ULTIMA 2.0

CS-435 Operating System

Phase # : 1

By:

Yazan Faham: yfaham@iu.edu

Joseph Venanzio: josekama@iu.edu

Date: 2/24/2019

# Phase Abstract:

The purpose of this phase implementation was to build some of the necessary operating system components. We utilized the functions of the semaphore and scheduler. The semaphore was used in order to gain control of resources while the scheduler was used to make sure the process gets enough time to finish its task. Once we discovered the right implementation of the semapore and scheduler functions, we used pthreads on our window to see the actions of the semaphore and scheduler. The results produced from combining everything gave us the same windows, where each thread printed on the appropriate screen window until it was done, then proceeded forward to the next window.

# Table Of Contents:

[**Phase Abstract:**](#_i8ts7q73gsv) **2**

[**Table Of Contents:**](#_39zl451jedw6) **2**

[**Phase Description:**](#_35wba4p6n8og) **3**

[**Design Diagrams:**](#_w9iv24t8cbj0) **4**

[Class Diagram:](#_8ian7q8p54pi) 4

[**Source Code:**](#_8oh0ctug0hf) **5**

[Main.cpp](#_lkutoj194bss) 5

[Queue.h](#_qboa956vglp6) 10

[MyScheduler.cpp](#_n3ed8429kqz6) 11

[MyScheduler.h](#_5uhowfti9gb1) 14

[MySemaphore.cpp](#_q9u7vloqci8) 15

[MySemaphore.h](#_ekaf6qf7vhjw) 18

[MyVector.h](#_7y7330byd0or) 19

[MyNode.h](#_pe1oo4fexigc) 23

[**Output Test Plan**](#_ecw5a9wuwyb2) **25**

[System Functionality](#_354mm0tprmjm) 27

# Phase Description:

This phase is aimed to construct the classes Scheduler and Semaphore to manage the threads that are running in the background and using shared resources in order to prevent them from giving unexpected results. In this phase we merged our classes with lab2 part2 to run and manage all the windows through their appropriate threads.

