OS Assignment – Semester One 2020 – Lift Simulation Using Multithreading and Multiprocessing

Author: Dylan Travers

ID: 19128043

COMP2006 – Operating Systems

Curtin University – Computing

Table of Contents

Introduction

The purpose of this assignment was to find solutions to the Producer-Consumer problem, where multiple threads and/or processes produce a shared resource for multiple other threads and/or processes to consume. Synchronization of access to these shared resources must be maintained. In this application of the Producer-Consumer problem, producers are represented by a Lift Request Server and consumers are represented by Lifts, which consume these requests served by the Lift Request Server. The requests are stored in a bounded shared buffer, which is represented as a FIFO queue of lift requests of a fixed size.

Mutual Exclusion and Multithreading

In the multithreading implementation of this assignment, the Pthreads API was used to implement both threading and mutual exclusion. The API provides thread creation, mutex locks and condition variables, all of which were used throughout the assignment.

Multithreading with Pthreads

Different threads of execution on a single process, known as a lightweight process, where used to simulate the Lift Request Server and multiple Lifts servicing those requests. In my implementation, Pthreads can be used to spawn multiple threads, execute some function/s, and then join the threads together again once the processing is complete.

To spawn threads, 4 thread IDs were made, which were then passed to several calls to the function `pthread_create()`, along with the runner functions and the respective parameters that each thread should execute on. These runner functions were `lift_R()` to represent the producer and `lift_consumer()` / `lift()`. Using threads to represent the producer consumer problem was beneficial, as threads can share memory from within the single process. Because of this, threads are more lightweight, as they do not need to perform costly operations to map shared memory between processes. This meant that a few key global variables could be used to coordinate resource access between threads in the source code found in `threads.c`. Pointers to C structs and Pthreads mutexes could be accessed between threads, as opposed to memory mapping these variables to shared memory for inter-process communication. The runner functions, `lift_R()` and `lift_consumer()` / `lift()` could then access the shared resources, namely the buffer that the producer would serve requests to and the consumer would consume requests from, and other structs that provided details based on the access to the buffer, such as total movement between consumer threads and the total number of requests being served to the buffer.

Mutual Exclusion with Pthreads

The Pthreads API also provides the ability to provide synchronization between threads, and even processes. This was important in this situation as access to the shared resources, namely the buffer, would need to be controlled between the threads. This is due to the fact that producing or consuming a request is a critical section problem, meaning that any thread performing any operation during this section should not be able to be interrupted whilst, preventing race conditions, satisfying the progress condition and preventing bounded waiting.

This is achieved with the Pthreads API using mutex locks and condition variables. One mutex lock and two condition variables were declared as global variables, meaning that all threads of execution had access to them. The condition variables were `canConsume`, which indicated that the buffer was not empty, and `canProduce`, which indicated that the buffer was not full.

For the producer thread, whilst there were still requests to be produced from the sim_input.txt file, the producer would obtain a mutex lock, and then using `pthread_cond_wait(canProduce),` it could wait for the buffer to have indexes available to push requests to. Once this condition was satisfied, the thread could then re-lock it's mutex, and enter it's critical section, where it would update the buffer with a new request. After this critical section, the producer thread can relinquish it's mutex lock, calling `pthread_mutex_unlock(),` and then signaling a condition variable with `pthread_cond_signal(canConsume)` to indicate to other waiting consumer threads that a request has been served to the buffer.

For the consumer threads, represented by lifts waiting to consume requests from the buffer, at the beginning of their runner function `lift()`, they enter their while loop and will wait for the producer signal `pthread_cond_signal(canConsume)` by using their wait command `pthread_cond_wait(canConsume)`, until it is signaled by the producer thread that a request is in the buffer, available to consume. Once a consumer exits its wait, it attains a mutex lock and enters the critical section, operates on the buffer and other shared memory variables, then relinquishes the mutex lock and signals the producer thread with `pthread_cond_signal(canProduce)`.

Using condition variables as opposed to semaphores in this situation prevents busy waiting.

Mutual Exclusion and Multiprocessing

In the second part of this assignment, multithreading was replaced by multiprocessing. Instead of multiple threads of execution operating as producers and consumers in a lightweight process, multiple heavy weight processes, that is processes with a single thread of execution, were used to represent producers and consumers.

Multiprocessing

By using UNIX systems calls such as `fork()` and `wait()`, multiple processes can be spawned during a program's execution to perform producer and consumer functions. Unlike threads, child processes do not share memory with their parent processes. This makes the task of sharing resources like the buffer more difficult than with threads, as threads have shared memory sections from the process that spawns them by default. This means immediately there is more overhead using multiple heavyweight processes as opposed to multithreaded lightweight processes because some form of inter-process communication must be established, such as shared memory segments or message passing.

Shared Memory and Multiprocessing

Individual processes can not automatically share memory, first, a region of shared memory must be established, with strict control on it's contents, as well as how many processes can operate on it at once. In the context of this assignment, the buffer, as well as information regarding the buffer needed to be stored in shared memory regions, so that producer and consumer processes could access and update this memory for other processes to see. The approach taken in this assignment was to use POSIX API for shared memory mapping, namely using the functions `shm_open()` to create a shared memory object, `ftruncate()` to configure the size of shared memory and `mmap()` to establish a memory mapped file containing the shared memory object.

