ULTIMA 2.0

CS-435 Operating System

Phase 2 – InterProcess Communication

By:

Drake Wood

drawood@iu.edu

James Giegerich

jgiegeri@iu.edu

3/18/19

Table of Contents

Phase Abstract

This report details the design and development of the second phase of the Ultima 2.0 project for C435. This program is written in C++ and is designed to run on a Unix operating system. This program and design are meant to be added to and upgraded in future iterations to add more features. This report includes the full source code as well as design diagrams.

Before this phase could be started, we had to rework Phase One. The first thing we did was assign the UI loop to its own thread. The second thing was set each thread status to “READY” upon creation. We developed a run() function in the scheduler class to set the first thread to “RUNNING”. Our yield() method was developed to set the current “RUNNING” thread to “READY”, and the find the next “READY” thread and set it to “RUNNING”. Lastly, we reworked our up() and down() methods in the semaphore class to operate as binary semaphores. If the semaphore was already being used, the thread requesting access will get queued until the current thread is finished with the resource for that semaphore.

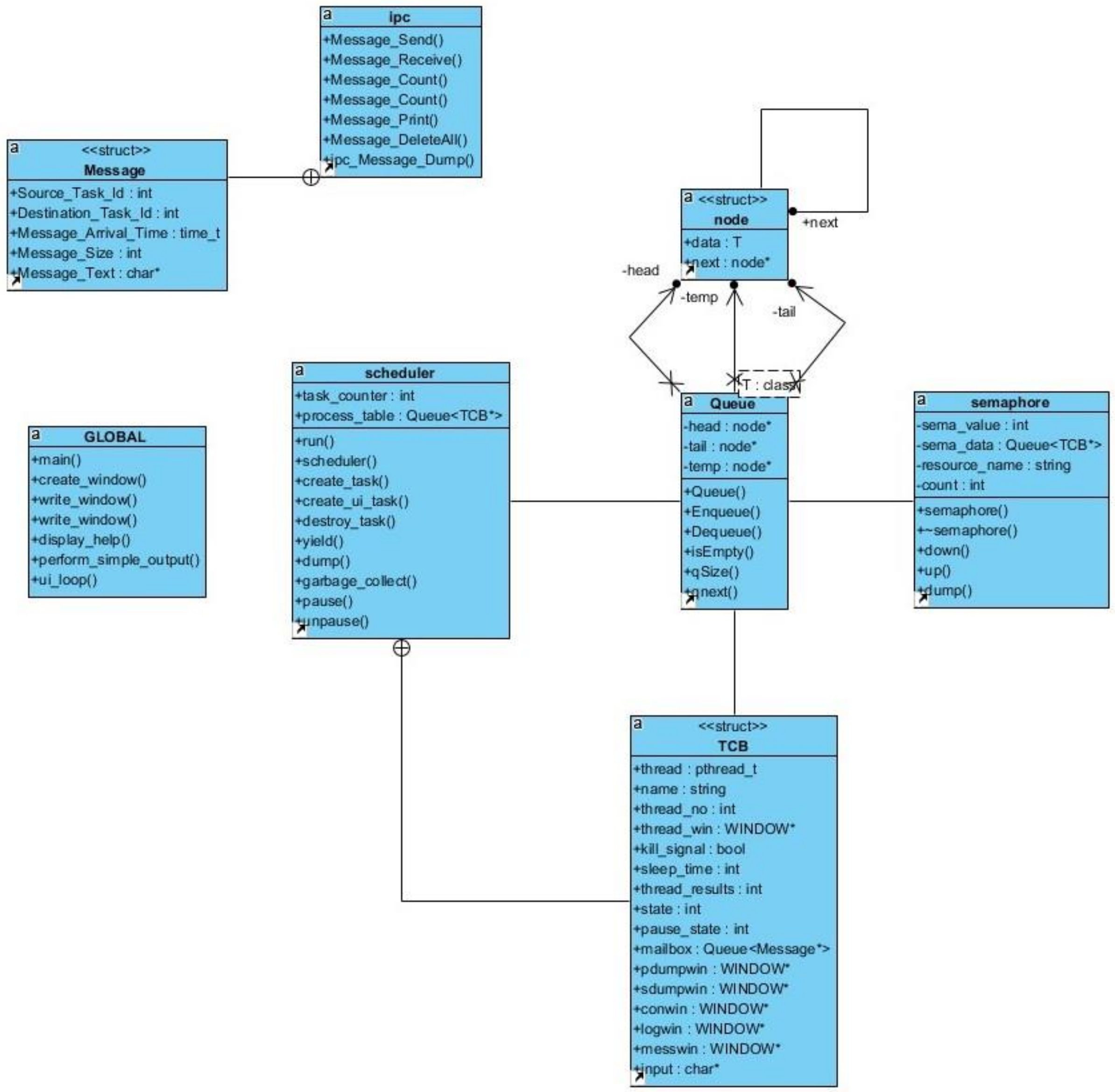
Phase Description

This phase is to develop and implement a simulated interprocess communication system for the threads in our simulated operating system. One task must be able to send and receive messages from another task. This is done by the use of a physical mailbox queue in the Task Control Block (TCB). The IPC class includes a struct object that includes data members such as the Task ID numbers of the source and destination threads, message arrival time, message size, and the text of the message itself. Included class methods are Message_Send() and Message_Receive() for sending and receiving, Message_Count(int Task_Id) to count the number of messages in a specific task’s mailbox and Message_Count() to tally the number of messages in all collective mailbox queues, Message_Print() to print all queued messages for any given Task ID, Message_DeleteAll() to delete all queued messages in a given task, and ipc_Message_Dump() to print all queued messages for all tasks.

- Each mailbox is given its own semaphore to keep threads from simultaneously writing to the same message queue.
- Message_Send() takes an object of type Message* and enqueues it to the mailbox queue of that message’s destination Task ID.
- Message_Receive() takes in a Task ID and dequeues the first message object on that task’s mailbox queue.
- Message_Print() takes in a Task ID and prints all messages for that task as well as the other Message object data members.
- Message_DeleteAll() takes a Task ID and deletes (dequeues) all messages in that task’s mailbox.
- ipc_Message_Dump() prints all queued messages for all tasks to the Message Dump Window by printing a table header for the data members and then calling Message_Print() once for each Task that has not yet been killed.

The text book Modern Operating Systems by Tanenbaum, C435 class notes, Unix manual, and some online research was used for external information.