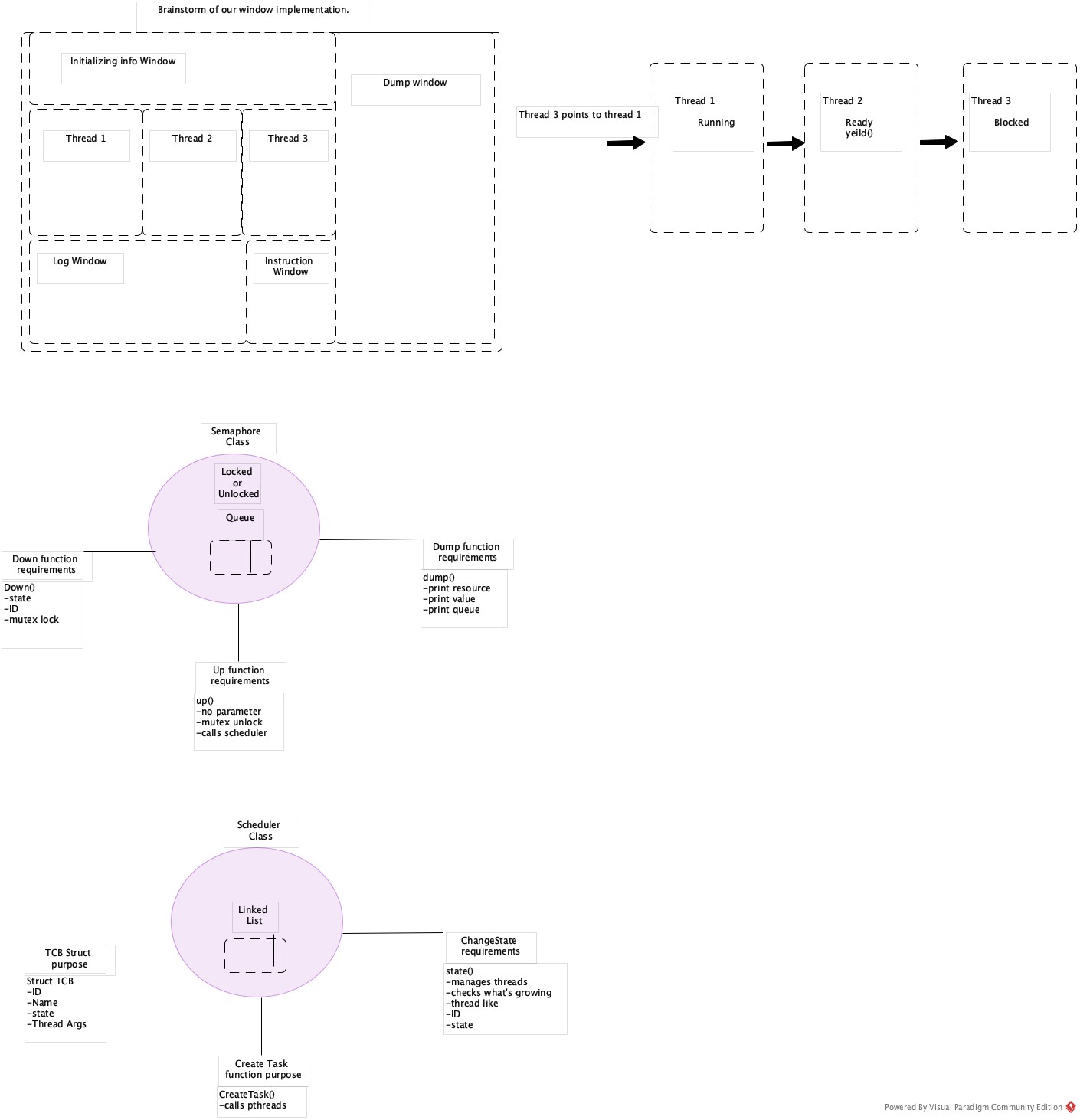
Our Semaphore, which manages mutual exclusion, holds a pointer to a Scheduler object, which manages the running threads. The Semaphore has a Queue that holds the TID of the thread waiting to access the shared resource. The Scheduler has a Vector (linked list) that holds information about running threads (TCB).

Some of the issues include figuring out how to appropriately pass the desired function for the Scheduler object to run it as a thread through a call to pthread\_create(). We also struggled with managing the threads’ access to the shared resource in the scheduler as well as in down() and up(), which we discussed many times with other peer groups.

# Design Diagrams:

## Class Diagram:

# 

****

# 

# Source Code:

## Main.cpp

#include <ncurses.h>

#include <unistd.h>

#include <stdarg.h>

#include <termios.h>

#include <fcntl.h>

#include <cstring>

#include "MyScheduler.h"

#include "MySemaphore.h"

using namespace std;

MyScheduler sched;

MySemaphore sema("Screen Sema");

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Purpose: Creates window

//Input : height, width, starty, startx

//Output : none

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

WINDOW \*create\_window(int height, int width, int starty, int startx);

void write\_window(WINDOW \*win, const char \*text);

void write\_window(WINDOW \*win, int x, int y, const char \*text);

void display\_help(WINDOW \*win);

void\* run\_task\_window(void \*args);

void\* run\_console\_window(void \*args);

int main() {

sema.set\_sched\_ptr(&sched);

pthread\_t t1, t2, t3, console\_t;

initscr();

int y, x;

int max\_x, max\_y;

getmaxyx(stdscr, max\_y, max\_x);

wprintw(stdscr, "Screen height = %d, Screen width = %d\n", max\_y, max\_x);

getyx(stdscr, y, x);

wprintw(stdscr, "Currecnt y = %d, Current x = %d\n", y, x);

sleep(1);

refresh();

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Purpose: Creates window with positions and heights and width, printing the