These functions are used in the processes.c file, where child processes are spawned, shared memory mapped and consumer and producer functions executed. The several child processes could simply name the memory mapped file and have access to it, so each runner function, `lift_R()` and `lift()`, creates their own mappings to the respective segments of shared memory for structures like the buffer, mutexes and other variables for logging information about I/O on the buffer.

Mutual Exclusion with Pthreads Mutex Attributes

To achieve mutual exclusion during the critical sections of both the producer and consumer processes, the Pthreads API was used again to create mutex lock and condition variables `canConsume` and `canProduce.`

However, if these mutex locks and condition variables are initialized the same way as the thread implementation, the mutex and condition variables will only block on a single thread of execution. Because we are using multiple heavy weight processes, simple calls to the locking functions will only block a single process, rather than multiple. To solve this issue, the Pthreads API provides mutex and condition variable attributes that can be assigned to these objects at their initialization in the parent process, then placed in a shared memory segment for following child processes to access. This meant that a child process, consumer or producer, could access these mutex objects in shared memory and acquire locks or signal to other processes during execution.

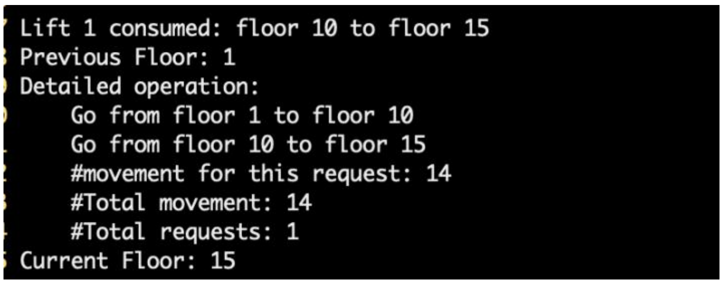
Sample I/O

sim_input.txt

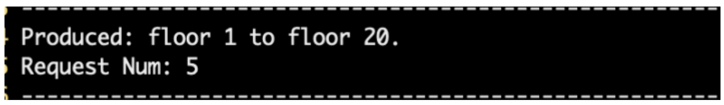
This text file is parsed by the program for requests. It contains a source floor and a destination floor, with between 50 and 100 requests in the format “<src> <dest>”. These values are stored as strings.

sim_out.txt

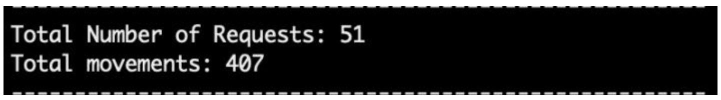
This file is generated during execution and appended to upon after every action taken on the buffer, either by the consumer or the producer. Each consumer keeps a track of some statistics that are updated when they fulfill a request, which are then output to this file. The producer also outputs its own data based on how many requests it has served. Once all requests have been fulfilled, a footer is appended to the end of the file which states the total requests that have been fulfilled and the total floors travelled by all of the lifts.



```
Lift 1 consumed: floor 10 to floor 15
Previous Floor: 1
Detailed operation:
    Go from floor 1 to floor 10
    Go from floor 10 to floor 15
    #movement for this request: 14
    #Total movement: 14
    #Total requests: 1
Current Floor: 15
```



```
-----------------------------------------------------
Produced: floor 1 to floor 20.
Request Num: 5
-----------------------------------------------------
```



```
-----------------------------------------------------
Total Number of Requests: 51
Total movements: 407
-----------------------------------------------------
```

Testing and other Thoughts

Running time

By using the UNIX command line argument `time` when executing both the multithreaded and the multiprocessing version of the program with a buffer of size 10 and a consumer sleeping time of one second, on average, both implementations took 17 seconds to execute.

Problems with shared memory and queues

When implementing the first part of this assignment, a FIFO queue using a linked list data structure was used to store requests. This was helpful as the buffer used only as much memory as there were items within it, as it dynamically grew and shrank in memory as producers made requests and consumers fulfilled them.

This became an issue however when trying to turn the multithreaded implementation into a multiprocessing one. This was because in shared memory, allocating memory using `malloc()` should be avoided, as the data stored in the pointed to addresses would be different between processes, as they do not share memory. This meant that when it came to the creating a queue, I instead opted for an array-based implementation of a queue. Instead of using pointers to point at the requests that are at the front/back of a linked list, I used indexes. If a consumer fulfilled a request, it would update what index that corresponded to the front of the queue.

Source Code

For readability, some print statements have been removed/shortened.

Lift_sim_A (multithreading)

README

```
# assignment
AUTHOR: DYLAN TRAVERS
PROGRAM: lift_sim_A (pthreads)
SEM 1 2020

An Operating Systems COMP200 assignment to design a Bounded-Buffer/Producer-Consumer
solution for multiple simulated elevators servicing different floors of a building. Written in C using pthreads.

usage:
~/./lift_sim_A <buffer_size> <sleep_time>

make commands:
~/make
   compile using gcc
~/make clean
   remove executable, .o files, output text file
~/make run
   run program with sample inputs of buffer size = 10 and sleep time = 1
~/make valgrind
   run with valgrind
~/make helgrind
   run with valgrind tool helgrind for race condiditons
```

main.c

```
/*
File: main.c
Author: Dylan Travers
ID: 19128043
Program: lift_sim

Purpose: runs the main method
```

```
********************************************************************************/
#include "main.h"

int main(int argc, char **argv)
{
    /* cmd line arguments for buffer size and sleep time */
    int m, t;
    Queue *requests = NULL;

    /* check that number of arguments == 3, otherwise provide usage info */
    if(argc == 3)
    {
        requests = inFileInit(requests);
    }
    /* usage information for running ./program */
    else
    {
        printf("Usage: %s <buffer size> <time for elevator to sleep", argv[0]);
        exit(-1);
    }

    /* get cmd line args */
    m = atoi(argv[1]);
    t = atoi(argv[2]);

    threadInit(requests, m, t);

    /* destroy request queue and all it's nodes */
    destroyQueue(requests);
}
```

