
- Create a trainer object, which will have the following:
    - A start or “not busy” method that the simulator can call to simulate a trainer becoming “available” which should:
        - Immediately check for customers just arriving (i.e. arriving state)
        - “Travel” to the waiting room (with no wait time because of scooter) and check for any occupied couches
        - “Wait” for any arriving customers if none in the waiting room (this is the “busy on their phone” part)
- See Appendix A for UML Diagrams

Synchronization of Customers/Trainers using Semaphores - Part 1 (b)

Overview

The second part of this section of the project is to employ semaphores to handle/mitigate the three boundary conditions as outlined in part (a) of this project.

Scope of Work

Either update any existing data structures, or implement new ones if they do not already exist, which will implement semaphores and allow for the three boundary conditions to be handled. This should be demonstrated via the simulator utilizing these new structures and showing the appropriate control flow to alleviate deadlock, starvation, and first-come-first-served violation.

Tasks and Design

- Update the gym structure to include semaphores for the following:
    - Number of customers arriving - As this is the first property checked by the trainers before “traveling” to the waiting room, it will ensure they do not miss a customer that is just now arriving

    - Number of customers waiting - This will allow for the travel time difference between the customer and the trainer to ensure the customer is not missed as they are walking from the arrival to the waiting area
    - Number of couches available - This will allow for the handling of multiple customers arriving in the waiting area
    - Wait and signal methods for each semaphore
- Update the simulator, customer, and trainer objects to allow for a flag to be sent which will instruct those objects of whether or not to utilize the new semaphores
- See Appendix A for UML Diagrams

Driver Files

All compilation and execution should occur in the root directory, i.e., the directory with src/, build/, include/, and data/ folders.

Part 1

Compilation

make part1

Execution

./part1 <num_trainers> <num_couches>

Overview

Launches the simulator with flags to cause a violation of the first-come-first-serve standard and starve clients of trainers. These situations include: when a customer and trainer are not available at the same time, when two customers arrive in the waiting room at the same time when only one couch space is available, or when one customer arrives in the waiting room and goes to take a seat on the couch and another customer walks in and the later customer is served first.

Part 2

Compilation

make part2

Execution

./part2 <num_trainers> <num_couches>

Overview

Launches the simulator that implements deadlock detection and recovery. This part deals with the weight room and things such as: weight plates, number of plates of each weight, current plates being used by each customer and the request to each weight type that will be made by each customer.

Part 3

Compilation

make part3

Execution

./part3 <num_trainers> <num_couches>

Overview

Launches the simulator that implements the record book and mutex synchronization. For each customer a trainer modifies the record. If two or more customers finish the exercise at the same time, two trainers cannot perform writing to the record book at the same time, so a mutex lock is utilized.

Work Completed

Collin Thornton

Description
Contribution can be broken into 6 main phases:

1. Development of resource manager API
    a. Create vector API to simplify implementation
2. Development of client/trainer linked list APIs
3. Implementation of deadlock detection/rollback algorithm
4. Development of client/trainer workout events
5. Development of simulation launch code
6. Implementation of makefile compilation

The resource manager API is designed for interfacing with the weight database. The weight database is represented as a text file stored in the data/ folder. It is automatically created and initialized on program launch. The file itself is divided into three sections separated by the string "---". The first lists the total weights possessed by the gym. The second lists weights which are currently checked out. The third lists requests to checkout specific plates. A public facing API abstracts the file I/O into a series of high-level commands, such as requestWeightAllocation(pid_t pid, Weight *weight) or releaseWeightAllocation(pid_t pid). Each such command utilizes private functions to read the allocation and request matrices from the file.

I created a simple vector library to work with the weight matrices to simplify the implementation of the resource manager API. Many of the private functions utilize the vector library to perform element-wise operations. These include comparisons, addition, and subtraction. A typical use case of the vector library occurs when a client is attempting to allocate a weight request. When this occurs, the resource manager adds the allocation and request matrices row-by-row The total request is subsequently found by adding the columns of the summation matrix. The resulting vector is compared against the max weight vector. If it's smaller, the allocation request is granted. This process is heavily dependent on the vector API.

Early on, we decided that linked lists (LL) are preferable over dynamic arrays to track lists of clients and trainers. LL are simple to expand/subtract, search, and traverse. I designed the vast majority of all code related to the client/trainer structs, node structs, and LL structs. Supporting code include initialization and deletion functions for all struct types, to_string functions for all structs, and LL operations such as add/remove and various search functions. Users can search the LLs for nodes with specific properties. For example, using the function call client_list_find_trainer(pid_t pid, ClientList* list), one may search a list for a client paired to a trainer with a process id of <pid>. These functions are heavily used throughout the simulator.

Deadlock detection was implemented based on the algorithm defined on page 339 of the class textbook. At a high level, it utilizes the resource manager to search for circular weight allocation requests. If found, the system will return the process ids of all involved clients for further processing. The parent process periodically runs the algorithm to check for deadlocked clients in the workout room. If there are any deadlocked processes, the parent searches for the process with the least number of completed sets. At this point, it places the process id of the chosen process in the shared memory space. Each process periodically checks this flag with performing a weight request. If the request is not immediately granted and the process notices that it's been selected as the victim, it releases all existing weight allocations and resets the current workout set.

The workout events encapsulate a simulated workout room. It is divided into two major events: the client workout event and the trainer workout event. Each event is ran by its respective process, managing actions, such as lifting weights or setting the total number of sets, along with IPC and synchronization with the other event. The client event waits for the trainer to set a workout, then executes it. Execution involves making a weight request, getting the weight allocation (this is where deadlock can occur), and lifting the weights. After every set it waits for the trainer to set a new weight for the next set. The trainer event performs the compliment action. It repeatedly generates the set weight and waits for client acknowledgement until the client finishes the workout.

As simulation complexity increased, the necessity of launch code became evident. Each driver file calls the same launch file, though they specify different run-time flags. The launch code spawns all clients and trainers, monitors for deadlock, starvation, and handles construction/deconstruction of shared memory spaces, semaphores, and other synchronization tools.

About halfway through the project I realized that, in utilizing a module coding style, we were going to be compiling over a dozen files every time we wanted to test the simulator. I created a makefile to simplify the process and expedite compilation. All built .o files are stored in the build/ directory. Executable files (ie part1, part2, and part3) are stored in the root directory.

Classes and Methods

Driver: part2.c

- Methods
    - test_resource_manager
        - Overview
            - Test code for the resource manager, focusing on the generation of an internal weight database from the communal file.
        - Parameters
            - void
        - Returns
            - void
    - main
        - Overview
            - Launch the test functions.
        - Parameters
            - Int argc -> Number of arguments
            - char** argv -> List of arguments. Will be used to specify number of customers, couches, and filenames as necessary
        - Returns
            - Int - status code

Additional Files:

Resource_manager.c

- Structs
    - None
- Methods
    - getGymResources()

        - @brief return the gym's total resources. must be deleted with weight_del()
        - @return (Weight*) Vector of total weights
    - removeWhiteSpace
        - Overview
            - Remove all whitespace from a given string. Used to simplify string processing.
        - Parameters
            - Char *str - String to be processed
        - Return
            - char* - Processed string
    - init_resource_manager
        - Overview
            - Initialize the semaphore. Should only be ran on the parent process
        - Return
            - Int - return code. Negative on error
    - open_resource_manager
        - Overview
            - Open the semaphore on the current process
        - Return
            - Int - return code. Negative on error
    - close_resource_manager
        - Overview
            - Close the semaphore on the current process
        - Return
            - Void
    - destroy_resource_manager
        - Overview
            - Destroy the semaphore. Should only be ran on the parent process
        - Return
            - Void
    - weight_matrix_init
        - Overview
            - Private function. Initialize a weightMatrix on the heap. All values NULL or 0
        - Return
            - WeightMatrix* - pointer to new matrix
    - weight_matrix_del
        - Overview
            - Free a WeightMatrix*, and all internal WeightRows* and Weights*
        - Parameters
            - WeightMatrix* - matrix to be deleted
        - Return
            - Int - 0 on success
    - Weight_matrix_add_req

        - Overview
            - Add a weight request to a weight matrix. Will add new row if pid is not found.
        - Parameters
            - Pid pid_t - pid of requesting process
            - Weight Weight* - weight to be added
            - Matrix WeightMatrix* - matrix to store summation
        - Return
            - Int - number of rows in matrix. Negative on error
    - Weight_matrix_sub_req
        - Overview
            - Subtract a weight request from a weight matrix. Will delete a row if result is 0 vector
        - Parameters
            - Pid pid_t - pid of requesting process
            - Weight Weight* - weight to be subtracted
            - Matrix WeightMatrix* - matrix to store summation
        - Return
            - Int - number of rows in matrix. Negative on error
    - Weight_matrix_search
        - Overview
            - Search a WeightMatrix for a pid
        - Parameters
            - pid (pid_t) - pid for which to search
            - matrix (WeightMatrix*) - matrix to be searched
            - row_number (int*) - storage for the row number. negative if the row is not found. can be set to NULL if not needed
        - Return
            - WeightMatrixRow* - pointer to the row. NULL if not found
    - Weight_matrix_to_string
        - Overview
            - Return a string representation of matrix
        - Parameters
            - Matrix WeightMatrix* - matrix to be returned as string
            - Buffer char[] - buffer to store string output
        - Return
            - Const char* - pointer to string. Same as buffer
    - getAvailableWeights
        - Overview
            - Return currently available weights
        - Return
            - Weight* - vector of current weight
    - getWeighFromFile
        - Overview

        - Private function. Not locked with semaphore. Get a weight from the input file
    - Parameter
        - Section (unsigned int) section number from which to read
    - Return
        - Weight* - allocation on heap
- getWeightAllocation
    - Overview
        - Return the currently allocated weights. Deleted with weight_matrix_del
    - Return
        - WeightMatrix* matrix of allocations allocated on heap
- getWeightRequest
    - Overview
        - Removes the request, allocates weights, and adjusts the currently available weights
    - Parameters
        - Pid pid_t - process to grant
    - Return
        - Int - return code. Negative on error
- getWeightMatrixFromFile
    - Overview
        - Read a weight matrix from the input file
    - Parameters
        - Section (unsigned int) - section number from which to read
    - Return
        - WeightMatrix* - pointer to weight matrix on heap
- grantWeightRequest
    - Overview
        - Removes the request, allocates weights, and adjusts the currently available weights
    - Parameters
        - Pid pid_t - process to grant
    - Return
        - Int - return code. Negative on error
- writeWeightAllocation
    - Overview
        - Write a new allocation to the file
    - Parameters
        - Pid pid_t - pid of process
        - Weight Weight* - new allocation
    - Return
        - Int - negative on failure
- writeWeightRequest

            - Overview
                - Write a new request to the file
            - Parameters
                - Pid pid_t - pid of process
                - Weight Weight* - new request
            - Return
                - Int - negative on failure
        - writeWeightMatrixToFile
            - Overview
                - Private function. Not locked with semaphore. Write a weight matrix to the input file. will delete the matrix
            - Parameters
                - Matrix WeightMatrix* - matrix to be written
                - Section (int) section number at which to write
            - Return
                - Int - negative on failure
        - releaseWeightAllocation
            - Overview
                - Removes the allocation and adjusts currently available weights
            - Parameters
                - Pid pid_t - process id to adjust
                - Weight Weight* - amount to change
            - Return
                - Int - return code. Negative on error
        - removeWeightRequest
            - Overview
                - Removes a request from the file. Will throw error if result is negative
            - Parameters
                - Pid pid_t - pid of process
                - Weight Weight* - weight to be subtracted
            - Return
                - Int - negative on failure
        - clearWeightFile
            - Overview
                - Clear allocation and request matrices from file
            - Return
                - Int - negative on failure

Gym.c

- Enumerations
    - PlateIndex
        - Overview
            - Enumeration of possible plate weights to aid in handling the plate array in the Weight struct

            - Values
                - TWO_HALF, FIVE, TEN, FIFTEEN, TWENTY, TWENTY_FIVE, THIRTY_FIVE, FORTY_FIVE
- Structs
    - Weight
        - Overview
            - Maintains information related to the number of weights in utilization.
        - Variables
            - Unsigned short num_plates[8] - Array of plates indexed with the PlateIndex enumeration. Values indicate the number of plates in use. Eg. num_plates[FIVE] = 2 indicates that 2 plates of weight 5 are in use.
            - Float total_weight - total weight of plates in num_plates.
- Methods
    - weight_init()
        - Overview
            - Allocate a Weight struct on the heap and initialize.
        - Parameters
            - Short plate_array[8] - Array with initial plate weights. Set as NULL if not yet defined.
            - Float total_weight - Sum of the weight of all plates. Set as 0 if plate_array not yet defined.
        - Return
            - Weight* - Pointer to a Weight strut allocated on the heap
    - weight_del()
        - Overview
            - Free a Weight struct from the heap
        - Parameters
            - Weight* weight - The struct to be freed
        - Return
            - Int - the return code from the function

Deadlock.c

- Methods
    - checkForDeadlock
        - Overview
            - Check the input file for deadlocked processes.
        - Parameters:
            - deadlocked_array (pid_t[]) array with length of number of processes (matrix rows)
        - Returns
            - Int - number of deadlocked processes
    - test_deadlock_detection
        - Overview

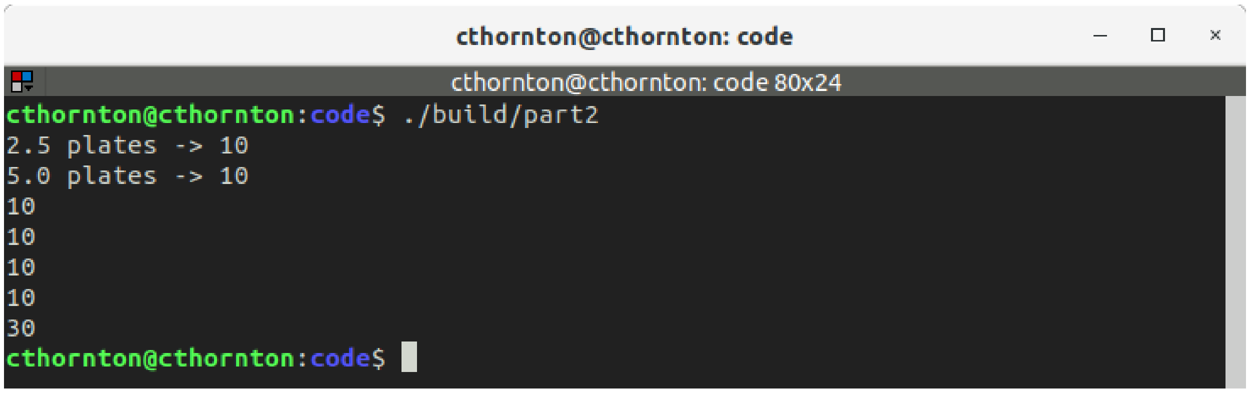
        - Tests to see if there is deadlock detection
    - Variables
        - int num_deadlocked = checkForDeadlock(deadlock_array); - sets the number of deadlock occurrences to the checkForDeadlock(deadlock_array) function
    - Returns
        - Void
-

Code:
See Appendix B for code

Input

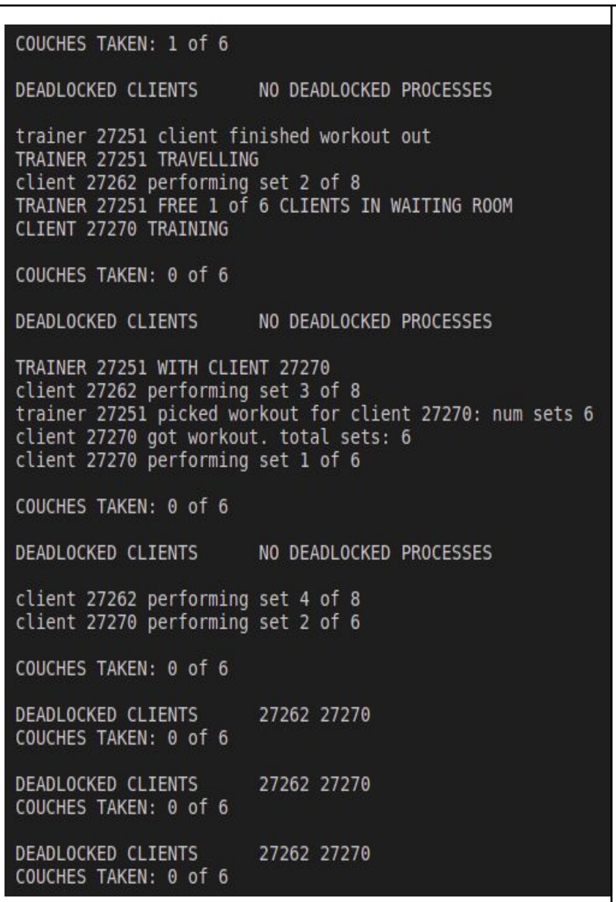
Weight.txt
```
# WEIGHT DATABASE FOR GYM
# SPECIAL CHARACTERS:
#   #  -> Comment. Program ignores line
#   ;  -> Delimeter. Primary delimeter
#   ,  -> Delimeter between plate weight and number of plates
#  --- -> Section divider

# TOTAL AVAILABLE WEIGHTS IN GYM
# FORMAT : plate weight, number available

2.5,10
5,10
10,10
15,10
20,10
25,10
35,10
45,30

# PROGRAM MODIFIES FILES BELOW DASHED LINES
---
# CURRENTLY CHECKED OUT PLATES
# FORMAT : customer id; plate weight, # plates; plate weight, # plates; ...
# ex. customer1;2.5,2;5,0;10,0;15,0;20,0;25,2;35,0;45,2

---
# CURRENT CHECK OUT REQUESTS
# FORMAT : customer id; plate weight, # plates; plate weight, # plates; ...
# ex. customer1;2.5,2;5,0;10,0;15,0;20,0;25,2;35,0;45,2
```

Output

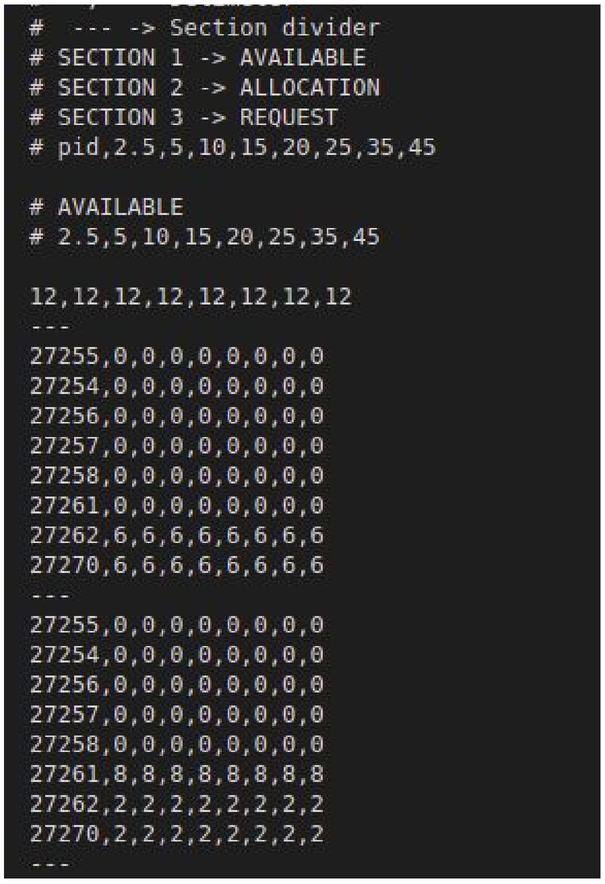


```
cthornton@cthornton: code
cthornton@cthornton: code 80x24
cthornton@cthornton:code$ ./build/part2
2.5 plates -> 10
5.0 plates -> 10
10
10
10
10
30
cthornton@cthornton:code$
```

The code successfully reads the database from weights.txt. The code has been verified to be memory safe with valgrind, a useful C/C++ tool for debugging memory related issues. Internally, the data is represented as a Weight struct allocated on heap and represents the total set of weights the gym possesses.

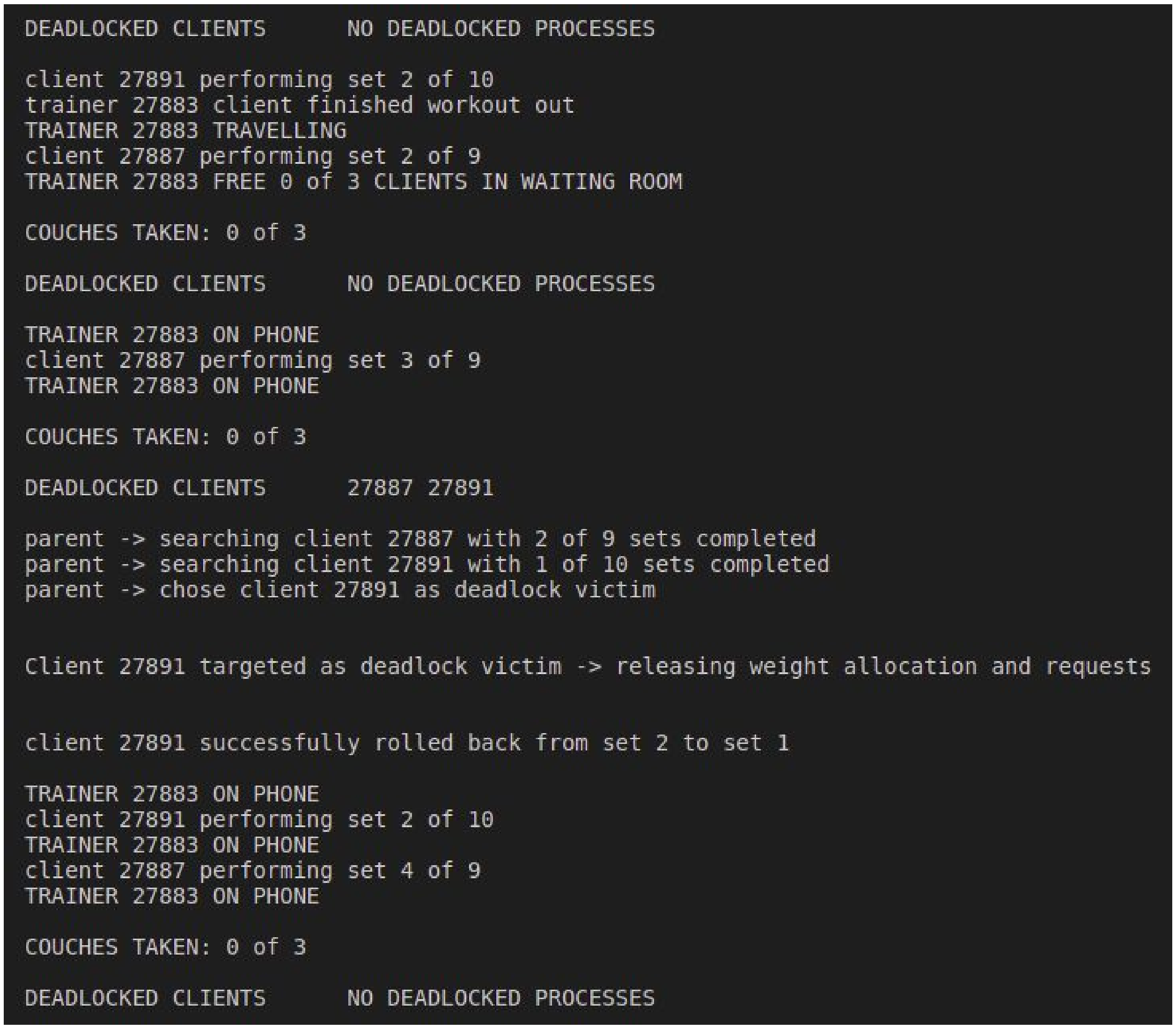


```
COUCHES TAKEN: 1 of 6

DEADLOCKED CLIENTS      NO DEADLOCKED PROCESSES

trainer 27251 client finished workout out
TRAINER 27251 TRAVELLING
client 27262 performing set 2 of 8
TRAINER 27251 FREE 1 of 6 CLIENTS IN WAITING ROOM
CLIENT 27270 TRAINING

COUCHES TAKEN: 0 of 6

DEADLOCKED CLIENTS      NO DEADLOCKED PROCESSES

TRAINER 27251 WITH CLIENT 27270
client 27262 performing set 3 of 8
trainer 27251 picked workout for client 27270: num sets 6
client 27270 got workout. total sets: 6
client 27270 performing set 1 of 6

COUCHES TAKEN: 0 of 6

DEADLOCKED CLIENTS      NO DEADLOCKED PROCESSES

client 27262 performing set 4 of 8
client 27270 performing set 2 of 6

COUCHES TAKEN: 0 of 6

DEADLOCKED CLIENTS      27262 27270
COUCHES TAKEN: 0 of 6

DEADLOCKED CLIENTS      27262 27270
COUCHES TAKEN: 0 of 6

DEADLOCKED CLIENTS      27262 27270
COUCHES TAKEN: 0 of 6
```



```
#  --- -> Section divider
# SECTION 1 -> AVAILABLE
# SECTION 2 -> ALLOCATION
# SECTION 3 -> REQUEST
# pid,2.5,5,10,15,20,25,35,45

# AVAILABLE
# 2.5,5,10,15,20,25,35,45

12,12,12,12,12,12,12,12
---
27255,0,0,0,0,0,0,0,0
27254,0,0,0,0,0,0,0,0
27256,0,0,0,0,0,0,0,0
27257,0,0,0,0,0,0,0,0
27258,0,0,0,0,0,0,0,0
27261,0,0,0,0,0,0,0,0
27262,6,6,6,6,6,6,6,6
27270,6,6,6,6,6,6,6,6
---
27255,0,0,0,0,0,0,0,0
27254,0,0,0,0,0,0,0,0
27256,0,0,0,0,0,0,0,0
27257,0,0,0,0,0,0,0,0
27258,0,0,0,0,0,0,0,0
27261,8,8,8,8,8,8,8,8
27262,2,2,2,2,2,2,2,2
27270,2,2,2,2,2,2,2,2
---
```

These screenshots show the successful detection of a deadlock scenario. The left subfigure depicts terminal output. The right depicts the current weight requests and allocations. As can be seen processes 27262 and 27270 have allocated all of the total weights and are both requesting more. This creates a circular situation as neither process is willing to release its currently allocated waits.



```
DEADLOCKED CLIENTS      NO DEADLOCKED PROCESSES

client 27891 performing set 2 of 10
trainer 27883 client finished workout out
TRAINER 27883 TRAVELLING
client 27887 performing set 2 of 9
TRAINER 27883 FREE 0 of 3 CLIENTS IN WAITING ROOM

COUCHES TAKEN: 0 of 3

DEADLOCKED CLIENTS      NO DEADLOCKED PROCESSES

TRAINER 27883 ON PHONE
client 27887 performing set 3 of 9
TRAINER 27883 ON PHONE

COUCHES TAKEN: 0 of 3

DEADLOCKED CLIENTS      27887 27891

parent -> searching client 27887 with 2 of 9 sets completed
parent -> searching client 27891 with 1 of 10 sets completed
parent -> chose client 27891 as deadlock victim


Client 27891 targeted as deadlock victim -> releasing weight allocation and requests


client 27891 successfully rolled back from set 2 to set 1

TRAINER 27883 ON PHONE
client 27891 performing set 2 of 10
TRAINER 27883 ON PHONE
client 27887 performing set 4 of 9
TRAINER 27883 ON PHONE

COUCHES TAKEN: 0 of 3

DEADLOCKED CLIENTS      NO DEADLOCKED PROCESSES
```

This figure depicts a deadlock rollback scenario. The parent process detects that clients 27887 and 27891 are deadlocked. It notes that client 27891 has completed fewer sets and choses it as the victim. Once the client detects that it's the victim, it rolls itself back and restarts the set. This resolves the deadlock.

Tyler Krebs

Description

- I have researched and completed the recordbook.c file and implemented all necessary functions for part e. Testing was an important part of my share of the project. Making sure functions could be used by other parts of the project was another key goal of mine.