// running threads

//Input : height, width, starty, startx

//Output : none

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

WINDOW \*heading\_win = newwin(12, 80, 3, 2);

box(heading\_win, 0, 0);

mvwprintw(heading\_win, 1, 2, "ULTIMA 2.0 (Spring 2019)");

mvwprintw(heading\_win, 2, 2, "Starting Thread 1.....");

mvwprintw(heading\_win, 3, 2, "Starting Thread 2.....");

mvwprintw(heading\_win, 4, 2, "Starting Thread 3.....");

mvwprintw(heading\_win, 5, 2, "Press ctrl-c to exit the program....");

wrefresh(heading\_win);

WINDOW \*t1\_win = create\_window(15, 25, 15, 2);

WINDOW \*t2\_win = create\_window(15, 25, 15, 30);

WINDOW \*t3\_win = create\_window(15, 25, 15, 57);

WINDOW \*log\_win = create\_window(10, 60, 30, 2);

WINDOW \*console\_win = create\_window(10, 20, 30, 62);

WINDOW \*d\_window = create\_window(37, 60, 3, 82);

write\_window(d\_window, 1, 1, "........dump() info........");

write\_window(log\_win, 1, 5, "....log....\n");

write\_window(console\_win, 1, 1, "....console....\n");

write\_window(console\_win, 2, 1, "ULTIMA #");

sema.set\_dump\_win(d\_window);

sched.setLogWindow(log\_win);

cbreak();

noecho();

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Purpose: Makes thread 1,2 and 3 start running then blocks the execution for a

// milisecond certain time. Afterwards function suspends execution of the

// calling thread until the target thread terminates, unless the target thread

// has already terminated.

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

sched.create\_task(&t1, run\_task\_window, t1\_win, "Thread 1", MyScheduler::RUNNING);

sleep(1);

sched.create\_task(&t2, run\_task\_window, t2\_win, "Thread 2", MyScheduler::RUNNING);

sleep(1);

sched.create\_task(&t3, run\_task\_window, t3\_win, "Thread 3", MyScheduler::RUNNING);

sleep(1);

sched.create\_task(&console\_t, run\_console\_window, console\_win, "Console", MyScheduler::RUNNING);

pthread\_join(t1, NULL);

pthread\_join(t2, NULL);

pthread\_join(t3, NULL);

pthread\_join(console\_t, NULL);

endwin();

return(0);

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Purpose: Creates window with attributes that makes the window scrollable.

//Input : height, width, starty, startx

//Output : win

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

WINDOW \*create\_window(int height, int width, int starty, int startx) {

WINDOW \*win = newwin(height, width, starty, startx);

scrollok(win, true);

scroll(win);

box(win, 0, 0);

wrefresh(win);

return win;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Purpose: Writes a text to a window and draws a box followed by refreshing the window.

//Input : win, text

//Output : none

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void write\_window(WINDOW \*win, const char \*text) {

wprintw(win, text);

box(win, 0, 0);

wrefresh(win);

}

void write\_window(WINDOW \*win, int y, int x, const char \*text) {

mvwprintw(win, y, x, text);

box(win, 0, 0);

wrefresh(win);

}

void display\_help(WINDOW \*win) {

wclear(win);

write\_window(win, 1, 1, "...help...");

write\_window(win, 2, 1, "1= run 1");

write\_window(win, 3, 1, "2= run 2");

write\_window(win, 4, 1, "3= run 3");

write\_window(win, 5, 1, "c= clear screen");

write\_window(win, 6, 1, "h= help screen");

write\_window(win, 7, 1, "d= dump");

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Purpose: prints to the window the runnings tasks by

//Input : \*args

//Output : none

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void\* run\_task\_window(void \*args) {

MyScheduler::TCB\* process = (MyScheduler::TCB\*)args;

int tid = process->tid;

char name[64];

strcpy(name, process->tname);

WINDOW\* win = process->win\_ptr;

WINDOW\* log\_win = process->log\_win\_ptr;

char buff[256];

sprintf(buff, " Starting %s....\n", name);

sema.down(tid);

write\_window(win, buff);

sema.up();

int CPU\_Quantum = 0;

while (true) {

if (process->state == MyScheduler::RUNNING) {

sprintf(buff, " Task %d running #%d\n", tid, CPU\_Quantum++);

sema.down(tid);

write\_window(win, buff);

write\_window(log\_win, buff);

sema.up();

sched.run\_next(tid);

}

else if (process->state == MyScheduler::DEAD)

break;

else

pthread\_yield();

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Purpose: runs window by providing functionality for the console window by reading

// the input and writing the corresponding output.