Main.h

```
#ifndef MAIN_H
#define MAIN_H

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

```
#include "fileIO.h"
#include "queue.h"
#include "threads.h"

#endif
```

FileIO.c

```
/*
File: fileIO.c
Author: Dylan Travers
ID: 19128043
Program: lift_sim

Purpose: to handle all file io between other functions and sim_in and sim_out
        files and their respective values
*******************************************************************************/

#include "fileIO.h"

Queue *inFileInit(Queue *requests)
{
   /* file pointer for sim_in.txt */
   FILE* inFile;
   /* each individual line that is read in */
   char line[MAX_LINE_LENGTH];
   char* inFileName = "sim_input.txt";
   int lines, i, src, dest;

   if((inFile = fopen(inFileName, "r")) != NULL)
   {
      /* get lines in sim_in.txt */
      lines = getLines(inFile);
      rewind(inFile);
      requests = initQueue(lines);
      if(requests == NULL)
      {
         fprintf(stderr, "Could not init request queue.\n");
      }
```

```
    /* iterate through lines */
    if(lines >= 50 && lines <= 100)
    {

        for(i = 0; i < lines; i++)
        {
            if(fgets(line, MAX_LINE_LENGTH, inFile) != NULL)
            {
                /* get instructions from line */
                sscanf(line, "%d %d", &src, &dest);
                /* make sure instructions are within the floors 1-20 */
                if(src >= 1 && src <= 20)
                {
                    if(dest >= 1 && dest <= 20)
                    {
                        enqueue(requests, src, dest);

                    }
                    else
                    {
                        printf("ERROR: Found incorrect input”);
                    }
                }
                else
                {
                    printf("ERROR: Found incorrect input”);
                }
            }
            else
            {
                perror("ERROR: sim_input.txt incorrectly formatted\n");
            }
        }
        fclose(inFile);
    }
    else
    {
        perror("ERROR: sim_input.txt is not working\n");
    }
}
```

```
    else
    {
        perror("ERROR: number of lift requests must equal 50 > n > 100”);
    }
    return requests;
}

int getLines(FILE* file)
{
    char ch;
    int lines;
    lines = 0;

    for(ch = fgetc(file); !feof(file); ch = fgetc(file))
    {
        if(ch == '\n')
        {
            lines++;
        }
    }
    return lines;
}
```

FileIO.h

```
#ifndef FILEIO_H
#define FILEIO_H

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

#include "queue.h"

#define MAX_LINE_LENGTH 8
Queue *inFileInit(Queue *requests);

int getLines(FILE* file);
```

```
#endif
```

Queue.c

```
/*
File: queue.c
Author: Dylan Travers
ID: 19128043
Program: lift_sim

Purpose: implementation of a FIFO queue
*****************************************************************************/

#include "queue.h"

/* init the queue, pass a limit to the queue, and it will allocate memory to
the queue, ensures that the limit is not 0 and sets it to 101. Returns a pointer
to a queue data structure. */
Queue *initQueue(int limit)
{
    Queue *queue = (Queue*) malloc(sizeof (Queue));
    if (queue == NULL)
    {
        return NULL;
    }
    if (limit <= 0)
    {
        limit = 101;
    }

    queue->limit = limit;
    queue->size = 0;
    queue->head = NULL;
    queue->tail = NULL;

    return queue;
}

/* IMPORT: pointer to a Queue
```

```
   EXPORT: nothing
   PURPOSE: frees items within the queue and frees the queue. */

void destroyQueue(Queue *queue)
{
    while(queue->size != 0)
    {
        dequeue(queue);
    }
    free(queue);
}
/*
IMPORT: pointer to a queue and a pointer to a node item.
EXPORT: either true or false, depending if the enqueue was successful
PURPOSE: Add a provided node item to the queue of other node items, basic
         error checking is done to make sure that the item and queue is non NULL
         and that the queue hasn't exceeded it's node limit
*/
int enqueue(Queue *queue, int src, int dest)
{
    node *_node;

    if (queue == NULL)
    {
        /* bad data passed */
        printf("\n\nError: bad data passed to queue.\n\n");
        return FALSE;
    }
    if (queue->size >= queue->limit)
    {
        /* queue full */
        printf("\n\nError: tried to add data to full queue\n\n");
        return FALSE;

    }
    _node = (node*)malloc(sizeof(node));

    _node->source = src;
    _node->destination = dest;
    _node->prev = NULL;
    /* queue empty */
    if (queue->size == 0)
    {
        queue->head = _node;
        queue->tail = _node;
    }
    else
    {
        queue->tail->prev = _node;
        queue->tail = _node;
    }
    queue->size++;
    return TRUE;
}