Classes and Methods

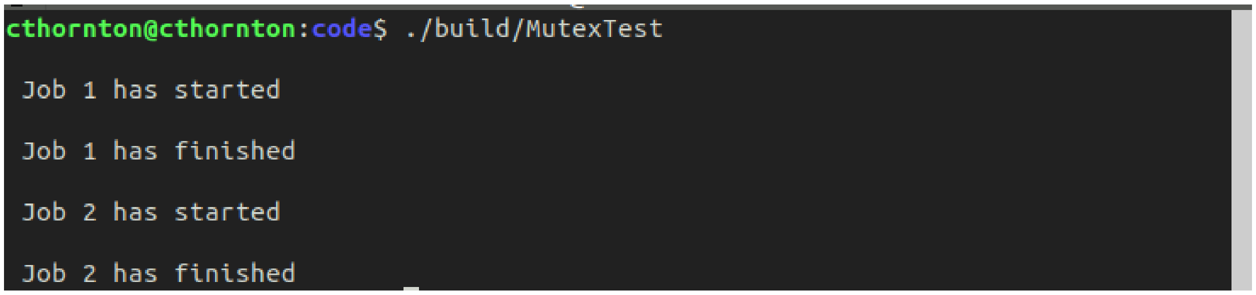
Driver: part3.c

- Methods
  - main
    - Overview
      - Test code for the resource manager, focusing on record book and outputs using synchronization
    - Parameters
      - void
    - Returns
      - void

MutexTest.c
- Methods
  - Main
    - Overview
      - This is the main method of this test file that utilizes mutex lock and calls for the “trythis” function to run. Creates two processes only allowing one to run at a time
    - Parameters:
      - Int i - keeps track of the number of processes, starts at 0, creates if less than 2.
      - Int error - equals the create pthread function and if error != 0 display error message.
    - Returns:
      - Return 0;
  - trythis
    - Overview
      - Creates Mutex lock and starts a counter for the job number. Displays info for starting and stopping the processes.
    - Parameters:
      - Counter - keeps track of the process/job number for displaying job status
    - Returns:
      - NULL

<u>Code</u>
See Appendix B for code

<u>Output:</u>


```
cthornton@cthornton:code$ ./build/MutexTest

 Job 1 has started

 Job 1 has finished

 Job 2 has started

 Job 2 has finished
```

recordbook.c

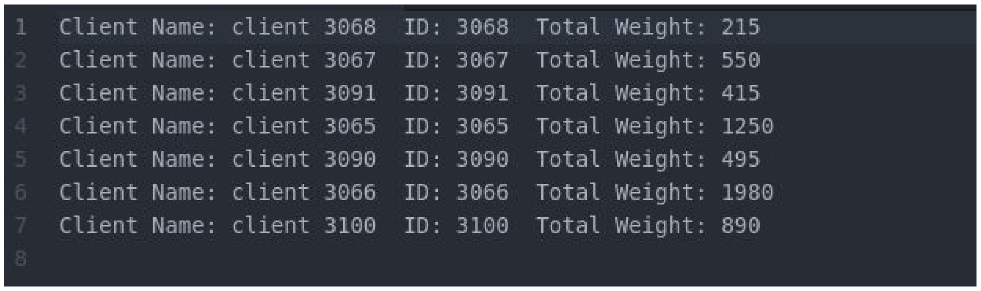
- Methods
  - addToRecordBook
    - Overview
      - This is the function that appends the emp struct of name, id, weight to the recordbook.txt file
    - Parameters:
      - File *fp = NULL; - file pointer for appending to the text file
    - Returns:
      - void
  - displayRecordBook
    - Overview
      - This function reads the recordbook.txt file and prints the contents to the console
    - Parameters:
      - File *fp = NULL; - file pointer for reading the contents of the text file
      - Struct emp empVal; - this is the main client struct that contains the client's name, id, and total weight used during the workout
    - Returns:
      - Void
  - clearRecordBook
    - Overview
      - This function just clears the text file
    - Parameters:
      - FILE *fp = NULL; - opens the file to close the fp
    - Returns:
      - Void
  - initRecordBook
    - Overview
      - This function initialized the record book and creates memory space and creates a shared mutex lock
    - Parameters:
      - pthread_mutexattr_t psharedm; - this creates the shared mutex attribute

        - int sharedMemoryID = shmget(SHARED_KEY, sizeof(SharedMutex), IPC_CREAT|0644); - this creates the shared memory space for the shared mutex
        - shared_mutex = shmat(sharedMemoryID, NULL, 0); - this attaches the shared mutex to the shared memory
    - Returns
        - Void
- openRecordBook
    - Overview
        - This function opens the record book. needs to be opened for each trainer to add to it
    - Parameters:
        - Int sharedMemoryID = shmget(SHARED_KEY, sizeof(SharedMutex), IPC_CREAT|0644); - This allocates the memory space for the shared mutex
    - Returns
        - Void
- closeRecordBook
    - Overview
        - function to close the record book and detach the shared memory
    - Parameter:
        - NA
    - Returns
        - Void
- destroyRecordBook
    - Overview
        - function destroys the record book and clears the memory allocated to it
    - Parameters:
        - Int sharedMemoryID = shmget(SHARED_KEY, sizeof(SharedMutex), IPC_CREAT|0644); - This allocates the memory space for the shared mutex
    - Returns
        - Void

<u>Code</u>

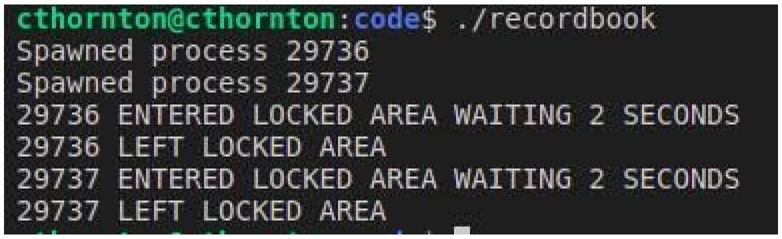
See Appendix B for code

<u>Output:</u>



```
Client Name: client 3068  ID: 3068  Total Weight: 215
Client Name: client 3067  ID: 3067  Total Weight: 550
Client Name: client 3091  ID: 3091  Total Weight: 415
Client Name: client 3065  ID: 3065  Total Weight: 1250
Client Name: client 3090  ID: 3090  Total Weight: 495
Client Name: client 3066  ID: 3066  Total Weight: 1980
Client Name: client 3100  ID: 3100  Total Weight: 890
```

This is the recordbook.txt file that is created with the recordbook.c file.



```
cthornton@cthornton:code$ ./recordbook
Spawned process 29736
Spawned process 29737
29736 ENTERED LOCKED AREA WAITING 2 SECONDS
29736 LEFT LOCKED AREA
29737 ENTERED LOCKED AREA WAITING 2 SECONDS
29737 LEFT LOCKED AREA
```

This is the output testing the mutex locking when multiple trainers are trying to modify the record book at the same time.

Robert Cook

Description

For our final project, I completed work in several areas to enable the demonstration of not only the boundary cases as stated in part 1 of the assignment but their solutions as well. This includes: the design and implementation of a shared Gym object, the mechanisms necessary to enable and manage shared memory within all aspects of the final program, and working with Collin to create all the code necessary to manage the various states and their transitions for both the Client and Trainer objects. I also worked in creating all UML diagrams utilized by both the progress and final reports.

Additionally, I created several aspects of the code in the first half of our project which were later refactored out of usage due to our decisions to change our approach to better manage the multi-processing and state changes of the Client and Trainer objects. This is an example of good collaboration within the team, to be able to recognize areas where we should regroup and reassess our solution from a different perspective to ultimately create a better product.

To solve the problems presented in part 1 of the assignment, we leveraged a struct called Gym which stored arrays of clients and trainers, as well as key properties to manage the workflow. This struct was placed into shared memory where it could be accessed across multiple processes, hence representing a real gym servicing many clients with multiple trainers. We also included several semaphores in various parts of the state changing process of both the Client and Trainer state machines. It was particularly important when a client was transitioning from the Arriving to Waiting states as this allowed us to prevent the starvation and deadlock scenarios presented. Lastly, a component of time was added to both the Client and Trainer state transitions, particularly around the transition to the waiting room. This was to help replicate the difference in traveling speed the client and trainer experienced due to one having a motorized scooter and the other being on foot.

Much of my code, by its nature, was integrated into the overall code files leveraged across the program, as it needed to create and manage the shared gym object along with handling the arriving states and their events for both Clients and Trainers. This code would be found in the gym.c, client.c, and trainer.c files.

<u>Classes and Methods</u>

Driver: part1.c

- Methods
  - main
    - Overview
      - This is the standard main function for any executable class. Within this one the simulator is launch with the necessary flags to cause the boundary cases to be shown
    - Parameters:
      - Int argc: the number of trainers and number of couches to run on the simulation
    - Returns:
      - void

Additional Files

The following represent sections of code files which I contributed.

gym.c

- Methods
  - init_shared_gym
    - Overview
      - Within this method I worked on the shared memory mechanisms for storing the shared Gym struct
  - open_shared_gym
    - Overview
      - Within this method I also worked on the shared memory mechanisms used to access the shared Gym struct
  - destroy_shared_gym
    - Overview
      - Within this method I contributed the code to detach and unlink the shared memory objects

client.c

- Methods
  - client_proc_state_machine

        - Overview
            - Within this method I contributed to the code which handles the ARRIVING state as well as the WAITING state for a client object. This is where the various boundary cases were either handled or allowed to happen

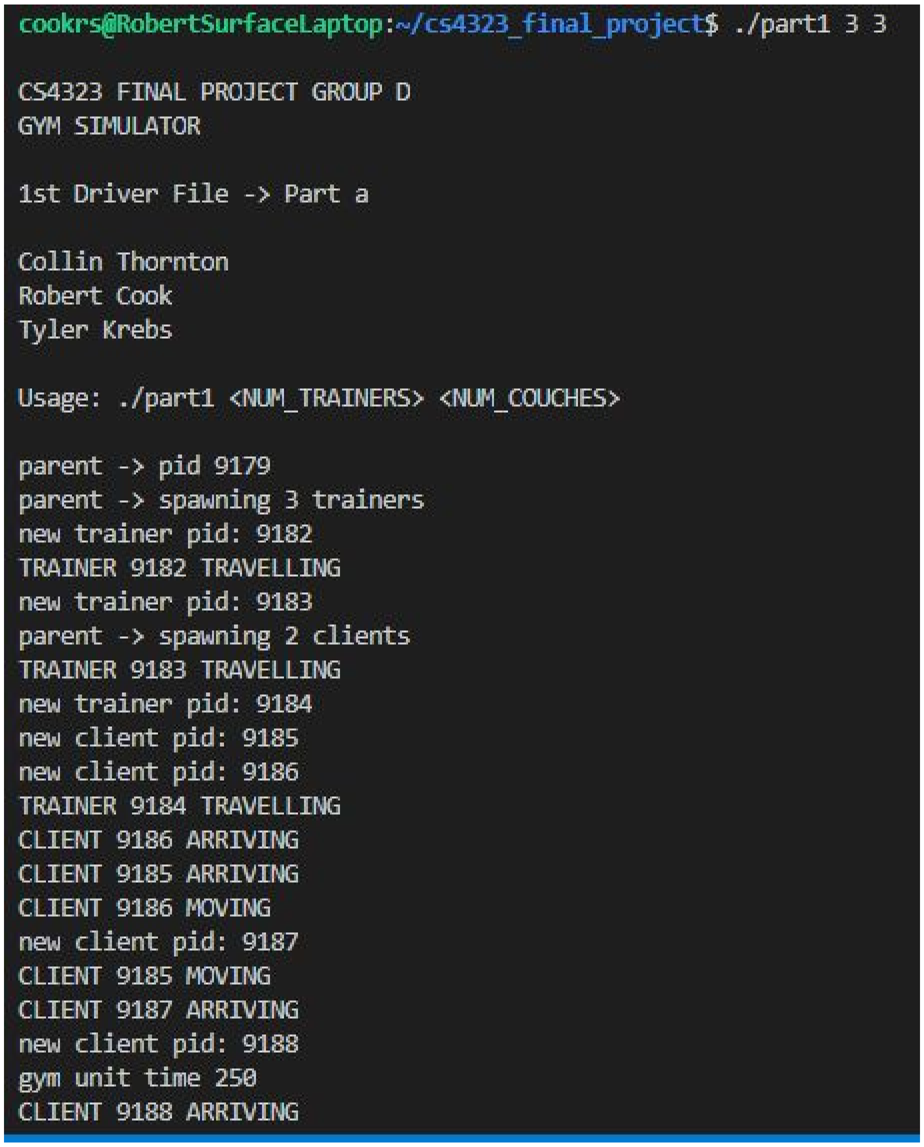
trainer.c

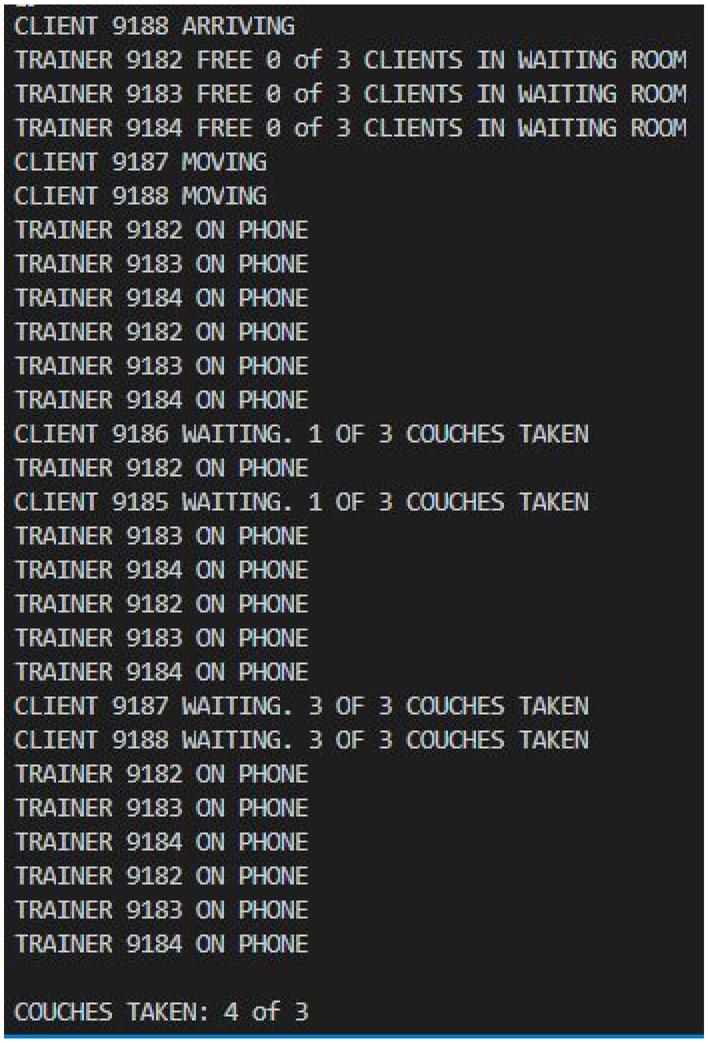
- Methods
    - trainer_proc_state_machine
        - Overview
            - Within this method I contributed code to handling the various trainer states, specifically the ON_PHONE and FREE states.

<u>Code</u>

See Appendix B for code.

<u>Output/Validation</u>

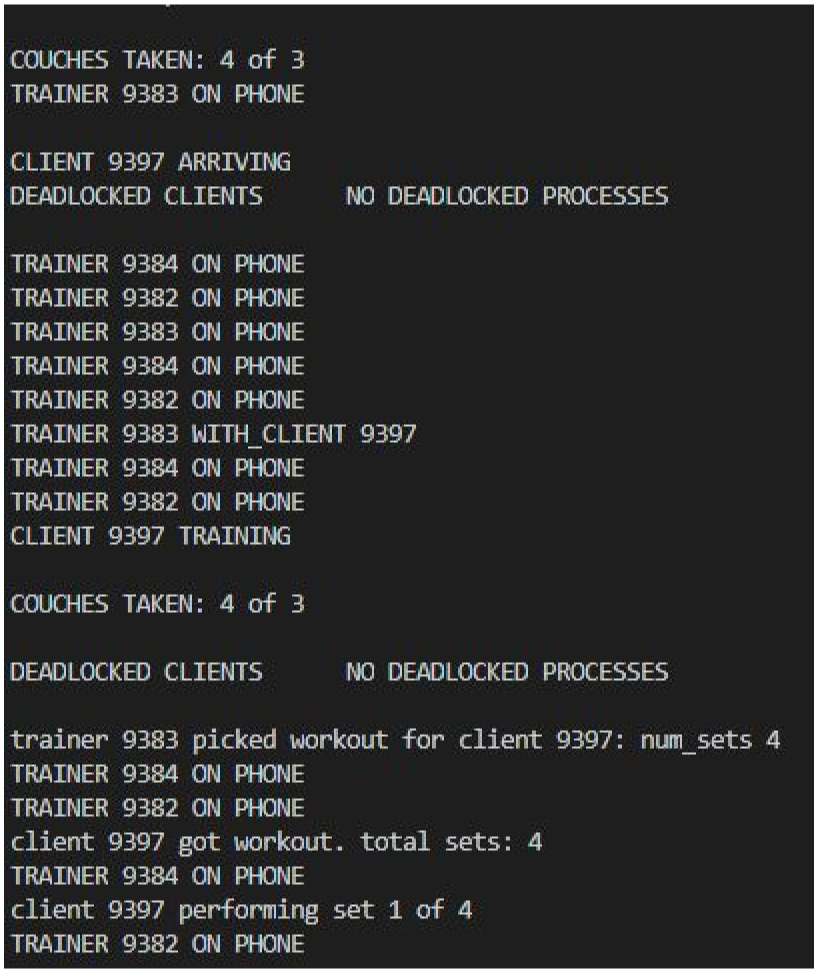

```
cookrs@RobertSurfaceLaptop:~/cs4323_final_project$ ./part1 3 3

CS4323 FINAL PROJECT GROUP D
GYM SIMULATOR

1st Driver File -> Part a

Collin Thornton
Robert Cook
Tyler Krebs

Usage: ./part1 <NUM_TRAINERS> <NUM_COUCHES>

parent -> pid 9179
parent -> spawning 3 trainers
new trainer pid: 9182
TRAINER 9182 TRAVELLING
new trainer pid: 9183
parent -> spawning 2 clients
TRAINER 9183 TRAVELLING
new trainer pid: 9184
new client pid: 9185
new client pid: 9186
TRAINER 9184 TRAVELLING
CLIENT 9186 ARRIVING
CLIENT 9185 ARRIVING
CLIENT 9186 MOVING
new client pid: 9187
CLIENT 9185 MOVING
CLIENT 9187 ARRIVING
new client pid: 9188
gym unit time 250
CLIENT 9188 ARRIVING
```



```
CLIENT 9188 ARRIVING
TRAINER 9182 FREE 0 of 3 CLIENTS IN WAITING ROOM
TRAINER 9183 FREE 0 of 3 CLIENTS IN WAITING ROOM
TRAINER 9184 FREE 0 of 3 CLIENTS IN WAITING ROOM
CLIENT 9187 MOVING
CLIENT 9188 MOVING
TRAINER 9182 ON PHONE
TRAINER 9183 ON PHONE
TRAINER 9184 ON PHONE
TRAINER 9182 ON PHONE
TRAINER 9183 ON PHONE
TRAINER 9184 ON PHONE
CLIENT 9186 WAITING. 1 OF 3 COUCHES TAKEN
TRAINER 9182 ON PHONE
CLIENT 9185 WAITING. 1 OF 3 COUCHES TAKEN
TRAINER 9183 ON PHONE
TRAINER 9184 ON PHONE
TRAINER 9182 ON PHONE
TRAINER 9183 ON PHONE
TRAINER 9184 ON PHONE
CLIENT 9187 WAITING. 3 OF 3 COUCHES TAKEN
CLIENT 9188 WAITING. 3 OF 3 COUCHES TAKEN
TRAINER 9182 ON PHONE
TRAINER 9183 ON PHONE
TRAINER 9184 ON PHONE
TRAINER 9182 ON PHONE
TRAINER 9183 ON PHONE
TRAINER 9184 ON PHONE

COUCHES TAKEN: 4 of 3
```

The above output demonstrates the boundary case described in the assignment where two clients arrive at the same time but only one couch is available and thus we get one more person in waiting than we have couches because we did not implement a semaphore to prevent this (or rather we have it turned off in this example to demonstrate the boundary case).

The above additionally demonstrates the boundary case where clients arrive with no trainers available, the travel to the waiting room and during that travel trainers check the waiting room and find no clients and thus go back to their phones. This will also set up the demonstration of the first come first served policy violation boundary case which can be seen in the output below.



```
COUCHES TAKEN: 4 of 3
TRAINER 9383 ON PHONE

CLIENT 9397 ARRIVING
DEADLOCKED CLIENTS      NO DEADLOCKED PROCESSES

TRAINER 9384 ON PHONE
TRAINER 9382 ON PHONE
TRAINER 9383 ON PHONE
TRAINER 9384 ON PHONE
TRAINER 9382 ON PHONE
TRAINER 9383 WITH_CLIENT 9397
TRAINER 9384 ON PHONE
TRAINER 9382 ON PHONE
CLIENT 9397 TRAINING

COUCHES TAKEN: 4 of 3

DEADLOCKED CLIENTS      NO DEADLOCKED PROCESSES

trainer 9383 picked workout for client 9397: num_sets 4
TRAINER 9384 ON PHONE
TRAINER 9382 ON PHONE
client 9397 got workout. total sets: 4
TRAINER 9384 ON PHONE
client 9397 performing set 1 of 4
TRAINER 9382 ON PHONE
```

Work Issues/Limitations

Collin Thornton

Early in the project we decided to utilize linked lists (LL) to organize groups of clients/trainers and shared memory for IPC. I handled the LL APIs and Robert took the shared memory API. This approach worked well until we began to integrate, at which point we realized that LL do not work well in shared memory. Linked lists are driven by pointers local to a specific process. Even though we could place the head of the list in shared memory, we'd then have to map every attribute of every node to the shared memory. This simply was not feasible. We faced a decision: change the APIs and fix all resultant problems or develop a workaround within the current API. The first option would involve a complete redesign of the simulator, replacing all linked lists with dynamic arrays. This was not desirable as more than one thousand lines of code had already been developed. As such, we decided to develop a workaround within the current API. In response, I designed the SharedGym API to copy all LL from the local space to arrays in the shared space. We locked the shared space with mutual exclusion semaphores to prevent race conditions on simulatenous access of the data.

Even with the semaphores, potential existed for synchronization problems. This primarily involved the overwriting of local changes or utilization of outdated shared data. In response, I set the following rules within the API:

1. When updating the local gym from the shared gym, **any** structs with current process ID **cannot** be altered
2. When updating the shared gym from the local gym, **only** structs with the current process ID **can** be altered.

Client and Trainer structs track the process ID of their respective process. When referencing “structs with current process ID”, we are referring to the situation where:

client->pid = getpid()   or   trainer->pid = getpid()

These rules establish two properties on the API. First, users can be sure that any pointers to structs with the current process ID will not be affected when syncing with the shared gym. Secondly, users can be sure that their pushes will not overwrite local data in any other process.

One negative attribute of this process is that pointers to structs which do not have the current process ID may be destroyed. As such, every struct with an open reference must be redefined after a call to update_gym().

Tyler Krebs

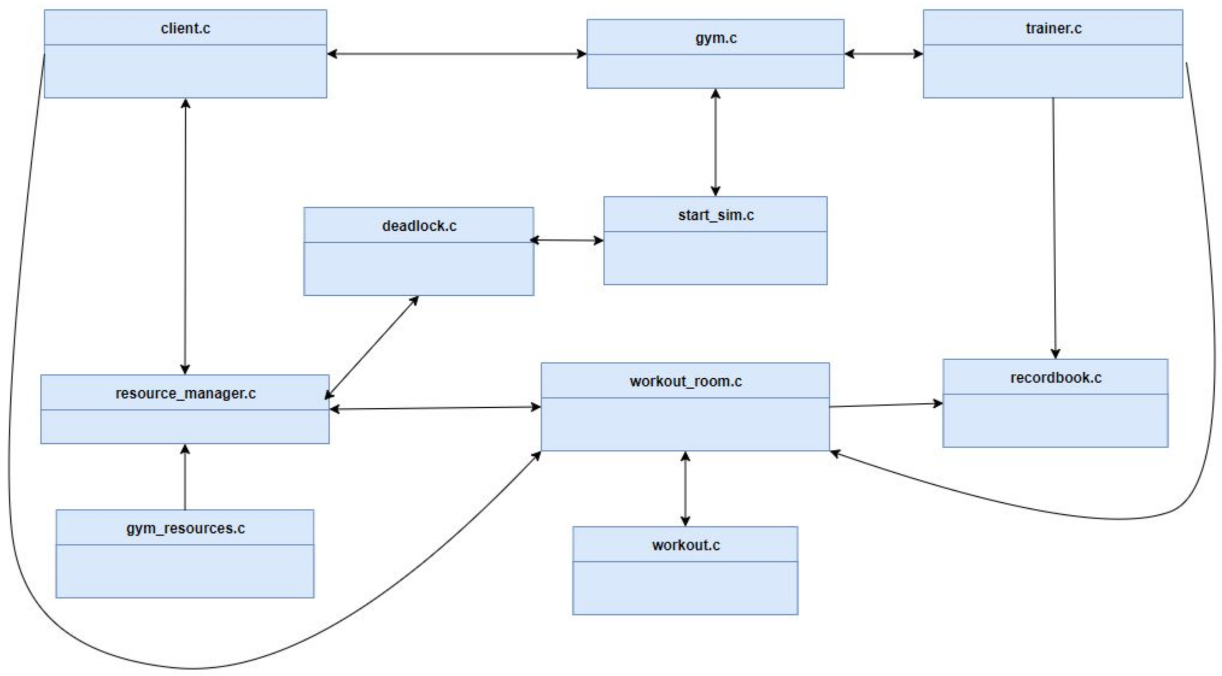
Throughout this project I experienced various work issues and limitations. The most important issue I experienced was trying to understand the other group members code. They did a very good job explaining everything but I am not the best at searching through multiple files for parameters that I need. I had a lot of help from the other group members getting the shared mutex to work as it should.

Robert Cook

As we approached the completion of the project, one aspect that was a real struggle was actually causing the boundary cases to occur. We had done such a good job in the design and implementation of our code that causes the issues to happen became nearly impossible, particularly with it came to simultaneous resource needs as our state machine was already handling this case well. In the end we actually had to code in several flags within the appropriately working code to cause it to disable these features to allow those boundary cases through. Doing this was actually quite difficult as it became an anti-pattern to our already completed code. Luckily we were able to get past this block and created several mechanisms which allowed us to switch these on or off as needed.