//Input : height, width, starty, startx

//Output : none

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void\* run\_console\_window(void \*args) {

MyScheduler::TCB\* process = (MyScheduler::TCB\*)args;

int tid = process->tid;

char name[64];

strcpy(name, process->tname);

WINDOW\* win = process->win\_ptr;

WINDOW\* log\_win = process->log\_win\_ptr;

nodelay(win, true);

char buf[256];

int input = ERR;

while (true) {

if (process->state == MyScheduler::RUNNING) {

sema.down(tid);

input = wgetch(win);

switch (input) {

case '1':

case '2':

case '3':

sprintf(buf, "%c\n", input);

write\_window(win, buf);

sprintf(buf, " You typed = %c\n", input);

write\_window(win, buf);

write\_window(log\_win, buf);

break;

case 'c':

wclear(win);

write\_window(win, 1, 1, "ULTIMA #");

break;

case 'h':

display\_help(win);

write\_window(win, 8, 1, "ULTIMA #");

break;

case 'd':

sema.dump(4);

wclear(win);

write\_window(win, 1, 1, "ULTIMA #\n press R to resume");

nodelay(win, false);

do {

input = wgetch(win);

} while (input != 'r');

nodelay(win, true);

sema.un\_dump();

break;

case ERR:

break;

default:

sprintf(buf, " %c\n", input);

write\_window(win, buf);

write\_window(win, " -Invalid command\n");

write\_window(log\_win, buf);

write\_window(log\_win, " -Invalid command\n");

write\_window(win, " ULTIMA #");

break;

}

sema.up();

sched.run\_next(tid);

}

else if (process->state == MyScheduler::DEAD)

break;

else

pthread\_yield();

}

}

## Queue.h

#ifndef MYQUEUE\_H

#define MYQUEUE\_H

#include <iostream>

using namespace std;

template <typename T>

class MyQueue {

public:

struct node {

T data;

node \*next;

};

node \*head, \*tail;

MyQueue() {

head = NULL;

tail = NULL;

length = 0;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Purpose: inserts newData in the queue

//Input : newData

//Output : none

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void enqueue(T newData) {

if (length == 0) {

tail = new node;

tail->next = NULL;

tail->data = newData;

head = tail;

}

else if (length > 0) {

tail->next = new node;

tail = tail->next;

tail->data = newData;

}

length++;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Purpose: removes data from the queue

//Input : none

//Output : returns a pointer to the data that was at the head.

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

T\* dequeue() {

if (length == 0) {

return NULL;

}

else if (length == 1) {

tail = NULL;

}

node \*temp = head;

head = head->next;

temp->next = NULL;

length--;

return &(temp->data);

}

T\* front() {

return &(head->data);

}

int getLength() {

return length;

}

bool isEmpty() {

return length == 0;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Purpose: removes all the data in the queue as long the length is bigger than 0.

//Input : none

//Output : none

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void clear() {

while (length > 0)

dequeue();

}

~MyQueue() {

clear();

}

private:

int length;

};

#endif

## MyScheduler.cpp

#define \_OPEN\_THREADS

#define \_GNU\_SOURCE

#include "MyScheduler.h"

#include <iostream>

#include <pthread.h>

#include <assert.h>

#include <time.h>

#include <unistd.h>

#include <ncurses.h>

#include <stdarg.h>

#include <termios.h>

#include <fcntl.h>

#include <cstdlib>

#include <cstring>

using namespace std;

MyScheduler::MyScheduler() {

next\_tid = 0;

}

void MyScheduler::setDumpWindow(WINDOW \*d\_window) {

dump\_window = d\_window;