/*
IMPORT: pointer to a queue
EXPORT: a node pointer to the dequeued object
PURPOSE: pop the first item off the queue and return it, basic error checking to
         make sure the queue isnt already empty. Makes the next item in line the
         new head.
*/
node *getRequest(Queue *queue)
{
    node *data;
    if (isEmpty(queue))
    {
        printf("ERROR: tried to get request on empty queue\n");
        return NULL;
    }
    else
    {
        data = queue->head;
        return data;
    }
}

void dequeue(Queue *queue)
{
    node* temp = queue->head;
    if(queue->head == NULL)
    {
        printf("ERROR: could not dequeue, queue is empty already...\n");
    }
    else
    {
        queue->head = temp->prev;
        queue->size--;
        free(temp);
    }
}

/*
IMPORT: pointer to a queue
EXPORT: either true or false, depending if the queue has items in it
PURPOSE: checks whether the queue has things in it.
*/
int isEmpty(Queue* queue)
{
    if (queue == NULL)
    {
        return FALSE;
    }
    if (queue->size == 0)
    {
        return TRUE;
    }
    if(queue->head == NULL)
    {
        return TRUE;
    }
    else
    {
        return FALSE;
    }
}
```

Queue.h

```
#ifndef QUEUE_H
#define QUEUE_H

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

#define TRUE 1
#define FALSE 0

typedef struct Node {
    int source;
    int destination;
    struct Node *prev;
} node;

typedef struct Queue {
    node *head;
    node *tail;
    int size;
    int limit;
} Queue;

Queue *initQueue(int limit);
void destroyQueue(Queue *queue);
int enqueue(Queue *queue, int src, int dest);
node *getRequest(Queue *queue);
void dequeue(Queue *queue);
int isEmpty(Queue* queue);

#endif
```

Threads.c

```
/*
*File: threads.c
*Author: Dylan travers
*ID: 19128043
*Program: lift_sim_A
```

```
*
*This file holds functions that init threads, run producer/conusmer threads,
*and destroys all at the end.
*******************************************************************************/

#include "threads.h"

/* mutex lock */
pthread_mutex_t bufferMutex = PTHREAD_MUTEX_INITIALIZER;

/* conditions */
pthread_cond_t canProduce = PTHREAD_COND_INITIALIZER;
pthread_cond_t canConsume = PTHREAD_COND_INITIALIZER;

/* buffer */
buffer_item *buffer;

/* totals */
totals_item *consumerTotals;

/* flag that represents that there are still things to consume */
int flag;

/* producer runner method for threads */
void* lift_R(void *param)
{
    Queue *allRequests = (Queue *)param;
    node *prodNode;
    int src, dest, totalRequests;
    totalRequests = 0;

    while (allRequests->size !=0)
    {


        /* lock mutex */
        /* wait condition whilst buffer is full */
        pthread_mutex_lock(&bufferMutex);
        while(buffer->bufferQ->size == buffer->bufferSize)
        {
            pthread_cond_wait(&canProduce, &bufferMutex);
        }

        /* remove a request from the total requests */
        prodNode = getRequest(allRequests);
        src = prodNode->source;
        dest = prodNode->destination;

        /* insert request into the buffer */
        enqueue(buffer->bufferQ, src, dest);
        /* increment the total requests counter */
        totalRequests++;
        producerLogger(src, dest, totalRequests);
        dequeue(allRequests);

        /* exit critical section */

        pthread_cond_signal(&canConsume);
        pthread_mutex_unlock(&bufferMutex);
    }
    return NULL;
}

void* lift_consumer(void *in_lift_item)
{
    lift_item *lift = (lift_item *) in_lift_item;
    node *consNode;
    int src, dest, movement;
    /* only consume whilst there are items left to consume */
    /* this is the only race condition I get in helgrind, fine though because
       the producer never alters the state of flag during it's lock. */
    while(flag == 0)
    {
        pthread_mutex_lock(&bufferMutex);
        /* 2nd statement prevents deadlock */
        while (buffer->bufferQ->size == 0 && flag == 0)
        {
            pthread_cond_wait(&canConsume, &bufferMutex);
        }
        /* check incase race condition occurs */
        if (flag == 0)
        {
            /* consume and update relevent data, write to file */
            consNode = getRequest(buffer->bufferQ);
            src = consNode->source;
            dest = consNode->destination;
            movement = abs((lift->prev - src) + (src - dest));
            lift->requestTotal++;
            lift->moveTotal += movement;
            consumerTotals->moves += movement;
            consumerTotals->requests++;
            consumerLogger(src, dest, movement, lift->prev, lift->liftName);
            lift->prev = dest;
            dequeue(buffer->bufferQ);
        }
        /* is there anything lef to consume? No? let other consumers know */
        if (consumerTotals->requests == consumerTotals->allRequests)
        {
            flag = 1;
        }

        pthread_cond_signal(&canProduce);
        pthread_mutex_unlock(&bufferMutex);
        sleep(buffer->sleepTime);
    }
    return NULL;
}


void threadInit(Queue* allRequests, int bufferSize, int sleepTime)
{
    flag = 0;
    pthread_t liftAID, liftBID, liftCID, liftRID;
    lift_item liftAItem, liftBItem, liftCItem;

    /* init the shared buffer */
    initBuffer(bufferSize, sleepTime);
    /* init each lift struct */
    initLifts(&liftAItem, 'A');
    initLifts(&liftBItem, 'B');
    initLifts(&liftCItem, 'C');
    /* init the consumerTotals struct */
    initTotals(allRequests->size);