Design Diagram



Source Code

Main.cpp

```
/*===================================================================================|
|   Assignment:     Ultima 2.0 Phase 2
|    File Name:     main.cpp
| Dependencies:     scheduler.h semaphore.h window.h queue.h
|      Authors:     Drake Wood, James Giegerich
|     Language:     C++
|     Compiler:     G++
|        Class:     C435 - Operating Systems
|   Instructor:     Dr. Hakimzadeh
| Date Created:     2/16/2019
| Last Updated:     3/18/2019
|     Due Date:     3/18/2019
|===================================================================================|
| Description: Contains the main function that interacts with the scheduler, semaphore, queue, and window classes.
|              Multiple curses windows and 3 pthreads are created which are then passed to other functions to handle.
|              The rest of main contains an input loop for the user to control what happens while the program is running.
|              Details of the way functions are used and their parameters as well as
|              computation details are given in the class descriptions.
*===================================================================================*/

#include <iostream>
#include <pthread.h>            // Needed for using the pthread library
#include <assert.h>
#include <time.h>
#include <unistd.h>             // Needed for sleep()
#include <ncurses.h>            // Needed for Curses windowing
#include <stdarg.h>             // Needed for formatted output to window
#include <termios.h>
#include <fcntl.h>
#include <cstdlib>
#include "queue.h"
#include "scheduler.h"
#include "semaphore.h"
#include "window.h"

using namespace std;
semaphore sema_screen(1, (char *)"Screen Print");           // creates semaphores
semaphore sema_t1mail(1, (char *)"t1mail");
semaphore sema_t2mail(1, (char *)"t2mail");
semaphore sema_t3mail(1, (char *)"t3mail");
semaphore sema_t4mail(1, (char *)"t4mail");
semaphore sema_ptable(1, (char *)"ptable");
scheduler sched; //creates scheduler
ipc IPC; // creates ipc

//-------------------------------------------------------------------------------
```

```
//-------------------------------MAIN--------------------------------------------
//-----------------------------------------------------------------------------

int main() {
	initscr();					// Start nCurses
	refresh();					// Refresh screen

//-----------------------------------------------------------------------------
//Creating heading window, printing content
//-----------------------------------------------------------------------------

	WINDOW * Heading_Win = newwin(12, 80, 3, 2);
	box(Heading_Win, 0,0);
	mvwprintw(Heading_Win, 2, 28, "ULTIMA 2.0 (Spring 2019)");
	mvwprintw(Heading_Win, 4, 2, "Starting ULTIMA 2.0.....");
	mvwprintw(Heading_Win, 5, 2, "Starting Thread 1....");
	mvwprintw(Heading_Win, 6, 2, "Starting Thread 2....");
	mvwprintw(Heading_Win, 7, 2, "Starting Thread 3....");
	mvwprintw(Heading_Win, 9, 2, "Press 'q' or Ctrl-C to exit the program...");
	wrefresh(Heading_Win);

//-----------------------------------------------------------------------------
//Creating log window
//-----------------------------------------------------------------------------

	WINDOW * Log_Win = create_window(15, 60, 30, 2);
	write_window(Log_Win, 1, 18, "...........Log...........\n");

//-----------------------------------------------------------------------------
//Creating console window
//-----------------------------------------------------------------------------

	WINDOW * Console_Win = create_window(15, 20, 30, 62);
	write_window(Console_Win, 1, 1, " ....Console....\n");
	write_window(Console_Win, 2, 1, "Ultima # ");
	write_window(Log_Win, " Main program started\n" );

//-----------------------------------------------------------------------------
//Creating 3 windows for tasks
//-----------------------------------------------------------------------------

	WINDOW * W1 = create_window(15, 25, 15, 2);
	WINDOW * W2 = create_window(15, 25, 15, 30);
	WINDOW * W3 = create_window(15, 25, 15, 57);

//-----------------------------------------------------------------------------
//Creating a process table dump window
//-----------------------------------------------------------------------------
```

```
        WINDOW * Process_Table = create_window(9, 80, 3, 83);
        write_window(Process_Table, 1, 5, "     PROCESS TABLE DUMP \n -------------------------------------\n");

//---------------------------------------------------------------------------
//Creating tasks with create_task passing each a name and task window
//---------------------------------------------------------------------------

        sched.create_task((char*)" Task 1", W1,Process_Table);
        sched.create_task((char*)" Task 2", W2,Process_Table);
        sched.create_task((char*)" Task 3", W3,Process_Table);

//----------------------------------------------------------------------------
//Creating a semaphore dump window
//----------------------------------------------------------------------------

        WINDOW * Sema_Dump = create_window(16, 80, 12, 83);
        write_window(Sema_Dump, 1, 5, "      SEMAPHORE DUMP \n -------------------------------------\n");

//---------------------------------------------------------------------------
//Create a Messaging dump window
//---------------------------------------------------------------------------

        WINDOW * Message_Dump = create_window(17, 80, 28, 83);
        write_window(Message_Dump, 1, 5, "      MESSAGING DUMP \n -------------------------------------\n");

//---------------------------------------------------------------------------
//Setup for user console input
//---------------------------------------------------------------------------

        cbreak();                           // Set up keyboard I/O processing
        noecho();                           // disable line buffering
        nodelay(Console_Win, true);         // disable automatic echo of characters read by getch(), wgetch()
                                            // nodelay causes getch to be a non-blocking call

//---------------------------------------------------------------------------
//Start threads and run till end of program
//---------------------------------------------------------------------------

        sched.run(); //start running all the tasks

        sched.create_ui_task(Process_Table, Sema_Dump, Console_Win, Log_Win, Message_Dump);

        while(sched.process_table.qSize() != 0){ // loop while threads run
                sleep(1);
        }

endwin();
return 0;
} // end of main
```

Semaphore.h

```
/*=================================================================================|
|   Assignment:      Ultima 2.0 Phase 2
|     File Name:     semaphore.h
|  Dependencies:     scheduler.h
|       Authors:     Drake Wood, James Giegerich
|      Language:     C++
|      Compiler:     G++
|         Class:     C435 - Operating Systems
|    Instructor:     Dr. Hakimzadeh
|  Date Created:     2/16/2019
|  Last Updated:     3/18/2019
|      Due Date:     3/18/2019
|=================================================================================|
|  Description: This is the header file which defines the class semaphore. There are 3 functions along with a
|                              constructor and deconstructor.
|                              -dump displays information related to this class in a window for debugging.
|                              -up creates a mutex and queues it, effectively locking a critical section.
|                              -down dequeues a mutex, unlocking it.
*=================================================================================*/

#ifndef SEMAPHORE_H
#define SEMAPHORE_H

#include <iostream>
#include <pthread.h>          // Needed for using the pthread library
#include <assert.h>
#include <time.h>
#include <unistd.h>                    // Needed for sleep()
#include <ncurses.h>          // Needed for Curses windowing
#include <stdarg.h>                    // Needed for formatted output to window
#include <termios.h>
#include <fcntl.h>
#include <cstdlib>
#include "queue.h"
#include "scheduler.h"
#include <string>

class semaphore {
        int sema_value;                        // 0 or 1 in the case of a binary semaphore

        Queue <scheduler::TCB*>sema_data;
        std::string resource_name; // The name of the resource being managed
        int count;

        public:
        semaphore(int sema_value, char* sema_name);
        ~semaphore();

        void down();                                    // Get the resource or get queued!
        void up();                                      // Release the resource
        void dump(int level, WINDOW * win); // Include some
};                                                      // Functions which will allow you to dump the
#endif                                                  // Contents of the semaphore in a readable format.
```

Semaphore.cpp

```
/*==================================================================================|

|    Assignment:      Ultima 2.0 Phase 2

|     File Name:      semaphore.cpp

|  Dependencies:      semaphore.h window.h

|       Authors:      Drake Wood, James Giegerich

|      Language:      C++

|      Compiler:      G++

|         Class:      C435 - Operating Systems

|    Instructor:      Dr. Hakimzadeh

|  Date Created:      2/16/2019

|  Last Updated:      3/18/2019

|      Due Date:      3/18/2019

|==================================================================================|

|  Description: Contains the definitions for the functions outlined in semaphore.h

*==================================================================================*/


#include "window.h"

#include "semaphore.h"

#include "scheduler.h"


extern semaphore sema_screen;

extern scheduler sched;

extern scheduler TCB;


void semaphore::down(){                                                  // Lock resource


                scheduler::TCB * tcb; // new tcb object


                if (sema_value > 0){ // if no 1 else has access

                        sema_value--;        // get access here

                } else{ // if someone else has access then we need to queue and block ourselves

                        tcb = sched.process_table.Dequeue();

                        tcb->state = 0; // blocked
```

```
                    sched.process_table.Enqueue(tcb);  //add back to process table

                    sema_data.Enqueue(tcb); // stick in sema queue waitlist


                    for (int i = 0; i < sched.process_table.qSize() -1 ; i++)

                    {

                            tcb = sched.process_table.Dequeue();

                            sched.process_table.Enqueue(tcb); //shuffle queue

                    }

            }


} // end of down


void semaphore::up(){
        // Release the resource


            if (sema_data.isEmpty()){ // if no one is in line then set value back to 1

                    sema_value++;

            } else {

                    scheduler::TCB * tcb = sema_data.Dequeue(); // get first in semaphore queue

                    scheduler::TCB * tcb2; // new tcb for comparison

                    for (int i = 0; i < sched.process_table.qSize() ; i++)

                    {

                            tcb2 = sched.process_table.Dequeue(); //get scheduler task

                            if (tcb->thread_no == tcb2->thread_no){ // if task id in scheduler matches the one in
semaphore

                                    tcb2->state = 1;  //ready // change it to ready from blocked

                            }

                            sched.process_table.Enqueue(tcb2); //shuffle queue

                    }

            }

} // end of up


void semaphore::dump(int level, WINDOW * win){

                                                                // giving semaphore a resource name
```

```
	char buff[256];



	sprintf(buff, " Resource: %s Sema Value: \t%d \n", resource_name.c_str(), sema_value); // Print the resource name of
the current semaphore

	write_window(win, buff);


	std::string name;

	if (sema_data.qSize() > 0){

		scheduler::TCB * tcb;


		for (int i = 0 ; sema_data.qSize(); i++){

			tcb = sema_data.Dequeue();

			name = tcb->name;

			sema_data.Enqueue(tcb);

			sprintf(buff, "%s -> ", name.c_str());

			write_window(win, buff);

		}

	} else{

			write_window(win, " \tSema Queue empty\n");

	}

} // end of dump


semaphore::semaphore(int sema_value, char* sema_name){

	count = 0;

	this->sema_value = sema_value;

	this->resource_name = sema_name;

}


semaphore::~semaphore(){

}
```