}

void MyScheduler::setLogWindow(WINDOW \*log\_win) {

log\_window = log\_win;

}

// void MyScheduler::start\_manage\_tasks(void \*ptr) {

// pthread\_t pt;

// pthread\_create(&pt, NULL, manage\_tasks, ptr);

// }

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Purpose: Starts the task passed in the parameter as a thread

//Input : pt\_t\_ptr: thread\_t address

// (\*fun): pointer to the passed funtion

// win: pointer to the corresponding window

// name: of the thread

// state: initial state of the thread

//Output : none

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void MyScheduler::create\_task(pthread\_t \*pt\_t\_ptr, void \*(\*fun) (void \*), WINDOW \*win, const char \*name, int state) {

process\_table.insert(\*(new TCB), process\_table.getLength());

TCB \*process = process\_table.at(process\_table.getLength() - 1);

process->state = state;

process->tid = next\_tid++;

process->win\_ptr = win;

process->log\_win\_ptr = log\_window;

strcpy(process->tname, name);

pthread\_create(pt\_t\_ptr, NULL, fun, process);

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Purpose: marks the thread with tid == id as DEAD

//Input : id

//Output : none

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void MyScheduler::destroy\_task(int id) {

for (int i = 0; i < process\_table.getLength(); i++) {

if (process\_table.at(i)->tid == id) {

process\_table.at(i)->state = DEAD;

break;

}

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Purpose: To set the thread with tid == id to BLOCKED

//Input : id

//Output : none

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void MyScheduler::yield(int id) {

for (int i = 0; i < process\_table.getLength(); i++)

if (process\_table.at(i)->tid == id)

process\_table.at(i)->state = BLOCKED;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Purpose: Prints the dump, content on the window, with details based on level.

//Input : level

//Output : none

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void MyScheduler::dump(int level) {

int console\_tid = -1;

TCB \*process;

for (int i = 0; i < process\_table.getLength(); i++) {

process = process\_table.at(i);

wprintw(dump\_window, " Name = %s", process->tname);

if (!strcmp(process->tname, "Console\0"))

console\_tid = process->tid;

if (level >= 1) {

wprintw(dump\_window, ", TID = %d", process->tid);

}

if (level >= 2) {

switch (process->state) {

case DEAD:

wprintw(dump\_window, ", state = DEAD");

break;

case BLOCKED:

wprintw(dump\_window, ", state = BLOCKED");

break;

case READY:

wprintw(dump\_window, ", state = READY");

break;

case RUNNING:

wprintw(dump\_window, ", state = RUNNING");

break;

}

}

wprintw(dump\_window, "\n");

box(dump\_window, 0, 0);

wrefresh(dump\_window);

}

change\_state(console\_tid, RUNNING);

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Purpose: Removes threads that are marked DEAD

//Input : none

//Output : none

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void MyScheduler::garbage\_collect() {

for (int i = 0; i < process\_table.getLength(); i++)

if (process\_table.at(i)->state == DEAD)

process\_table.remove(i);

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Purpose: changes state of the thread with tid == id passed in.

//Input : id, newState.

//Output : none

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void MyScheduler::change\_state(int id, int newState) {

for (int i = 0; i < process\_table.getLength(); i++)

if (process\_table.at(i)->tid == id)

process\_table.at(i)->state = newState;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Purpose: Runs the next id by implicitly blocking the currrent id

//Input : id

//Output : none

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void MyScheduler::run\_next(int id) {

int i;

for (i = 0; i < process\_table.getLength(); i++) {

if (process\_table.at(i)->tid == id) {

process\_table.at(i)->state = BLOCKED;

i = (i + 1) % process\_table.getLength();

break;

}

}

process\_table.at(i)->state = RUNNING;

}

// void\* MyScheduler::manage\_tasks(void \*ptr) {

// int i = 0;

// MyScheduler \*sched\_ptr = (MyScheduler\*) ptr;

// TCB \*process;

// while (true) {

// if (!sched\_ptr->process\_table.isEmpty()) {

// process = sched\_ptr->process\_table.at(i);