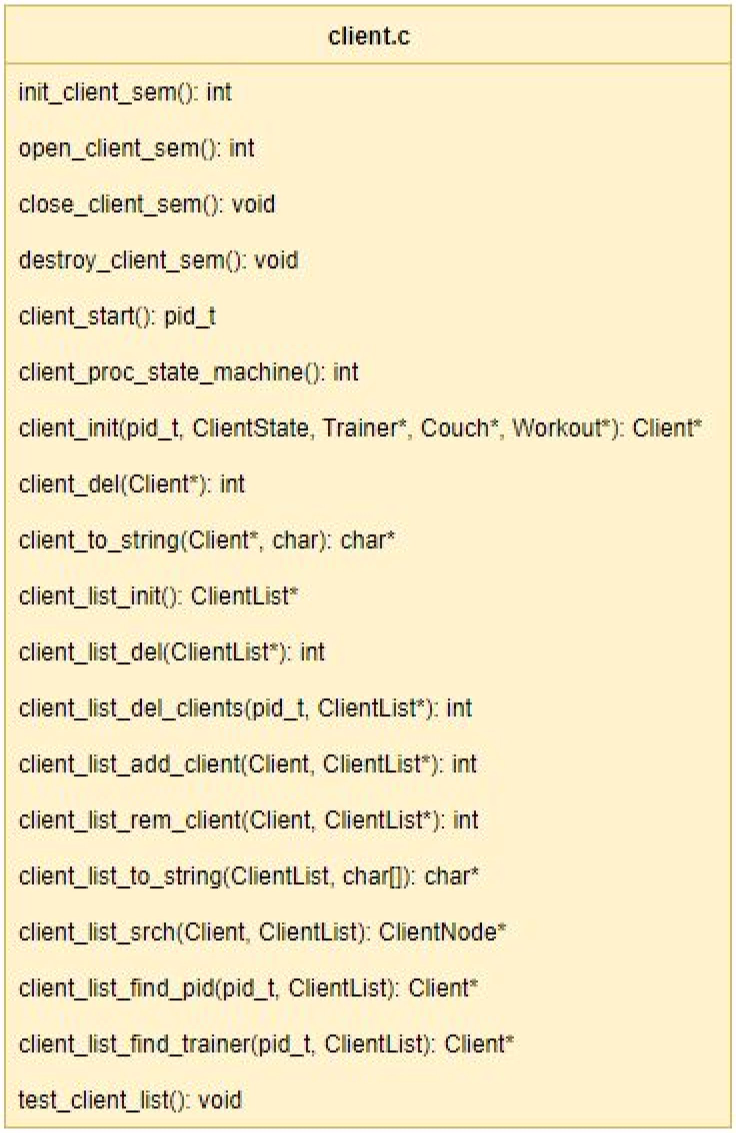
Appendix A - UML Diagrams

Workflow UML



Detailed Class UML

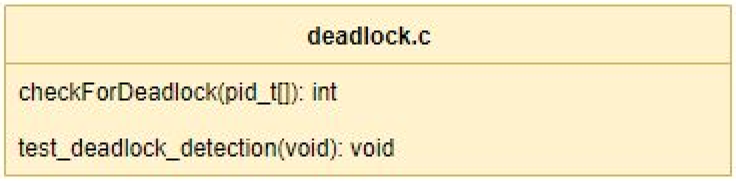
client.c



client.c

init_client_sem(): int
open_client_sem(): int
close_client_sem(): void
destroy_client_sem(): void
client_start(): pid_t
client_proc_state_machine(): int
client_init(pid_t, ClientState, Trainer*, Couch*, Workout*): Client*
client_del(Client*): int
client_to_string(Client*, char): char*
client_list_init(): ClientList*
client_list_del(ClientList*): int
client_list_del_clients(pid_t, ClientList*): int
client_list_add_client(Client, ClientList*): int
client_list_rem_client(Client, ClientList*): int
client_list_to_string(ClientList, char[]): char*
client_list_srch(Client, ClientList): ClientNode*
client_list_find_pid(pid_t, ClientList): Client*
client_list_find_trainer(pid_t, ClientList): Client*
test_client_list(): void

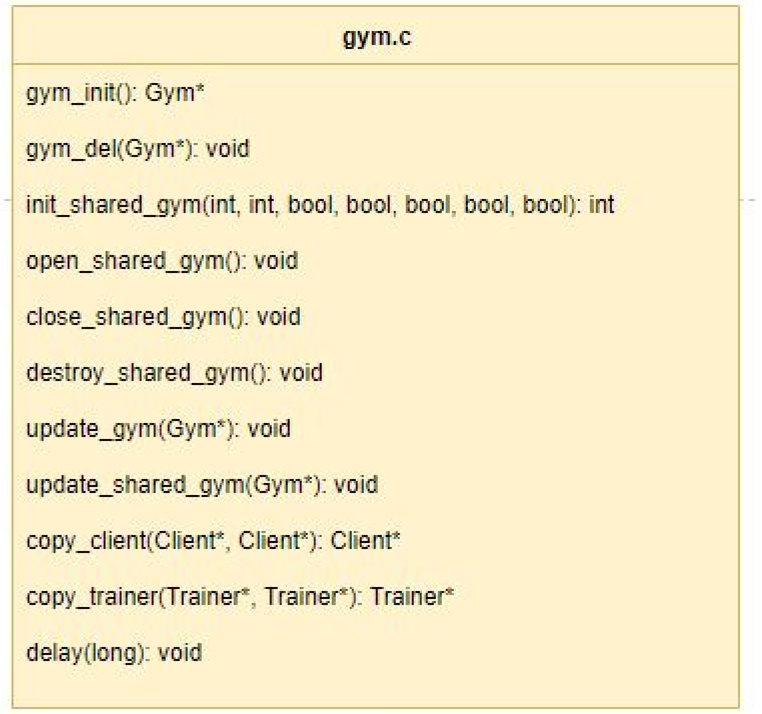
deadlock.c



deadlock.c

checkForDeadlock(pid_t[]): int
test_deadlock_detection(void): void

gym.c


gym.c

gym_init(): Gym*

gym_del(Gym*): void

init_shared_gym(int, int, bool, bool, bool, bool, bool): int

open_shared_gym(): void

close_shared_gym(): void

destroy_shared_gym(): void

update_gym(Gym*): void

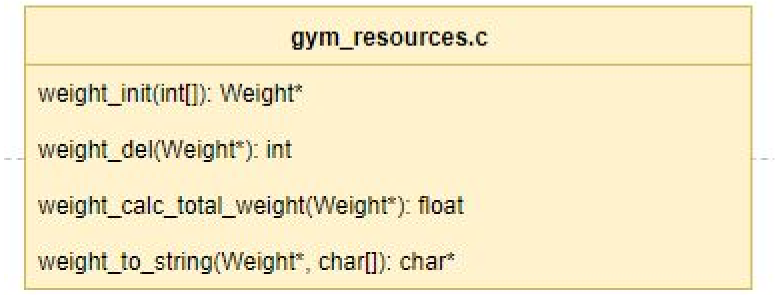
update_shared_gym(Gym*): void

copy_client(Client*, Client*): Client*

copy_trainer(Trainer*, Trainer*): Trainer*

delay(long): void

gym_resources.c


gym_resources.c

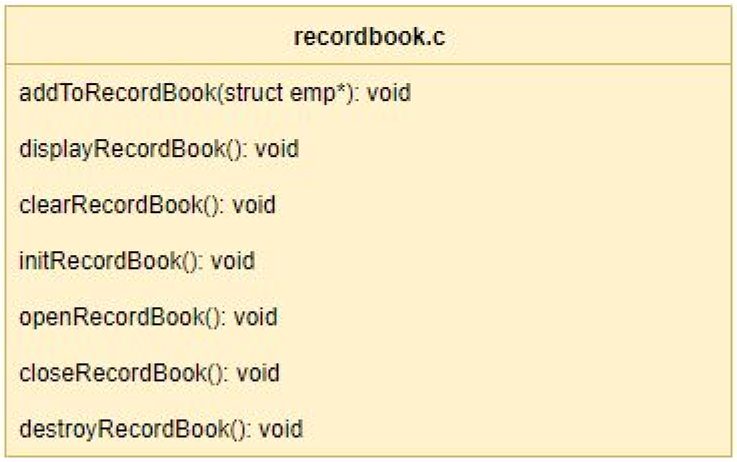
weight_init(int[]): Weight*

weight_del(Weight*): int

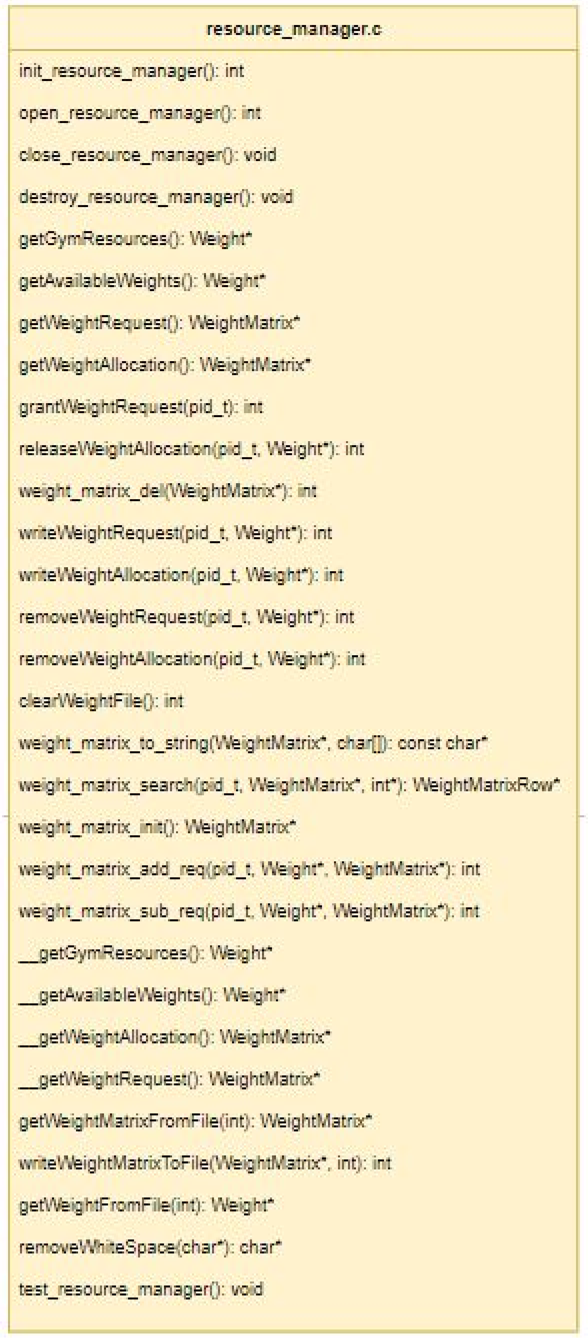
weight_calc_total_weight(Weight*): float

weight_to_string(Weight*, char[]): char*

recordbook.c



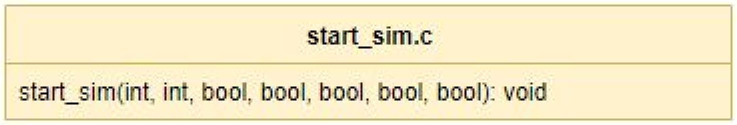
resource_manager.c



resource_manager.c

init_resource_manager(): int
open_resource_manager(): int
close_resource_manager(): void
destroy_resource_manager(): void
getGymResources(): Weight*
getAvailableWeights(): Weight*
getWeightRequest(): WeightMatrix*
getWeightAllocation(): WeightMatrix*
grantWeightRequest(pid_t): int
releaseWeightAllocation(pid_t, Weight*): int
weight_matrix_del(WeightMatrix*): int
writeWeightRequest(pid_t, Weight*): int
writeWeightAllocation(pid_t, Weight*): int
removeWeightRequest(pid_t, Weight*): int
removeWeightAllocation(pid_t, Weight*): int
clearWeightFile(): int
weight_matrix_to_string(WeightMatrix*, char[]): const char*
weight_matrix_search(pid_t, WeightMatrix*, int*): WeightMatrixRow*
weight_matrix_init(): WeightMatrix*
weight_matrix_add_req(pid_t, Weight*, WeightMatrix*): int
weight_matrix_sub_req(pid_t, Weight*, WeightMatrix*): int
__getGymResources(): Weight*
__getAvailableWeights(): Weight*
__getWeightAllocation(): WeightMatrix*
__getWeightRequest(): WeightMatrix*
getWeightMatrixFromFile(int): WeightMatrix*
writeWeightMatrixToFile(WeightMatrix*, int): int
getWeightFromFile(int): Weight*
removeWhiteSpace(char*): char*
test_resource_manager(): void

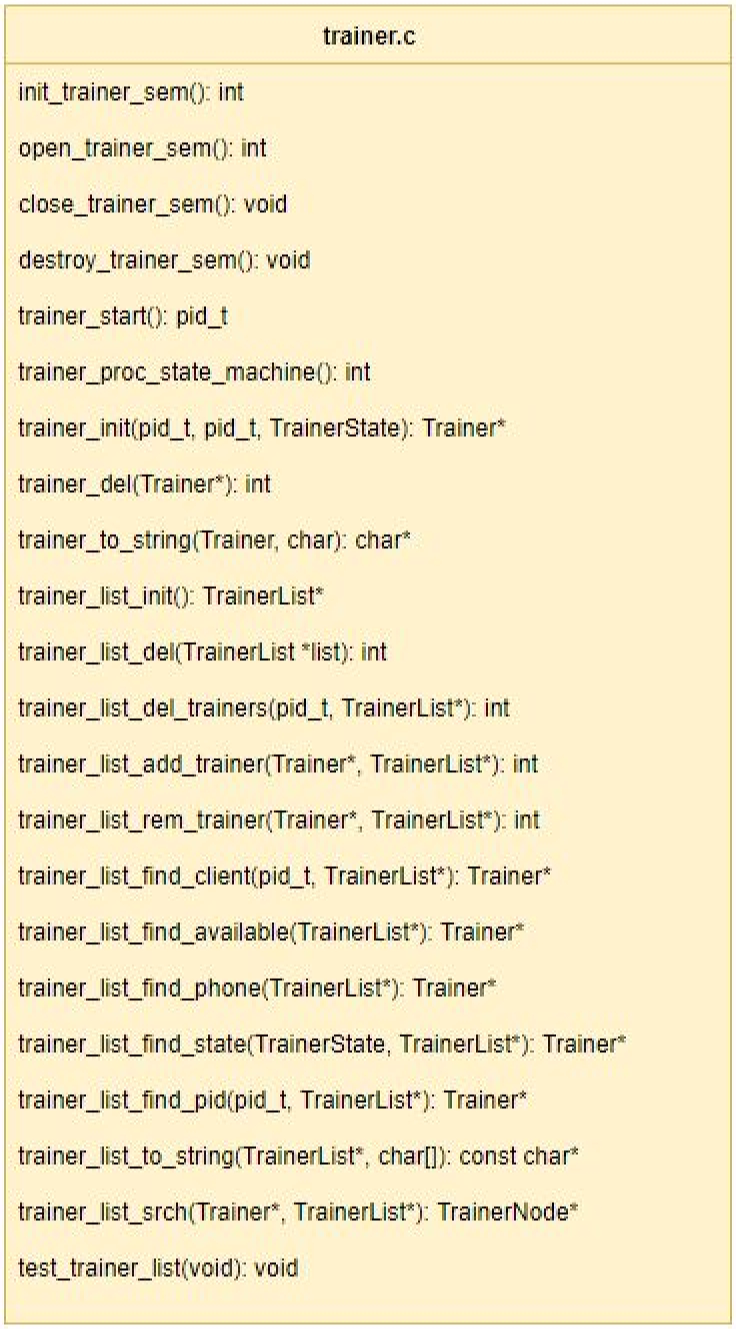
start_sim.c



start_sim.c

start_sim(int, int, bool, bool, bool, bool, bool): void

trainer.c


trainer.c

init_trainer_sem(): int

open_trainer_sem(): int

close_trainer_sem(): void

destroy_trainer_sem(): void

trainer_start(): pid_t

trainer_proc_state_machine(): int

trainer_init(pid_t, pid_t, TrainerState): Trainer*

trainer_del(Trainer*): int

trainer_to_string(Trainer, char): char*

trainer_list_init(): TrainerList*

trainer_list_del(TrainerList *list): int

trainer_list_del_trainers(pid_t, TrainerList*): int

trainer_list_add_trainer(Trainer*, TrainerList*): int

trainer_list_rem_trainer(Trainer*, TrainerList*): int

trainer_list_find_client(pid_t, TrainerList*): Trainer*

trainer_list_find_available(TrainerList*): Trainer*

trainer_list_find_phone(TrainerList*): Trainer*

trainer_list_find_state(TrainerState, TrainerList*): Trainer*

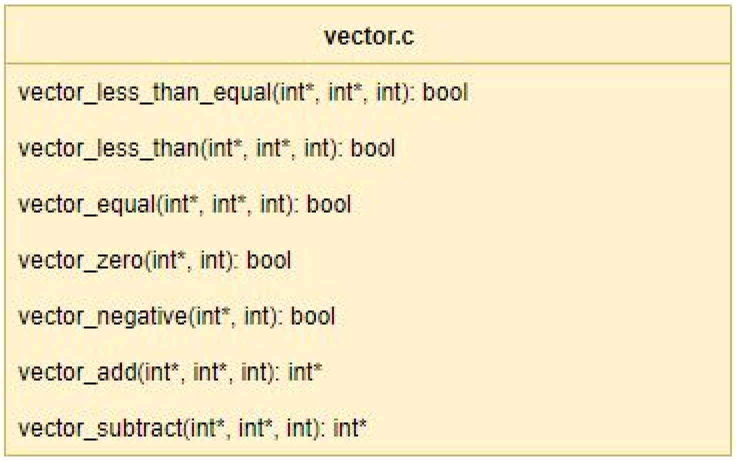
trainer_list_find_pid(pid_t, TrainerList*): Trainer*