    /* create all threads for lift request producer and the three lift consumers */
    pthread_create(&liftRID, NULL, lift_R, allRequests);
    pthread_create(&liftAID, NULL, lift_consumer, (void*)&liftAItem);
    pthread_create(&liftBID, NULL, lift_consumer, (void*)&liftBItem);
    pthread_create(&liftCID, NULL, lift_consumer, (void*)&liftCItem);

    /* join all and destroy everything */
    pthread_join(liftRID, NULL);
    pthread_join(liftAID, NULL);
    pthread_join(liftBID, NULL);
    pthread_join(liftCID, NULL);

    totalLogger(consumerTotals->requests, consumerTotals->moves);
    pthread_mutex_destroy(&bufferMutex);
    pthread_cond_destroy(&canConsume);
    pthread_cond_destroy(&canProduce);

    destroyQueue(buffer->bufferQ);
    free(consumerTotals);
    free(buffer);
}

void initBuffer(int _bufferSize, int _sleepTime)
{
    /* initialise buffer */
    buffer = (buffer_item*)malloc(sizeof(buffer_item));
    buffer->bufferSize = _bufferSize;
    buffer->bufferQ = initQueue(_bufferSize);
    buffer->sleepTime = _sleepTime;
}

void initLifts(lift_item *lift, char _liftName)
{
    /* initialise struct fields */
    lift->liftName = _liftName;
    lift->prev = 1;
    lift->moveTotal = 0;
    lift->requestTotal = 0;
}

/* init the totals */
void initTotals(int size)
{
    consumerTotals = (totals_item*)malloc(sizeof(totals_item));
    consumerTotals->requests = 0;
    consumerTotals->moves = 0;
    consumerTotals->allRequests = size;
}

/* debug method for displaying info to console, not file */
void displayInfo(int src, int dest, int prev, int movement, char name)
{
    printf("---\nLift %c consumed: floor %d to floor %d\n", name, src, dest);
    printf("Previous Floor: %d\n", prev);
    printf("Detailed operation:\n\tGo from floor %d to floor %d\n\t", prev, src);
    printf("Go from floor %d to floor %d\n\t", src, dest);
    printf("#movement for this request: %d\n\t", movement);
    printf("#Total movement: %d\n\t", consumerTotals->moves);
    printf("#Total requests: %d\n", consumerTotals->requests);
    printf("Current Floor: %d\n", dest);
}

/* logs lift/consumer information */
void consumerLogger(int src, int dest, int movement, int prev, char name)
{
    FILE* logfile;

    if((logfile = fopen("sim_out.txt", "a")) != NULL)
    {
        fprintf(logfile, "------------------------------------------------------\n");
        fprintf(logfile, "Lift %c consumed: floor %d to floor %d\n", name, src, dest);
        fprintf(logfile, "Previous Floor: %d\n", prev);
        fprintf(logfile, "Detailed operation:\n\tGo from floor %d to floor %d\n\t", prev, src);
        fprintf(logfile, "Go from floor %d to floor %d\n\t", src, dest);
        fprintf(logfile, "#movement for this request: %d\n\t", movement);
        fprintf(logfile, "#Total movement: %d\n\t", consumerTotals->moves);
        fprintf(logfile, "#Total requests: %d\n", consumerTotals->requests);
        fprintf(logfile, "Current Floor: %d\n", dest);
    }
    else
    {
        perror("ERROR: couldnt open output file");
    }
    fclose(logfile);
}

/* log the total requests and moves at then end of file */
void totalLogger(int requests, int moves)
{
    FILE* logfile;

    if((logfile = fopen("sim_out.txt", "a")) != NULL)
    {
        fprintf(logfile, "-------------------------------------------------------\n");
        fprintf(logfile, "Total Number of Requests: %d\nTotal movements: %d\n", requests, moves);
        fprintf(logfile, "-------------------------------------------------------\n");
    }
    else
    {
        perror("ERROR: couldnt open output file");
    }
    fclose(logfile);
}

void producerLogger(int src, int dest, int totalRequests)
{
    FILE* logfile;

    if((logfile = fopen("sim_out.txt", "a")) != NULL)
    {
        fprintf(logfile, "-------------------------------------------------------\n");
        fprintf(logfile, "Produced: floor %d to floor %d.\nRequest Num: %d\n", src, dest, totalRequests);
    }
    else
    {
        perror("ERROR: couldnt open output file");
    }
    fclose(logfile);
}
```

Threads.h

```
#ifndef THREADS_H
#define THREADS_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>

#include "queue.h"

typedef struct _totals_item {
    int moves;
    int requests;
    int allRequests;
} totals_item;

typedef struct _buffer_item {
    int bufferSize;
    Queue *bufferQ;
    int sleepTime;
} buffer_item;

typedef struct _lift_item {
    char liftName;
    int moveTotal;
    int requestTotal;
    int prev;
} lift_item;


void* lift_R(void *param);
```

```
void* lift_consumer(void *in_lift_item);

void threadInit(Queue* allRequests, int bufferSize, int sleepTime);

void initBuffer(int _bufferSize, int _sleepTime);

void initLifts(lift_item *lift, char _liftName);

void initTotals(int size);

void displayInfo(int src, int dest, int prev, int movement, char name);

void consumerLogger(int src, int dest, int movement, int prev, char name);

void totalLogger(int requests, int moves);

void producerLogger(int src, int dest, int totalRequests);

#endif
```

Lift_sim_B (multiprocessesing)

README

```
# assignmentProcesses
Student: Dylan Travers
ID: 19128043
Assignment: OS 2020 Sem 1, part B
PROGRAM: lift_sim_B (processes)

An Operating Systems COMP200 assignment to design a Bounded-Buffer/Producer-Consumer
solution for multiple simulated elevators servicing different floors of a building. Written in C.