//

// if (process->state == READY)

// process->state = RUNNING;

// else

// i = (i + 1) % sched\_ptr->process\_table.getLength();

// }

// else pthread\_yield();

// }

// return NULL;

// }

## MyScheduler.h

#ifndef MYSCHEDULER\_H

#define MYSCHEDULER\_H

#define \_OPEN\_THREADS

#define \_GNU\_SOURCE

#include "MyVector.h"

#include <iostream>

#include <pthread.h>

#include <assert.h>

#include <time.h>

#include <unistd.h>

#include <ncurses.h>

#include <stdarg.h>

#include <termios.h>

#include <fcntl.h>

#include <cstdlib>

using namespace std;

class MyScheduler {

public:

MyScheduler();

void setDumpWindow(WINDOW \*d\_window);

void setLogWindow(WINDOW \*log\_win);

static const int DEAD = -1, BLOCKED = 0, READY = 1, RUNNING = 2;

void create\_task(pthread\_t \*pt\_t\_ptr, void \*(\*fun) (void \*), WINDOW \*win, const char \*name, int state);

void destroy\_task(int id);

void yield(int id);

void dump(int level);

void garbage\_collect();

void change\_state(int id, int state);

void run\_next(int id);

struct TCB {

int state, tid;

char tname[64];

WINDOW \*win\_ptr;

WINDOW \*log\_win\_ptr;;

};

MyVector<TCB> process\_table;

private:

int next\_tid;

WINDOW \*dump\_window;

WINDOW \*log\_window;

};

#endif

## MySemaphore.cpp

#define \_OPEN\_THREADS

#define \_GNU\_SOURCE

#include "MyScheduler.h"

#include <iostream>

#include <pthread.h>

#include <assert.h>

#include <time.h>

#include <unistd.h>

#include <ncurses.h>

#include <stdarg.h>

#include <termios.h>

#include <fcntl.h>

#include <cstdlib>

#include <cstring>

using namespace std;

MyScheduler::MyScheduler() {

next\_tid = 0;

}

void MyScheduler::setDumpWindow(WINDOW \*d\_window) {

dump\_window = d\_window;

}

void MyScheduler::setLogWindow(WINDOW \*log\_win) {

log\_window = log\_win;

}

// void MyScheduler::start\_manage\_tasks(void \*ptr) {

// pthread\_t pt;

// pthread\_create(&pt, NULL, manage\_tasks, ptr);

// }

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Purpose: Starts the task passed in the parameter as a thread

//Input : pt\_t\_ptr: thread\_t address

// (\*fun): pointer to the passed funtion

// win: pointer to the corresponding window

// name: of the thread

// state: initial state of the thread

//Output : none

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void MyScheduler::create\_task(pthread\_t \*pt\_t\_ptr, void \*(\*fun) (void \*), WINDOW \*win, const char \*name, int state) {

process\_table.insert(\*(new TCB), process\_table.getLength());

TCB \*process = process\_table.at(process\_table.getLength() - 1);

process->state = state;

process->tid = next\_tid++;

process->win\_ptr = win;

process->log\_win\_ptr = log\_window;

strcpy(process->tname, name);

pthread\_create(pt\_t\_ptr, NULL, fun, process);

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Purpose: marks the thread with tid == id as DEAD

//Input : id

//Output : none

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void MyScheduler::destroy\_task(int id) {

for (int i = 0; i < process\_table.getLength(); i++) {

if (process\_table.at(i)->tid == id) {

process\_table.at(i)->state = DEAD;

break;

}

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Purpose: To set the thread with tid == id to BLOCKED

//Input : id

//Output : none

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void MyScheduler::yield(int id) {

for (int i = 0; i < process\_table.getLength(); i++)

if (process\_table.at(i)->tid == id)

process\_table.at(i)->state = BLOCKED;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Purpose: Prints the dump, content on the window, with details based on level.