trainer_list_to_string(TrainerList*, char[]): const char*

trainer_list_srch(Trainer*, TrainerList*): TrainerNode*

test_trainer_list(void): void

vector.c



vector.c

vector_less_than_equal(int*, int*, int): bool

vector_less_than(int*, int*, int): bool

vector_equal(int*, int*, int): bool

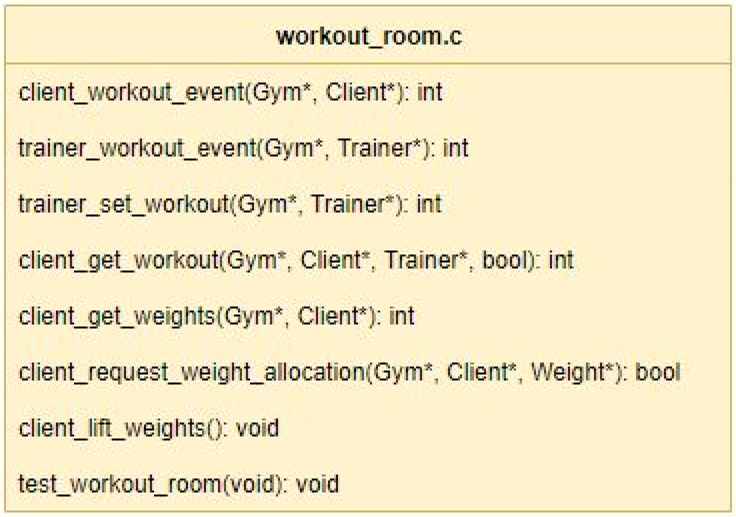
vector_zero(int*, int): bool

vector_negative(int*, int): bool

vector_add(int*, int*, int): int*

vector_subtract(int*, int*, int): int*

workout_room.c



workout_room.c

client_workout_event(Gym*, Client*): int

trainer_workout_event(Gym*, Trainer*): int

trainer_set_workout(Gym*, Trainer*): int

client_get_workout(Gym*, Client*, Trainer*, bool): int

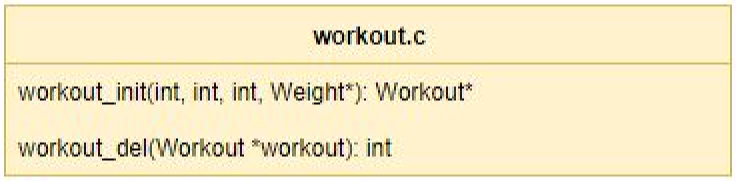
client_get_weights(Gym*, Client*): int

client_request_weight_allocation(Gym*, Client*, Weight*): bool

client_lift_weights(): void

test_workout_room(void): void

workout.c



workout.c

workout_init(int, int, int, Weight*): Workout*

workout_del(Workout *workout): int

Appendix B - Source Code

Header File Code

```
// ########################################
//
//    Author  -   Collin Thornton
//    Email   -   collin.thornton@okstate.edu
//    Brief   -   Final Project Client include
//    Date    -   11-15-20
//
// ########################################


#ifndef CLIENT_H
#define CLIENT_H

#include "trainer.h"
#include "workout.h"
#include "gym_resources.h"

#define MAX_CLIENTS 10

typedef enum {
    ARRIVING,
    WAITING,
    MOVING,
    TRAINING,
    LEAVING
} ClientState;

/**
 * @brief Maintain information relative to a specific client
 */
typedef struct Client {
    pid_t pid;
    ClientState state;

    Trainer current_trainer;
```

```c
    Couch current_couch;        //TODO Should this just be a semaphore?
    Workout workout;            // Set by trainer
} Client;

/**
 * @brief Node for a linked list of clients
 */
typedef struct ClientNode {
    Client* node;
    struct ClientNode* prev;
    struct ClientNode* next;
} ClientNode;

/**
 * @brief Linked list of clients
 */
typedef struct {
    ClientNode *HEAD, *TAIL;

    int len;
} ClientList;


// EACH CLIENT SHOULD BE ON A DIFFERENT THREAD
// - should maintain a finite state machine

//////////////////////////////
//
// Semaphore handling
//

/**
 * @brief Inititialize the client semaphore. Should be called on the
parent process
 * @return (int) return code. Negative on failure
 */
int init_client_sem();

/**
```

```
 * @brief Open the client semaphore. Should be called on the client
processes
 * @return (int) negative on failure
 */
int open_client_sem();

/**
 * @brief close the client semaphore. Should be called after
open_client_sem()
 */
void close_client_sem();

/**
 * @brief free the client semaphore. Should be called after
init_client_sem() on parent
 */
void destroy_client_sem();




//////////////////////////////
//
// Client process
//

/**
 * @brief Spawn a client child process. New process will launch
client_proc_state_machine()
 * @return (pid_t) Process ID of new child process
 */
pid_t client_start();

/**
 * @brief Execute the client state machine. Should be run by client
process
 * @return (int) return code. negative on error
 */
int client_proc_state_machine();
```

```c
//////////////////////////////
//
// Client struct
//

/**
 * @brief Initialize a client struct on the heap
 * @param pid (pid_t) Process ID of client
 * @param state (ClientState) Initial state of client
 * @param trainer (Trainer*) Current trainer (NULL if none)
 * @param couch (Couch*) Current couch of client (NULL if none)
 * @param worktout (Workout*) Current workout of client (NULL if none)
 * @return (Client*) Struct initialized on heap
 */
Client* client_init(pid_t pid, ClientState state, Trainer* trainer, Couch
*couch, Workout *workout);

/**
 * @brief Free a client struct from the heap
 * @param client (Client*) struct to be deleted
 * @return (int) return code. negative on error
 */
int client_del(Client* client);

/**
 * @brief stringify a client struct
 * @param client (Client*) struct to be stringified
 * @param buffer (char[]) string buffer
 * @return (const char*) same as buffer
 */
const char* client_to_string(Client *client, char buffer[]);


//////////////////////////////
//
// Client list
//
```

```c
/**
 * @brief Initalize a client LL on the heap
 * @return (ClientList*) newly allocated LL
 */
ClientList* client_list_init();

/**
 * @brief Delete a client LL from the heap
 * @param list (ClientList*) list to be deleted
 * @return (int) return code. negative on error
 */
int client_list_del(ClientList *list);

/**
 * @brief Delete clients from LL. Will free the client structs
 * @param exclude (pid_t) Client PID to exclude from deletion
 * @param list (ClientList*) LL from which to delete clients
 * @return (int) return code. negative on error
 */
int client_list_del_clients(pid_t exclude, ClientList *list);

/**
 * @brief Add a client to the LL
 * @param client (Client*) client to be added
 * @param list (ClientList*) list to which client is added
 * @return (int) return code. negative on error
 */
int client_list_add_client(Client *client, ClientList* list);

/**
 * @brief Remove a client from an LL
 * @param client (Client*) client to be removed
 * @param list (ClientList*) list from which to remove client
 * @return (int) return code. negative on error
 */
int client_list_rem_client(Client *client, ClientList *list);

/**
 * @brief stringify a client LL
 * @param list (ClientList*) LL to be stringified
 * @param bufer (char[]) buffer to store new string
 * @return (const char*) same as buffer
 */
const char* client_list_to_string(ClientList *list, char buffer[]);


/**
 * @brief search a client LL for a given client
 * @param client (Client*) needle
 * @param list (ClientList*) haystack
 * @return (ClientNode*) the node containing the requested client. NULL if
not found
 */
ClientNode* client_list_srch(Client *client, ClientList *list);

/**
 * @brief search a client LL for a given client
 * @param pid (pid_t) needle
 * @param list (ClientList*) haystack
 * @return (Client*) pointer to client struct. NULL if not found
 */
Client* client_list_find_pid(pid_t pid, ClientList *list);

/**
 * @brief search a client LL for a paired trainer
(client->current_trainer.pid)
 * @param pid (pid_t) needle -> pid of trainer
 * @param list (ClientList*) haystack
 * @return (Client*) poitner to client struct with specified trainer. NULL
if not found
 */
Client* client_list_find_trainer(pid_t pid, ClientList *list);


void test_client_list();

#endif // CLIENT_H// ##########################################
//
//   Author  -   Collin Thornton
//   Email   -   collin.thornton@okstate.edu
//   Brief   -   Final Project deadlock detection include
//   Date    -   11-23-20
//
// ##########################################


#ifndef DEADLOCK_DETECTION_H
#define DEADLOCK_DETECTION_H

#include <stdbool.h>

#include "resource_manager.h"


/**
 * @brief Check the input file for deadlocked processes.
 * @param deadlocked_array (pid_t[]) array with length of number of
processes (matrix rows)
 * @return (int) number of deadlocked processes
 */
int checkForDeadlock(pid_t deadlock_array[]);


void test_deadlock_detection(void);




#endif // DEADLOCK_DEFINITION_H//
##########################################
//
//   Author  -   Collin Thornton
//   Email   -   collin.thornton@okstate.edu
//   Brief   -   Final Project Gym include
//   Date    -   11-15-20
//
// ##########################################


#ifndef GYM_H
```

```c
#define GYM_H

#include <stdbool.h>
#include <fcntl.h>

#include "trainer.h"
#include "client.h"

#define BUFFER_SIZE 1024

// #define VERBOSE


/**
 * @brief Non-pointer version of Gym struct. Used for shared memory.
Represents total resources of gym
 */
typedef struct {
    // List of client on a couch (in waiting room)
    Client waitingList[MAX_CLIENTS];

    // List of clients arriving
    Client arrivingList[MAX_CLIENTS];

    // List of client training
    Client workoutList[MAX_CLIENTS];

    // List of trainers
    Trainer trainerList[MAX_TRAINERS];


    // PID of deadlock victim chosen by paren
    pid_t deadlock_victim;

    // Constants of gym
    int maxCouches;
    int num_trainers;
    int unit_time; // milliseconds
    // Execution flags
    bool boundary_case;
    bool realistic;
    bool fix_deadlock;
    bool detect_deadlock;
    bool trainer_log;

} SharedGym;


/**
 * @brief Represents total resources of gym as related to process
 */
typedef struct {
    // List of clients on a couch
    ClientList* waitingList;

    // List of arriving clients
    ClientList* arrivingList;

    // List of training clients
    ClientList* workoutList;

    // List of trainers
    TrainerList* trainerList;


    // PID of deadlock victim chosen by parent
    pid_t deadlock_victim;

    // Constants of gym
    int maxCouches;
    int num_trainers;
    int unit_time; // milliseconds

    // Execution flags
    bool boundary_case;
    bool realistic;
    bool fix_deadlock;
    bool detect_deadlock;
    bool trainer_log;
} Gym;



//////////////////////////////
//
// Gym funcitons
//


/**
 * @brief Initialize a gym on the heap
 * @return (Gym*) newly allocated struct
 */
Gym* gym_init();


/**
 * @brief Free a gym from the heap
 * @param gym (Gym*) gym to be freed
 */
void gym_del(Gym *gym);




//////////////////////////////
//
// Shared memory and semaphore functions
//


/**
 * @brief Initialize the shared memory space and semaphore. Should only be
called by parent process
 * @param maxCouches (int) Max number of clients in waiting room
 * @param numTrainers (int) total number of trainers in simulation
 * @param boundary_case (bool) Flag to solve for Part B
 * @param realistic (bool) Flag to toggle how clients choose weights
 * @param detectDeadlock (bool) Flag to solve for Part C
 * @param fixDeadlock (bool) Flag to solve for Part D
 * @param trainerLog (bool) Flag to solve for Part E
 */
int init_shared_gym(int maxCouches, int numTrainers, bool boundary_case,
bool realistic, bool detectDeadlock, bool fixDeadlock, bool trainerLog);


/**
 * @brief Open the shared memory and semaphore in the current process.
 */
void open_shared_gym();


/**
 * @brief Close the shared memory and semaphore in the current process
 */
void close_shared_gym();


/**
 * @brief Free the shared memory and semaphore. Should follow a call to
init_share_gym() in the parent process.
 */
void destroy_shared_gym();



///////////////////////////////
//
// Update shared and local memory
//


/**
 * @brief Copy a locally stored, LL-based gym, to the shared memory space.
Will only modify structs with the current process ID
 * @param gym (Gym*) gym to be copied
 */
void update_gym(Gym *gym);
```

```c
/**
 * @brief Copy the shared space, array-based gym, to the local memory
space. Will not modify structs with the current process ID
 */
void update_shared_gym(Gym* gym);



//////////////////////////////
//
// Helper functions
//


/**
 * @brief Copy a client struct from src to dest
 * @param dest (Client*) Destination of copy
 * @param src (Client*) Source of copy
 * @return (Client*) same as dest
 */
Client* copy_client(Client *dest, Client *src);


/**
 * @brief Copy a trainer from src to dest
 * @param src (Trainer*) destination of copy
 * @param dest (Trainer*) src of copy
 * @return (Trainer*) same as dest
 */
Trainer* copy_trainer(Trainer* dest, Trainer *src);


/**
 * @brief Delay in milliseconds
 * @param mS (long) delay time
 */
void delay(long mS);
```

```c
#endif // GYM_H// ##########################################
//
//   Author  -   Collin Thornton
//   Email   -   collin.thornton@okstate.edu
//   Brief   -   Final Project Gym include
//   Date    -   11-15-20
//
// ##########################################

#ifndef GYM_RESOURCES_H
#define GYM_RESOURCES_H

#include <semaphore.h>

#define NUMBER_WEIGHTS 8

#define MIN_COUCHES 3
#define MAX_COUCHES 6


typedef struct {
    sem_t couch_mutex;       //! MAY BE EASIER TO JUST MAKE A SEMAPHORE
} Couch;                     //TODO Remove this. It's unused.


typedef enum {
    TWO_HALF,
    FIVE,
    TEN,
    FIFTEEN,
    TWENTY,
    TWENTY_FIVE,
    THIRTY_FIVE,
    FORTY_FIVE
} PlateIndex;


/**
 * @brief Organize the grip plates
 */
```

```c
typedef struct {
    int num_plates[NUMBER_WEIGHTS];                         // Use PlateIndex
as index for the array (will help w/ keeping track)
    float total_weight;                                     // Summation of
plate weights
} Weight;




/**
 * @brief Allocate Weight struct on heap
 * @param plate_array (int[NUMBER_WEIGHTS]) Array with each indice a
number of grip plates of corresponding PlateIndice. Set to NULL if not yet
known
 * @return (Weight*) struct allocated on heap
 */
Weight* weight_init(int plate_array[NUMBER_WEIGHTS]);


/**
 * @brief free a Weight struct from the heap
 * @param weight (Weight*) struct to be freed
 * @return (int) return code. negative on failure
 */
int weight_del(Weight *weight);


/**
 * @brief Calculate the total weight represented by the num_plates array
 * @param weight (Weight*) pointer to a Weight struct
 * @return (float) total weight represented by that struct
 */
float weight_calc_total_weight(Weight *weight);


/**
 * @brief stringify a weight struct
 * @param weight (Weight*) struct to be stringified
 * @param buffer (char[]) buffer to store string
 * @return (const char*) same as buffer
 */
const char* weight_to_string(Weight *weight, char buffer[]);

#endif // GYM_RESOURCES_H
#ifndef RECORDKEEPING_RECORDBOOK_H
#define RECORDKEEPING_RECORDBOOK_H

#include <pthread.h>

#define MAX_NAME_LEN 100


typedef struct emp
{
    char name[MAX_NAME_LEN];
    int id;
    int weight;
} Emp;

void addToRecordBook(struct emp *empValue);
void displayRecordBook();
void clearRecordBook();
void initRecordBook();
void openRecordBook();
void closeRecordBook();
void destroyRecordBook();

void test_recordbook();

#endif //RECORDKEEPING_RECORDBOOK_H
// ##########################################
//
//    Author  -   Collin Thornton
//    Email   -   collin.thornton@okstate.edu
//    Brief   -   Final Project gym resource include
//    Date    -   11-15-20
//
// ##########################################
```

```c
#ifndef RESOURCE_MANAGER_H
#define RESOURCE_MANAGER_H

#include <semaphore.h>

#include "gym.h"


typedef struct {
    pid_t pid;
    Weight *weight;
} WeightMatrixRow;

typedef struct{
    WeightMatrixRow* rows;
    int num_rows;
} WeightMatrix;


//////////////////////////////
//
//  USER FUNCTIONS
//


// FUNTIONS TO INITALIZE THE SEMAPHORE

/**
 * @brief Initialize the semaphore. Should only be ran on the parent
process
 * @return (int) return code. negative on error
 */
int init_resource_manager();

/**
 * @brief Open the semaphore on the current process.
 * @return (int) return code. Negative on error
 */
```

```
int open_resource_manager();

/**
 * @brief Close the semaphore on the current process
 */
void close_resource_manager();

/**
 * @brief Desotry the semaphore. Should only be ran on the parent process.
 */
void destroy_resource_manager();


// FUNCTIONS TO GET WEIGHT REQUESTS FROM FILE


/**
 * @brief return the gym's total resources. must be deleted with
weight_del()
 * @return (Weight*) Vector of total weights
 */
Weight* getGymResources();

/**
 * @brief return currently available weights
 * @return (Weight*) vector of current weight
 */
Weight* getAvailableWeights();


/**
 * @brief return the current weight requests. deleted with
weight_matrix_del
 * @return (WeightMatrix*) matrix of requests allocated on heap
 */
WeightMatrix* getWeightRequest();


/**
```

```
 * @brief return the currently allocated weights. deleted with
weight_matrix_del
 * @return (WeightMatrix*) matrix of allocations allocated on heap
 */
WeightMatrix* getWeightAllocation();


/**
 * @brief removes the request, allocates weights, and adjusts the
currently available weights
 * @param pid (pid_t) process to grant
 * @return (int) return code. negative on error
 */
int grantWeightRequest(pid_t pid);

/**
 * @brief removes the allocation and adjusts currently available weights
 * @param pid (pid_t) process id to adjust
 * @param weight (Weight*) amount to change
 * @return (int) return code. negative on error
 */
int releaseWeightAllocation(pid_t pid, Weight* weight);


/**
 * @brief Free a WeightMatrix*, and all internal WeightRows* and Weights*
 * @param matrix (WeightMatrix*) matrix to be deleted
 * @return (int) 0 on success
 */
int weight_matrix_del(WeightMatrix *matrix);

// FUNCTIONS TO WRITE WEIGHT REQUESTS TO FILE

/**
 * @brief write a new request to the file
 * @param pid (pid_t) pid of process
 * @param weight (Weight*) new request
 * @return (int) negative on failure
 */
int writeWeightRequest(pid_t pid, Weight *weight);
```

```
/**
 * @brief write a new allocation to the file
 * @param pid (pid_t) pid of process
 * @param weight (Weight*) new allocation
 * @return (int) negative on failure
 */
int writeWeightAllocation(pid_t pid, Weight *weight);


// FUNCTIONS TO REMOVE WEIGHT REQUEST FROM FILE

/**
 * @brief remove a request from the file. will throw error if result is
negative
 * @param pid (pid_t) pid of process
 * @param weight (Weight*) weight to be subtracted
 * @return (int) negative on failure
 */
int removeWeightRequest(pid_t pid, Weight *weight);


/**
 * @brief remove an allocation from the file. will throw error if result
is negative
 * @param pid (pid_t) pid of process
 * @param weight (Weight*) weight to be subtracted
 * @return (int) negative on failure
 */
int removeWeightAllocation(pid_t pid, Weight *weight);


/**
 * @brief clear allocation and request matrices from file
 * @return (int) negative on failure
 */
int clearWeightFile();
```

```
/**
 * @brief Return a string representative of matrix
 * @param matrix (WeightMatrix*) matrix to be returned as string
 * @param buffer (char[]) buffer to store string output
 * @return (const char*) pointer to string. same as buffer
 */
const char* weight_matrix_to_string(WeightMatrix *matrix, char buffer[]);


/**
 * @brief Search a WeightMatrix for a pid
 * @param pid (pid_t) pid for which to search
 * @param matrix (WeightMatrix*) matrix to be searched
 * @param row_number (int*) storage for the row number. negative if row
not found. can be set to NULL if not neededd
 * @return (WeightMatrixRow*) pointer to the row. NULL if not found
 */
WeightMatrixRow* weight_matrix_search(pid_t pid, WeightMatrix *matrix, int
*row_number);




//////////////////////////////
//
//  HELPER FUNCTIONS
//


/**
 * @brief Private function. Initailize a WeightMatrix on heap. All values
NULL or 0
 * @return (WeightMatrix*) pointer to new matrix
 */
static WeightMatrix* weight_matrix_init();



/**
```

```
 * @brief Add a weight request to a weight matrix. Will add new row if pid is not found
 * @param pid (pid_t) pid of requesting process
 * @param weight (Weight*) weight to be added
 * @param matrix (WeightMatrix*) matrix to store summation
 * @return (int) number of rows in matrix. Negative on error
 */
static int weight_matrix_add_req(pid_t pid, Weight *weight, WeightMatrix *matrix);


/**
 * @brief Subtract a weight request from a weight matrix. Will delete a row if result is 0 vector
 * @param pid (pid_t) pid of process
 * @param weight (Weight*) weight to be subtracted
 * @param matrix (WeightMatrix*) matrix to store difference
 * @return (int) nuber of rows in matrix. Negative on error
 */
static int weight_matrix_sub_req(pid_t pid, Weight *weight, WeightMatrix *matrix);


/**
 * @brief Private function. Not locked with semaphore. See getGymResources()
 */
static Weight* __getGymResources();

/**
 * @brief Privat function. Not locked with semaphore. See getAvalaibleWeights()
 */
static Weight* __getAvailableWeights();

/**
 * @brief Private function. Not locked with semaphore. See getWeightAllocation()
 */
static WeightMatrix* __getWeightAllocation();
```

```
/**
 * @brief Private function. Not locked with semaphore. See
getWeightRequest()
 */
static WeightMatrix* __getWeightRequest();


/**
 * @brief read a weight matrix from the input file
 * @param section (unsigned int) section number from which to read
 * @return (WeightMatrix*) pointer to weight matrix on heap
 */
static WeightMatrix* getWeightMatrixFromFile(unsigned int section);


/**
 * @brief Private function. Not locked with semaphore. write a weight
matrix to the input file. will delete the matrix
 * @param matrix (WeightMatrix*) matrix to be written
 * @param section (int) section number at which to write
 * @return (int) negative on failure
 */
static int writeWeightMatrixToFile(WeightMatrix *matrix, int section);


/**
 * @brief Private function. Not locked with semaphore. get a weight from
the input file
 * @param section (unsigned int) section number from which to read
 * @return (Weight*) allocation on heap
 */
static Weight* getWeightFromFile(unsigned int section);

/**
 * @brief Private function. remove all whitespace from a string
 * @param str (char*) input string. will be changed
 * @return (const char*) output string. same as str
 */
static char* removeWhiteSpace(char* str);
```

```c
void test_resource_manager();

#endif // RESOURCE_MANAGER_H// ###########################################
//
//    Author  -   Collin Thornton
//    Email   -   collin.thornton@okstate.edu
//    Brief   -   Final Project start include
//    Date    -   11-30-20
//
// ###########################################


#ifndef PART2_H
#define PART2_H

#include <stdbool.h>

#include "workout_room.h"
#include "deadlock.h"
#include "recordbook.h"


/**
 * @brief Startup the sim with flags. Used in driver files
 * @param num_trainer (int) total number of trainers in simulation
 * @param num_couches (int) max clients in waiting room
 * @param boundary_case (const bool) solve for part b
 * @param realistc (const bool) toggle realistic weight algorithm in
client
 * @param detect_deadlock (const bool) solve for part c
 * @param fix_deadlock (const bool) solve for part d
 * @param trainer_log (const bool) solve for part e
 */
void start_sim(int num_trainers, int num_couches, const bool
boundary_case, const bool realistic,
    const bool detect_deadlock, const bool fix_deadlock, const bool
trainer_log);
```

```
#endif // PART2_H// ##########################################
//
//   Author  -   Collin Thornton
//   Email   -   collin.thornton@okstate.edu
//   Brief   -   Final Project Trainer include
//   Date    -   11-15-20
//
// ##########################################

#ifndef TRAINER_H
#define TRAINER_H

#include <stdlib.h>
#include <sys/wait.h>

#include "workout.h"

#define MIN_TRAINERS 3
#define MAX_TRAINERS 5


typedef enum {
    FREE,
    ON_PHONE,
    TRAVELLING,
    WITH_CLIENT
} TrainerState;

/**
 * @brief Maintain information relative to a specific trainer
 */
typedef struct Trainer {
    pid_t pid;
    pid_t client_pid;
    TrainerState state;
    Workout workout;
} Trainer;
```

```
/**
 * @brief Node for a linked list of trainers
 */
typedef struct TrainerNode {
    Trainer* node;
    struct TrainerNode* prev;
    struct TrainerNode* next;
} TrainerNode;

/**
 * @brief Linked list of trainers
 */
typedef struct {
    TrainerNode *HEAD, *TAIL;

    int len;
} TrainerList;


// EACH CLIENT SHOULD BE ON A DIFFERENT PROCESS
// - should maintain a finite state machine

//////////////////////////////
//
// Semaphore handling
//


/**
 * @brief Initalize the trainer semaphore. Should be called on the parent
process.
 * @return (int) return code. negative on failure.
 */
int init_trainer_sem();


/**
 * @brief Open the trainer sempaahore. SHould be called on the trainer
process
 * @return (int) return code. negative on error
 */
int open_trainer_sem();


/**
 * @brief close the trainer semaphore. should be called after
open_trainer_sem()
 */
void close_trainer_sem();


/**
 * @brief free the trainer semaphore. should be called after
init_trainer_sem() on parent
 */
void destroy_trainer_sem();



//////////////////////////////
//
// Trainer process
//



/**
 * @brief Spawn a trainer child process. new process will launch
trainer_proc_state_machin()
 * @return (pid_t) Process ID of new child process
 */
pid_t trainer_start();


/**
 * @brief Execute the trainer state machine. Should only be run by trianer
proccess
 * @return (int) return code. negative on error
 */
int trainer_proc_state_machine();
```

```
//////////////////////////////
//
// Client struct
//

/**
 * @brief Allocate trainer on heap. Init params as NULL or negative if
unavailable
 * @param pid (pid_t) Process ID of trainer
 * @param client_pid (pid_t) Process ID of client. -1 if unavailable
 * @param state (TrainerState) Inital state of trainer
 * @return (Trainer*) pointer to a trainer struct
 */
Trainer* trainer_init(pid_t pid, pid_t client_pid, TrainerState state);


/**
 * @brief Delete a trainer from the heap
 * @param trainer (Trainer*) trainer to be deleted
 * @return (int)return code. negative on erro
 */
int trainer_del(Trainer* trainer);


/**
 * @brief Stringify a trainer struct
 * @param trainer (Traienr*) struct to be stringified
 * @param buffer (char[]) buffer to store new string
 * @return (const char*) same as buffer
 */
const char* trainer_to_string(Trainer *trainer, char buffer[]);


//////////////////////////////
//
// Trainer list
//
```

```
/**
 * @brief Initalize a trainer LL on the heap
 * @return (TrainerList*) pointer to newly allocated LL
 */
TrainerList* trainer_list_init();


/**
 * @brief Free a trainer list and all internal nodes. Does not free the
individual trainers
 * @param list (TrainerList*) Pointer to the current shared list
 * @return (int) return code. negative on error
 */
int trainer_list_del(TrainerList *list);


/**
 * @brief Delete trainers from a given list
 * @param exclude (pid_t) Process ID of exclude trainer
 * @param list (TrainerList*) List of current trainers
 * @return (int) return code. negative on error
 */
int trainer_list_del_trainers(pid_t exclude, TrainerList *list);


/**
 * @brief Add a trainer to a list
 * @param trainer (Trianer*) trainer to be added
 * @param list (TrainerList*) target list
 * @return (int) return code. negative on error
 */
int trainer_list_add_trainer(Trainer *trainer, TrainerList* list);


/**
 * @brief Remove a trainer from the list
 * @param trainer (Trainer*) trainer to be removed
 * @param list (TrainerList*) list from which to remove the trainer
 * @return (int) return code. negative on error.
 */
int trainer_list_rem_trainer(Trainer *trainer, TrainerList *list);


/**
 * @brief Search for a client in trainer LL
 * @param client_pid (pid_t) needle -> pid of client in question
 * @param list (TrainerList*) poitner to LL with client
 * @return (Trainer*) null if not found, otherwise pointer to the trainer
 */
Trainer* trainer_list_find_client(pid_t client_pid, TrainerList *list);


/**
 * @brief Find a trainer currently on their available
 * @param list (TrainerList*) list to be searched
 * @return (Trainer*) null if not found, othewise poitner to the trainer
 */
Trainer* trainer_list_find_available(TrainerList *list);


/**
 * @brief Find a trainer currently on their phone
 * @param list (TrainerList*) list to be searched
 * @return (Trainer*) null if not found, othewise poitner to the trainer
 */
Trainer* trainer_list_find_phone(TrainerList *list);


/**
 * @brief Find a trainer currently at a given state
 * @param list (TrainerList*) list to be searched
 * @return (Trainer*) null if not found, othewise poitner to the trainer
 */
Trainer* trainer_list_find_state(TrainerState state, TrainerList *list);


/**
 * @brief Find a trainer currently with pid in LL
 * @param list (TrainerList*) list to be searched
 * @return (Trainer*) null if not found, othewise poitner to the trainer
 */
Trainer* trainer_list_find_pid(pid_t pid, TrainerList *list);


/**
 * @brief Stringify a trainer list
 * @param list (TrainerList*) list to be stringified
 * @param buffer (char[]) buffer to store new string
 * @return (const char*) same as buffer
 */
const char* trainer_list_to_string(TrainerList *list, char buffer[]);


/**
 * @brief Search a trainer LL for a given trainer
 * @param trainer (Trainer*) needle
 * @param list (TrainerLIst*) haystack
 * @return (TrainerNode*) pointer to node containing the Trainer struct
 */
TrainerNode* trainer_list_srch(Trainer *trainer, TrainerList *list);


void test_trainer_list(void);

#endif // TRAINER_H// ###########################################
//
//    Author  -   Collin Thornton
//    Email   -   collin.thornton@okstate.edu
//    Brief   -   Final Project vector include
//    Date    -   11-24-20
//
// ###########################################

#ifndef VECTOR_H
#define VECTOR_H


/**
 * @brief test if v1 <= v2
 * @param v1 (int*) left vector
 * @param v2 (int*) right vector
 * @param size (int) number of elements in vectors. assumed to be same for
both
 * @return (bool) true if v1 <= v2. else false
 */
bool vector_less_than_equal(int *v1, int *v2, int size);

/**
 * @brief test if v1 < v2
 * @param v1 (int*) left vector
 * @param v2 (int*) right vector
 * @param size (int) number of elements in vectors. assumed to be same for
both
 * @return (bool) true if v1 < v2. else false
 */
bool vector_less_than(int *v1, int *v2, int size);

/**
 * @brief test if v1 == v2
 * @param v1 (int*) left vector
 * @param v2 (int*) right vector
 * @param size (int) number of elements in vectors. assumed to be same for
both
 * @return (bool) true if v1 == v2. else false
 */
bool vector_equal(int *v1, int *v2, int size);


/**
 * @brief test if v1[i] == 0 for all i < size
 * @param v1 (int*) vector to check
 * @param size (int) number of elements to check
 * @return (bool) true if v1[i] == 0 for all i < size. else false
 */
bool vector_zero(int *v1, int size);

/**
 * @brief check if v1[i] < 0 for any i < size
 * @param v1 (int*) vector to check
 * @param size (int) number of elements to check
 * @return (bool) true of any v1[i] < 0
 */
bool vector_negative(int *v1, int size);



/**
 * @brief perform v1[i] += v2[i] for all i < size
 * @param v1 (int*) left vector
 * @param v2 (int*) right vector
 * @param size (int) number of elements to add
 * @return (int*) v1 = v1 + v2
 */
int* vector_add(int *v1, int *v2, int size);


/** @brief perform v1[i] -= v2[i] for all i < size
 * @param v1 (int*) left vector
 * @param v2 (int*) right vector
 * @param size (int) number of elements to add
 * @return (int*) v1 = v1 - v2
 */
int* vector_subtract(int *v1, int *v2, int size);



#endif // VECTOR_H// ###########################################
//
//    Author  -   Collin Thornton
//    Email   -   collin.thornton@okstate.edu
//    Brief   -   Final Project workout include
//    Date    -   11-26-20
//
// ###########################################



#ifndef WORKOUT_H
#define WORKOUT_H
```

```c
#include "gym_resources.h"

#define MAX_WEIGHT 500
#define MIN_WEIGHT 100

// Store data for a single workout
typedef struct {
    int total_sets;
    int sets_left;

    int total_weight;
    Weight in_use;
} Workout;



//////////////////////////////
//
// Workout functions
//



/**
 * @brief Allocate a workout on the heap
 * @param total_sets (int) Total sets in workout
 * @param sets_left (int) Sets left in workout
 * @param total_weight (int) Total weight of workout
 * @param in_use (Weight*) Points to weights currently being used
 */
Workout* workout_init(int total_sets, int sets_left, int total_weight,
Weight *in_use);


/**
 * @brief Free a workout from the heap
 * @param workout (Worktout*) workout to be deleted
 * @return (int) return code. negative on failure
 */
int workout_del(Workout *workout);
```

```
#endif // WORKOUT_H// ##########################################
//
//    Author  -   Collin Thornton
//    Email   -   collin.thornton@okstate.edu
//    Brief   -   Final Project par2 include
//    Date    -   11-20-20
//
// ##########################################


#ifndef WORKOUT_ROOM_H
#define WORKOUT_ROOM_H

#include <stdbool.h>

#include "gym.h"
#include "client.h"




//////////////////////////////
//
// Event functions
//



/**
 * @brief Execute the workout event on client process.
 * @param gym (Gym*) gym struct used for IPC
 * @param client (Client*) client in workout event
 * @return (int) return code. negative on error
 */
int client_workout_event(Gym *gym, Client *client);



/**
 * @brief Execute the workout event on trainer proccess.
 * @param gym (Gym*) gym struct used for IPC
 * @param trianer (Trainer*) trainer in workout event
 * @return (int) return code. negative on error
 */
int trainer_workout_event(Gym *gym, Trainer *trainer);



//////////////////////////////
//
// Helper functions
//



/**
 * @brief Choose total weight and total sets. Place in IPC
 * @param gym (Gym*) gym struct for IPC
 * @param trainer (Trainer*) trainer in event
 * @return (int) return code. negative on error
 */
int trainer_set_workout(Gym *gym, Trainer *trainer);


/**
 * @brief Get total weight and total sets from trianer over shared memory
 * @param gym (Gym*) gym struct for IPC
 * @param client (Client*) client in event
 * @param trainer (Trainer*) trainer that's paired with client
 * @param first_time (bool) flag to toggle whether the trainer can change
the total number of sets
 */
int client_get_workout(Gym *gym, Client *client, Trainer *trainer, bool
first_time);


/**
 * @brief Get grip plates from the resource manager
 * @param gym (Gym*) gym struct for IPC
 * @param client (Client*) client in event
 */
int client_get_weights(Gym *gym, Client *client);
```

```c
/**
 * @brief Make a weight request from the resource manaegr
 * @param gym (Gym*) gym struct for IPC
 * @param client (Client*) client in event
 * @param weight (Weight*) request
 * @return bool true on succes. else false
 */
bool client_request_weight_allocation(Gym *gym, Client *client, Weight
*weight);


/**
 * @brief Delay for a bit and set flags as the client lifts weights
 */
void client_lift_weights();


void test_workout_room(void);

#endif // WORKOUT_ROOM_H
```