This directory contains source code, Makefile and sample input of 51 lift requests.

ATTENTION: may require the inclusion of the linker flag -lrt on some systems, may cause errors on others.
Add/remove it if necessary in the Makefile
```

to compile:

```
    ~/assignmentProcesses$ make
```

to run with sample input buffer size = 5 and sleep time = 5:

```
    ~/assignmentProcesses$ make run
```

to run with valgrind (im not sure how well this works with multiprocessing) and sample cmd args:

```
    ~/assignmentProcesses$ make valgrind
```

to run with vaglrind tool helgrind (im not sure how well it handles pthread condition variables and multiple processes)

```
    ~/assignmentProcesses make helgrind
```

Main.h

```c
/*
File: main.c
Author: Dylan Travers
ID: 19128043
Program: lift_sim

Purpose: runs the main method, error checks cmd line args
*******************************************************************************/
#include "main.h"

int main(int argc, char **argv)
{
    /* cmd line arguements for buffer size and sleep time */
    int m, t, allReqBufFd;
    buffer_item* allReqBuf;
    const char* reqBufName = "OS_REQUESTS";

    /* check that number of arguements == 3, otherwise provide usage info */
    if(argc == 3)
    {
        m = atoi(argv[1]);
        t = atoi(argv[2]);
    }
    /* usage information for running ./program */
    else
    {
        printf("Usage: %s <buffer size> <time for elevator to sleep", argv[0]);
        exit(-1);
    }

    allReqBufFd = shm_open(reqBufName, O_CREAT | O_RDWR, 0666);
    if(allReqBufFd < 0)
    {
        perror("allReqBuf shmopen error\n");
    }
    ftruncate(allReqBufFd, sizeof(buffer_item));
    allReqBuf = mmap(NULL, sizeof(buffer_item), PROT_READ | PROT_WRITE, MAP_SHARED, allReqBufFd, 0);

    allReqBuf = inFileInit(allReqBuf);

    processInit(m, t);



    munmap(allReqBuf, sizeof(buffer_item));
    close(allReqBufFd);
    shm_unlink(reqBufName);
}
```

Main.h

```
#ifndef MAIN_H
#define MAIN_H

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

#include "processes.h"
#include "fileIO.h"


#endif
```

fileIO.c

```
/*
File: fileIO.c
Author: Dylan Travers
ID: 19128043
Program: lift_sim

Purpose: to handle all file io between other functions and sim_in and sim_out
        files and their respective values
*******************************************************************************/

#include "fileIO.h"

buffer_item *inFileInit(buffer_item *buf)
{
    /* file pointer for sim_in.txt */
    FILE* inFile;
    /* each individual line that is read in */
    char line[MAX_LINE_LENGTH];
    char* inFileName = "sim_input.txt";
    int lines, i, src, dest;
    request_item request;


    if((inFile = fopen(inFileName, "r")) != NULL)
    {
        /* get lines in sim_in.txt */
        lines = getLines(inFile);
        rewind(inFile);
        /* iterate through lines */
        if(lines >= 50 && lines <= 100)
        {
            buf->bufSize = lines;
            buf->front = buf->back = 0;

            for(i = 0; i < lines; i++)
            {
                if(fgets(line, MAX_LINE_LENGTH, inFile) != NULL)
                {
                    /* get instructions from line */
                    sscanf(line, "%d %d", &src, &dest);
                    /* make sure instructions are within the floors 1-20 */
                    if(src >= 1 && src <= 20)
                    {
                        if(dest >= 1 && dest <= 20)
                        {
                            /* add request to buffer */
                            request.src = src;
                            request.dest = dest;
                            buf->buffer[buf->back] = request;
                            buf->back = (buf->back + 1) % buf->bufSize;
                        }
                        else
                        {
                            printf("ERROR: Found incorrect input in %s, %d is an invalid floor. Line: %d\n", inFileName,
dest, i+1);
                        }
                    }
                    else
                    {
                        printf("ERROR: Found incorrect input in %s, %d is an invalid floor. Line: %d\n", inFileName, src,
i+1);
                    }
                }
                else
                {
                    perror("ERROR: sim_input.txt incorrectly formatted\n");
                }
            }
            fclose(inFile);
        }
        else
        {
            perror("ERROR: sim_input.txt is not working\n");
        }
    }
    else
    {
        perror("ERROR: Lift requests in sim_input.txt is out side of range 50 <= number of requests <= 100\n");
    }
    return buf;
}

int getLines(FILE* file)
{
    char ch;
    int lines;
    lines = 0;

    for(ch = fgetc(file); !feof(file); ch = fgetc(file))
    {
        if(ch == '\n')
        {
            lines++;
        }
    }
    return lines;
}

void consumerLogger(int src, int dest, int movement, lift_item* lift, vars_item* vars)
{
    FILE* logfile;