//Input : level

//Output : none

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void MyScheduler::dump(int level) {

int console\_tid = -1;

TCB \*process;

for (int i = 0; i < process\_table.getLength(); i++) {

process = process\_table.at(i);

wprintw(dump\_window, " Name = %s", process->tname);

if (!strcmp(process->tname, "Console\0"))

console\_tid = process->tid;

if (level >= 1) {

wprintw(dump\_window, ", TID = %d", process->tid);

}

if (level >= 2) {

switch (process->state) {

case DEAD:

wprintw(dump\_window, ", state = DEAD");

break;

case BLOCKED:

wprintw(dump\_window, ", state = BLOCKED");

break;

case READY:

wprintw(dump\_window, ", state = READY");

break;

case RUNNING:

wprintw(dump\_window, ", state = RUNNING");

break;

}

}

wprintw(dump\_window, "\n");

box(dump\_window, 0, 0);

wrefresh(dump\_window);

}

change\_state(console\_tid, RUNNING);

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Purpose: Removes threads that are marked DEAD

//Input : none

//Output : none

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void MyScheduler::garbage\_collect() {

for (int i = 0; i < process\_table.getLength(); i++)

if (process\_table.at(i)->state == DEAD)

process\_table.remove(i);

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Purpose: changes state of the thread with tid == id passed in.

//Input : id, newState.

//Output : none

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void MyScheduler::change\_state(int id, int newState) {

for (int i = 0; i < process\_table.getLength(); i++)

if (process\_table.at(i)->tid == id)

process\_table.at(i)->state = newState;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Purpose: Runs the next id by implicitly blocking the currrent id

//Input : id

//Output : none

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void MyScheduler::run\_next(int id) {

int i;

for (i = 0; i < process\_table.getLength(); i++) {

if (process\_table.at(i)->tid == id) {

process\_table.at(i)->state = BLOCKED;

i = (i + 1) % process\_table.getLength();

break;

}

}

process\_table.at(i)->state = RUNNING;

}

// void\* MyScheduler::manage\_tasks(void \*ptr) {

// int i = 0;

// MyScheduler \*sched\_ptr = (MyScheduler\*) ptr;

// TCB \*process;

// while (true) {

// if (!sched\_ptr->process\_table.isEmpty()) {

// process = sched\_ptr->process\_table.at(i);

//

// if (process->state == READY)

// process->state = RUNNING;

// else

// i = (i + 1) % sched\_ptr->process\_table.getLength();

// }

// else pthread\_yield();

// }

// return NULL;

// }

## MySemaphore.h

#ifndef MYSEMAPHORE\_H

#define MYSEMAPHORE\_H

#include "MyQueue.h"

#include "MyScheduler.h"

#include <string.h>

using namespace std;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Purpose : To control access of resources provided in the window.

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

class MySemaphore {

public:

MySemaphore (char \*name)

{

this->sema\_value = 1;

strcpy(resource\_name, name);

this->sema\_queue;

dumping = false;

}

void set\_dump\_win(WINDOW \*win);

void set\_sched\_ptr(MyScheduler \*ptr);

void down(int);

void up();

void dump(int level);

void un\_dump();

private:

char resource\_name[64];

int sema\_value;

MyQueue<int> sema\_queue;

MyScheduler \*Scheduler;

MyVector<MyScheduler::TCB> states;

bool dumping;

WINDOW \*dump\_window;

};

#endif

## MyVector.h

#ifndef MYVECTOR\_H

#define MYVECTOR\_H

#include <iostream>

using namespace std;

template <typename T>

class MyVector {

public:

MyVector() {

head = NULL;

tail = NULL;

length = 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Purpose: Inserts newData in the linked list at position.

Returns true on success

Input: newData, position

Output: True on success, false otherwise.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

bool insert(T newData, int position) {

if (position < 0 || position > length)

return false;

if (length < 0) {

cout << "\nError in MyVector::insert(), length < 0.\n";

return false;

}

else if (length == 0) {

tail = new node;

head = tail;

head->next = NULL;

head->data = newData;

}

else {

node \*newNode = new node;

newNode->data = newData;

if (position == 0) {

newNode->next = head;

head = newNode;

}

else if (position == length) {

tail->next = newNode;

tail = newNode;

}

else {

node \*prev = head;

int i = 0;

while (prev->next && i < position) {

prev = prev->next;

i++;

}

newNode->next = prev->next;

prev->next = newNode;

}

}

length++;

return true;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Purpose: Deletes nodes starting from start up until last, inclusive.