C File Code

```c
// ##################################################
//
//   Author  -   Collin Thornton
//   Email   -   collin.thornton@okstate.edu
//   Brief   -   Final Project client struct source
//   Date    -   11-16-20
//
// ##################################################

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
#include <unistd.h>
#include <sys/wait.h>
#include <semaphore.h>
```

```
#include "client.h"
#include "gym.h"
#include "workout_room.h"
#include "resource_manager.h"


// Initialize semaphore names
static const char CLIENT_ARRIVING_SEM_NAME[] = "/sem_client_arriving";
static const char CLIENT_WAITING_SEM_NAME[] = "/sem_client_waiting";

// Declare semaphores
static sem_t *client_arriving_sem;
static sem_t *client_waiting_sem;


//////////////////////////////
//
// Client process functions
//


/**
 * @brief Spawn a client child process. New process will launch client_proc_state_machine()
 * @return (pid_t) Process ID of new child process
 */
pid_t client_start() {
    pid_t pid = fork();

    if(pid < 0) {
        perror("client_start() fork");
        return pid;
    }
    else if(pid == 0) {
        printf("new client pid: %d\r\n", getpid());
        int ret = client_proc_state_machine();
        exit(ret);
    } else {
        return pid;
    }
}


/**
 * @brief Execute the client state machine. Should be run by client process
 * @return (int) return code. negative on error
 */
int client_proc_state_machine() {

    // Intialize semaphores and shared memory

    open_client_sem();
    open_resource_manager();

    int pid = getpid();

    Gym *gym = gym_init();
    open_shared_gym();
    update_gym(gym);

    if(gym == NULL) {
        perror("client_proc_state_machine() get_shared_gym");
        return -1;
    }


    // Intialize a new client struct
    Client *client = client_init(getpid(), ARRIVING, NULL, NULL, NULL);
    Trainer *trainer;

    char buffer[BUFFER_SIZE] = "\0";

    bool shutdown = false;
    bool has_couch = false;


    while(!shutdown) {
        // Execute state machine

        switch(client->state) {
            case ARRIVING:

                if(gym->boundary_case) {
                    sem_wait(client_arriving_sem);
                }

                printf("CLIENT %d ARRIVING\r\n", pid);

                client_list_add_client(client, gym->arrivingList);
```

```
update_shared_gym(gym);


if(gym->boundary_case) {
    // WAIT FOR ALL TRAINERS TO FINISH TRAVELLING

    trainer = trainer_list_find_state(TRAVELLING, gym->trainerList);

    while(trainer != NULL) {
        delay(1*gym->unit_time);
        update_gym(gym);
        trainer = trainer_list_find_state(TRAVELLING, gym->trainerList);
    }
}

// NOW FIND TRAINER ON PHONE
update_gym(gym);

trainer = trainer_list_find_phone(gym->trainerList);
while(trainer != NULL && trainer->client_pid > 0) {
    update_gym(gym);
    trainer = trainer_list_find_phone(gym->trainerList);
    delay(1*gym->unit_time);
}

if(trainer != NULL && trainer->client_pid <= 0) {
    client->current_trainer = *trainer;
    update_shared_gym(gym);

    while(trainer != NULL && trainer->client_pid != getpid()) {
        // wait for the trainer to also claim us

        update_gym(gym);
        trainer = trainer_list_find_pid(client->current_trainer.pid, gym->trainerList);
        delay(1*gym->unit_time);
    }
    client->state = TRAINING;
}
else {
    client->state = MOVING;
}

delay(2*gym->unit_time);
client_list_rem_client(client, gym->arrivingList);
```

```
            // CLOSE SEMAPHORE
            if(gym->boundary_case) {
                sem_post(client_arriving_sem);
            }

            break;

        case WAITING:

            if(gym->boundary_case)
                sem_wait(client_waiting_sem);

            if(gym->waitingList->len >= gym->maxCouches && !has_couch) {
                client->state = LEAVING;

                // RELEASE SEMAPHORE
                if(gym->boundary_case)
                    sem_post(client_waiting_sem);

                printf("CLIENT %d WAITING ROOM FULL WITH %d COUCHES OCCUPIED.
LEAVING\r\n", pid, gym->waitingList->len);

                break;
            }

            if(!gym->boundary_case)
                delay(1*gym->unit_time);

            client_list_add_client(client, gym->waitingList);
            update_shared_gym(gym);


            if(!has_couch)
                printf("CLIENT %d WAITING. %d OF %d COUCHES TAKEN\r\n", getpid(),
gym->waitingList->len, gym->maxCouches);

            has_couch = true;


            // Check if a trainer has picked us up
            trainer = trainer_list_find_client(getpid(), gym->trainerList);
            if(trainer != NULL) {
                has_couch = false;
            client->state = TRAINING;
            client_list_rem_client(client, gym->waitingList);
            copy_trainer(&client->current_trainer, trainer);
        }

        delay(1*gym->unit_time);

        if(gym->boundary_case)
            sem_post(client_waiting_sem);
        break;

    case MOVING:

        printf("CLIENT %d MOVING\r\n", pid);

        delay(6*gym->unit_time);
        client->state = WAITING;
        break;

    case TRAINING:
        printf("CLIENT %d TRAINING\r\n", pid);

        client_list_add_client(client, gym->workoutList);
        update_shared_gym(gym);

        delay(2*gym->unit_time);

        client_workout_event(gym, client);
        client->state = LEAVING;

        break;

    case LEAVING:
        printf("CLIENT %d LEAVING\r\n", pid);

        shutdown = true;
        break;

    default:
        printf("CLIENT %d UNKOWN STATE \r\n", getpid());

        shutdown = true;
}
```

```
        // Update local & shared memory
        update_shared_gym(gym);
        update_gym(gym);
    }

    // Remove client from any lists & close semaphores

    printf("Client %d destroying data\r\n", getpid());

    client_list_rem_client(client, gym->arrivingList);
    client_list_rem_client(client, gym->waitingList);
    client_list_rem_client(client, gym->workoutList);
    update_shared_gym(gym);

    client_del(client);
    gym_del(gym);

    close_shared_gym();
    close_resource_manager();
    close_trainer_sem();
    close_client_sem();

    printf("Client %d exiting\r\n", getpid());

    return 0;
}





//////////////////////////////
//
// Client struct functions
//


/**
 * @brief Initialize a client struct on the heap
 * @param pid (pid_t) Process ID of client
 * @param state (ClientState) Initial state of client
 * @param trainer (Trainer*) Current trainer (NULL if none)
 * @param couch (Couch*) Current couch of client (NULL if none)
 * @param worktout (Workout*) Current workout of client (NULL if none)
 * @return (Client*) Struct initialized on heap
 */
Client* client_init(pid_t pid, ClientState state, Trainer* trainer, Couch* couch, Workout* workout)
{
    Client* client = (Client*)malloc(sizeof(Client));

    if(client == NULL) {
        perror("client_init malloc()");
        return NULL;
    }

    client->pid = pid;
    client->state = state;

    if(trainer != NULL)
        client->current_trainer = *trainer;

    else
    {
        Trainer *tmp = trainer_init(-1, -1, FREE);
        client->current_trainer = *tmp;
        trainer_del(tmp);
    }

    if(couch != NULL)
        client->current_couch = *couch;

    if(workout != NULL)
        client->workout = *workout;

    else
    {
        Workout *tmp = workout_init(-1, -1 -1, -1, NULL);
        client->workout = *tmp;
        workout_del(tmp);
    }

    return client;
}


/**
 * @brief Free a client struct from the heap
 * @param client (Client*) struct to be deleted
 * @return (int) return code. negative on error
 */
int client_del(Client* client) {
    if(client == NULL) return -1;

    free(client);
    return 0;
}


/**
 * @brief stringify a client struct
 * @param client (Client*) struct to be stringified
 * @param buffer (char[]) string buffer
 * @return (const char*) same as buffer
 */
const char* client_to_string(Client *client, char buffer[]) {
    if(client == NULL) return NULL;

    sprintf(buffer, "pid: %d", client->pid);
    sprintf(buffer + strlen(buffer), "   state: %d", client->state);

    sprintf(buffer + strlen(buffer), "   trainer: %d", client->current_trainer.pid);

    int sem_val;
    sprintf(buffer + strlen(buffer), "   couch: %d",
sem_getvalue(&client->current_couch.couch_mutex, &sem_val));


    sprintf(buffer + strlen(buffer), "   workout: %d", client->workout.total_weight);

    return buffer;
}





//////////////////////////////
//
// Client list functions
//
```

```
/**
 * @brief Initalize a client LL on the heap
 * @return (ClientList*) newly allocated LL
 */
ClientList* client_list_init() {
    ClientList* list = (ClientList*)malloc(sizeof(ClientList));

    if(list == NULL) {
        perror("client_list_init malloc()");
        return NULL;
    }

    list->HEAD = NULL;
    list->TAIL = NULL;
    list->len = 0;
    return list;
}


/**
 * @brief Delete a client LL from the heap
 * @param list (ClientList*) list to be deleted
 * @return (int) return code. negative on error
 */
int client_list_del_clients(pid_t exclude, ClientList *list) {
    if(list == NULL) return 0;

    ClientNode *tmp = list->HEAD;

    while(tmp != NULL) {
        if(tmp->node->pid != exclude) client_del(tmp->node);
        tmp = tmp->next;
    }
    return 0;
}


/**
 * @brief Delete clients from LL. Will free the client structs
 * @param exclude (pid_t) Client PID to exclude from deletion
 * @param list (ClientList*) LL from which to delete clients
 * @return (int) return code. negative on error
 */
```

```c
int client_list_del(ClientList *list) {
    if(list == NULL) return 0;

    ClientNode *tmp = list->HEAD;

    while(tmp != NULL) {
        ClientNode *next = tmp->next;
        free(tmp);
        tmp = next;
    }

    free(list);

    return 0;
}


/**
 * @brief Add a client to the LL
 * @param client (Client*) client to be added
 * @param list (ClientList*) list to which client is added
 * @return (int) return code. negative on error
 */
int client_list_add_client(Client *client, ClientList *list) {
    ClientNode *new_node = (ClientNode*)malloc(sizeof(ClientNode));

    if(new_node == NULL) {
        perror("client_list_add_client malloc()");
        return -1;
    }

    // Check if we're already in list
    if(client_list_find_pid(client->pid, list) != NULL) {
        free(new_node);
        return 1;
    }

    new_node->node = client;
    new_node->next = NULL;
    new_node->prev = list->TAIL;

    if(list->HEAD == NULL) {
        list->HEAD = new_node;
        list->TAIL = new_node;
    }
    else {
        list->TAIL->next = new_node;
        list->TAIL = new_node;
    }

    ++list->len;
    return list->len;
}


/**
 * @brief Remove a client from an LL
 * @param client (Client*) client to be removed
 * @param list (ClientList*) list from which to remove client
 * @return (int) return code. negative on error
 */
int client_list_rem_client(Client *client, ClientList *list) {
    if(list == NULL || client == NULL) {
        perror("client_list_rem_client() invalid_argument");
        return -1;
    }

    ClientNode *tmp = client_list_srch(client, list);

    if(tmp == NULL) return list->len;

    if(tmp == list->HEAD && tmp == list->TAIL) {
        free(tmp);
        list->HEAD = NULL;
        list->TAIL = NULL;
    }
    else if(tmp == list->HEAD) {
        ClientNode *new_head = list->HEAD->next;
        free(tmp);
        list->HEAD = new_head;
        new_head->prev = NULL;
    }
    else if(tmp == list->TAIL) {
        ClientNode *new_tail = list->TAIL->prev;
        free(tmp);
        list->TAIL = new_tail;
        new_tail->next = NULL;
    }
    else {
        ClientNode *prevNode = tmp->prev;
        ClientNode *nextNode = tmp->next;
        free(tmp);
        prevNode->next = nextNode;
        nextNode->prev = prevNode;
    }

    --list->len;
    return list->len;
}


/**
 * @brief stringify a client LL
 * @param list (ClientList*) LL to be stringified
 * @param bufer (char[]) buffer to store new string
 * @return (const char*) same as buffer
 */
const char* client_list_to_string(ClientList* list, char buffer[]) {
    if(list == NULL) return NULL;

    if(list->HEAD == NULL) {
        sprintf(buffer, "EMPTY\n");
        return buffer;
    }

    ClientNode *tmp = list->HEAD;
    char buff[1024];
    client_to_string(tmp->node, buff);
    sprintf(buffer, "%s   HEAD", buff);
    tmp = tmp->next;

    while(tmp != NULL) {
        char buff[1024];
        client_to_string(tmp->node, buff);
        sprintf(buffer + strlen(buffer), "\n%s", buff);
        tmp = tmp->next;
    }
    sprintf(buffer + strlen(buffer), "   TAIL\n");
    return buffer;
}
```

```
/**
 * @brief search a client LL for a given client
 * @param client (Client*) needle
 * @param list (ClientList*) haystack
 * @return (ClientNode*) the node containing the requested client. NULL if not found
 */
ClientNode* client_list_srch(Client *client, ClientList *list) {
    if (client == NULL || list == NULL) return NULL;

    ClientNode *tmp = list->HEAD;
    while(tmp != NULL) {
        if(tmp->node == client) return tmp;
        tmp = tmp->next;
    }
    return NULL;
}


/**
 * @brief search a client LL for a given client
 * @param pid (pid_t) needle
 * @param list (ClientList*) haystack
 * @return (Client*) pointer to client struct. NULL if not found
 */
Client* client_list_find_pid(pid_t pid, ClientList *list) {
    if(list == NULL) return NULL;

    ClientNode *tmp = list->HEAD;
    while(tmp != NULL) {
        if(tmp->node->pid == pid) return tmp->node;
        tmp = tmp->next;
    }
    return NULL;
}


/**
 * @brief search a client LL for a paired trainer (client->current_trainer.pid)
 * @param pid (pid_t) needle -> pid of trainer
 * @param list (ClientList*) haystack
 * @return (Client*) poitner to client struct with specified trainer. NULL if not found
 */
Client* client_list_find_trainer(pid_t pid, ClientList *list) {
    if(list == NULL) return NULL;
    ClientNode *tmp = list->HEAD;
    while(tmp != NULL) {
        if(tmp->node->current_trainer.pid == pid) return tmp->node;
        tmp = tmp->next;
    }
    return NULL;
}




//////////////////////////////
//
// Semaphore handling
//


/**
 * @brief Inititialize the client semaphore. Should be called on the parent process
 * @return (int) return code. Negative on failure
 */
int init_client_sem() {
    sem_unlink(CLIENT_ARRIVING_SEM_NAME);
    client_arriving_sem = sem_open(CLIENT_ARRIVING_SEM_NAME, O_CREAT, 0644, 1);
    if(client_arriving_sem == SEM_FAILED) {
        perror("client_sem_init failed to open arriving sem");
        exit(1);
    }

    sem_unlink(CLIENT_WAITING_SEM_NAME);
    client_waiting_sem = sem_open(CLIENT_WAITING_SEM_NAME, O_CREAT, 0644, 1);
    if(client_waiting_sem == SEM_FAILED) {
        perror("client_sem_init failed to open waiting sem");
        exit(1);
    }
}


/**
 * @brief Open the client semaphore. Should be called on the client processes
 * @return (int) negative on failure
 */
int open_client_sem() {
    client_arriving_sem = sem_open(CLIENT_ARRIVING_SEM_NAME, O_CREAT, 0644, 1);
    if(client_arriving_sem == SEM_FAILED) {
        perror("client_sem_init failed to open arriving sem");
        exit(1);
    }

    client_waiting_sem = sem_open(CLIENT_WAITING_SEM_NAME, O_CREAT, 0644, 1);
    if(client_waiting_sem == SEM_FAILED) {
        perror("client_sem_init failed to open waiting sem");
        exit(1);
    }
    return 0;
}


/**
 * @brief close the client semaphore. Should be called after open_client_sem()
 */
void close_client_sem() {
    sem_close(client_arriving_sem);
    sem_close(client_waiting_sem);
    return;
}


/**
 * @brief free the client semaphore. Should be called after init_client_sem() on parent
 */
void destroy_client_sem() {
    sem_unlink(CLIENT_ARRIVING_SEM_NAME);
    sem_unlink(CLIENT_WAITING_SEM_NAME);
    return;
}

//////////////////////////////
//
// Client test function
//

void test_client_list() {
    printf("\r\n");

    Client *client_one = client_init(1, ARRIVING, NULL, NULL, NULL);
    Client *client_two = client_init(2, ARRIVING, NULL, NULL, NULL);
    ClientList *client_list = client_list_init();
    client_list_add_client(client_one, client_list);
    client_list_add_client(client_two, client_list);

    int buffer_size = 1024*client_list->len;
    char buffer[buffer_size];
    client_list_to_string(client_list, buffer);

    printf("INITIAL LIST: \r\n%s\r\n", buffer);

    client_list_rem_client(client_one, client_list);
    client_list_to_string(client_list, buffer);

    printf("CLIENT ONE REMOVED: \r\n%s\r\n", buffer);

    client_list_rem_client(client_two, client_list);
    client_list_to_string(client_list, buffer);

    printf("CLIENT TWO REMOVED:\r\n%s\r\n", buffer);

    client_del(client_one);
    client_del(client_two);
    client_list_del(client_list);
}
// ############################################
//
//   Author  -   Collin Thornton
//   Email   -   collin.thornton@okstate.edu
//   Brief   -   Final Project deadlock detection src
//   Date    -   11-23-20
//
// ############################################

#include <stdbool.h>
#include <stdlib.h>
#include <stdio.h>

#include "deadlock.h"
#include "vector.h"



/**
```

```
 * @brief Check the input file for deadlocked processes.
 * @param deadlocked_array (pid_t[]) array with length of number of processes (matrix rows)
 * @return (int) number of deadlocked processes
 */
int checkForDeadlock(pid_t deadlock_array[]) {
    // BASED ON ALGORITHM DESCRIBED ON PG 339 OF TEXTBOOK


    // Step 1

    Weight *available = getAvailableWeights();
    WeightMatrix *allocation = getWeightAllocation();
    WeightMatrix *request = getWeightRequest();

    Weight work = *available;

    unsigned int num_procs = allocation->num_rows;
    bool finished[num_procs];

    for(int i=0; i<num_procs; ++i) {
        if(vector_zero(allocation->rows[i].weight->num_plates, NUMBER_WEIGHTS)) {
            finished[i] = true;
        }
        else {
            finished[i] = false;
        }
    }


    bool exec_step_three = false;
    do {
        // Step 2

        int row;
        for(int i=0; i<num_procs; ++i) {
            if(!finished[i] && vector_less_than_equal(request->rows[i].weight->num_plates,
work.num_plates, NUMBER_WEIGHTS)) {
                exec_step_three = true;
                row = i;
                break;
            } else  {
                exec_step_three = false;
            }
        }
        if(exec_step_three) {
            // Step 3

            vector_add(work.num_plates, allocation->rows[row].weight->num_plates,
NUMBER_WEIGHTS);
            finished[row] = true;
        }
    } while(exec_step_three == true);


    // Step 4

    int num_deadlocked=0;
    for(int i=0; i<num_procs; ++i) {
        if(!finished[i]) {
            if(deadlock_array != NULL) deadlock_array[num_deadlocked] = allocation->rows[i].pid;
            ++num_deadlocked;
        }
    }

    if(deadlock_array != NULL) {
        for(int i=num_deadlocked; i<num_procs; ++i) {
            deadlock_array[i] = -1;
        }
    }

    weight_del(available);
    weight_matrix_del(allocation);
    weight_matrix_del(request);

    return num_deadlocked;
}



void test_deadlock_detection() {
    // WE WILL ASSUME THE WEIGHT FILE IS PRELOADED AND AS SUCH NOT CLEAR IT

    WeightMatrix *allocation = getWeightAllocation();
    int num_procs = allocation->num_rows;

    weight_matrix_del(allocation);
    pid_t deadlock_array[num_procs];
    int num_deadlocked = checkForDeadlock(deadlock_array);

    printf("\r\n");

    for(int i=0; i<num_deadlocked; ++i) {
        printf("%d\r\n", deadlock_array[i]);
    }

    if(num_deadlocked == 0) printf("NO DEADLOCKED PROCESSES\r\n");
    printf("\r\n");
}


// ###########################################
//
//   Author  -   Collin Thornton / Robert Cook
//   Email   -   robert.cook@okstate.edu
//   Brief   -   Final Project gym resource source
//   Date    -   11-20-20
//
// ###########################################


#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <errno.h>
#include <time.h>

#include "gym.h"


// Relative speed of simulation
#define UNIT_TIME 250       // Milliseconds

// Key for shared memory
#define SHARED_KEY 0x1234
```

```
// Semaphore for accessing shared memory
static sem_t *shared_gym_sem;

// Name of semaphore
static char SHARED_GYM_SEM_NAME[] = "/sem_shmem_gym";

// SharedGym in shared memory
static SharedGym *sharedGym;



//////////////////////////////
//
// Gym functions
//


/**
 * @brief Initialize a gym on the heap
 * @return (Gym*) newly allocated struct
 */
Gym* gym_init() {
   Gym *gym = (Gym*)malloc(sizeof(Gym));
   gym->arrivingList = client_list_init();
   gym->waitingList = client_list_init();
   gym->workoutList = client_list_init();
   gym->trainerList = trainer_list_init();
   gym->unit_time = UNIT_TIME;
   gym->num_trainers = 0;
   gym->boundary_case = true;
   gym->realistic = true;
   gym->detect_deadlock = true;
   gym->fix_deadlock = true;
   gym->trainer_log = false;
   gym->maxCouches = 0;
   gym->deadlock_victim = -1;

   return gym;
}


/**
 * @brief Free a gym from the heap
 * @param gym (Gym*) gym to be freed
 */
void gym_del(Gym *gym) {
   client_list_del_clients(-100, gym->arrivingList);
   client_list_del_clients(-100, gym->waitingList);
   client_list_del_clients(-100, gym->workoutList);
   trainer_list_del_trainers(-100, gym->trainerList);
   client_list_del(gym->arrivingList);
   client_list_del(gym->waitingList);
   client_list_del(gym->workoutList);
   trainer_list_del(gym->trainerList);
   free(gym);
}



//////////////////////////////
//
// Shared memory and semaphore functions
//



/**
 * @brief Initialize the shared memory space and semaphore. Should only be called by parent
process
 * @param maxCouches (int) Max number of clients in waiting room
 * @param numTrainers (int) total number of trainers in simulation
 * @param boundary_case (bool) Flag to solve for Part B
 * @param realistic (bool) Flag to toggle how clients choose weights
 * @param detectDeadlock (bool) Flag to solve for Part C
 * @param fixDeadlock (bool) Flag to solve for Part D
 * @param trainerLog (bool) Flag to solve for Part E
 */
int init_shared_gym(int maxCouches, int numTrainers, bool boundaryCase, bool realistic, bool
detectDeadlock, bool fixDeadlock, bool trainerLog){
   sem_unlink(SHARED_GYM_SEM_NAME);
   shared_gym_sem = sem_open(SHARED_GYM_SEM_NAME, O_CREAT, 0644, 1);
   if(shared_gym_sem == SEM_FAILED) {
      perror("init_shared_gym sem_failed_to_open");
      exit(1);
   }

   int sharedMemoryID;
   int *sharedMemoryAddress;
```

```
sharedMemoryID = shmget(SHARED_KEY, sizeof(SharedGym), IPC_CREAT|0644);

if (sharedMemoryID == -1){
   // FAILSAFE FOR MISHANDLED SHARED MEMORY DEALLOCATION

   system("ipcrm -M 4660");
   sharedMemoryID = shmget(SHARED_KEY, sizeof(SharedGym), IPC_CREAT|0644);

   if(sharedMemoryID == -1) {
      perror("Something went wrong allocating the shared memory space");
      return 1;
   }
}

// TAKE THE SEMAPHORE
sem_wait(shared_gym_sem);

// ATTACH TO SHARED MEMORY
sharedGym = shmat(sharedMemoryID, NULL, 0);

if (sharedGym == (void *) -1){
   sem_post(shared_gym_sem);
   perror("Could not attached to the shared memory\n");
   return 1;
}

// INIT CLIENTS
for(int i=0; i<MAX_CLIENTS; ++i) {
   sharedGym->arrivingList[i].pid      = -1;
   sharedGym->arrivingList[i].state    = -1;
   sharedGym->waitingList[i].pid       = -1;
   sharedGym->waitingList[i].state     = -1;
   sharedGym->workoutList[i].pid       = -1;
   sharedGym->workoutList[i].state     = -1;
}

// INIT TRAINERS
for(int i=0; i<MAX_TRAINERS; ++i) {
   sharedGym->trainerList[i].client_pid = -1;
   sharedGym->trainerList[i].pid        = -1;
   sharedGym->trainerList[i].state      = FREE;
}
```

```
    // INIT CONSTANTS AND FLAGS
    sharedGym->maxCouches = maxCouches;
    sharedGym->num_trainers = numTrainers;
    sharedGym->boundary_case = boundaryCase;
    sharedGym->realistic = realistic;
    sharedGym->fix_deadlock = fixDeadlock;
    sharedGym->detect_deadlock = detectDeadlock;
    sharedGym->trainer_log = trainerLog;
    sharedGym->unit_time = UNIT_TIME;
    sharedGym->deadlock_victim = -1;


    // DETATCH FROM SHARED MEMORY
    if (shmdt(sharedGym) == -1 && errno != EINVAL){
        sem_post(shared_gym_sem);
        perror("Something happened trying to detach from shared memory\n");
        return 1;
    }

    // RELEASE THE SEMAPHORE
    sem_post(shared_gym_sem);

    return 0;
}


/**
 * @brief Open the shared memory and semaphore in the current process.
 */
void open_shared_gym(){

    shared_gym_sem = sem_open(SHARED_GYM_SEM_NAME, O_CREAT, 0644, 1);
    if(shared_gym_sem == SEM_FAILED) {
        perror("init_shared_gym sem_failed_to_open");
        exit(1);
    }

    //First get shared object from memory
    int sharedMemoryID;
    int *sharedMemoryAddress;

    sharedMemoryID = shmget(SHARED_KEY, sizeof(SharedGym), IPC_CREAT|0644);

    if (sharedMemoryID == -1){
        //something went wrong here
        perror("Something went wrong allocating the shared memory space\n");
        return;
    }

    // TAKE THE SEMAPHORE
    sem_wait(shared_gym_sem);

    sharedGym = shmat(sharedMemoryID, NULL, 0);

    if (sharedGym == (void *) -1){
        sem_post(shared_gym_sem);
        perror("Could not attached to the shared memory\n");
        return;
    }

    // RELEASE THE SEMAPHORE
    sem_post(shared_gym_sem);
    return;
}


/**
 * @brief Close the shared memory and semaphore in the current process
 */
void close_shared_gym(){
    sem_wait(shared_gym_sem);

    // IF ERRNO == EINVAL, MEMORY HAS ALREADY BEEN DETACHED -> WE CAN
IGNORE
    if (shmdt(sharedGym) == -1 && errno != EINVAL) {
        perror("Something happened trying to detach from shared memory\n");
        return;
    }
    sem_post(shared_gym_sem);
    sem_close(shared_gym_sem);
}


/**
 * @brief Free the shared memory and semaphore. Should follow a call to init_share_gym() in
the parent process.
 */
void destroy_shared_gym() {
    int sharedMemoryID = shmget(SHARED_KEY, sizeof(SharedGym), IPC_CREAT|0644);

    if (shmctl(sharedMemoryID,IPC_RMID,0) == -1) {
        // It's already been closed by another process. Just ignore.
        perror("Something went wrong with the shmctl function\n");
        return;
    }

    sem_unlink(SHARED_GYM_SEM_NAME);
}




//////////////////////////////
//
// Update shared and local memory
//


/**
 * @brief Copy a locally stored, LL-based gym, to the shared memory space. Will only modify
structs with the current process ID
 * @param gym (Gym*) gym to be copied
 */
void update_shared_gym(Gym *gym) {
    //! CANNOT MODIFY OTHER PROCESS STATES

    // SAVE THE POINTER TO THE CURRENT PROCESS, IF IT EXISTS
    Client *client = client_list_find_pid(getpid(), gym->arrivingList);
    if(client == NULL) client = client_list_find_pid(getpid(), gym->waitingList);
    if(client == NULL) client = client_list_find_pid(getpid(), gym->workoutList);

    Trainer *trainer = NULL;
    if(client == NULL) trainer = trainer_list_find_pid(getpid(), gym->trainerList);


    // SEE IF WE'RE CURRENTLY IN THE LIST
    bool in_list = (client != NULL || trainer != NULL) ? true : false;

    // SEE WHETHER WE'RE A CLIENT OR TRAINER IN THE LIST
    bool is_client = (client == NULL) ? false : true;
    // TAKE THE SEMAPHORE
    sem_wait(shared_gym_sem);

    // Only the parent can set a victim for deadlock rollback
    if(!in_list)  sharedGym->deadlock_victim = gym->deadlock_victim;

    //! CURRENT VICTIM AUTOMATICALLY UNSETS ITSELF WHEN PUSHING TO SHARED
MEM
    if(getpid() == sharedGym->deadlock_victim) sharedGym->deadlock_victim = -1;


    // 1.) Iterate through array
    // 1a.) If entry is in wrong list, set pid and state to -1
    // 1b.) If entry should be in list, set equal to client
    // 1bi.) Add to first empty spot


    // Delete everything with same pid

    for(int i=0; i < MAX_CLIENTS; ++i) {
        pid_t tmp_pid = sharedGym->arrivingList[i].pid;
        if(tmp_pid == getpid()) {
            sharedGym->arrivingList[i].pid = -1;
            sharedGym->arrivingList[i].state = -1;
        }

        tmp_pid = sharedGym->waitingList[i].pid;
        if(tmp_pid == getpid()) {
            sharedGym->waitingList[i].pid = -1;
            sharedGym->waitingList[i].state = -1;
        }

        tmp_pid = sharedGym->workoutList[i].pid;
        if(tmp_pid == getpid()) {
            sharedGym->workoutList[i].pid = -1;
            sharedGym->workoutList[i].state = -1;
        }
    }

    for(int i=0; i < MAX_TRAINERS; ++i) {
        pid_t tmp_pid = sharedGym->trainerList[i].pid;
        if(tmp_pid == getpid()) {
            sharedGym->trainerList[i].pid = -1;
            sharedGym->trainerList[i].client_pid = -1;
            sharedGym->trainerList[i].state = -1;
        }
    }


    // Add us back to the correct list

    if(in_list && is_client) {
        if(client->state == ARRIVING) {
            for(int i=0; i < MAX_CLIENTS; ++i) {
                if(sharedGym->arrivingList[i].pid == -1) {
                    copy_client(&sharedGym->arrivingList[i], client);
                    break;
                }
            }
        }
        else if(client->state == WAITING) {
            for(int i=0; i < MAX_CLIENTS; ++i) {
                if(sharedGym->waitingList[i].pid == -1) {
                    copy_client(&sharedGym->waitingList[i], client);
                    break;
                }
            }
        }
        else if(client->state == TRAINING) {
            for(int i=0; i < MAX_CLIENTS; ++i) {
                if(sharedGym->workoutList[i].pid == -1) {
                    copy_client(&sharedGym->workoutList[i], client);
                    break;
                }
            }
        }
    }
    else if(in_list && !is_client) {
        for(int i=0; i < MAX_TRAINERS; ++i) {
            if(sharedGym->trainerList[i].pid == -1) {
                copy_trainer(&sharedGym->trainerList[i], trainer);
                break;
            }
        }
    }


    sem_post(shared_gym_sem);
```

```
}


/**
 * @brief Copy the shared space, array-based gym, to the local memory space. Will not modify
structs with the current process ID
 */
void update_gym(Gym *gym)  {
    //! CANNOT MODIFY CURRENT PROCESS STATE
    // Easiest to just delete everything except current process and start from scratch


    Client *client = client_list_find_pid(getpid(), gym->arrivingList);
    if(client == NULL) client = client_list_find_pid(getpid(), gym->waitingList);
    if(client == NULL) client = client_list_find_pid(getpid(), gym->workoutList);

    Trainer *trainer = NULL;
    if(client == NULL) trainer = trainer_list_find_pid(getpid(), gym->trainerList);

    bool in_list = (client != NULL || trainer != NULL) ? true : false;
    bool is_client = (client == NULL) ? false : true;

    // Spare current process from deletion
    client_list_del_clients(getpid(), gym->arrivingList);
    client_list_del_clients(getpid(), gym->waitingList);
    client_list_del_clients(getpid(), gym->workoutList);
    trainer_list_del_trainers(getpid(), gym->trainerList);

    client_list_del(gym->arrivingList);
    client_list_del(gym->waitingList);
    client_list_del(gym->workoutList);
    trainer_list_del(gym->trainerList);

    gym->arrivingList = client_list_init();
    gym->waitingList = client_list_init();
    gym->workoutList = client_list_init();
    gym->trainerList = trainer_list_init();


    sem_wait(shared_gym_sem);

    // UPDATE GYM WITH CONSTANTS & FLAGS FROM SHARED SPACE
    gym->deadlock_victim = sharedGym->deadlock_victim;
    gym->maxCouches = sharedGym->maxCouches;
```

```
gym->unit_time = sharedGym->unit_time;
gym->boundary_case = sharedGym->boundary_case;
gym->num_trainers = sharedGym->num_trainers;
gym->realistic = sharedGym->realistic;
gym->fix_deadlock = sharedGym->fix_deadlock;
gym->detect_deadlock = sharedGym->detect_deadlock;
gym->trainer_log = sharedGym->trainer_log;

// Update everything with differnet pid
for(int i=0; i < MAX_CLIENTS; ++i) {
    pid_t tmp_pid = sharedGym->arrivingList[i].pid;
    if(tmp_pid != getpid() && tmp_pid != -1) {
        Client *tmp_client = client_init(0, ARRIVING, NULL, NULL, NULL);
        copy_client(tmp_client, &sharedGym->arrivingList[i]);
        client_list_add_client(tmp_client, gym->arrivingList);
    }

    tmp_pid = sharedGym->waitingList[i].pid;
    if(tmp_pid != getpid() && tmp_pid != -1) {
        Client *tmp_client = client_init(0, ARRIVING, NULL, NULL, NULL);
        copy_client(tmp_client, &sharedGym->waitingList[i]);
        //printf("pid %d received client w/ pid %d\r\n", getpid(), tmp_client->pid);
        client_list_add_client(tmp_client, gym->waitingList);
    }

    tmp_pid = sharedGym->workoutList[i].pid;
    if(tmp_pid != getpid() && tmp_pid != -1) {
        Client *tmp_client = client_init(0, ARRIVING, NULL, NULL, NULL);
        copy_client(tmp_client, &sharedGym->workoutList[i]);
        client_list_add_client(tmp_client, gym->workoutList);
    }
}

for(int i=0; i < MAX_TRAINERS; ++i) {
    pid_t tmp_pid = sharedGym->trainerList[i].pid;
    if(tmp_pid != getpid() && tmp_pid != -1) {
        Trainer *tmp_trainer = trainer_init(-1, -1, FREE);
        copy_trainer(tmp_trainer, &sharedGym->trainerList[i]);
        trainer_list_add_trainer(tmp_trainer, gym->trainerList);
    }
}

// RELEASE SEMAPHORE
sem_post(shared_gym_sem);
```

```
    // update current process
    if(in_list && is_client) {
        switch(client->state) {
            case WAITING:
                //printf("my pid: %d\r\n", getpid());
                client_list_add_client(client, gym->waitingList);
                //client_list_to_string(gym->waitingList, buffer);
                //printf("%s\r\n", buffer);
                break;
            case TRAINING:
                client_list_add_client(client, gym->workoutList);
                break;
            case ARRIVING:
                client_list_add_client(client, gym->arrivingList);
                break;
        }
    }
    else if(in_list && !is_client) {
        trainer_list_add_trainer(trainer, gym->trainerList);
    }
}



//////////////////////////////
//
// Helper functions
//


/**
 * @brief Copy a client struct from src to dest
 * @param dest (Client*) Destination of copy
 * @param src (Client*) Source of copy
 * @return (Client*) same as dest
 */
Client* copy_client(Client *dest, Client *src) {
    if(dest == NULL || src == NULL) {
        perror("copy_client invalid_argument");
        return NULL;
    }
    *dest = *src;
    return dest;
}


/**
 * @brief Copy a trainer from src to dest
 * @param src (Trainer*) destination of copy
 * @param dest (Trainer*) src of copy
 * @return (Trainer*) same as dest
 */
Trainer* copy_trainer(Trainer *dest, Trainer *src) {
    if(dest == NULL || src == NULL) {
        perror("copy_trainer invalide_argument");
        return NULL;
    }

    *dest = *src;
    return dest;
}


/**
 * @brief Delay in milliseconds
 * @param mS (long) delay time
 */
void delay(long mS) {
    struct timespec ts;

    ts.tv_sec = mS / 1000;
    ts.tv_nsec = (mS % 1000) * 1000000L;

    int ret;
    do {
        ret = nanosleep(&ts, &ts);
    } while(ret == -1 && errno == EINTR);
}// ##############################################
//
//   Author  -   Collin Thornton / Robert Cook
//   Email   -   robert.cook@okstate.edu
//   Brief   -   Final Project gym resource source
//   Date    -   11-20-20
//
// ##############################################
```