Window.h

```
/*=================================================================================|
|   Assignment:     Ultima 2.0 Phase 2
|    File Name:     window.h
|  Dependencies:    none
|      Authors:     Drake Wood, James Giegerich
|     Language:     C++
|     Compiler:     G++
|        Class:     C435 - Operating Systems
|   Instructor:     Dr. Hakimzadeh
|  Date Created:    2/16/2019
|  Last Updated:    3/18/2019
|     Due Date:     3/18/2019
|=================================================================================|
|  Description: This is the header file which defines the class window. There are 5 functions.
|                           -create_window draws a new window at a specified size and position, it also turns on
scrolling.
|                           -write_window sends text to a specific window either with a position or without.
|                           -display_help prints a help window with console options for the user.
|                           -perform_simple_output is the function that the thread will run while alive, it prints an
incrementing message.
*=================================================================================*/


#ifndef WINDOW_H
#define WINDOW_H


#include <iostream>
#include <pthread.h>        // Needed for using the pthread library
#include <assert.h>
#include <time.h>
#include <unistd.h>         // Needed for sleep()
#include <ncurses.h>        // Needed for Curses windowing
#include <stdarg.h>         // needed for formatted output to window
#include <termios.h>
```

```
#include <fcntl.h>
#include <cstdlib>

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 *perform_simple_output(void *arguments);
void *ui_loop(void *arguments);

#endif
```

Window.cpp

```
/*==============================================================================|
|   Assignment:     Ultima 2.0 Phase 2
|    File Name:     window.cpp
|  Dependencies:    semaphore.h window.h
|      Authors:     Drake Wood, James Giegerich
|     Language:     C++
|     Compiler:     G++
|        Class:     C435 - Operating Systems
|   Instructor:     Dr. Hakimzadeh
|  Date Created:    2/16/2019
|  Last Updated:    3/18/2019
|      Due Date:    3/18/2019
|==============================================================================|
|  Description: Contains the definitions for the functions outlined in window.h
*==============================================================================*/

#include "window.h"
#include "semaphore.h"
#include "ipc.h"
#include "time.h"



extern semaphore sema_screen;
extern semaphore sema_t1mail;
extern semaphore sema_t2mail;
extern semaphore sema_t3mail;
extern semaphore sema_t4mail;
extern semaphore sema_ptable;
extern scheduler sched;
extern ipc IPC;

WINDOW *create_window(int height, int width, int starty, int startx)
{
        sema_screen.down();
        WINDOW *Win;
        Win = newwin(height, width, starty, startx);
        scrollok(Win, TRUE);            // Allow scrolling of the window
        scroll(Win);                                        // scroll the window
        box(Win, 0 , 0);                    // 0, 0 gives default characters for the vertical and horizontal lines
        wrefresh(Win);                              // draw the window
        sema_screen.up();

        return Win;
} // end of create_window

void write_window(WINDOW * Win, const char* text)
{
        sema_screen.down();
        wprintw(Win, text);
        box(Win, 0 , 0);
        wrefresh(Win);                      // Draw the window
        sema_screen.up();
 } // end of write_window

void write_window(WINDOW * Win, int y, int x, const char* text)
{
        sema_screen.down();
        mvwprintw(Win, y, x, text);
        box(Win, 0 , 0);
        wrefresh(Win);                      // Draw the window
        sema_screen.up();
} // end of write_window

void display_help(WINDOW * Win)
{
        sema_screen.down();
        wclear(Win);
        sema_screen.up();                                       // Write window already has its own lock

        write_window(Win, 1, 1, "1: Kill Task 1");

        write_window(Win, 2, 1, "2: Kill Task 2");

        write_window(Win, 3, 1, "3: Kill Task 3");

        write_window(Win, 4, 1, "c: Clear Screen");

        write_window(Win, 5, 1, "d: Pause + Dump");

        write_window(Win, 6, 1, "h: Help Screen");

        write_window(Win, 7, 1, "q: Quit");

        write_window(Win, 8, 1, "g: Garbage Collect");

        write_window(Win, 9, 1, "z: Message testing");


 } // end of display_help


void *perform_simple_output(void *arguments)

{

        scheduler::TCB * tcb = (scheduler::TCB *) arguments;        // Extract the thread arguments: (method 1)

        int thread_no = tcb->thread_no;                                                 // Cast arguments in to
thread_data

        WINDOW * Win = tcb->thread_win;

        WINDOW * pdumpwin = tcb->pdumpwin;

        int CPU_Quantum =0;

        int yield_quantum = 0;

        char buff[256];

        time_t messagetime;





        while (tcb->state != 3){ // not dead

                while(tcb->state == 2) { // running

                        if(yield_quantum == 0){ // updates process table when thread gets cpu time

                                sched.dump(1, pdumpwin);

                        }


                        if (CPU_Quantum == 0) { // First thing a task does is send messages
                for (int i = 1 ; i < 4; i++){
                        ipc::Message * message = new ipc::Message; // create message
                        message->Source_Task_Id = thread_no; // source = this task
                        message->Destination_Task_Id = i; // destination is every task
                        message->Message_Text = "message text"; // placeholder text
                        time(&messagetime); // get time
                        message->Message_Arrival_Time = messagetime; // store time
                        message->Message_Size = sizeof(message); // get and store size of message
                        if (IPC.Message_Send(message) == 1){ // send the message
                                sprintf(buff, " Message sent\n");
                                write_window(Win, buff);
                        }
                }
        }



        if(tcb->kill_signal !=1){ //for some reason we cant die before printing or get corruption for now
                sprintf(buff, " Task-%d running #%d\n", thread_no, CPU_Quantum++);
                write_window(Win, buff);
        }
        yield_quantum++;
        if (tcb->kill_signal == 1){ // set to be killed
                        write_window(Win, " I'm dying...\n");
                        tcb->state = 3;
                        sched.yield();
                }
        if (yield_quantum == 1001){// if quantum is up
                        yield_quantum = 0; //reset
                        write_window(Win, " I'm yielding...\n");
                        sched.yield();
                        //sleep(1);
                }
        }
} // end while
```

```
        return 0;
}

void *ui_loop(void *arguments)
{
        scheduler::TCB * tcb = (scheduler::TCB *) arguments;     // Extract the thread arguments: (method 1)
                                                          // Cast arguments in to tcb
        WINDOW * pdumpwin = tcb->pdumpwin;
        WINDOW * sdumpwin = tcb->sdumpwin;
        WINDOW * conwin = tcb->conwin;
        WINDOW * logwin = tcb->logwin;
        WINDOW * messwin = tcb->messwin;
        //char* input = tcb->input;
        char buff[256];

                while (tcb->state != 3){ // not dead
                while(tcb->state == 2) { // running
                         // updates process table when thread gets cpu time
                                sched.dump(1, pdumpwin);

                        switch(wgetch(conwin))
                {
                        case '1':
                                        write_window(conwin, "1 \n Ultima # ");
                                        if (sched.destroy_task(1)){
                                                write_window(logwin, " Task 1 killed. \n");
                                        }else{
                                                write_window(logwin, " Task 1 was already dead... \n ");
                                        }
                                        sched.yield();
                                        break;
                        case '2':
                                        write_window(conwin, "2 \n Ultima # ");
                if (sched.destroy_task(2)){
                        write_window(logwin, " Task 2 killed. \n");
                }else{
                        write_window(logwin, " Task 2 was already dead... \n ");
                }
                break;
case '3':
                write_window(conwin, "3 \n Ultima # ");
                if (sched.destroy_task(3)){
                        write_window(logwin, " Task 3 killed. \n");
                }else{
                        write_window(logwin, " Task 3 was already dead... \n ");
                }
                break;
case 'c':                                       // CLEAR

        sema_screen.down();
        refresh();                      // Clear the entire screen (in case it is corrupted)
        wclear(conwin);         // Clear the Console window
        sema_screen.up();

        write_window(conwin, 1, 1, "Ultima # ");
        break;
case 'd':
        write_window(conwin, "d \n Ultima # ");
        write_window(logwin, " Paused, press any key to continue... \n");
        //sched.pause();
        sched.dump(1, pdumpwin);    // updates process dump windows

        sema_screen.down();
        refresh();                      // Clear the entire screen (in case it is corrupted)
        wclear(sdumpwin);               // Clear the sema dump window
        wclear(messwin);    // Clear the message dump window
        sema_screen.up();
```

```
                    write_window(sdumpwin, 1, 5, "        SEMAPHORE DUMP \n -------------------------------
------\n");

                    sema_screen.dump(1, sdumpwin);

                    sema_t1mail.dump(1, sdumpwin);

                    sema_t2mail.dump(1, sdumpwin);

                    sema_t3mail.dump(1, sdumpwin);

                    sema_t4mail.dump(1, sdumpwin);

                    sema_ptable.dump(1, sdumpwin);


                    write_window(messwin, 1, 5, "                    MESSAGING DUMP \n -------------------------
------------\n");

                    IPC.ipc_Message_Dump(messwin);

                    std::cin.get();

                    write_window(logwin, " Unpaused... \n");

                    break;

               case 'h':                                        // HELP


                    display_help(conwin);


                    break;

               case 'g':

                    sched.garbage_collect();

                    write_window(conwin, "g \n Ultima # ");

                    write_window(logwin, " Garbage collect\n");

                    sched.yield();

                    break;

               case 'q':                                        // QUIT

                    write_window(logwin," Quiting the main program....\n" );

                    for (int i = 1; i < 4; i++){

                         if (sched.destroy_task(i)){                    // Killed

                              sprintf(buff, " Task %d to be killed\n", i);

                              write_window(logwin, buff);

                         }else{
     // Already dead
```

```
                        sprintf(buff, " Task %d already dead\n", i);
                        write_window(logwin, buff);
                        } // end else
                } // end if
                tcb->kill_signal = 1;
                sched.yield();
        break;
case 'z': // message test case
        IPC.Message_Print(1, messwin); // print messages
        sprintf(buff, " # of messages in task1 box = %d \n" , IPC.Message_Count(1));
        write_window(messwin, buff);
        sprintf(buff, " # of messages in all boxes = %d \n" , IPC.Message_Count());
        write_window(messwin, buff);
        sprintf(buff, " Deleting messages in task 1 box \n");
        write_window(messwin, buff);
        IPC.Message_DeleteAll(1);
        sprintf(buff, " # of messages in task1 box = %d \n" , IPC.Message_Count(1));
        write_window(messwin, buff);
        sprintf(buff, " # of messages in all boxes = %d \n" , IPC.Message_Count());
        write_window(messwin, buff);
        break;
case ERR:       // If wgetch() return ERR, that means no keys were pressed
        if (tcb->kill_signal == 1){ // set to be killed
                write_window(logwin, " UI window dying...\n");
                tcb->state = 3;
        } else {
//write_window(logwin, " NO INPUT, UI yielding...\n");
                sched.yield();
        }

        break;
default:
        write_window(conwin, "\n -Invalid Command\n");
        write_window(logwin, " -Invalid Command\n");
```

```
                        write_window(conwin, " Ultima # \n");

                    break;
                } // end switch

                if (tcb->state == 3){
                write_window(logwin," Ultima 2.0 shutting down...\n" );
                sched.garbage_collect();
                }
        } // while running

} // while not dead

        return 0;

}
```