Input: start, last.

Output: none.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void remove\_index(int start, int last) {

if (last < start || length < 1)

return;

node \*temp = head;

int i = 0;

if (start == 0) {

while (i <= last && temp) {

head = head->next;

delete temp;

length--;

i++;

temp = head;

}

if (head == NULL)

tail = NULL;

return;

}

node \*prev = NULL;

while (i < start && temp) {

prev = temp;

temp = temp->next;

i++;

}

while (i <= last && temp) {

prev->next = temp->next;

delete temp;

length--;

i++;

temp = prev->next;

}

if (temp == NULL)

tail = prev;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Purpose: Deletes nodes starting from start, up to (size) number of nodes

Input: newData, position

Output: none

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void remove\_size(int start, int size) {

if (size < 1 || length < 1)

return;

node \*temp = head;

int i = 0;

if (start == 0) {

while (i < size && temp) {

head = head->next;

delete temp;

length--;

i++;

temp = head;

}

if (head == NULL)

tail = NULL;

return;

}

node \*prev = NULL;

while (i < start && temp) {

prev = temp;

temp = temp->next;

i++;

}

i = 0;

while (i < size && temp) {

prev->next = temp->next;

delete temp;

length--;

i++;

temp = prev->next;

}

if (temp == NULL)

tail = prev;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Purpose: Deletes a single node at position

Input: position

Output: none

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void remove(int position) {

if (position < 0 || length < 1)

return;

node \*temp = head;

if (position == 0) {

head = head->next;

delete temp;

length--;

if (head == NULL)

tail = NULL;

return;

}

int i = 0;

node \*prev = NULL;

while (i < position && temp) {

prev = temp;

temp = temp->next;

i++;

}

prev->next = temp->next;

delete temp;

length--;

if (temp == tail)

tail = prev;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Purpose: Returns a pointer to TCB at position

Input: position

Output: Pointer to TCB.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

T\* at(int position) {

node\* temp = head;

int i = 0;

while (i < position && temp) {

temp = temp->next;

i++;

}

return &(temp->data);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Purpose: length getter

Input: none

Output: length value

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int getLength() {

return length;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Purpose: Checks if the linked list is empty

Input: none

Output: True if empty, false otherwise.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

bool isEmpty() {

return length == 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Purpose: Empties the entire linked list

Input: none

Output: none

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void clear() {

remove\_size(0, length);

}

~MyVector() {

clear();

}

private:

struct node {

T data;

node \*next;

};

node \*head, \*tail;

int length;

};

#endif

## MyNode.h

#ifndef MYNODE\_H

#define MYNODE\_H

#include <cstddef>

using namespace std;

template <typename T>

class MyNode {

public:

MyNode() {

next = NULL;

}

MyNode(T &newData) {

data = newData;

next = NULL;

}

T getData() {

return data;

}

void setData(T &newData) {

data = newData;

}

MyNode<T>\* getNext() {

return next;

}

void setNext(MyNode<T>\* newNext) {

next = newNext;

}

~MyNode() {

}

private:

T data;

MyNode \*next;

};

#endif

# Output Test Plan

## 

## 

## 

## 

## 

## **System Functionality**

The system is working because we met the requirements that the professor requested: running threads and organizing them using scheduler and semaphore classes. It also worked because we asked for clarification on a certain task. Having a clear understanding on some of the requirements made us able to implement the objectives.

Objectives Requirements

The objectives were attained by having the windows run through threads while be managed by semaphores and scheduler. The required buttons for the window to operate are 1,2, 3, c, h, d. Where buttons 1, 2, 3 are recognized through a text in console window, c clears the console window and fixes the screen in case it glitched on startup, h brings the help information and d dumps content information to the dump window.