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

#include "gym_resources.h"



//////////////////////////////
//
// Weight functions
//


/**
 * @brief Allocate Weight struct on heap
 * @param plate_array (int[NUMBER_WEIGHTS]) Array with each indice a number of grip
plates of corresponding PlateIndice. Set to NULL if not yet known
 * @return (Weight*) struct allocated on heap
 */
Weight* weight_init(int plate_array[8]) {
    Weight* weight = malloc(sizeof(Weight));

    if(weight == NULL) {
        perror("gym.c weight_init malloc()");
        return NULL;
    }

    weight->total_weight = 0;
    if(plate_array == NULL) {
        for(int i=0; i<8; ++i) weight->num_plates[i] = 0;
    } else {
        for(int i=0; i<8; ++i) {
            weight->num_plates[i] = plate_array[i];
        }
    }

    weight->total_weight = weight_calc_total_weight(weight);
    return weight;
}
```

```
/**
 * @brief free a Weight struct from the heap
 * @param weight (Weight*) struct to be freed
 * @return (int) return code. negative on failure
 */
int weight_del(Weight *weight) {
    if(weight == NULL) return 1;
    free(weight);
    weight = NULL;

    return 0;
}


/**
 * @brief Calculate the total weight represented by the num_plates array
 * @param weight (Weight*) pointer to a Weight struct
 * @return (float) total weight represented by that struct
 */
float weight_calc_total_weight(Weight *weight) {
    if(weight == NULL) return 0;

    float total_weight = 0;

    for(int i=TWO_HALF; i<=FORTY_FIVE; ++i) {
        switch (i) {
            case TWO_HALF:
                total_weight += weight->num_plates[i]*2.5;
                break;
            case FIVE:
                total_weight += weight->num_plates[i]*5.0;
                break;
            case TEN:
                total_weight += weight->num_plates[i]*10.0;
                break;
            case FIFTEEN:
                total_weight += weight->num_plates[i]*15.0;
                break;
            case TWENTY:
                total_weight += weight->num_plates[i]*20.0;
                break;
            case TWENTY_FIVE:
                total_weight += weight->num_plates[i]*25.0;
                break;
            case THIRTY_FIVE:
                total_weight += weight->num_plates[i]*35.0;
                break;
            case FORTY_FIVE:
                total_weight += weight->num_plates[i]*45.0;
                break;
        }
    }
    return total_weight;
}


/**
 * @brief stringify a weight struct
 * @param weight (Weight*) struct to be stringified
 * @param buffer (char[]) buffer to store string
 * @return (const char*) same as buffer
 */
const char* weight_to_string(Weight *weight, char buffer[]) {
    buffer[0] = '\0';

    for(int j=TWO_HALF; j<=FORTY_FIVE; ++j) {
        sprintf(buffer+strlen(buffer), "%d,", weight->num_plates[j]);
    }
    buffer[strlen(buffer)-1] = '\0';
    return buffer;
}// ############################################
//
//   Author  -   Collin Thornton
//   Email   -   collin.thornton@okstate.edu
//   Brief   -   Final Project part1 driver
//   Date    -   11-20-20
//
// ############################################


#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>

#include "start_sim.h"
```

```
int main(int argc, char **argv) {
    printf("\r\nCS4323 FINAL PROJECT GROUP D\r\n");
    printf("GYM SIMULATOR\r\n\r\n");
    printf("1st Driver File -> Part a\r\n\r\n");
    printf("Collin Thornton\r\nRobert Cook\r\nTyler Krebs\r\n\r\n");
    printf("Usage: ./part1 <NUM_TRAINERS> <NUM_COUCHES>\n\n");


    if(argc != 3) {
        exit(-1);
    }


    int num_trainers = 0;
    int num_couches = 0;

    if((num_trainers = strtol(argv[1], NULL, 10)) == 0 && errno != 0) {
        exit(-1);
    }

    if((num_couches = strtol(argv[2], NULL, 10)) == 0 && errno != 0) {
        exit(-1);
    }


    start_sim(num_trainers, num_couches, false, false, true, false, false);
}// ###############################################
//
//   Author  -   Collin Thornton
//   Email   -   collin.thornton@okstate.edu
//   Brief   -   Final Project part2 driver
//   Date    -   11-20-20
//
// ###############################################


#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>

#include "start_sim.h"
```

```
int main(int argc, char **argv) {
    printf("\r\nCS4323 FINAL PROJECT GROUP D\r\n");
    printf("GYM SIMULATOR\r\n\r\n");
    printf("2nd Driver File -> Parts b & c\r\n\r\n");
    printf("Collin Thornton\r\nRobert Cook\r\nTyler Krebs\r\n\r\n");
    printf("Usage: ./part2 <NUM_TRAINERS> <NUM_COUCHES>\n\n");


    if(argc != 3) {
        exit(-1);
    }


    int num_trainers = 0;
    int num_couches = 0;

    if((num_trainers = strtol(argv[1], NULL, 10)) == 0 && errno != 0) {
        exit(-1);
    }

    if((num_couches = strtol(argv[2], NULL, 10)) == 0 && errno != 0) {
        exit(-1);
    }


    start_sim(num_trainers, num_couches, true, false, true, false, false);
}// ##########################################
//
//   Author  -   Collin Thornton
//   Email   -   collin.thornton@okstate.edu
//   Brief   -   Final Project part3 driver
//   Date    -   11-30-20
//
// ##########################################


#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>

#include "start_sim.h"
```

```
int main(int argc, char **argv) {
    printf("\r\nCS4323 FINAL PROJECT GROUP D\r\n");
    printf("GYM SIMULATOR\r\n\r\n");
    printf("3rd Driver File -> Parts b, d, & e\r\n\r\n");
    printf("Collin Thornton\r\nRobert Cook\r\nTyler Krebs\r\n\r\n");
    printf("Usage: ./part3 <NUM_TRAINERS> <NUM_COUCHES>\n\n");

    if(argc != 3) {
        exit(-1);
    }


    int num_trainers = 0;
    int num_couches = 0;

    if((num_trainers = strtol(argv[1], NULL, 10)) == 0 && errno != 0) {
        exit(-1);
    }

    if((num_couches = strtol(argv[2], NULL, 10)) == 0 && errno != 0) {
        exit(-1);
    }


    start_sim(num_trainers, num_couches, true, true, true, true, true);
}
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <errno.h>
#include <time.h>

#include "recordbook.h"
```

```
#define SHARED_KEY 0x2345

#define TEST_RECORDBOOK


/*Semaphore for the ensuring mutual exclusion*/
typedef struct SharedMutex {
   pthread_mutex_t mutex;
} SharedMutex;

//defines the name for the record book text file
#define DEFAULT_RECORD_FILE "./data/recordbook.txt"

/*Pointer to store name of the file.*/
char *RecordFileName = NULL;


//creates the shared mutex
SharedMutex *shared_mutex;


static bool TEST = false;

//This function takes the struct 'emp' and adds the name, id, and weight to the record book text
file
void addToRecordBook(struct emp *empValue)
{
   FILE *fp = NULL;

   //starts in locked mode
   pthread_mutex_lock(&shared_mutex->mutex);

   if(TEST) {
      printf("%d ENTERED LOCKED AREA WAITING 2 SECONDS\r\n", getpid());
      sleep(2);
   }

   //opens the file in append mode
   fp = fopen(RecordFileName, "a");
   if (fp == NULL)
   {
      perror("Failed to open the file. \n");
   }
```

```
    char buffer[1024];

    //print all values to the record book text file in the correct format
    sprintf(buffer, "Client Name: %s\t", empValue->name);
    sprintf(buffer + strlen(buffer), "ID: %d\t", empValue->id);
    sprintf(buffer + strlen(buffer), "Total Weight: %d\r\n", empValue->weight);

    fputs(buffer, fp);

    fclose(fp);

    //unlocks the mutex lock

    if(TEST) {
        printf("%d LEFT LOCKED AREA\r\n", getpid());
    }

    pthread_mutex_unlock(&shared_mutex->mutex);
}

//function for printing out the information from the record book text file
void displayRecordBook()
{
    FILE *fp = NULL;
    struct emp empVal;

    //starts in locked mode
    pthread_mutex_lock(&shared_mutex->mutex);

    //opens the file in read mode
    fp = fopen(RecordFileName, "r");
    if (fp == NULL)
    {
        perror("Failed to open the file. \n");
    }

    //prints the information line by line to the console
    while (fread(&empVal, sizeof(empVal), 1, fp) ==  1)
    {
        printf("Name = %s \t Id = %d \t Weight = %d \n", empVal.name, empVal.id,
empVal.weight);
        memset(&empVal, 0x00, sizeof(empVal));
    }
    fclose(fp);

    //unlocks the mutex lock
    pthread_mutex_unlock(&shared_mutex->mutex);
}

// Opens the file and truncates it, implying all the records are cleared
void clearRecordBook()
{
    FILE *fp = NULL;

    //opens in locked mode
    pthread_mutex_lock(&shared_mutex->mutex);

    fp = fopen(RecordFileName, "w");
    if (fp == NULL)
    {
        perror("Failed to open the file. \n");
    }
    fclose(fp);

    //unlocks the mutex lock
    pthread_mutex_unlock(&shared_mutex->mutex);
}




//function for initializing the record book. The function MUST be called before invoking any of the
record logging operation.
void initRecordBook()
{
    //creating a shared mutex attribute
    pthread_mutexattr_t psharedm;

    //initializing the shared mutex attribute
    pthread_mutexattr_init(&psharedm);
    pthread_mutexattr_setpshared(&psharedm, PTHREAD_PROCESS_SHARED);

    //creating the shared memory space for the shared mutex
    int sharedMemoryID = shmget(SHARED_KEY, sizeof(SharedMutex), IPC_CREAT|0644);

    //if no shared memory space then creates it, after creation and theres still non exit
    if (sharedMemoryID == -1){
        // FAILSAFE FOR MISHANDLED SHARED MEMORY DEALLOCATION
        system("ipcrm -M 9029");
        sharedMemoryID = shmget(SHARED_KEY, sizeof(SharedMutex), IPC_CREAT|0644);

        if(sharedMemoryID == -1) {
            perror("Something went wrong allocating the shared memory space");
            exit(-1);
        }
    }


    //attaching the shared mutex to the shared memory
    shared_mutex = shmat(sharedMemoryID, NULL, 0);

     if (shared_mutex == (void *) -1){
        perror("Could not attached to the shared memory\n");
        return;
    }

    //initializing the shared mutex
    pthread_mutex_init(&shared_mutex->mutex, &psharedm);

    if (shmdt(shared_mutex) == -1 && errno != EINVAL){
        perror("Something happened trying to detach from shared memory\n");
        return;
    }

}

//function to opens the record book. needs to be opened for each trainer to add to it
void openRecordBook() {
    //First get shared object from memory
    int sharedMemoryID;
    int *sharedMemoryAddress;

    //allocates the memory space for the shared mutex
    sharedMemoryID = shmget(SHARED_KEY, sizeof(SharedMutex), IPC_CREAT|0644);

    if (sharedMemoryID == -1){
        //something went wrong here
        perror("Something went wrong allocating the shared memory space\n");
        return;
    }
    //attatching the shared mutex to the shared memory
    shared_mutex = shmat(sharedMemoryID, NULL, 0);

    if (shared_mutex == (void *) -1){
        perror("Could not attached to the shared memory\n");
        return;
    }

    RecordFileName = DEFAULT_RECORD_FILE;

    return;
}

//function to close the record book and detach the shared memory
void closeRecordBook() {
    if (shmdt(shared_mutex) == -1 && errno != EINVAL) {
        perror("Something happened trying to detach from shared memory\n");
        return;
    }
}

//function destroys the record book
void destroyRecordBook() {
    int sharedMemoryID = shmget(SHARED_KEY, sizeof(SharedMutex), IPC_CREAT|0644);

    if (shmctl(sharedMemoryID,IPC_RMID,0) == -1){
        // It's already been closed by another process. Just ignore.
        perror("Something went wrong with the shmctl function\n");
        return;
    }
}



void test_recordbook() {

    TEST = true;

    initRecordBook();
    closeRecordBook();

    for(int i=0; i<2; ++i) {
        pid_t pid = fork();
        if(pid < 0) {
            perror("fork");
            exit(1);
        }
        else if(pid == 0) {
            // CHILD PROCESS
            openRecordBook();

            srand(getpid());
            Emp entry;

            sprintf(entry.name, "Process %d", getpid());
            entry.id = getpid();
            entry.weight = rand() % 500;

            addToRecordBook(&entry);

            closeRecordBook();

            exit(0);
        }
        else {
            // PARENT PROCESS -> DON'T DO MUCH
            printf("Spawned process %d\r\n", pid);
        }
    }

    for(int i=0; i<2; ++i) wait(NULL);

    TEST = false;

}
// ##############################################
//
//   Author  -   Collin Thornton
//   Email   -   collin.thornton@okstate.edu
//   Brief   -   Final Project gym resource source
//   Date    -   11-20-20
//
// ##############################################