Queue.h

```
/*==================================================================================|
|    Assignment:       Ultima 2.0 Phase 1
|      File Name:  queue.h
| Dependencies:     none
|         Authors:     Drake Wood, James Giegerich
|      Language:   C++
|       Compiler:  G++
|           Class:    C435 - Operating Systems
|       Instructor:    Dr. Hakimzadeh
|  Date Created:    12/16/2019
|  Last Updated:       3/18/2019
|       Due Date:  3/18/2019
|==================================================================================|
|  Description: This is the header file which defines the queue template class used by the semaphore
|             and scheduler classes.
*==================================================================================*/

#ifndef QUEUE_H
#define QUEUE_H

#include <iostream>
#include <pthread.h> // Needed for using the pthread library
#include <assert.h>
#include <time.h>
#include <unistd.h> // Needed for sleep()
#include <ncurses.h> // Needed for Curses windowing
#include <stdarg.h> // needed for formatted output to window
#include <termios.h>
#include <fcntl.h>
#include <cstdlib>

template <class T>
```

```
  class Queue{
    private:
      struct node{
        T data;
        node* next;
        };
      node* head;
      node* tail;
      node* temp;

    public:
      Queue();
      void Enqueue(const T data);
      T Dequeue();
      bool isEmpty();
      int qSize();
      T qnext();
};



// Constructor.
template <class T>
Queue<T>::Queue(){
  head = NULL;
  tail = NULL;
  temp = NULL;
}

template <class T>
int Queue<T>::qSize(){
  int count = 0;
  temp = head;
```

```
    if (isEmpty()){
        return count;
    } // end if
    else if (temp->next == NULL){
        count++;
        return count;
    } // end if
    else{
        while (temp != NULL){
            count++;
            temp = temp->next;
        } // end while

        return count;
    } // end else
} // end of qSize

//get data of next item, will probably be able to use this in yield
//so that we can pass the cpu to the next ID in the QUIT
//also in semaphore dump to show the Q there
template <class T>
T Queue<T>::qnext(){
    T returnData;
    returnData = head->next->data;
    return returnData;
}

// Enqueues new node and populates the data
// with T data as passed in.
template <class T>
void Queue<T>::Enqueue(const T data){
    if (tail == NULL){
        head = tail = new node;
        tail->next = NULL;
        tail->data = data;
    } // end if
    else {
    tail->next = new node;
    tail->next->data = data;
    tail->next->next = NULL;
    tail = tail->next;
    } // end else
} // end of Enqueue

// Dequeues node and returns data stored
// in dequeued node.
template <class T>
T Queue<T>::Dequeue(){
    T returnData;
    returnData = head->data;
    temp = head->next;
    delete head;
    head = temp;
    tail = (!head? NULL: tail);
    return returnData;
}

// Checks if the queue is empty.
template <class T>
bool Queue<T>::isEmpty(){
  return (!tail);
}

#endif
```

Queue.cpp

```
/*=================================================================================|
|      Assignment:       Ultima 2.0 Phase 1
|      File Name:  queue.cpp
|      Dependencies:     queue.h
|       Authors:     Drake Wood, James Giegerich
|      Language:   C++
|      Compiler:  G++
|       Class:    C435 - Operating Systems
|       Instructor:    Dr. Hakimzadeh
|      Date Created:    2/16/2019
|      Last Updated:     3/18/2019
|      Due Date:           3/19/2019
|=================================================================================|
| Description: Contains the definitions for the functions outlined in queue.h
*=================================================================================*/

#include "queue.h"

// Constructor
template <class T>
Queue<T>::Queue(){
  head = NULL;
  tail = NULL;
  temp = NULL;
}

template <class T>
int Queue<T>::qSize(){
   int count = 0;
   temp = head;

   if (isEmpty()){
       return count;
   } // end if
   else if (temp->next == NULL){
       count++;
       return count;
   } // end if
   else{
       while (temp != NULL){
           count++;
           temp = temp->next;
       } // end while
       return count;
   } // end else
} // end of qSize

// Enqueues new node and populates the data
// with T data as passed in.
template <class T>
void Queue<T>::Enqueue(const T data){
   if (tail == NULL){
   head = tail = new node;
   tail->next = NULL;
   tail->data = data;
   } // end if
   else {
       tail->next = new node;
       tail->next->data = data;
       tail->next->next = NULL;
       tail = tail->next;
   } // end else
} // end Enqueue

// Dequeues node and returns data stored
// in dequeued node.
template <class T>
T Queue<T>::Dequeue(){
   T returnData;
   returnData = head->data;
   temp = head->next;
   delete head;
   head = temp;
   tail = (!head? NULL: tail);
   return returnData;
}

// Checks if the queue is empty.
template <class T>
bool Queue<T>::isEmpty(){
   return (!tail);
}
struct thread_data{
      int thread_no;
      char* thread_name;
      WINDOW *thread_win;
      bool kill_signal;
      int sleep_time;
      int thread_results;
      bool state;
};
```

Scheduler.h

```
/*=================================================================================|
|   Assignment:     Ultima 2.0 Phase 2
|    File Name:     scheduler.h
|  Dependencies:    queue.h
|      Authors:     Drake Wood, James Giegerich
|     Language:     C++
|     Compiler:     G++
|        Class:     C435 - Operating Systems
|   Instructor:     Dr. Hakimzadeh
|  Date Created:    2/16/2019
|  Last Updated:    3/18/2019
|     Due Date:     3/18/2019
|=================================================================================|
|  Description: This is the header file which defines the class scheduler. There are 4 functions along with a
|                   constructor, deconstructor, and data structs.
|                   -dump displays information related to this class in a window for debugging.
|                   -garbage_collect finds threads that have been killed adn removes them from the process table.
|                   -create_task creates a new thread running simple output, its information is added to the process table.
|                   -destroy_task finds the task that the user wishes to kill and changes it status, stopping it.
|                   -TCB is the struct containing the thread_data which is stored in the process queue.
*=================================================================================*/


#ifndef SCHEDULER_H
#define SCHEDULER_H


#include <iostream>
#include <pthread.h>            // Needed for using the pthread library
#include <assert.h>
#include <time.h>
#include <unistd.h>             // Needed for sleep()
#include <ncurses.h>            // Needed for Curses windowing
#include <stdarg.h>             // Needed for formatted output to window
#include <termios.h>
```

```
#include <fcntl.h>

#include <cstdlib>

#include "queue.h"

#include <string>

#include "ipc.h"


class scheduler {

        public:

        struct TCB {

                pthread_t thread;

                std::string name;

                int thread_no;                          // Thread number

                WINDOW *thread_win;                     // Thread's window

                bool kill_signal;               // Kill signal flag live/kill

                int sleep_time;                         // Sleep time

                int thread_results;             // Results

                int state;

                int pause_state;

                Queue<ipc::Message*> mailbox;

                WINDOW *pdumpwin; // these are for the main input

                WINDOW *sdumpwin;

                WINDOW *conwin;

                WINDOW *logwin;

                WINDOW *messwin;

                char *input;

        };


        void run();

        scheduler();

        int task_counter;

        Queue <TCB*>process_table;

        int create_task(char* name, WINDOW *win, WINDOW *pdumpwin);             // Create appropriate data structures and
calls coroutine()

        int create_ui_task(WINDOW *pdumpwin, WINDOW *sdumpwin, WINDOW *conwin, WINDOW *logwin,
WINDOW * messwin);
```

```
        bool destroy_task(int tasknumber);                          // to kill a task (Set its status to DEAD)

        void yield();
                // Strict round robin process switch.

        void dump(int level, WINDOW * win);                                 // Debugging function with level
indicating the verbosity of the dump

        void garbage_collect();             // Include some functions which will allow you to dump the contents of the

        void pause();

        void unpause();

};
                                            // process table in a readable format.  See the expected output section

#endif
                        // (below) for suggestions.


                                            // remove dead task, free their resources, etc.
```