    if((logfile = fopen("sim_out.txt", "a")) != NULL)
    {
        fprintf(logfile, "--------------------------------------------------------\n");
        fprintf(logfile, "Lift %d consumed: floor %d to floor %d\n", lift->liftName, src, dest);
        fprintf(logfile, "Previous Floor: %d\n", lift->prev);
        fprintf(logfile, "Detailed operation:\n\tGo from floor %d to floor %d\n\t", lift->prev, src);
        fprintf(logfile, "Go from floor %d to floor %d\n\t", src, dest);
        fprintf(logfile, "#movement for this request: %d\n\t", movement);
        fprintf(logfile, "#Total movement: %d\n\t", vars->totalMoves);
        fprintf(logfile, "#Total requests: %d\n", vars->requestsConsumed);
        fprintf(logfile, "Current Floor: %d\n", dest);
    }
    else
    {
        perror("ERROR: couldnt open output file");
    }
    fclose(logfile);
}

void totalLogger(int requests, int moves)
{
    FILE* logfile;

    if((logfile = fopen("sim_out.txt", "a")) != NULL)
    {
        fprintf(logfile, "-------------------------------------------------------\n");
        fprintf(logfile, "Total Number of Requests: %d\nTotal movements: %d\n", requests, moves);
        fprintf(logfile, "-------------------------------------------------------\n");
    }
    else
    {
        perror("ERROR: couldnt open output file");
    }
    fclose(logfile);
}

void producerLogger(int src, int dest, int totalRequests)
{
    FILE* logfile;

    if((logfile = fopen("sim_out.txt", "a")) != NULL)
    {
        fprintf(logfile, "-------------------------------------------------------\n");
        fprintf(logfile, "Produced: floor %d to floor %d.\nRequest Num: %d\n", src, dest, totalRequests);
    }
    else
    {
        perror("ERROR: couldnt open output file");
    }
    fclose(logfile);
}
```

fileIO.h

```
#ifndef FILEIO_H
```

```
#define FILEIO_H

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

#include "processes.h"

#define MAX_LINE_LENGTH 8

buffer_item *inFileInit(buffer_item *buf);
int getLines(FILE* file);
void consumerLogger(int src, int dest, int movement, lift_item* lift, vars_item* vars);
void totalLogger(int requests, int moves);
void producerLogger(int src, int dest, int totalRequests);

#endif
```

Processes.c

```
/*
File: processes.c
Author: Dylan Travers
ID: 19128043
Program: lift_sim

Purpose: init all shared memory blocks, fork all processes, clean up.
*********************************************************************************/


#include "processes.h"

/* is passed the cmd line args, init all shared memory for processes */
void processInit(int m, int t)
{
    int varsFd, bufFd, i;
    vars_item* vars = NULL;
    buffer_item* buf = NULL;
    const char *varsName = "OS_VARS";
    const char *bufName = "OS_BUF";
    pthread_condattr_t condattr;
    pthread_mutexattr_t mtxattr;



    /* initialize the buffer control variables */
    varsFd = shm_open(varsName, O_CREAT | O_RDWR, 0666);
    if (varsFd < 0)
    {
        perror("vars shmopen error\n");
    }
    ftruncate(varsFd, sizeof(vars_item));
    pthread_mutexattr_init(&mtxattr);
    pthread_mutexattr_setpshared(&mtxattr, PTHREAD_PROCESS_SHARED);
    pthread_condattr_init(&condattr);
    pthread_condattr_setpshared(&condattr, PTHREAD_PROCESS_SHARED);

    vars = mmap(0, sizeof(vars_item), PROT_READ | PROT_WRITE, MAP_SHARED, varsFd, 0);
    vars->requestsInBuffer = 0;
    vars->requestsConsumed = 0;
    vars->requestsProduced = 0;
    vars->totalMoves = 0;
    vars->totalRequests = 0;
    vars->flag = 0;
    vars->sleepTime = t;
    pthread_mutex_init(&vars->mutex, &mtxattr);
    pthread_cond_init(&vars->canProduce, &condattr);
    pthread_cond_init(&vars->canConsume, &condattr);

    /* initialize the buffer */
    bufFd = shm_open(bufName, O_CREAT | O_RDWR, 0666);
    if (bufFd < 0)
    {
        perror("buf shmopen error\n");
    }
    ftruncate(bufFd, sizeof(buffer_item));
    buf = mmap(0, sizeof(buffer_item), PROT_READ | PROT_WRITE, MAP_SHARED, bufFd, 0);
    buf->front = 0;
    buf->back = 0;
```

```
    buf->bufSize = m;

    for (i = 0; i < 4; i++)
    {
        int pid = fork();

        if (pid == 0 && i == 0)
            lift_R();
        else if (pid == 0)
            lift(i);
        /*else pid == 0, which implies the parent process --
        it loops around to start the next child.*/
    }
    for (i = 0; i < 4; i++)
    {
        wait(NULL);
    }

    /* log totals in out file */
    totalLogger(vars->requestsConsumed, vars->totalMoves);


    /* clean up all mutexes, condition variables, mutex attributes and shared mem */
    pthread_mutex_destroy(&vars->mutex);
    pthread_cond_destroy(&vars->canProduce);
    pthread_cond_destroy(&vars->canConsume);

    pthread_mutexattr_destroy(&mtxattr);
    pthread_condattr_destroy(&condattr);

    munmap(vars, sizeof(vars_item));
    munmap(buf, sizeof(buffer_item));

    close(varsFd);
    close(bufFd);

    shm_unlink(varsName);
    shm_unlink(bufName);
}
```

```
/*
 LIFT REQUEST PRODUCER
 */
void lift_R()
{
    int i, varsFd, bufFd, allReqBufFd;
    vars_item* vars;
    buffer_item* buffer;
    buffer_item* allReqBuf;
    request_item request;