#include "resource_manager.h"
#include "vector.h"
```

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


#include <sys/stat.h>

#define MAX_LINE_SIZE 1024

static sem_t *resource_manager_sem;


// #define SUB_DELETE_ROW // UNCOMMENT TO DELETE MATRIX ROW WHEN
SUBTRACTION = 0

// Setup global variables for filenames
static const char* FILENAME = "data/weight_allocation.txt";
static const char* TMP_FILENAME = "data/weight_allocation.tmp";

// Setup global variable for semaphore
static char RESOURCE_MANAGER_SEM_NAME[] = "/sem_resource_manager";






/**
 * @brief Initialize the semaphore. Should only be ran on the parent process
 * @return (int) return code. negative on error
 */
int init_resource_manager() {
   sem_unlink(RESOURCE_MANAGER_SEM_NAME);
   resource_manager_sem = sem_open(RESOURCE_MANAGER_SEM_NAME, O_CREAT,
0644, 1);
   if(resource_manager_sem == SEM_FAILED) {
      perror("init_resource_manager sem failed to open");
      exit(1);
   }

   mkdir("./data", 0644);
```

```
    const char *INITIAL_WRITE = "\n"
        "# WEIGHT ALLOCATION MATRICES FOR GYM\n"
        "# SPECIAL CHARACTERS:\n"
        "#   #  -> Comment. Program ignores line\n"
        "#   ,  -> Delimeter\n"
        "#  --- -> Section divider\n"
        "# SECTION 1 -> AVAILABLE\n"
        "# SECTION 2 -> ALLOCATION\n"
        "# SECTION 3 -> REQUEST\n"
        "# pid,2.5,5,10,15,20,25,35,45\n"
        "\n"
        "# AVAILABLE\n"
        "# 2.5,5,10,15,20,25,35,45\n"
        "\n"
        "12,12,12,12,12,12,12,12\n"
        "---\n"
        "---\n"
        "---\n"
        "\n";

    // TAKE THE SEMAPHORE
    sem_wait(resource_manager_sem);


    // initalize the file
    FILE *file = fopen(FILENAME, "w");
    fputs(INITIAL_WRITE, file);
    fclose(file);

    sem_post(resource_manager_sem);

    return 0;
}




/**
 * @brief Open the semaphore on the current process.
 * @return (int) return code. Negative on error
 */
int open_resource_manager() {
    resource_manager_sem = sem_open(RESOURCE_MANAGER_SEM_NAME, O_CREAT,
0644, 1);
    if(resource_manager_sem == SEM_FAILED) {
        perror("init_resource_manager sem failed to open");
        exit(1);
    }
    return 0;
}


/**
 * @brief Close the semaphore on the current process
 */
void close_resource_manager() {
    sem_close(resource_manager_sem);
    return;
}


/**
 * @brief Desotry the semaphore. Should only be ran on the parent process.
 */
void destroy_resource_manager() {
    sem_unlink(RESOURCE_MANAGER_SEM_NAME);
    return;
}


/**
 * @brief Private function. Initailize a WeightMatrix on heap. All values NULL or 0
 * @return (WeightMatrix*) pointer to new matrix
 */
static WeightMatrix* weight_matrix_init() {
    WeightMatrix* matrix = malloc(sizeof(WeightMatrix));

    if(matrix == NULL) {
        perror("weight_matrix_init malloc()");
        return NULL;
    }

    matrix->rows = NULL;
    matrix->num_rows = 0;
    return matrix;
}


/**
```

```
 * @brief Free a WeightMatrix*, and all internal WeightRows* and Weights*
 * @param matrix (WeightMatrix*) matrix to be deleted
 * @return (int) 0 on success
 */
int weight_matrix_del(WeightMatrix* matrix) {
    if(matrix != NULL) {
        if(matrix->rows != NULL) {
            for(int i=0; i<matrix->num_rows; ++i) {
                if(matrix->rows[i].weight != NULL) free(matrix->rows[i].weight);
            }
            free(matrix->rows);
        }
        free(matrix);
    }

    return 0;
}


/**
 * @brief Add a weight request to a weight matrix. Will add new row if pid is not found
 * @param pid (pid_t) pid of requesting process
 * @param weight (Weight*) weight to be added
 * @param matrix (WeightMatrix*) matrix to store summation
 * @return (int) number of rows in matrix. Negative on error
 */
static int weight_matrix_add_req(pid_t pid, Weight* weight, WeightMatrix *matrix) {
    if(matrix == NULL) {
        perror("weight_matrix_add_req invalid_argument matrix");
        return -1;
    }
    if(weight == NULL) {
        perror("weight_matrix_add_req invalid_argument weight");
        return -1;
    }

    WeightMatrixRow* row = weight_matrix_search(pid, matrix, NULL);

    if(row == NULL) {
        // ADD A ROW IF PID NOT FOUND
        if(matrix->num_rows == 0) matrix->rows =
(WeightMatrixRow*)malloc(sizeof(WeightMatrixRow));
        else matrix->rows = (WeightMatrixRow*)realloc(matrix->rows,
(matrix->num_rows+1)*sizeof(WeightMatrixRow));
```

```
        if(matrix->rows == NULL) {
            perror("resourceManager allocate row");
            weight_del(weight);
            return -1;
        }

        ++matrix->num_rows;

        matrix->rows[matrix->num_rows-1].pid = pid;
        matrix->rows[matrix->num_rows-1].weight = weight;

        return matrix->num_rows;
    }

    // ELSE ADD weight TO CURRENT ROW

    for(int i=TWO_HALF; i<=FORTY_FIVE; ++i) {
        row->weight->num_plates[i] += weight->num_plates[i];

    }
    weight_del(weight);

    return matrix->num_rows;
}


/**
 * @brief Subtract a weight request from a weight matrix. Will delete a row if result is 0 vector
 * @param pid (pid_t) pid of process
 * @param weight (Weight*) weight to be subtracted
 * @param matrix (WeightMatrix*) matrix to store difference
 * @return (int) nuber of rows in matrix. Negative on error
 */
static int weight_matrix_sub_req(pid_t pid, Weight *weight, WeightMatrix *matrix) {
    int row_number;
    WeightMatrixRow *row = weight_matrix_search(pid, matrix, &row_number);

    if(row == NULL) {
        weight_del(weight);
        return -1;
    }

    vector_subtract(row->weight->num_plates, weight->num_plates, NUMBER_WEIGHTS);
    if(vector_negative(row->weight->num_plates, NUMBER_WEIGHTS)) {
        perror("weight_matrix_sub_req invalid request");
        weight_del(weight);
        return -1;
    }
    row->weight->total_weight = weight_calc_total_weight(row->weight);

    if(row->weight->total_weight >= 0) {
        // FINISHED
        weight_del(weight);
        return matrix->num_rows;
    }

    // ELSE WE NEED TO DELETE ROW AND RESTRUCTURE MATRIX
    #ifdef SUB_DELETE_ROW
    weight_del(weight);
    weight_del(row->weight);

    for(int i=row_number+1; i<matrix->num_rows; ++i) {
        matrix->rows[i-1].pid = matrix->rows[i].pid;
        matrix->rows[i-1].weight = matrix->rows[i].weight;
    }

    --matrix->num_rows;

    if(matrix->num_rows > 0) {
        matrix->rows = (WeightMatrixRow*)realloc(matrix->rows,
(matrix->num_rows)*sizeof(WeightMatrixRow));
    } else {
        free(matrix->rows);
        matrix->rows = 0;
    }
    #endif

    return matrix->num_rows;
}


/**
 * @brief Search a WeightMatrix for a pid
 * @param pid (pid_t) pid for which to search
 * @param matrix (WeightMatrix*) matrix to be searched
 * @param row_number (int*) storage for the row number. negative if row not found. can be set
to NULL if not neededd
```

```
 * @return (WeightMatrixRow*) pointer to the row. NULL if not found
 */
WeightMatrixRow* weight_matrix_search(pid_t pid, WeightMatrix *matrix, int *row_number) {
   if(matrix == NULL || matrix->rows == NULL) return NULL;

   for(int i=0; i<matrix->num_rows; ++i) {
      if(row_number != NULL) *row_number = i;
      if(matrix->rows[i].pid == pid) return &matrix->rows[i];
   }

   if(row_number != NULL) *row_number = -1;
   return NULL;
}


/**
 * @brief Return a string representative of matrix
 * @param matrix (WeightMatrix*) matrix to be returned as string
 * @param buffer (char[]) buffer to store string output
 * @return (const char*) pointer to string. same as buffer
 */
const char* weight_matrix_to_string(WeightMatrix *matrix, char buffer[]) {
   buffer[0] = '\0';

   for(int i=0; i<matrix->num_rows; ++i) {
      char line[MAX_LINE_SIZE];
      sprintf(buffer+strlen(buffer), "%d,", matrix->rows[i].pid);

      weight_to_string(matrix->rows[i].weight, line);
      sprintf(buffer+strlen(buffer), "%s\n", line);
   }
   return buffer;
}


/**
 * @brief return the gym's total resources. must be deleted with weight_del()
 * @return (Weight*) Vector of total weights
 */
Weight* getGymResources() {
   sem_wait(resource_manager_sem);
   Weight *database = __getGymResources();
   sem_post(resource_manager_sem);
   return database;
}

/**
 * @brief return currently available weights
 * @return (Weight*) vector of current weight
 */
Weight* getAvailableWeights() {
   sem_wait(resource_manager_sem);
   Weight *total = __getAvailableWeights();
   sem_post(resource_manager_sem);
   return total;
}


/**
 * @brief Private function. Not locked with semaphore. See getGymResources()
 */
static Weight* __getGymResources() {
   Weight *database = getWeightFromFile(0);
   return database;
}

/**
 * @brief Privat function. Not locked with semaphore. See getAvalaibleWeights()
 */
static Weight* __getAvailableWeights() {
   Weight *total = __getGymResources();
   WeightMatrix *allocated = __getWeightAllocation();

   Weight *total_used = weight_init(NULL);
   for(int i=0; i<allocated->num_rows; ++i) {
      vector_add(total_used->num_plates, allocated->rows[i].weight->num_plates,
NUMBER_WEIGHTS);
   }

   vector_subtract(total->num_plates, total_used->num_plates, NUMBER_WEIGHTS);

   weight_matrix_del(allocated);
   weight_del(total_used);
   return total;
}


/**
```

```
 * @brief Private function. Not locked with semaphore. get a weight from the input file
 * @param section (unsigned int) section number from which to read
 * @return (Weight*) allocation on heap
 */
static Weight* getWeightFromFile(unsigned int section) {
    FILE *file = fopen(FILENAME, "r");
    if(file == NULL) {
        perror("getGymResources fopen()");
        return NULL;
    }

    Weight* database = weight_init(NULL);

    char line[MAX_LINE_SIZE] = "\0";

    // THROW OUT FIRST LINE
    fgets(line, MAX_LINE_SIZE-1, file);

    for(int i=0; i<section; ++i) {
        while(fgets(line, MAX_LINE_SIZE-1, file) != NULL && !feof(file)) {
            if(line[0] == '#' || line[0] == '\n' || line[0] == '\r') continue;
            if(strstr(line, "---") != NULL) break;
        }
    }


    while(fgets(line, MAX_LINE_SIZE-1, file) != NULL && !feof(file)) {
        removeWhiteSpace(line);

        if(line[0] == '#' || line[0] == '\n' || line[0] == '\r') continue;
        if(strstr(line, "---") != NULL) break;

        #ifdef VERBOSE
        printf("%s", line);
        #endif // VERBOSE

        char *number_plates = strtok(line, ",");
        if(number_plates == NULL) {
            perror("getGymResources inital string token");
            weight_del;
            fclose(file);
            return NULL;
        }
        for(int i=TWO_HALF; i<=FORTY_FIVE; ++i) {
            if(number_plates == NULL && i != FORTY_FIVE) {
                perror("getGymResources field_processing");
                printf("%d\n", i);
                weight_del(database);
                fclose(file);
                return NULL;
            }

            char *end;
            int num_plates = 0;
            float total_weight;

            if((num_plates = strtol(number_plates, &end, 10)) == 0 && errno != 0) {
                perror("getGymResources num_plates");
                weight_del(database);
                fclose(file);
                return NULL;
            }

            database->num_plates[i] = num_plates;

            number_plates = strtok(NULL, ",");
        }

    }

    fclose(file);

    database->total_weight = weight_calc_total_weight(database);
    return database;
}


/**
 * @brief return the currently allocated weights. deleted with weight_matrix_del
 * @return (WeightMatrix*) matrix of allocations allocated on heap
 */
WeightMatrix* getWeightAllocation() {
    sem_wait(resource_manager_sem);
    WeightMatrix* ret = __getWeightAllocation();
    sem_post(resource_manager_sem);

   return ret;
```

```
}


/**
 * @brief removes the request, allocates weights, and adjusts the currently available weights
 * @param pid (pid_t) process to grant
 * @return (int) return code. negative on error
 */
WeightMatrix* getWeightRequest() {
    sem_wait(resource_manager_sem);
    WeightMatrix* ret = __getWeightRequest();
    sem_post(resource_manager_sem);

    return ret;
}


/**
 * @brief Private function. Not locked with semaphore. See getWeightAllocation()
 */
static WeightMatrix* __getWeightAllocation() {
   return getWeightMatrixFromFile(1);
}


/**
 * @brief Private function. Not locked with semaphore. See getWeightRequest()
 */
static WeightMatrix* __getWeightRequest() {
    return getWeightMatrixFromFile(2);;
}


/**
 * @brief read a weight matrix from the input file
 * @param section (unsigned int) section number from which to read
 * @return (WeightMatrix*) pointer to weight matrix on heap
 */
static WeightMatrix* getWeightMatrixFromFile(unsigned int section) {
    if(section > 2 || section == 0) {
        perror("getWeightMatrixFromFile section");
        return NULL;
    }
```

```
FILE *file = fopen(FILENAME, "r");
if(file == NULL) {
    perror("getWeightMatrixFromFile fopen()");
    return NULL;
}

WeightMatrix* database = weight_matrix_init();

char line[MAX_LINE_SIZE];

for(int i=0; i<section; ++i) {
    while(fgets(line, MAX_LINE_SIZE-1, file) != NULL && !feof(file)) {
        if(line[0] == '#' || line[0] == '\n' || line[0] == '\r') continue;
        if(strstr(line, "---") != NULL) break;
    }
}

while(fgets(line, MAX_LINE_SIZE-1, file) != NULL && !feof(file)) {
    removeWhiteSpace(line);

    if(strstr(line, "---") != NULL) break;

    #ifdef VERBOSE
    printf("%s", line);
    #endif // VERBOSE

    Weight* weight = weight_init(NULL);
    pid_t pid;


    char *tmp = strtok(line, ",");
    char *end;

    if(tmp == NULL) {
        perror("getWeightMatrixFromFile pid token");
        weight_del(weight);
        weight_matrix_del(database);
        fclose(file);
        return NULL;
    }
    if((pid = strtol(tmp, &end, 10)) == 0 && errno != 0) {
        perror("getWeightMatrixFromFile pid");
        weight_del(weight);
        weight_matrix_del(database);
```

```
            fclose(file);
            return NULL;
        }


        for(int i=TWO_HALF; i<=FORTY_FIVE; ++i) {
            tmp = strtok(NULL, ",");

            if(tmp == NULL) {
                perror("getWeightMatrixFromFile field_processing");
                weight_del(weight);
                weight_matrix_del(database);
                fclose(file);
                return NULL;
            }

            int num_plates = 0;
            float total_weight;

            errno = 0;
            if((num_plates = strtol(tmp, &end, 10)) == 0 && errno != 0) {
                perror("getWeightMatrixFromFile num_plates");
                weight_del(weight);
                weight_matrix_del(database);
                fclose(file);
                return NULL;
            }

            weight->num_plates[i] = num_plates;
        }
        weight->total_weight = weight_calc_total_weight(weight);
        weight_matrix_add_req(pid, weight, database);

    }

    fclose(file);
    return database;
}


/**
 * @brief removes the request, allocates weights, and adjusts the currently available weights
 * @param pid (pid_t) process to grant
 * @return (int) return code. negative on error
 */
int grantWeightRequest(pid_t pid) {
    sem_wait(resource_manager_sem);
    WeightMatrix *tot_request = __getWeightRequest();
    WeightMatrix *tot_allocation = __getWeightAllocation();
    Weight *currently_availble = __getAvailableWeights();

    Weight *tmp = weight_init(NULL);
    weight_matrix_add_req(pid, tmp, tot_allocation);

    int req_row, alloc_row;
    WeightMatrixRow *request = weight_matrix_search(pid, tot_request, &req_row);
    WeightMatrixRow *allocation = weight_matrix_search(pid, tot_allocation, &alloc_row);

    if(request == NULL) {
        perror("grantWeightRequest() pid doesn't exit");
        weight_matrix_del(tot_allocation);
        weight_matrix_del(tot_request);
        weight_del(currently_availble);
        sem_post(resource_manager_sem);
        return -1;
    }


    vector_add(allocation->weight->num_plates, request->weight->num_plates,
NUMBER_WEIGHTS);

    if(vector_less_than_equal(request->weight->num_plates, currently_availble->num_plates,
NUMBER_WEIGHTS) == false) {
        //perror("grantWeightRequest() allocation out of bounds");
        weight_matrix_del(tot_allocation);
        weight_matrix_del(tot_request);
        weight_del(currently_availble);
        sem_post(resource_manager_sem);
        return -2;
    }
    vector_subtract(request->weight->num_plates, request->weight->num_plates,
NUMBER_WEIGHTS);

    writeWeightMatrixToFile(tot_allocation, 1);
    writeWeightMatrixToFile(tot_request, 2);

    weight_matrix_del(tot_allocation);
    weight_matrix_del(tot_request);
    weight_del(currently_availble);

    sem_post(resource_manager_sem);
    return 0;
}

/**
 * @brief write a new allocation to the file
 * @param pid (pid_t) pid of process
 * @param weight (Weight*) new allocation
 * @return (int) negative on failure
 */
int writeWeightAllocation(pid_t pid, Weight *weight) {
    sem_wait(resource_manager_sem);
    WeightMatrix *alloc_matrix = __getWeightAllocation();
    WeightMatrix *req_matrix = __getWeightRequest();
    Weight *tmp_weight = weight_init(NULL);

    weight_matrix_add_req(pid, weight, alloc_matrix);
    weight_matrix_add_req(pid, tmp_weight, req_matrix);

    int ret = writeWeightMatrixToFile(alloc_matrix, 1);
    writeWeightMatrixToFile(req_matrix, 2);
    weight_matrix_del(alloc_matrix);
    weight_matrix_del(req_matrix);
    sem_post(resource_manager_sem);
    return ret;
}


/**
 * @brief write a new request to the file
 * @param pid (pid_t) pid of process
 * @param weight (Weight*) new request
 * @return (int) negative on failure
 */
int writeWeightRequest(pid_t pid, Weight *weight) {
    sem_wait(resource_manager_sem);
    WeightMatrix *req_matrix = __getWeightRequest();
    WeightMatrix *alloc_matrix = __getWeightAllocation();
    Weight *tmp_weight = weight_init(NULL);

    weight_matrix_add_req(pid, weight, req_matrix);
    weight_matrix_add_req(pid, tmp_weight, alloc_matrix);
    int ret = writeWeightMatrixToFile(req_matrix, 2);
    writeWeightMatrixToFile(alloc_matrix, 1);

    weight_matrix_del(req_matrix);
    weight_matrix_del(alloc_matrix);
    sem_post(resource_manager_sem);
    return ret;
}


/**
 * @brief Private function. Not locked with semaphore. write a weight matrix to the input file. will
delete the matrix
 * @param matrix (WeightMatrix*) matrix to be written
 * @param section (int) section number at which to write
 * @return (int) negative on failure
 */
static int writeWeightMatrixToFile(WeightMatrix *matrix, int section) {
    if(section > 2 || section == 0) {
        perror("writeWeightToFile section");
        return 1;
    }

    FILE *file = fopen(FILENAME, "r");
    FILE *new_file = fopen(TMP_FILENAME, "w");

    short file_flag = 0;
    if(file == NULL) {
        perror("writeWeightMatrixToFile file");
        file_flag = 1;
    }
    if(new_file == NULL) {
        perror("writeWeightMatrixToFile new_file");
        file_flag += 2;
    }

    switch(file_flag) {
        case 0:
            break;
        case 1:
            fclose(new_file);
            return 1;
        case 2:
            fclose(file);
            return 1;
        case 3:
            return 1;
    }

    char line[MAX_LINE_SIZE];

    for(int i=0; i<section; ++i) {
        while(fgets(line, MAX_LINE_SIZE-1, file) != NULL && !feof(file)) {
            fputs(line, new_file);
            if(strcmp(line, "---\n") == 0) break;
        }
    }

    int buff_size = matrix->num_rows*MAX_LINE_SIZE;
    char buffer[buff_size];

    weight_matrix_to_string(matrix, buffer);
    fprintf(new_file, "%s", buffer);
    fputs("---\n", new_file);

    while(fgets(line, MAX_LINE_SIZE-1, file) != NULL && !feof(file)) {
        if(strcmp(line, "---\n") == 0) break;
    }

    while(fgets(line, MAX_LINE_SIZE-1, file) != NULL && !feof(file)) {
        fputs(line, new_file);
    }


    fclose(file);
    fclose(new_file);

    remove(FILENAME);
    rename(TMP_FILENAME, FILENAME);

    return 0;
}


/**
 * @brief removes the allocation and adjusts currently available weights
 * @param pid (pid_t) process id to adjust
 * @param weight (Weight*) amount to change
 * @return (int) return code. negative on error
 */
int releaseWeightAllocation(pid_t pid, Weight *weight) {
    sem_wait(resource_manager_sem);
    WeightMatrix *matrix = __getWeightAllocation();

    WeightMatrixRow *row = weight_matrix_search(pid, matrix, NULL);
    if (row == NULL) {
        weight_matrix_del(matrix);
        sem_post(resource_manager_sem);
       return -1;
    }

    vector_subtract(row->weight->num_plates, weight->num_plates, NUMBER_WEIGHTS);
    if(vector_negative(row->weight->num_plates, NUMBER_WEIGHTS)) {
        perror("removeWeightAllocation() invalid argument");
        weight_matrix_del(matrix);
        sem_post(resource_manager_sem);
        return -1;
    }

    int ret = writeWeightMatrixToFile(matrix, 1);
    weight_matrix_del(matrix);
    sem_post(resource_manager_sem);
    return ret;
}


/**
 * @brief remove a request from the file. will throw error if result is negative
 * @param pid (pid_t) pid of process
 * @param weight (Weight*) weight to be subtracted
 * @return (int) negative on failure
 */
int removeWeightRequest(pid_t pid, Weight *weight) {
    sem_wait(resource_manager_sem);
    WeightMatrix *matrix = __getWeightRequest();
    weight_matrix_sub_req(pid, weight, matrix);
    int ret = writeWeightMatrixToFile(matrix, 2);
    weight_matrix_del(matrix);
    sem_post(resource_manager_sem);
    return ret;
}
```

```
/**
 * @brief clear allocation and request matrices from file
 * @return (int) negative on failure
 */
int clearWeightFile() {
    sem_wait(resource_manager_sem);
    WeightMatrix *matrix = weight_matrix_init();
    int ret1 = writeWeightMatrixToFile(matrix, 1);
    int ret2 = writeWeightMatrixToFile(matrix, 2);
    weight_matrix_del(matrix);

    remove(TMP_FILENAME);

    sem_post(resource_manager_sem);

    if(ret2 < 0) return ret2;
    return ret1;
}


/**
 * @brief Private function. remove all whitespace from a string
 * @param str (char*) input string. will be changed
 * @return (const char*) output string. same as str
 */
static char* removeWhiteSpace(char* str) {
    const char* d = str;
    do {
        while(*d == ' ' || *d == '\t') {
            ++d;
        }
    } while(*str++ = *d++);
}



void test_resource_manager(void) {
    char line[1024];

    ////////////////////////
    //
    // TEST CLEAR FUNCTION
```

```
//

clearWeightFile();


/////////////////////////
//
// TEST WRITING FUNCITONS
//

pid_t pid = 2;
int weights[8] = {2, 2, 2, 2, 2, 2, 2, 2};

printf("Process ID: %d\r\n\r\n", pid);


char in[5];

// TEST writeWeigthAllocation (section 2 of input file). Deletes weight
//writeWeightAllocation(pid+4, weight);



printf("Press enter to write a weight request\r\n");
fflush(stdin);
fgets(in, 2, stdin);

// TEST writeWeightRequest (section 3 of input file). Deletes weight
Weight *weight = weight_init(weights);
writeWeightRequest(pid, weight);

weight = getAvailableWeights();
weight_to_string(weight, line);
weight_del(weight);
printf("currently availble: %s\r\n\r\n\r\n", line);



printf("Press enter to grant the weight request\r\n");
fflush(stdin);
fgets(in, 2, stdin);

// TEST removeWeightAllocation (section 2 of input file). Deletes weight
//weight = weight_init(weights);
```

```
grantWeightRequest(pid);

weight = getAvailableWeights();
weight_to_string(weight, line);
weight_del(weight);
printf("currently availble: %s\r\n\r\n\r\n", line);


printf("Press enter to remove the weight allocation\r\n");
fflush(stdin);
fgets(in, 2, stdin);


// TEST removeWeightRequest (section 3 of input file). Deletes weight
pid_t request_pid = pid;
weight = weight_init(weights);
releaseWeightAllocation(request_pid, weight);
weight_del(weight);

weight = getAvailableWeights();
weight_to_string(weight, line);
weight_del(weight);
printf("currently availble: %s\r\n\r\n\r\n", line);


/////////////////////////
//
// TEST READING FUNCTIONS
//

// TEST getGymResources (section 1 of input file)
Weight *database = getGymResources();
if(database == NULL) exit(1);
weight_to_string(database, line);
weight_del(database);

printf("\r\nGYM RESOURCES\r\n\r\n");
printf("%s\r\n", line);


// TEST getWeightAllocation (section 2 of input file)
WeightMatrix *allocationMatrix = getWeightAllocation();
if(allocationMatrix == NULL) exit(1);
weight_matrix_to_string(allocationMatrix, line);
```

```
    weight_matrix_del(allocationMatrix);

    printf("\r\n----------\r\n\r\n");
    printf("WEIGHT ALLOCATION\r\n\r\n");
    printf("%s\r\n", line);


    // TEST getWeightRequest (section 3 of input file)
    WeightMatrix *requestMatrix = getWeightRequest();
    if(requestMatrix == NULL) exit(1);
    weight_matrix_to_string(requestMatrix, line);
    weight_matrix_del(requestMatrix);

    printf("\r\n----------\r\n\r\n");
    printf("WEIGHT REQUEST\r\n\r\n");
    printf("%s\r\n", line);

    printf("\r\n----------\r\n\r\n");
}
// ##############################################
//
//   Author  -   Collin Thornton
//   Email   -   collin.thornton@okstate.edu
//   Brief   -   Final Project part2 source
//   Date    -   11-20-20
//
// ##############################################


#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <time.h>

#include "start_sim.h"


/**
 * @brief Startup the sim with flags. Used in driver files
 * @param num_trainer (int) total number of trainers in simulation
 * @param num_couches (int) max clients in waiting room
 * @param boundary_case (const bool) solve for part b
 * @param realistc (const bool) toggle realistic weight algorithm in client
 * @param detect_deadlock (const bool) solve for part c
 * @param fix_deadlock (const bool) solve for part d
 * @param trainer_log (const bool) solve for part e
 */
void start_sim(const int num_trainers, const int num_couches, const bool boundary_case, const
bool realistic, const bool detect_deadlock, const bool fix_deadlock, const bool trainer_log) {

    if(num_trainers < 3 || num_trainers > MAX_TRAINERS) {
        printf("Number of trainers must be between %d and %d\r\n\r\n", MIN_TRAINERS,
MAX_TRAINERS);
        exit(-1);
    }

    if(num_couches < 3 || num_couches > MAX_COUCHES) {
        printf("Number of couches must be between %d and %d\r\n\r\n", MIN_COUCHES,
MAX_COUCHES);
        exit(-1);
    }


    // Set the random generator seed
    srand(time(0));

    //////////////////////////
    //
    // Initalized semaphores and shared memory
    //

    if(init_shared_gym(num_couches, num_trainers, boundary_case, realistic, detect_deadlock,
fix_deadlock, trainer_log) == 1) exit(1);
    init_resource_manager();
    init_trainer_sem();
    init_client_sem();
    initRecordBook();

    openRecordBook();
    clearRecordBook();

    //sem_close(shared_gym_sem);
    close_shared_gym();
    close_resource_manager();
    close_trainer_sem();
    close_client_sem();
```

```
closeRecordBook();

/////////////////////////
//
// Launch child processes
//

printf("parent -> pid %d\r\n", getpid());

printf("parent -> spawning %d trainers\r\n", num_trainers);
pid_t trainer_pids[num_trainers];
for(int i=0; i<num_trainers; ++i) trainer_pids[i] = trainer_start();



int num_clients = num_couches - 1;
printf("parent -> spawning %d clients\r\n", num_clients);
pid_t client_pids[num_clients];
for(int i=0; i<num_clients; ++i) client_pids[i] = client_start();


/////////////////////////
//
// Setup shared memory
//

open_shared_gym();
open_resource_manager();
open_trainer_sem();
open_client_sem();
openRecordBook();

Gym *gym = gym_init();
update_gym(gym);

delay(2*gym->unit_time);


/////////////////////////
//
// Setup couch failure
//

close_shared_gym();
```

```
close_resource_manager();
close_trainer_sem();
close_client_sem();
closeRecordBook();
gym_del(gym);

client_start();
client_start();

open_shared_gym();
open_resource_manager();
open_trainer_sem();
open_client_sem();
openRecordBook();
gym = gym_init();
update_gym(gym);


//////////////////////////
//
// Run parent tasks
// - Deadlock detection, check couches taken, etc.
//

printf("gym unit time %d\r\n", gym->unit_time);

delay(15*gym->unit_time);


//////////////////////////
//
// Setup FCFS failure
//

close_shared_gym();
close_resource_manager();
close_trainer_sem();
close_client_sem();
closeRecordBook();
gym_del(gym);

client_start();

open_shared_gym();
```

```
    open_resource_manager();
    open_trainer_sem();
    open_client_sem();
    openRecordBook();
    gym = gym_init();
    update_gym(gym);

    update_gym(gym);

    int print_delay = 0;

    bool first_time = true;

    while(gym->trainerList->len > 0) {


        if(print_delay % 5 == 0) {
            printf("\r\nCOUCHES TAKEN: %d of %d\r\n", gym->waitingList->len,
gym->maxCouches);
        }


        if((gym->detect_deadlock || gym->fix_deadlock) && print_delay % 5 == 0) {

            pid_t deadlock_array[MAX_CLIENTS];
            int num_deadlocked = checkForDeadlock(deadlock_array);

            printf("\r\nDEADLOCKED CLIENTS      ");

            for(int i=0; i<num_deadlocked; ++i) {
                printf("%d ", deadlock_array[i]);
            }

            if(num_deadlocked == 0) {
                printf("NO DEADLOCKED PROCESSES\r\n\r\n");
            }
            else {
                if(gym->fix_deadlock) {
                    printf("\r\n\r\n");

                    pid_t deadlock_victim;
                    int least_sets = 9999;
                    for(int i=0; i<num_deadlocked; ++i) {
                        Client *tmp = client_list_find_pid(deadlock_array[i], gym->workoutList);
                        if(tmp != NULL && (tmp->workout.total_sets - tmp->workout.sets_left) <
least_sets) {
                        printf("parent -> searching client %d with %d of %d sets completed\r\n",
tmp->pid, tmp->workout.total_sets-tmp->workout.sets_left, tmp->workout.total_sets);
                        deadlock_victim = tmp->pid;
                        }
                    }

                    gym->deadlock_victim = deadlock_victim;
                    printf("parent -> chose client %d as deadlock victim\r\n\r\n", deadlock_victim);
                }
            }
        }

        update_shared_gym(gym);
        delay(1*gym->unit_time);
        update_gym(gym);

        ++print_delay;
    }



    /////////////////////////
    //
    // Wait for child processes to exit
    //

    printf("parent -> waiting for processes to exit\r\n");
    for(int i=0; i<num_clients+3; ++i)  waitpid(client_pids[i], NULL, 0);
    for(int i=0; i<num_trainers; ++i) waitpid(trainer_pids[i], NULL, 0);



    /////////////////////////
    //
    // Cleanup the mess
    //

    printf("parent -> destorying data\r\n");

    gym_del(gym);
    close_shared_gym();
    destroy_shared_gym();
```

```
    close_resource_manager();
    destroy_resource_manager();

    close_trainer_sem();
    destroy_trainer_sem();

    close_client_sem();
    destroy_client_sem();

    closeRecordBook();
    destroyRecordBook();
}
// ############################################
//
//   Author  -   Collin Thornton
//   Email   -   collin.thornton@okstate.edu
//   Brief   -   Final Project part2 driver
//   Date    -   11-20-20
//
// ############################################


#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>

#include "start_sim.h"



int main(int argc, char **argv) {
    printf("\r\nCS4323 FINAL PROJECT GROUP D\r\n");
    printf("GYM SIMULATOR\r\n\r\n");
    printf("test_deadklock_rollback -> Parts c\r\n\r\n");
    printf("Collin Thornton\r\nRobert Cook\r\nTyler Krebs\r\n\r\n");
    printf("Usage: ./test_deadlock_rollback <NUM_TRAINERS> <NUM_COUCHES>\n\n");


    if(argc != 3) {
        exit(-1);
    }
    int num_trainers = 0;
    int num_couches = 0;

    if((num_trainers = strtol(argv[1], NULL, 10)) == 0 && errno != 0) {
        exit(-1);
    }

    if((num_couches = strtol(argv[2], NULL, 10)) == 0 && errno != 0) {
        exit(-1);
    }


    start_sim(num_trainers, num_couches, true, false, true, true, false);
}// ###########################################
//
//   Author  -   Collin Thornton
//   Email   -   collin.thornton@okstate.edu
//   Brief   -   Final Project part2 driver
//   Date    -   11-20-20
//
// ###########################################


#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>

#include "start_sim.h"



int main(int argc, char **argv) {
    printf("\r\nCS4323 FINAL PROJECT GROUP D\r\n");
    printf("GYM SIMULATOR\r\n\r\n");
    printf("Recordbook Test -> Part e\r\n\r\n");
    printf("Collin Thornton\r\nRobert Cook\r\nTyler Krebs\r\n\r\n");
    printf("Usage: ./test_recordbook\n\n");


    test_recordbook();
}// ###########################################
//
```

```
//   Author  -   Collin Thornton
//   Email   -   collin.thornton@okstate.edu
//   Brief   -   Final Project trainer struct source
//   Date    -   11-24-20
//
// #########################################

#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

#include "trainer.h"
#include "workout_room.h"
#include "resource_manager.h"
#include "recordbook.h"
#include "gym.h"


// Define semaphore name
static const char TRAINER_SEM_NAME[] = "/sem_trainer";


// Semaphore to synchronize trainers
static sem_t *trainer_sem;



/////////////////////////////
//
// Trainer process functions
//


/**
 * @brief Spawn a trainer child process. new process will launch trainer_proc_state_machin()
 * @return (pid_t) Process ID of new child process
 */
pid_t trainer_start() {
    pid_t pid = fork();

    if(pid < 0) {
        perror("trainer_start() fork");
        return pid;
    }
    else if(pid == 0) {
        printf("new trainer pid: %d\r\n", getpid());
        int ret = trainer_proc_state_machine();
        exit(ret);
    }
    else {
        return pid;
    }
}


/**
 * @brief Execute the trainer state machine. Should only be run by trianer proccess
 * @return (int) return code. negative on error
 */
int trainer_proc_state_machine() {

    // Set the seed for the random number generator. Used when setting workouts for the client
    srand(getpid());

    open_trainer_sem();
    open_resource_manager();
    openRecordBook();

    int pid = getpid();

    Gym *gym = gym_init();

    open_shared_gym();
    update_gym(gym);

    if(gym == NULL) {
        perror("trainer_proc_state_machine get shared gym");
        return -1;
    }


    // Initialize the trainer struct
    Trainer *trainer = trainer_init(pid, -1, TRAVELLING);
    Client *client;

    // Add ourself to the gym's list of trainers and refresh the list