Scheduler.cpp

```
/*=====================================================================================|
|   Assignment:                    Ultima 2.0 Phase 2
|    File Name:                    scheduler.cpp
| Dependencies:       scheduler.h semaphore.h window.h
|      Authors:          Drake Wood, James Giegerich
|     Language:          C++
|     Compiler:          G++
|        Class:          C435 - Operating Systems
|   Instructor:          Dr. Hakimzadeh
| Date Created:          2/16/2019
| Last Updated:                    3/18/2019
|     Due Date:          3/18/2019
|=====================================================================================|
|  Description: Contains the definitions for the functions outlined in scheduler.h
*=====================================================================================*/

#include "semaphore.h"
#include "scheduler.h"
#include "window.h"

#define BLOCKED 0
#define READY 1
#define RUNNING 2
#define DEAD 3

extern semaphore sema_screen;
extern semaphore sema_ptable;

void scheduler::dump(int level, WINDOW * win) {
        sema_screen.down();
        wclear(win);
        sema_screen.up();
        write_window(win, 1, 5, "      PROCESS TABLE DUMP \n -------------------------------------\n");
        char buff[256];
        int procnum1;
        int size = process_table.qSize();

        if (size == 0)                                  // Check if anything is in the queue
                write_window(win, " No tasks currently running...\n");
        sema_ptable.down();
        for(int i = 0 ; i < size ; i++){                // Search for a dead task
                TCB *tcb = process_table.Dequeue();
                process_table.Enqueue(tcb);
                procnum1 = tcb->thread_no;

                if ( tcb->state == READY){
                        sprintf(buff, " Task #%d status Ready\n", procnum1);
                }
                else if (tcb->state == RUNNING){
                        sprintf(buff, " Task #%d status Running\n", procnum1);
                }
                else if (tcb->state == BLOCKED){
                        sprintf(buff, " Task #%d status Blocked\n", procnum1);
                }
                else if (tcb->state == DEAD){
                        sprintf(buff, " Task #%d status Dead\n", procnum1);
                }// end else
                write_window(win, buff);
        } // end for
        sema_ptable.up();
        write_window(win, " -------------------------------------\n");
}// end of dump

void scheduler::garbage_collect() {                     // Delete those with  dead status
        sema_ptable.down();
        int size = process_table.qSize();
        for(int i = 0 ; i < size ; i++){
                TCB *tcb = process_table.Dequeue();
                if (tcb->state != DEAD){
                        process_table.Enqueue(tcb);
                } // end if
        } // end for
        sema_ptable.up();
} // end of garbage_collect


int scheduler::create_task(char* name, WINDOW *win, WINDOW *pdumpwin){
        int result_code;
        TCB * tcb = new TCB;
        this->task_counter++;
        tcb->thread_win = win;
        tcb->pdumpwin = pdumpwin;
        write_window(tcb->thread_win, 13, 1, "Starting Thread.....\n");
        tcb->thread_no = this->task_counter;
        tcb->name = name;
        tcb->state = READY;

        // create thread running simple output in its own window
        result_code = pthread_create(&tcb->thread, NULL, perform_simple_output, tcb);
        assert(!result_code);                           // if there is any problems with result code. display it and end
program.
        sema_ptable.down();
        process_table.Enqueue(tcb);             // add to process table
        sema_ptable.up();
        return 0;
} // end of create_task


int scheduler::create_ui_task(WINDOW *pdumpwin, WINDOW *sdumpwin, WINDOW *conwin, WINDOW *logwin,
WINDOW * messwin){
        int result_code;
        TCB * tcb = new TCB;
        this->task_counter++;
        tcb->pdumpwin = pdumpwin;
        tcb->sdumpwin = sdumpwin;
        tcb->conwin = conwin;
        tcb->logwin = logwin;
        tcb->messwin = messwin;

        tcb->thread_no = this->task_counter;
        tcb->name = (char*) "UI Thread";
        tcb->state = READY;

        // create thread ui loop
        result_code = pthread_create(&tcb->thread, NULL, ui_loop, tcb);
        assert(!result_code);                   // if there is any problems with result code. display it and end
program.
        sema_ptable.down();
        process_table.Enqueue(tcb);         // add to process table
        sema_ptable.up();
        return 0;
}
scheduler::scheduler(){
        task_counter = 0;
}

bool scheduler::destroy_task(int tasknumber) {
        bool flag = false;
        sema_ptable.down();
        for( int i =0; i < process_table.qSize(); i++){
                if (!process_table.isEmpty()){                                  // this block
searches Q for a specific process
                        TCB *tcb = process_table.Dequeue();
                        process_table.Enqueue(tcb);                     // if it is found kill signal is
changed

                        if (tcb->thread_no == tasknumber){
```

```
                                        if (tcb->kill_signal == 0){                        // other processes are added back to Q
                                                tcb->kill_signal = 1;
                                                flag = true;
                                        }else {                 //already killed.
                                                //return 0;
                                        } // end else
                                } // end if
                        } // end if
                } // end for
                sema_ptable.up();
                return flag;
        } // end of destroy_task

        void scheduler::yield() { // Give scheduler the option to change state to ready or to continue running
                sema_ptable.down();
                // Change current thread state to READY.
                TCB * tcb = process_table.Dequeue();
                if (tcb->state != DEAD){ //dont ready if yielding after death
                        tcb->state = READY;
                }
                process_table.Enqueue(tcb);

                // Find a thread that is READY.
                tcb = process_table.Dequeue();
                while (tcb->state != READY){ // might get stuck here?
                        process_table.Enqueue(tcb);
                        tcb = process_table.Dequeue();
                }

                tcb->state = RUNNING;
                process_table.Enqueue(tcb);

                // Shuffle the queue to put runnning thread back at front of queue.
                for (int i = 0; i < (process_table.qSize() - 1); i++){
                        tcb = process_table.Dequeue();
                        process_table.Enqueue(tcb);
            }
sema_ptable.up();
}

void scheduler::run(){
            sema_ptable.down();
            TCB *tcb = process_table.Dequeue();
            tcb->state = RUNNING;
            process_table.Enqueue(tcb);

            // Shuffle the queue to put runnning thread back at front of queue.
            for (int i = 0; i < (process_table.qSize() - 1); i++){
                        tcb = process_table.Dequeue();
                        process_table.Enqueue(tcb);
            }
            sema_ptable.up();
}

void scheduler::pause(){
            sema_ptable.down();
            for (int i = 0; i < (process_table.qSize()); i++){
                        TCB * tcb = process_table.Dequeue();
                        tcb->pause_state = tcb->state;
                        tcb->state = BLOCKED;
                        process_table.Enqueue(tcb);
            }
            sema_ptable.up();
}

void scheduler::unpause(){
            sema_ptable.down();
            for (int i = 0; i < (process_table.qSize()); i++){
                        TCB * tcb = process_table.Dequeue();
                        if (tcb->state != DEAD){
                        tcb->state = tcb->pause_state;
                        }

                        process_table.Enqueue(tcb);
            }
            sema_ptable.up();
}
```