    /* all requests buffer */
    allReqBufFd = shm_open("OS_REQUESTS", O_RDWR, 0666);
    allReqBuf = mmap(NULL, sizeof(buffer_item), PROT_READ | PROT_WRITE, MAP_SHARED, allReqBufFd, 0);

    /* buffer variables */
    varsFd = shm_open("OS_VARS", O_RDWR, 0666);
    vars = mmap(NULL, sizeof(vars_item), PROT_READ | PROT_WRITE, MAP_SHARED, varsFd, 0);

    /* the bounded buffer itself */
    bufFd = shm_open("OS_BUF", O_RDWR, 0666);
    buffer = mmap(NULL, sizeof(buffer_item), PROT_READ | PROT_WRITE, MAP_SHARED, bufFd, 0);

    vars->totalRequests = allReqBuf->bufSize;
    printf("Producer starting\n");

    /*begin*/
    for (i = 0; i < allReqBuf->bufSize; i++)
    {
        /* get next request */
        request = allReqBuf->buffer[allReqBuf->front];
        allReqBuf->front = (allReqBuf->front + 1) % allReqBuf->bufSize;

        /* critical section */
        pthread_mutex_lock(&vars->mutex);
        while(vars->requestsInBuffer == buffer->bufSize)
        {
            pthread_cond_wait(&vars->canProduce, &vars->mutex);
        }
        /* lock attained */
        buffer->buffer[buffer->back] = request;
        buffer->back = (buffer->back + 1) % buffer->bufSize;
        vars->requestsInBuffer++;
        vars->requestsProduced++;

        producerLogger(request.src, request.dest, vars->requestsProduced);

        /* end critical section */
        pthread_cond_signal(&vars->canConsume);
        pthread_mutex_unlock(&vars->mutex);
    }
    printf("producer exiting\n");

    /*clean up*/
    munmap(vars, sizeof(vars_item));
    munmap(buffer, sizeof(buffer_item));
    munmap(allReqBuf, sizeof(buffer_item));

    close(varsFd);
    close(bufFd);
    close(allReqBufFd);
    exit(0);
}
/*
 LIFT CONSUMERS
 */
void lift(int id)
{
    int varsFd, bufFd, movement;
    vars_item* vars;
    buffer_item* buffer;
    request_item request;
    lift_item* lift;

    /* init a lift struct */
    lift = (lift_item*)malloc(sizeof(lift_item));
    lift->liftName = id;
    lift->prev = 1;
    lift->moveTotal = 0;
```

```
lift->requestTotal = 0;

/* buffer varaibles */
varsFd = shm_open("OS_VARS", O_RDWR, 0666);
vars = mmap(0, sizeof(vars_item), PROT_READ | PROT_WRITE, MAP_SHARED, varsFd, 0);

/* the bounded buffer */
bufFd = shm_open("OS_BUF", O_RDWR, 0666);
buffer = mmap(0, sizeof(buffer_item), PROT_READ | PROT_WRITE, MAP_SHARED, bufFd, 0);

printf("\nLift %d starting\n", id);

/*begin*/
while(vars->flag == 0)
{
    /* critical section*/
    pthread_mutex_lock(&vars->mutex);
    while((vars->requestsInBuffer == 0) && (vars->flag ==0))
    {
        pthread_cond_wait(&vars->canConsume, &vars->mutex);
    }

    /* lock attained, check flag to see if there are any requests left */
    if(vars->flag == 0)
    {
        request = buffer->buffer[buffer->front];
        buffer->front = (buffer->front + 1) % buffer->bufSize;
        vars->requestsInBuffer--;
        vars->requestsConsumed++;
        lift->requestTotal++;
        movement = abs((lift->prev - request.src) + (request.src - request.dest));
        lift->moveTotal += movement;
        vars->totalMoves += movement;

        consumerLogger(request.src, request.dest, movement, lift, vars);

        lift->prev = request.dest;

    }
    /* if that was the last request, flip flag to let other processes know */
```

```
        if (vars->requestsConsumed == vars->totalRequests)
        {
            vars->flag = 1;
        }
        /* end critical section */
        pthread_cond_signal(&vars->canProduce);
        pthread_mutex_unlock(&vars->mutex);
        sleep(vars->sleepTime);
    }

    printf("\nLift %d exiting\n", id);
    /* clean up */
    munmap(vars, sizeof(vars_item));
    munmap(buffer, sizeof(buffer_item));

    close(varsFd);
    close(bufFd);
    free(lift);
    exit(0);
}
```

Processes.h

```
#ifndef PROCESSES_H
#define PROCESSES_H

#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <pthread.h>
#include <sys/mman.h>
#include <fcntl.h>

typedef struct _request_item {
    int src;
    int dest;
} request_item;

typedef struct _buffer_item {
    int front;
    int back;
    int bufSize;
    request_item buffer[100];
} buffer_item;

typedef struct _lift_item {
    int liftName;
    int moveTotal;
    int requestTotal;
    int prev;
} lift_item;

typedef struct _vars_item {
    int requestsInBuffer;
    int requestsConsumed;
    int requestsProduced;
    int totalRequests;
    int totalMoves;
    int flag;
    int sleepTime;
    pthread_mutex_t mutex;
    pthread_cond_t canProduce;
    pthread_cond_t canConsume;
} vars_item;

#include "fileIO.h"

void processInit(int m, int t);
void lift_R();
void lift(int id);

#endif
```