    trainer_list_add_trainer(trainer, gym->trainerList);
    update_shared_gym(gym);
    update_gym(gym);

    char buffer[BUFFER_SIZE] = "\0";

    bool shutdown = false;

    const int MAX_WAIT = 30;
    int num_wait = 0;


    // START STATE MACHINE
    while(!shutdown) {


        switch(trainer->state) {
            case FREE:
                printf("TRAINER %d FREE %d of %d CLIENTS IN WAITING ROOM\r\n", pid,
gym->waitingList->len, gym->maxCouches);

                sem_wait(trainer_sem);
                int val;
                sem_getvalue(trainer_sem, &val);

                update_gym(gym);

                if(gym->waitingList->len > 0) {
                    ClientNode *node = gym->waitingList->HEAD;
                    while(node != NULL) {
                        // Check if client is already claimed by a trainer
                        Trainer *tmp = trainer_list_find_client(node->node->pid, gym->trainerList);
                        if(tmp == NULL) break;
                        node = node->next;
                    }
                    if(node != NULL) {
                        trainer->state = WITH_CLIENT;
                        trainer->client_pid = node->node->pid;
                    }
                }
                else {
                    trainer->state = ON_PHONE;
                }
                update_shared_gym(gym);

                sem_post(trainer_sem);

                delay(2*gym->unit_time);
                break;

            case ON_PHONE:
                // CHECK IF WE'VE BEEN CLAIMED BY A CLIENT
                printf("TRAINER %d ON PHONE\r\n", pid);

                client = client_list_find_trainer(getpid(), gym->arrivingList);

                if(client != NULL) {
                    trainer->client_pid = client->pid;
                    trainer->state = WITH_CLIENT;
                    update_shared_gym(gym);
                    num_wait = 0;
                }
                else {
                    ++num_wait;
                    if(num_wait == MAX_WAIT) shutdown = true;
                }

                delay(2*gym->unit_time);

                break;

            case TRAVELLING:
                printf("TRAINER %d TRAVELLING\r\n", pid);
                trainer->client_pid = -1;

                delay(3*gym->unit_time);
                trainer->state = FREE;

                break;

            case WITH_CLIENT:
                printf("TRAINER %d WITH_CLIENT %d\r\n", pid, trainer->client_pid);
                delay(2*gym->unit_time);
                trainer_workout_event(gym, trainer);
                trainer->state = TRAVELLING;

                break;
                }

                update_shared_gym(gym);
                update_gym(gym);
            }

            // Remove trainer from lists & destroy semaphores

            printf("Trainer %d destroying data\r\n", getpid());

            trainer_list_rem_trainer(trainer, gym->trainerList);
            update_shared_gym(gym);

            trainer_del(trainer);
            gym_del(gym);

            close_shared_gym();
            close_resource_manager();
            close_trainer_sem();
            closeRecordBook();

            printf("Trainer %d exiting\r\n", getpid());

            return 0;
}



/////////////////////////////
//
// Sempahore handling
//

/**
 * @brief Initalize the trainer semaphore. Should be called on the parent process.
 * @return (int) return code. negative on failure.
 */
int init_trainer_sem() {
    sem_unlink(TRAINER_SEM_NAME);
    trainer_sem = sem_open(TRAINER_SEM_NAME, O_CREAT, 0644, 1);
    if(trainer_sem == SEM_FAILED) {
        perror("trainer_sem_init failed to open");
        exit(1);
    }
```

```
}


/**
 * @brief Open the trainer sempaahore. SHould be called on the trainer process
 * @return (int) return code. negative on error
 */
int open_trainer_sem() {
    trainer_sem = sem_open(TRAINER_SEM_NAME, O_CREAT, 0644, 1);
    if(trainer_sem == SEM_FAILED) {
        perror("trainer_sem_init failed to open");
        exit(1);
    }
    return 0;
}


/**
 * @brief close the trainer semaphore. should be called after open_trainer_sem()
 */
void close_trainer_sem() {
    sem_close(trainer_sem);
    return;
}


/**
 * @brief free the trainer semaphore. should be called after init_trainer_sem() on parent
 */
void destroy_trainer_sem() {
    sem_unlink(TRAINER_SEM_NAME);
    return;
}


////

//////////////////////////
//
// Trainer struct functions
//


/**
```

```
 * @brief Allocate trainer on heap. Init params as NULL or negative if unavailable
 * @param pid (pid_t) Process ID of trainer
 * @param client_pid (pid_t) Process ID of client. -1 if unavailable
 * @param state (TrainerState) Inital state of trainer
 * @return (Trainer*) pointer to a trainer struct
 */
Trainer* trainer_init(pid_t pid, pid_t client_pid, TrainerState state) {
    Trainer* trainer = (Trainer*)malloc(sizeof(Trainer));

    if(trainer == NULL) {
        perror("trainer_init malloc()");
        return NULL;
    }

    trainer->pid = pid;
    trainer->state = state;
    trainer->client_pid = client_pid;

    Workout *tmp = workout_init(-1, -1, -1, NULL);
    trainer->workout = *tmp;
    workout_del(tmp);

    return trainer;
}


/**
 * @brief Delete a trainer from the heap
 * @param trainer (Trainer*) trainer to be deleted
 * @return (int)return code. negative on erro
 */
int trainer_del(Trainer* trainer) {
    if(trainer == NULL) return -1;

    free(trainer);
    return 0;
}


/**
 * @brief Stringify a trainer struct
 * @param trainer (Traienr*) struct to be stringified
 * @param buffer (char[]) buffer to store new string
 * @return (const char*) same as buffer
 */
const char* trainer_to_string(Trainer *trainer, char buffer[]) {
    if(trainer == NULL) return NULL;

    sprintf(buffer, "pid: %d", trainer->pid);
    sprintf(buffer + strlen(buffer), "   state: %d", trainer->state);
    sprintf(buffer + strlen(buffer), "   client: %d", trainer->client_pid);
    return buffer;
}



//////////////////////////////
//
// Trainer list functions
//



/**
 * @brief Initalize a trainer LL on the heap
 * @return (TrainerList*) pointer to newly allocated LL
 */
TrainerList* trainer_list_init() {
    TrainerList* list = (TrainerList*)malloc(sizeof(TrainerList));

    if(list == NULL) {
        perror("trainer_list_init malloc()");
        return NULL;
    }

    list->HEAD = NULL;
    list->TAIL = NULL;
    list->len = 0;
    return list;
}


/**
 * @brief Free a trainer list and all internal nodes. Does not free the individual trainers
 * @param list (TrainerList*) Pointer to the current shared list
 * @return (int) return code. negative on error
 */
int trainer_list_del_trainers(pid_t exclude, TrainerList *list) {
    if(list == NULL) return 0;
    TrainerNode *tmp = list->HEAD;

    while(tmp != NULL) {
        if(tmp->node->pid != exclude) trainer_del(tmp->node);
        tmp = tmp->next;
    }
    return 0;
}


/**
 * @brief Delete trainers from a given list
 * @param exclude (pid_t) Process ID of exclude trainer
 * @param list (TrainerList*) List of current trainers
 * @return (int) return code. negative on error
 */
int trainer_list_del(TrainerList *list) {
    if(list == NULL) return 0;

    TrainerNode *tmp = list->HEAD;

    while(tmp != NULL) {
        TrainerNode *next = tmp->next;
        free(tmp);
        tmp = next;
    }

    free(list);

    return 0;
}

/**
 * @brief Add a trainer to a list
 * @param trainer (Trianer*) trainer to be added
 * @param list (TrainerList*) target list
 * @return (int) return code. negative on error
 */
int trainer_list_add_trainer(Trainer *trainer, TrainerList *list) {
    TrainerNode *new_node = (TrainerNode*)malloc(sizeof(TrainerNode));

    if(new_node == NULL) {
        perror("trainer_list_add_trainer malloc()");
        return -1;
    }

    if(trainer_list_find_pid(trainer->pid, list) != NULL) {
        free(new_node);
        return 1;
    }

    new_node->node = trainer;
    new_node->next = NULL;
    new_node->prev = list->TAIL;

    if(list->HEAD == NULL) {
        list->HEAD = new_node;
        list->TAIL = new_node;
    }
    else {
        list->TAIL->next = new_node;
        list->TAIL = new_node;
    }

    ++list->len;
    return list->len;
}


/**
 * @brief Remove a trainer from the list
 * @param trainer (Trainer*) trainer to be removed
 * @param list (TrainerList*) list from which to remove the trainer
 * @return (int) return code. negative on error.
 */
int trainer_list_rem_trainer(Trainer *trainer, TrainerList *list) {
    if(list == NULL || trainer == NULL) {
        perror("trainer_list_rem_trainer() invalid_argument");
        return -1;
    }

    TrainerNode *tmp = trainer_list_srch(trainer, list);

    if(tmp == NULL) return list->len;

    if(tmp == list->HEAD && tmp == list->TAIL) {
        free(tmp);
        list->HEAD = NULL;
        list->TAIL = NULL;
    }
    else if(tmp == list->HEAD) {
        TrainerNode *new_head = list->HEAD->next;
        free(tmp);
        list->HEAD = new_head;
        new_head->prev = NULL;
    }
    else if(tmp == list->TAIL) {
        TrainerNode *new_tail = list->TAIL->prev;
        free(tmp);
        list->TAIL = new_tail;
        new_tail->next = NULL;
    }
    else {
        TrainerNode *prevNode = tmp->prev;
        TrainerNode *nextNode = tmp->next;
        free(tmp);
        prevNode->next = nextNode;
        nextNode->prev = prevNode;
    }

    --list->len;
    return list->len;
}


/**
 * @brief Stringify a trainer list
 * @param list (TrainerList*) list to be stringified
 * @param buffer (char[]) buffer to store new string
 * @return (const char*) same as buffer
 */
const char* trainer_list_to_string(TrainerList* list, char buffer[]) {
    if(list == NULL) return NULL;

    if(list->HEAD == NULL) {
        sprintf(buffer, "EMPTY\n");
        return buffer;
    }

    TrainerNode *tmp = list->HEAD;
    char buff[1024];
    trainer_to_string(tmp->node, buff);
    sprintf(buffer, "%s   HEAD", buff);
    tmp = tmp->next;

    while(tmp != NULL) {
        char buff[1024];
        trainer_to_string(tmp->node, buff);
        sprintf(buffer + strlen(buffer), "\n%s", buff);
        tmp = tmp->next;
    }
    sprintf(buffer + strlen(buffer), "   TAIL\n");
    return buffer;
}


/**
 * @brief Search for a client in trainer LL
 * @param client_pid (pid_t) needle -> pid of client in question
 * @param list (TrainerList*) poitner to LL with client
 * @return (Trainer*) null if not found, otherwise pointer to the trainer
 */
Trainer* trainer_list_find_client(pid_t client_pid, TrainerList *list) {
    if(list == NULL || client_pid < 0) return NULL;

    TrainerNode *tmp = list->HEAD;
    while(tmp != NULL) {
        if(tmp->node->client_pid == client_pid) return tmp->node;
        tmp = tmp->next;
    }
    return NULL;
}


/**
 * @brief Find a trainer currently on their phone
 * @param list (TrainerList*) list to be searched
 * @return (Trainer*) null if not found, othewise poitner to the trainer
 */
Trainer* trainer_list_find_phone(TrainerList *list) {
    return trainer_list_find_state(ON_PHONE, list);
}


/**
```

```
 * @brief Find a trainer currently on their available
 * @param list (TrainerList*) list to be searched
 * @return (Trainer*) null if not found, othewise poitner to the trainer
 */
Trainer* trainer_list_find_available(TrainerList *list) {
    return trainer_list_find_state(FREE, list);
}


/**
 * @brief Find a trainer currently at a given state
 * @param list (TrainerList*) list to be searched
 * @return (Trainer*) null if not found, othewise poitner to the trainer
 */
Trainer* trainer_list_find_state(TrainerState state, TrainerList *list) {
    if(list == NULL) return NULL;

    TrainerNode *tmp = list->HEAD;
    while(tmp != NULL) {
        if(tmp->node->state == state) return tmp->node;
        tmp = tmp->next;
    }
    return NULL;
}


/**
 * @brief Find a trainer currently with pid in LL
 * @param list (TrainerList*) list to be searched
 * @return (Trainer*) null if not found, othewise poitner to the trainer
 */
Trainer* trainer_list_find_pid(pid_t pid, TrainerList *list) {
    if(list == NULL) return NULL;

    TrainerNode *tmp = list->HEAD;
    while(tmp != NULL) {
        if(tmp->node->pid == pid) return tmp->node;
        tmp = tmp->next;
    }
    return NULL;
}


/**
```

```
 * @brief Search a trainer LL for a given trainer
 * @param trainer (Trainer*) needle
 * @param list (TrainerLIst*) haystack
 * @return (TrainerNode*) pointer to node containing the Trainer struct
 */
TrainerNode* trainer_list_srch(Trainer *trainer, TrainerList *list) {
    if (trainer == NULL || list == NULL) return NULL;

    TrainerNode *tmp = list->HEAD;
    while(tmp != NULL) {
        if(tmp->node == trainer) return tmp;
        tmp = tmp->next;
    }
    return NULL;
}




void test_trainer_list() {
    printf("\r\n");

    Trainer *trainer_one = trainer_init(1, 0, FREE);
    Trainer *trainer_two = trainer_init(2, 0, FREE);
    Trainer *trainer_three = trainer_init(3, 0, FREE);

    TrainerList *trainer_list = trainer_list_init();
    trainer_list_add_trainer(trainer_one, trainer_list);
    trainer_list_add_trainer(trainer_two, trainer_list);
    trainer_list_add_trainer(trainer_three, trainer_list);

    int buffer_size = 1024*trainer_list->len;
    char buffer[buffer_size];
    trainer_list_to_string(trainer_list, buffer);

    printf("INITIAL LIST: \r\n%s\r\n", buffer);

    trainer_list_rem_trainer(trainer_one, trainer_list);
    trainer_list_to_string(trainer_list, buffer);

    printf("TRAINER ONE REMOVED: \r\n%s\r\n", buffer);

    trainer_list_rem_trainer(trainer_two, trainer_list);
    trainer_list_to_string(trainer_list, buffer);
    printf("TRAINER TWO REMOVED:\r\n%s\r\n", buffer);

    trainer_list_rem_trainer(trainer_three, trainer_list);
    trainer_list_to_string(trainer_list, buffer);

    printf("TRAINER THREE REMOVED\r\n%s\r\n", buffer);

    trainer_del(trainer_one);
    trainer_del(trainer_two);
    trainer_del(trainer_three);
    trainer_list_del(trainer_list);
}// ##############################################
//
//   Author  -   Collin Thornton
//   Email   -   collin.thornton@okstate.edu
//   Brief   -   Final Project vecotr src
//   Date    -   11-24-20
//
// ##############################################


#include <stdbool.h>
#include <stdlib.h>

#include "vector.h"




/**
 * @brief test if v1 <= v2
 * @param v1 (int*) left vector
 * @param v2 (int*) right vector
 * @param size (int) number of elements in vectors. assumed to be same for both
 * @return (bool) true if v1 <= v2. else false
 */
 bool vector_less_than_equal(int *v1, int *v2, int size) {
    if(v1 == NULL || v2 == NULL) return false;
    for(int i=0; i<size; ++i) {
        if(v1[i] > v2[i]) return false;
    }
    return true;
}
```

```
/**
 * @brief test if v1 < v2
 * @param v1 (int*) left vector
 * @param v2 (int*) right vector
 * @param size (int) number of elements in vectors. assumed to be same for both
 * @return (bool) true if v1 < v2. else false
 */
 bool vector_less_than(int *v1, int *v2, int size) {
   if(v1 == NULL || v2 == NULL) return false;
   for(int i=0; i<size; ++i) {
       if(v2[i] >= v1[i]) return false;
   }
   return true;
}

/**
 * @brief test if v1 == v2
 * @param v1 (int*) left vector
 * @param v2 (int*) right vector
 * @param size (int) number of elements in vectors. assumed to be same for both
 * @return (bool) true if v1 == v2. else false
 */
 bool vector_equal(int *v1, int *v2, int size) {
   if(v1 == NULL || v2 == NULL) return false;
   for(int i=0; i<size; ++i) {
       if(v1[i] != v2[i]) return false;
   }
   return true;
}


/**
 * @brief test if v1[i] == 0 for all i < size
 * @param v1 (int*) vector to check
 * @param size (int) number of elements to check
 * @return (bool) true if v1[i] == 0 for all i < size. else false
 */
bool vector_zero(int *v1, int size) {
   if(v1 == NULL) return false;
   for(int i=0; i<size; ++i) if(v1[i] != 0) return false;
   return true;
}
```

```
/**
 * @brief check if v1[i] < 0 for any i < size
 * @param v1 (int*) vector to check
 * @param size (int) number of elements to check
 * @return (bool) true of any v1[i] < 0
 */
bool vector_negative(int *v1, int size) {
    if(v1 == NULL) return false;
    for(int i=0; i<size; ++i) if(v1[i] < 0) return true;
    return false;
}

/**
 * @brief perform v1[i] += v2[i] for all i < size
 * @param v1 (int*) left vector
 * @param v2 (int*) right vector
 * @param size (int) number of elements to add
 * @return (int*) v1 = v1 + v2
 */
 int* vector_add(int *v1, int *v2, int size) {
    if(v1 == NULL || v2 == NULL) return NULL;
    for(int i=0; i<size; ++i) v1[i] += v2[i];
    return v1;
}


/**
 * @brief perform v1[i] -= v2[i] for all i < size
 * @param v1 (int*) left vector
 * @param v2 (int*) right vector
 * @param size (int) number of elements to add
 * @return (int*) v1 = v1 - v2
 */
 int* vector_subtract(int *v1, int *v2, int size) {
    if(v1 == NULL || v2 == NULL) return NULL;
    for(int i=0; i<size; ++i) v1[i] -= v2[i];
    return v1;
}// ##########################################
//
//   Author  -   Collin Thornton
//   Email   -   collin.thornton@okstate.edu
//   Brief   -   Final Project workout src
//   Date    -   11-26-20
//
```

```
// ##########################################


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

#include "workout.h"


/**
 * @brief Allocate a workout on the heap
 * @param total_sets (int) Total sets in workout
 * @param sets_left (int) Sets left in workout
 * @param total_weight (int) Total weight of workout
 * @param in_use (Weight*) Points to weights currently being used
 */
Workout* workout_init(int total_sets, int sets_left, int total_weight, Weight *weight) {
    Workout *workout = (Workout*)malloc(sizeof(Workout));

    if(workout == NULL) {
        perror("workout_init malloc");
        return NULL;
    }

    workout->total_weight = total_sets;
    workout->sets_left = sets_left;
    workout->total_weight = total_weight;
    if(weight != NULL) {
        workout->in_use = *weight;
    }
    else {
        Weight *tmp_weight = weight_init(NULL);
        workout->in_use = *tmp_weight;
        weight_del(tmp_weight);
    }

    return workout;
}


/**
 * @brief Free a workout from the heap
 * @param workout (Worktout*) workout to be deleted
 * @return (int) return code. negative on failure
 */
int workout_del(Workout *workout) {
    if(workout == NULL) return -1;
    //weight_del(workout->in_use);
    free(workout);
    return 0;
}// ############################################
//
//   Author  -   Collin Thornton
//   Email   -   collin.thornton@okstate.edu
//   Brief   -   Final Project part2 source
//   Date    -   11-20-20
//
// ############################################


#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <math.h>

#include "client.h"
#include "workout_room.h"
#include "resource_manager.h"
#include "workout.h"
#include "vector.h"
#include "recordbook.h"



// Set by each trainer to track the starting weight. Is randomized. Weight grows from here.
static int init_weight;



/////////////////////////////
//
// Event functions
//


/**
 * @brief Execute the workout event on client process.
 * @param gym (Gym*) gym struct used for IPC
 * @param client (Client*) client in workout event
 * @return (int) return code. negative on error
 */
int client_workout_event(Gym *gym, Client *client) {
    if(gym == NULL || client == NULL) {
        perror("client_workout_event invalid_argument");
        return -1;
    }
    if(client->current_trainer.pid < 0) {
        perror("client_workout_event no trainer");
        return -1;
    }

    Trainer *trainer = trainer_list_find_pid(client->current_trainer.pid, gym->trainerList);
    if(trainer == NULL) {
        perror("client_workout_event() trainer not in list");
        return -1;
    }

    // MAKE SURE WE'VE ADDED OURSELF TO THE CORRECT LIST
    client_list_rem_client(client, gym->arrivingList);
    client_list_rem_client(client, gym->waitingList);
    client_list_add_client(client, gym->workoutList);
    update_shared_gym(gym);


    // GET THE INTIAL WORKOUT FROM THE TRAINER
    client_get_workout(gym, client, trainer, true);
    update_shared_gym(gym);

    if(!gym->realistic)
        printf("client %d got workout. total sets: %d\r\n", getpid(), client->workout.total_sets);

    else
        printf("client %d got workout. total weight: %d\r\n", getpid(), client->workout.total_weight);

    for(int i=TWO_HALF; i<=FORTY_FIVE; ++i) client->workout.in_use.num_plates[i] = 0;
    client->workout.sets_left = client->workout.total_sets;


    // RUN THROUGH SETS
    //TODO MIGHT NEED TO ADD EXTRA WEIGHT REQUESTS TO ENCOURAGE
DEADLOCK
```

```
    //! SHOULD BE IN A DIFFERENT FUNCTION FOR ROLLBACK
    for(int i=0; i<client->workout.total_sets; ++i) {
        delay(1*gym->unit_time);

        if(!gym->realistic)
            printf("client %d performing set %d of %d\r\n", getpid(), i+1, client->workout.total_sets);

        else
            printf("client %d performing set %d of %d with %d weight\r\n", getpid(), i+1,
client->workout.total_sets, client->workout.total_weight);

        // Redo the set if we're the deadlock victim
        if(client_get_weights(gym, client) == -2) {

            printf("\r\nclient %d successfully rolled back from set %d to set %d\r\n\r\n", getpid(), i+1,
i);
            --i;
            continue;
        }
        //client->workout.total_weight = -1;
        //update_shared_gym(gym);

        client_lift_weights(gym, client);
        update_shared_gym(gym);

        client_get_workout(gym, client, trainer, false);
    }

    releaseWeightAllocation(getpid(), &client->workout.in_use);

    printf("client %d finished workout\r\n", getpid());


    // REPLACE WEIGHTS
    Workout *tmp = workout_init(client->workout.total_sets, client->workout.sets_left,
client->workout.total_weight, NULL);
    client->workout = *tmp;
    workout_del(tmp);


    update_shared_gym(gym);
}
```

```
/**
 * @brief Execute the workout event on trainer proccess.
 * @param gym (Gym*) gym struct used for IPC
 * @param trianer (Trainer*) trainer in workout event
 * @return (int) return code. negative on error
 */
int trainer_workout_event(Gym *gym, Trainer *trainer) {
    if(gym == NULL || trainer == NULL) {
        perror("trainer_workout_event invalid_argument");
        return -1;
    }
    if(trainer->client_pid <= 0) {
        perror("trainer_workout_event no client");
        return -1;
    }

    update_gym(gym);

    Client *client = client_list_find_pid(trainer->client_pid, gym->workoutList);

    while(client == NULL) {
        delay(1*gym->unit_time);
        update_gym(gym);
        client = client_list_find_pid(trainer->client_pid, gym->workoutList);
    }


    // CREATE THE WORKOUT AND SEND TO CLIENT
    init_weight = rand();

    trainer_set_workout(gym, trainer);
    update_shared_gym(gym);

    if(!gym->realistic)
        printf("trainer %d picked workout for client %d: num_sets %d\r\n", getpid(),
trainer->client_pid, trainer->workout.total_sets);

    else
        printf("trainer %d picked workout for client %d: total_weight %d, num_sets %d\r\n",
getpid(), trainer->client_pid, trainer->workout.total_weight, trainer->workout.total_sets);
```

```
    // WAIT FOR CLIENT TO ACKNOWLEDGE WORKOUT
    while(client != NULL && client->workout.total_weight <= 0) {
        delay(2*gym->unit_time);
        update_gym(gym);
        trainer = trainer_list_find_pid(getpid(), gym->trainerList);
        client = client_list_find_pid(trainer->client_pid, gym->workoutList);
    }

    int total_weight = client->workout.total_weight;

    // NOW WAIT FOR CLIENT TO LEAVE THE WORKOUT

    while(client != NULL) {

        // WAIT FOR CLIENT TO GRAB WEIGHTS
        while(client != NULL && client->workout.total_weight > 0) {
            delay(0.5*gym->unit_time);
            update_gym(gym);
            trainer = trainer_list_find_pid(getpid(), gym->trainerList);
            client = client_list_find_pid(trainer->client_pid, gym->workoutList);
        }

        trainer_set_workout(gym, trainer);
        update_shared_gym(gym);
        update_gym(gym);

        trainer = trainer_list_find_pid(getpid(), gym->trainerList);
        client = client_list_find_pid(trainer->client_pid, gym->workoutList);

        if(gym->realistic && client != NULL)
            printf("trainer %d picked %d weight for client %d\r\n", getpid(),
trainer->workout.total_weight, trainer->client_pid);

        // WAIT FOR CLIENT TO ACKNOWLEDGE WORKOUT
        while(client != NULL && client->workout.total_weight <= 0) {
            delay(0.5*gym->unit_time);
            update_gym(gym);
            trainer = trainer_list_find_pid(getpid(), gym->trainerList);
            client = client_list_find_pid(trainer->client_pid, gym->workoutList);
        }

        if(client != NULL) total_weight += client->workout.total_weight;
        delay(2*gym->unit_time);

    }

    trainer->workout.sets_left = -1;
    trainer->workout.total_sets = -1;
    trainer->workout.total_weight = -1;


    if(gym->trainer_log) {
        Emp record_entry;
        record_entry.id = trainer->client_pid;
        record_entry.weight = total_weight;

        for(int i=0; i<MAX_NAME_LEN; ++i) record_entry.name[i] = '\0';

        sprintf(record_entry.name, "client %d", trainer->client_pid);

        addToRecordBook(&record_entry);
    }

    printf("trainer %d client finished workout out\r\n", getpid());

    // FINISHED OUR JOB
    return 0;
}





//////////////////////////////
//
// Helper functions
//


/**
 * @brief Choose total weight and total sets. Place in IPC
 * @param gym (Gym*) gym struct for IPC
 * @param trainer (Trainer*) trainer in event
 * @return (int) return code. negative on error
 */
int trainer_set_workout(Gym *gym, Trainer *trainer) {
    // TODO This will need to setup to cause deadlock

    // GENERATE TOTAL WEIGHT, SHOULD BE A MULTIPLE OF 5

    static int weight_increase = 0;
    weight_increase += 5;

    int total_weight = ((init_weight) % (MAX_WEIGHT - MIN_WEIGHT + 1)/2) + MIN_WEIGHT +
weight_increase;
    total_weight = (total_weight > MAX_WEIGHT*2) ? 2*MAX_WEIGHT : total_weight;
    total_weight = 5 * (total_weight/5);

    // GENERATE TOTAL SETS
    int total_sets = (rand() % 11) + 1;;
    int sets_left = -1;                    // Decided by the client

    trainer->workout.sets_left = sets_left;
    trainer->workout.total_sets = total_sets;
    trainer->workout.total_weight = total_weight;

    return 0;
}


/**
 * @brief Get total weight and total sets from trianer over shared memory
 * @param gym (Gym*) gym struct for IPC
 * @param client (Client*) client in event
 * @param trainer (Trainer*) trainer that's paired with client
 * @param first_time (bool) flag to toggle whether the trainer can change the total number of
sets
 */
int client_get_workout(Gym *gym, Client *client, Trainer *trainer, bool first_time) {
    // WAIT FOR THE TRAINER TO SEND US A WORKOUT

    client->workout.total_weight = -1;

    update_shared_gym(gym);
    update_gym(gym);

    trainer = trainer_list_find_client(getpid(), gym->trainerList);
    while(trainer->workout.total_weight <= 0) {
        #ifdef VERBOSE
        printf("client %d waiting for workout\r\n", getpid());
        #endif //VERBOSE
        delay(1*gym->unit_time);

        update_gym(gym);
        client = client_list_find_pid(getpid(), gym->workoutList);
        trainer = trainer_list_find_pid(client->current_trainer.pid, gym->trainerList);
    }

    // TRAINER SENT WORKOUT. COPY TO CLIENT
    if(first_time) {
        client->workout.total_sets = trainer->workout.total_sets;
        client->workout.sets_left = client->workout.total_sets;
    }

    client->workout.total_weight = trainer->workout.total_weight;

    update_shared_gym(gym);
    trainer = trainer_list_find_client(getpid(), gym->trainerList);

    return 0;
}


/**
 * @brief Get grip plates from the resource manager
 * @param gym (Gym*) gym struct for IPC
 * @param client (Client*) client in event
 */
int client_get_weights(Gym *gym, Client *client) {
    // FIGURE OUT HOW MANY PLATES WE NEED WHILE UTILIZING THE SMALLEST
NUMBER
    int weight_left = client->workout.total_weight;

    int weights[NUMBER_WEIGHTS];

    if(gym->realistic) {
        // If we're running the realistic algorithm, pick an optimal number of weights with restrictions

        weights[FORTY_FIVE] = 2*(weight_left/45/2);
        weights[FORTY_FIVE] = (weights[FORTY_FIVE] > 4) ? 4 : weights[FORTY_FIVE];
        weight_left -= 45*weights[FORTY_FIVE];

        weights[THIRTY_FIVE] = 2*(weight_left/35/2);
        weights[THIRTY_FIVE] = (weights[THIRTY_FIVE] > 4) ? 4 : weights[THIRTY_FIVE];
        weight_left -= 35*weights[THIRTY_FIVE];

        weights[TWENTY_FIVE] = 2*(weight_left/25/2);
        weights[TWENTY_FIVE] = (weights[TWENTY_FIVE] > 4) ? 4 : weights[TWENTY_FIVE];
        weight_left -= 25*weights[TWENTY_FIVE];

        weights[TWENTY] = 2*(weight_left/20/2);
        weights[TWENTY] = (weights[TWENTY] > 4) ? 4 : weights[TWENTY];
        weight_left -= 20*weights[TWENTY];

        weights[FIFTEEN] = 2*(weight_left/15/2);
        weights[FIFTEEN] = (weights[FIFTEEN] > 4) ? 4 : weights[FIFTEEN];
        weight_left -= 15*weights[FIFTEEN];

        weights[TEN] = 2*(weight_left/10/2);
        weights[TEN] = (weights[TEN] > 4) ? 4 : weights[TEN];
        weight_left -= 10*weights[TEN];

        weights[FIVE] = 2*(weight_left/5/2);
        weights[FIVE] = (weights[FIVE] > 4) ? 4 : weights[FIVE];
        weight_left -= 5*weights[FIVE];

        weights[TWO_HALF] = 2*(int)(weight_left/2.5/2);
        weights[TWO_HALF] = (weights[TWO_HALF] > 4) ? 4 : weights[TWO_HALF];
        weight_left -= (int)(2.5*weights[TWO_HALF]);

        if(weight_left != 0) {
            perror("client_workout_event weight_left nonzero");
            printf("%d\r\n", weight_left);
            return -1;
        }
    }
    else {
        // Pick nonrealistic weights to cause deadlock. Ignore the trainer's recommended total
weight

        #ifdef VERBOSE
        printf("Client %d has %d sets left\r\n", getpid(), client->workout.sets_left);
        #endif // VERBOSE

        switch(client->workout.sets_left) {
            case 11:
                for(int i=TWO_HALF; i<=FORTY_FIVE; ++i) weights[i] = 1;
                break;
            case 10:
                for(int i=TWO_HALF; i<=FORTY_FIVE; ++i) weights[i] = 3;
                break;
            case 9:
                for(int i=TWO_HALF; i<=FORTY_FIVE; ++i) weights[i] = 8;
                break;
            case 8:
                for(int i=TWO_HALF; i<=FORTY_FIVE; ++i) weights[i] = 9;
                break;
            case 7:
                for(int i=TWO_HALF; i<=FORTY_FIVE; ++i) weights[i] = 10;
                break;
            case 6:
                for(int i=TWO_HALF; i<=FORTY_FIVE; ++i) weights[i] = 6;
                break;
            case 5:
                for(int i=TWO_HALF; i<=FORTY_FIVE; ++i) weights[i] = 8;
                break;
            case 4:
                for(int i=TWO_HALF; i<=FORTY_FIVE; ++i) weights[i] = 5;
                break;
            case 3:
                for(int i=TWO_HALF; i<=FORTY_FIVE; ++i) weights[i] = 7;
                break;
            case 2:
                for(int i=TWO_HALF; i<=FORTY_FIVE; ++i) weights[i] = 6;
                break;
            case 1:
                for(int i=TWO_HALF; i<=FORTY_FIVE; ++i) weights[i] = 5;
                break;
        }
    }

    client->workout.total_weight = 0;
    update_shared_gym(gym);

    Weight *request = weight_init(weights);
    vector_subtract(request->num_plates, client->workout.in_use.num_plates,
NUMBER_WEIGHTS);

    Weight req = *request;

    #ifdef VERBOSE
    char buffer[BUFFER_SIZE] = "\0";
    weight_to_string(&client->workout.in_use, buffer);

    printf("client %d in use\r\n%s\r\n", getpid(), buffer);
    weight_to_string(request, buffer);
    printf("client %d making weight request\r\n%s\r\n", getpid(), buffer);
    #endif // VERBOSE

    int ret;
    while((ret = writeWeightRequest(getpid(), request)) < 0) {
        #ifdef VERBOSE
        printf("client %d request denied: %d\r\n", getpid(), ret);
        #endif // VERBOSE
        delay(1*gym->unit_time);
    }

    #ifdef VERBOSE
    printf("client %d successfully requested weights\r\n", getpid());
    #endif // VERBOSE
    delay(2*gym->unit_time);



    //! THIS IS WHERE WE MIGHT DEADLOCK
    bool success = false;
    do {
        success = client_request_weight_allocation(gym, client, &req);

        // CHECK IF WE'RE SET AS THE DEADLOCK VICTIM
        if(gym->deadlock_victim == getpid()) {
            printf("\r\nClient %d targeted as deadlock victim -> releasing weight allocation and
requests\r\n\r\n", getpid());


            releaseWeightAllocation(getpid(), &client->workout.in_use);
            vector_subtract(client->workout.in_use.num_plates, client->workout.in_use.num_plates,
NUMBER_WEIGHTS);

            Weight *tmp_req = weight_init(req.num_plates);
            removeWeightRequest(getpid(), tmp_req);

            #ifdef VERBOSE
            weight_to_string(&client->workout.in_use, buffer);
            printf("Client %d released allocation\r\n%s\r\n", getpid(), buffer);
            weight_to_string(&req, buffer);
            printf("Client %d release request\r\n%s\r\n", getpid(), buffer);
            #endif // VERBOSE

            // Roll back a set
            //++client->workout.sets_left;
            update_shared_gym(gym);
            update_gym(gym);
            return -2;
        }

    } while(success == false);


    vector_add(client->workout.in_use.num_plates, req.num_plates, NUMBER_WEIGHTS);
    return 0;
}


/**
 * @brief Make a weight request from the resource manaegr
 * @param gym (Gym*) gym struct for IPC
 * @param client (Client*) client in event
 * @param weight (Weight*) request
 * @return bool true on succes. else false
 */
bool client_request_weight_allocation(Gym *gym, Client *client, Weight *weight) {
    int ret;
    while((ret = grantWeightRequest(getpid())) < 0 && gym->deadlock_victim != getpid()) {
        #ifdef VERBOSE
        printf("Client %d allocation denied\r\n", getpid());
        #endif // VERBOSE

        delay(1*gym->unit_time);
        update_gym(gym);
    }
    return (ret == 0) ? true : false;
}


/**
 * @brief Delay for a bit and set flags as the client lifts weights
 */
void client_lift_weights(Gym *gym, Client *client) {
    delay(2*gym->unit_time);
    --client->workout.sets_left;
}

void test_workout_room() {
    test_resource_manager();
    return;
}
```