ipc.h

```
/*=================================================================================|
|   Assignment:     Ultima 2.0 Phase 2
|    File Name:     ipc.h
|  Dependencies:    none
|      Authors:     Drake Wood, James Giegerich
|     Language:     C++
|     Compiler:     G++
|        Class:     C435 - Operating Systems
|   Instructor:     Dr. Hakimzadeh
|  Date Created:    3/3/2019
|  Last Updated:    3/18/2019
|     Due Date:     3/18/2019
|=================================================================================|
|  Description: This is the header file which defines the class ipc.
|                         This class contains 7  funtions and a message structure.
|                         -Message_Send takes a message as parameter and delivers it to destination.
|                         -Message_Receive reads first message in mailbox for given task.
|                         -Message_Count (single) returns the number of messages for given task.
|                         -Message_Count (all) returns total number of messages in all mailboxes.
|                         -Message_Print prints all messages for given task.
|                         -Message_DeleteAll delets all messages for given task.
|                         -ipc_Message_Dump prints all messages for all mailboxes.
*=================================================================================*/

#ifndef IPC_H
#define IPC_H

 class ipc {
 public:
        struct Message {
                int Source_Task_Id;
                int Destination_Task_Id;
```

```
		time_t Message_Arrival_Time; // research time.h, time_t, and tm

		int   Message_Size;

		char *Message_Text;

		//etc.

	} ;

	int Message_Send(Message *message);					// returns -1 if error occurred. Return 1
if successful.

	int Message_Receive(int Task_id, Message *message);	// returns 0 if no more messages are available, loads the
Message structure with

				// the first message from the mailbox and remove the message from the mailbox.

				// Return -1 if an error occurs.

	int Message_Count(int Task_id);								// return the number
of messages in Task-id's  message queue.

	int Message_Count();
	// return the total number of messages in all the message queues.

	void Message_Print(int Task_id, WINDOW * Win);
	// print the all messages for a given Task-id.

	int Message_DeleteAll(int Task_id);							// delete all the messages for
Task_id

	void ipc_Message_Dump(WINDOW * Win);
	// print all the messages in the message queue, but do not delete them from the queue.

				// (note that this function may be best placed in the scheduler!)

};

#endif
```

ipc.cpp

```
/*==================================================================================|
|   Assignment:                     Ultima 2.0 Phase 2
|    File Name:                     ipc.cpp
| Dependencies:      ipc.h window.h queue.h scheduler.h semaphore.h
|      Authors:          Drake Wood, James Giegerich
|     Language:          C++
|     Compiler:          G++
|        Class:          C435 - Operating Systems
|   Instructor:          Dr. Hakimzadeh
| Date Created:          3/3/2019
| Last Updated:                     3/18/2019
|     Due Date:          3/18/2019
|==================================================================================|
|  Description: Contains the definitions for the functions outlined in ipc.h
*==================================================================================*/


#include "window.h"
#include "queue.h"
#include "scheduler.h"
#include "ipc.h"
#include "semaphore.h"


extern semaphore sema_screen;
extern semaphore sema_t1mail;
extern semaphore sema_t2mail;
extern semaphore sema_t3mail;
extern semaphore sema_t4mail;
extern semaphore sema_ptable;
extern scheduler sched;
extern scheduler TCB;


// Sends message using destination in the message which is passed
int ipc::Message_Send( Message *message){
```

```
// returns -1 if error occurred. Return 1 if successful.
        sema_ptable.down();
        int flag = -1;
        int dtID = message->Destination_Task_Id; // gets destination task
        scheduler::TCB * tcb;

        switch(dtID) // switch for using correct semaphore
        {
                case '1':
                        sema_t1mail.down();
                        break;
                case '2':
                        sema_t2mail.down();
                        break;
                case '3':
                        sema_t3mail.down();
                        break;
                case '4':
                        sema_t4mail.down();
                break;
        }



        for (int i = 0; i < sched.process_table.qSize(); i++){ // searches process table for the destination task
                tcb = sched.process_table.Dequeue();
                if (dtID == tcb->thread_no){


                        tcb->mailbox.Enqueue(message); // enqueues the message in the tasks mailbox
                        flag = 1;
                }
                sched.process_table.Enqueue(tcb);  //add back to process table.
        }
        switch(dtID) // switch for using correct semaphore
        {
                case '1':
                        sema_t1mail.up();
                        break;
                case '2':
                        sema_t2mail.up();
                        break;
                case '3':
                        sema_t3mail.up();
                        break;
                case '4':
                        sema_t4mail.up();
                break;
        }
sema_ptable.up();
        return flag;
}

int ipc::Message_Receive(int Task_id, Message *message){
// returns 0 if no more messages are available, loads the Message structure with
        sema_ptable.down();
        // the first message from the mailbox and remove the message from the mailbox.

                        // Return -1 if an error occurs.
        int flag = -1;
        int dtID = message->Destination_Task_Id;
        scheduler::TCB * tcb;

        switch(dtID)// switch for using correct semaphore
        {
                case '1':
                        sema_t1mail.down();
                        break;
                case '2':
                        sema_t2mail.down();
                        break;
                case '3':
                        sema_t3mail.down();
                        break;
                case '4':
                        sema_t4mail.down();
                break;
        }

        for (int i = 0; i < sched.process_table.qSize(); i++){
                tcb = sched.process_table.Dequeue();
                if (dtID == tcb->thread_no){ // finds task
                        if (!tcb->mailbox.isEmpty()){ // gets first message from its mailbox
                        message = tcb->mailbox.Dequeue();
                        flag = 1; // message found
                        }
                        else{
                                flag = 0; // message not found
                        }
                }
                sched.process_table.Enqueue(tcb);  //add back to process table.
        }

                switch(dtID)// switch for using correct semaphore
        {
                case '1':
                        sema_t1mail.up();
                        break;
                case '2':
                        sema_t2mail.up();
                        break;
                case '3':
                        sema_t3mail.up();
                        break;
                case '4':
```

```
                        sema_t4mail.up();
                break;
        }
        sema_ptable.up();
        return flag;
}


int ipc::Message_Count(int Task_id){                    // return the number of messages in Task-id’s  message queue.
        sema_ptable.down();
        int count = 0;
        scheduler::TCB * tcb;

        for (int i = 0; i < sched.process_table.qSize(); i++){
                tcb = sched.process_table.Dequeue(); // find task
                if (Task_id == tcb->thread_no){
                        count = tcb->mailbox.qSize(); // gets number of messages
                }
                sched.process_table.Enqueue(tcb);  //add back to process table.
        }

        sema_ptable.up();
        return count;
}

int ipc::Message_Count(){                               // return the total number of messages in all the message queues.
        sema_ptable.down();
        int count = 0;
        scheduler::TCB * tcb;

        for (int i = 0; i < sched.process_table.qSize(); i++){ // cycle all tasks
                tcb = sched.process_table.Dequeue();
                count += tcb->mailbox.qSize(); // add up number of messages
                sched.process_table.Enqueue(tcb);                           //add back to process table.
```

```
		}
		sema_ptable.up();
		return count;
}


void ipc::Message_Print(int Task_id, WINDOW * Win){											// print
the all messages for a given Task-id.
		sema_ptable.down();
		scheduler::TCB * tcb;
		char buff[256];
		Message * mess;
		struct tm * timeinfo; // create the time data struct for extracting the format from the Message_Arrival_Time


		for (int i = 0; i < sched.process_table.qSize(); i++){
			tcb = sched.process_table.Dequeue();
			if (Task_id == tcb->thread_no){


			switch(Task_id)// switch for using correct semaphore
		{
			case '1':
				sema_t1mail.down();
				break;
			case '2':
				sema_t2mail.down();
				break;
			case '3':
				sema_t3mail.down();
				break;
			case '4':
				sema_t4mail.down();
			break;
		}


				sprintf(buff, " Time\t\tSize\tContent\t\tDestination\tSource\n"); // collumn names for message
printing
```

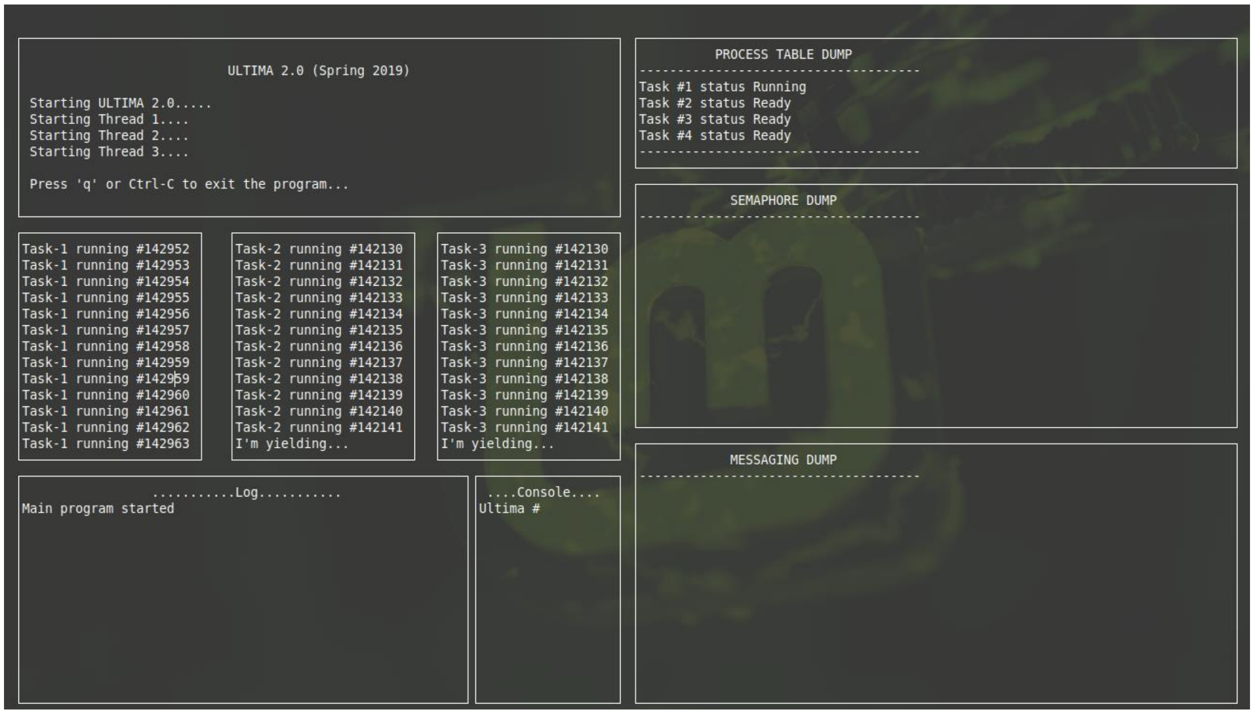
```
                    write_window(Win, buff);
                    for (int j = 0; j < tcb->mailbox.qSize(); j++){
                              mess = tcb->mailbox.Dequeue();
                              timeinfo = localtime (&mess->Message_Arrival_Time); // change time to local time
                                                                                   // and store in the new struct

                              // formatting can be found in time.h for day month year ect.
sprintf(buff, " %d:%d:%d\t%d\t%s\t%d\t\t%d \n", timeinfo->tm_hour,timeinfo->tm_min,timeinfo->tm_sec,
          mess->Message_Size, mess->Message_Text, mess->Destination_Task_Id, mess->Source_Task_Id);
                              write_window(Win, buff);
                    // single line output of all message data
                              tcb->mailbox.Enqueue(mess);
                    }
          }
          sched.process_table.Enqueue(tcb);  //add back to process table.

          switch(Task_id)// switch for using correct semaphore
     {
          case '1':
                    sema_t1mail.up();
                    break;
          case '2':
                    sema_t2mail.up();
                    break;
          case '3':
                    sema_t3mail.up();
                    break;
          case '4':
                    sema_t4mail.up();
          break;
     }
     }
     sema_ptable.up();
}
```

```
int ipc::Message_DeleteAll(int Task_id){                                    // delete all the messages for Task_id
        sema_ptable.down();
        scheduler::TCB * tcb;

        for (int i = 0; i < sched.process_table.qSize(); i++){
                tcb = sched.process_table.Dequeue();
                if (Task_id == tcb->thread_no){

                switch(Task_id)// switch for using correct semaphore
        {
                case '1':
                        sema_t1mail.down();
                        break;
                case '2':
                        sema_t2mail.down();
                        break;
                case '3':
                        sema_t3mail.down();
                        break;
                case '4':
                        sema_t4mail.down();
                break;
        }

                        while (!tcb->mailbox.isEmpty()){
                                tcb->mailbox.Dequeue();
                        }

        switch(Task_id)// switch for using correct semaphore
        {
                case '1':
                        sema_t1mail.up();
                        break;
                case '2':
                        sema_t2mail.up();
                        break;
                case '3':
                        sema_t3mail.up();
                        break;
                case '4':
                        sema_t4mail.up();
                break;
        }

                }
                sched.process_table.Enqueue(tcb);  //add back to process table.
        }
sema_ptable.up();
        return 1;
}

void ipc::ipc_Message_Dump(WINDOW * Win){
// print all the messages in the message queue, but do not delete them from the queue.

        for (int i = 1; i < 5; i++){
                        Message_Print(i, Win);
        }
}
```

Makefile

```
# Make file for Ultima 2.0

# Drake Wood, James Giegerich


# Variables to control Makefile operation


CXX = g++

LINKS = -lpthread -lncurses

CXXFLAGS = -Wall -g


# Targets needed to bring the executable up to date


main: main.o scheduler.o semaphore.o window.o ipc.o

	$(CXX) $(CXXFLAGS) -o main main.o scheduler.o semaphore.o window.o ipc.o $(LINKS)


main.o: main.cpp scheduler.h semaphore.h queue.h window.h

	$(CXX) $(CXXFLAGS) -c main.cpp $(LINKS)


window.o: window.h window.cpp semaphore.h ipc.h

	$(CXX) $(CXXFLAGS) -c window.cpp $(LINKS)


scheduler.o: scheduler.h scheduler.cpp queue.h window.h

	$(CXX) $(CXXFLAGS) -c scheduler.cpp $(LINKS)


semaphore.o: semaphore.h semaphore.cpp queue.h window.h

	$(CXX) $(CXXFLAGS) -c semaphore.cpp $(LINKS)


ipc.o: ipc.h ipc.cpp scheduler.h queue.h window.h

		$(CXX) $(CXXFLAGS) -c ipc.cpp $(LINKS)

clean:

	rm *.o
```

Output



```
ULTIMA 2.0 (Spring 2019)

Starting ULTIMA 2.0.....
Starting Thread 1....
Starting Thread 2....
Starting Thread 3....

Press 'q' or Ctrl-C to exit the program...
```

```
Task-1 running #142952
Task-1 running #142953
Task-1 running #142954
Task-1 running #142955
Task-1 running #142956
Task-1 running #142957
Task-1 running #142958
Task-1 running #142959
Task-1 running #142959
Task-1 running #142960
Task-1 running #142961
Task-1 running #142962
Task-1 running #142963
```

```
Task-2 running #142130
Task-2 running #142131
Task-2 running #142132
Task-2 running #142133
Task-2 running #142134
Task-2 running #142135
Task-2 running #142136
Task-2 running #142137
Task-2 running #142138
Task-2 running #142139
Task-2 running #142140
Task-2 running #142141
I'm yielding...
```

```
Task-3 running #142130
Task-3 running #142131
Task-3 running #142132
Task-3 running #142133
Task-3 running #142134
Task-3 running #142135
Task-3 running #142136
Task-3 running #142137
Task-3 running #142138
Task-3 running #142139
Task-3 running #142140
Task-3 running #142141
I'm yielding...
```

```
...........Log..........
Main program started
```

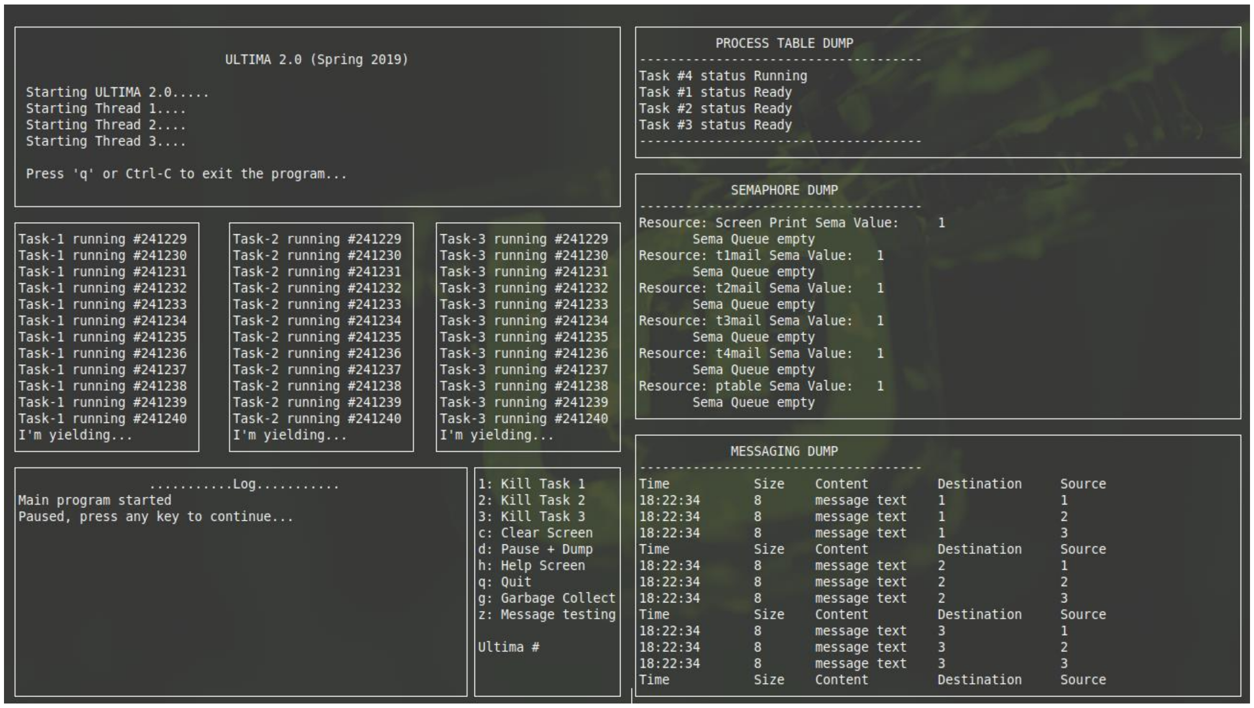
```
....Console....
Ultima #
```

```
         PROCESS TABLE DUMP
-------------------------------------
Task #1 status Running
Task #2 status Ready
Task #3 status Ready
Task #4 status Ready
-------------------------------------
```

```
           SEMAPHORE DUMP
-------------------------------------
```

```
           MESSAGING DUMP
-------------------------------------
```

Main screen showing processes yielding to each other in order with the process table keeping track of which is currently running. The semaphore and message dump windows are left blank until dump is called in the next screenshot.



```
ULTIMA 2.0 (Spring 2019)

Starting ULTIMA 2.0.....
Starting Thread 1....
Starting Thread 2....
Starting Thread 3....

Press 'q' or Ctrl-C to exit the program...
```

```
Task-1 running #241229
Task-1 running #241230
Task-1 running #241231
Task-1 running #241232
Task-1 running #241233
Task-1 running #241234
Task-1 running #241235
Task-1 running #241236
Task-1 running #241237
Task-1 running #241238
Task-1 running #241239
Task-1 running #241240
I'm yielding...
```

```
Task-2 running #241229
Task-2 running #241230
Task-2 running #241231
Task-2 running #241232
Task-2 running #241233
Task-2 running #241234
Task-2 running #241235
Task-2 running #241236
Task-2 running #241237
Task-2 running #241238
Task-2 running #241239
Task-2 running #241240
I'm yielding...
```

```
Task-3 running #241229
Task-3 running #241230
Task-3 running #241231
Task-3 running #241232
Task-3 running #241233
Task-3 running #241234
Task-3 running #241235
Task-3 running #241236
Task-3 running #241237
Task-3 running #241238
Task-3 running #241239
Task-3 running #241240
I'm yielding...
```

```
...........Log..........
Main program started
Paused, press any key to continue...
```

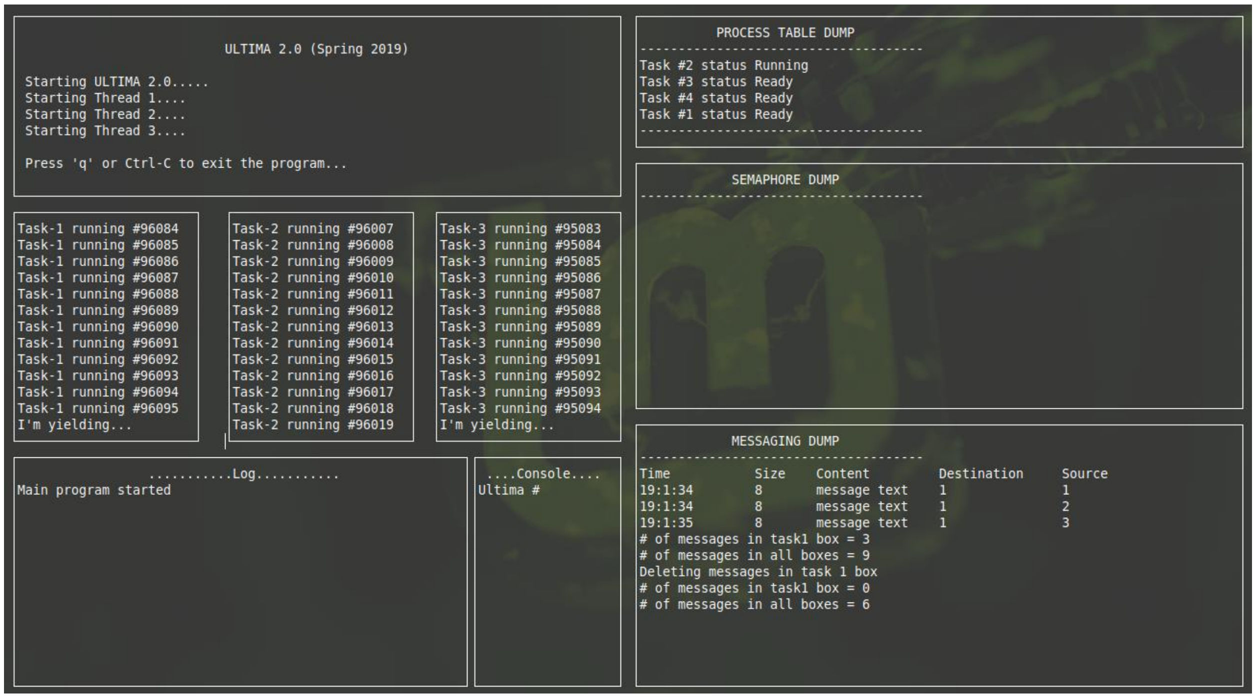
```
1: Kill Task 1
2: Kill Task 2
3: Kill Task 3
c: Clear Screen
d: Pause + Dump
h: Help Screen
q: Quit
g: Garbage Collect
z: Message testing

Ultima #
```

```
         PROCESS TABLE DUMP
-------------------------------------
Task #4 status Running
Task #1 status Ready
Task #2 status Ready
Task #3 status Ready
-------------------------------------
```

```
           SEMAPHORE DUMP
-------------------------------------
Resource: Screen Print Sema Value:      1
       Sema Queue empty
Resource: t1mail Sema Value:    1
       Sema Queue empty
Resource: t2mail Sema Value:    1
       Sema Queue empty
Resource: t3mail Sema Value:    1
       Sema Queue empty
Resource: t4mail Sema Value:    1
       Sema Queue empty
Resource: ptable Sema Value:    1
       Sema Queue empty
```

```
           MESSAGING DUMP
-------------------------------------
Time           Size    Content         Destination     Source
18:22:34       8       message text    1               1
18:22:34       8       message text    1               2
18:22:34       8       message text    1               3
Time           Size    Content         Destination     Source
18:22:34       8       message text    2               1
18:22:34       8       message text    2               2
18:22:34       8       message text    2               3
Time           Size    Content         Destination     Source
18:22:34       8       message text    3               1
18:22:34       8       message text    3               2
18:22:34       8       message text    3               3
Time           Size    Content         Destination     Source
```

Here is the dump function, each process sends a message to the others and itself and they are displayed.



```
                    ULTIMA 2.0 (Spring 2019)

 Starting ULTIMA 2.0.....
 Starting Thread 1....
 Starting Thread 2....
 Starting Thread 3....

 Press 'q' or Ctrl-C to exit the program...
```

```
Task-1 running #96084    Task-2 running #96007    Task-3 running #95083
Task-1 running #96085    Task-2 running #96008    Task-3 running #95084
Task-1 running #96086    Task-2 running #96009    Task-3 running #95085
Task-1 running #96087    Task-2 running #96010    Task-3 running #95086
Task-1 running #96088    Task-2 running #96011    Task-3 running #95087
Task-1 running #96089    Task-2 running #96012    Task-3 running #95088
Task-1 running #96090    Task-2 running #96013    Task-3 running #95089
Task-1 running #96091    Task-2 running #96014    Task-3 running #95090
Task-1 running #96092    Task-2 running #96015    Task-3 running #95091
Task-1 running #96093    Task-2 running #96016    Task-3 running #95092
Task-1 running #96094    Task-2 running #96017    Task-3 running #95093
Task-1 running #96095    Task-2 running #96018    Task-3 running #95094
I'm yielding...          Task-2 running #96019    I'm yielding...
```

```
          ...........Log...........
Main program started
```

```
 ....Console....
Ultima #
```

```
          PROCESS TABLE DUMP
-------------------------------------
Task #2 status Running
Task #3 status Ready
Task #4 status Ready
Task #1 status Ready
-------------------------------------
```

```
            SEMAPHORE DUMP
-------------------------------------
```

```
            MESSAGING DUMP
-------------------------------------
Time           Size    Content         Destination     Source
19:1:34        8       message text    1               1
19:1:34        8       message text    1               2
19:1:35        8       message text    1               3
# of messages in task1 box = 3
# of messages in all boxes = 9
Deleting messages in task 1 box
# of messages in task1 box = 0
# of messages in all boxes = 6
```

An option z was added for input which ran a series of functions for testing messaging. First the entire mailbox for task 1 was displayed, like before it has 3 messages. The functions for single task count and all task count are called followed by the deletion of all task 1's messages. You can see when the message count functions are called afterwards that they all perform correctly.