ULTIMA 2.0

CS-435 Operating System

Phase 3—Memory Management

By:

Drake Wood

drawood@iu.edu

James Giegerich

jgiegeri@iu.edu

4/8/19

Table of Contents

Phase Abstract

This report details the design and development of the third 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.

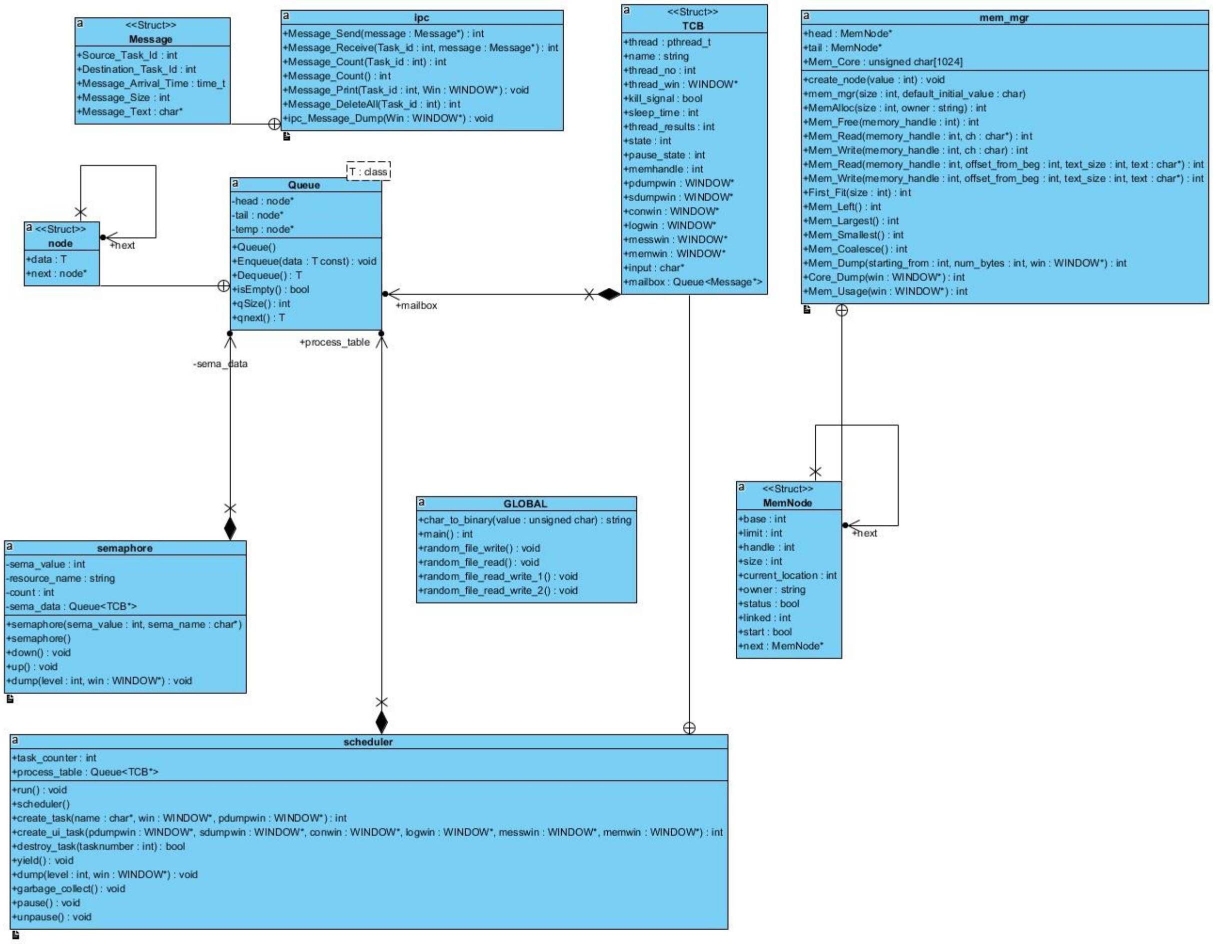
Phase Description

This phase is to develop and implement a simulated a memory management system for the threads in our simulated operating system. One task must be able to write and read to memory according to a first-fit algorithm. Memory is a linked list of eight MemNode structs of 128 “bits” that add up to a total memory cap of 1024 “bits”. The mem_mgr class contains the MemNode struct, which includes type int data members base, limit, handle, linked and current_location, as well as a string type owner, two bool type status and start, and then a MemNode pointer next. Class mem_mgr methods include a constructor, a create_node(), MemAlloc(), Mem_Free(), Mem_Read() (including an overloaded version), Mem_Write() (including an overloaded version), First_Fit(), Mem_Left(), Mem_Smallest(), Mem_Coalesce(), Mem_Dump(), Core_Dump(), Mem_Usage(). Lastly, an array of char called Mem_Core[] is included to contain the actual contents of the memory space, whether written to, empty, or freed.

- Each time a thread is set to RUN, it writes a single random character to the first available memory block it finds, and that section of memory is “owned” by that thread so that other threads cannot write to it.
- Before the scheduler tells a thread to yield, it then reads the first character in the memory block that it owns.
- Once the character is read, it can be overwritten by the next Mem_write() call as long as that call is performed by the owning thread.
- Once a thread is killed, its memory is “freed” by replacing any characters within the block with # signs.
- Other methods are included to find the size of the smallest and largest blocks of free memory, as well as how much total memory is free in the whole system.

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

Design Diagram



<<Struct>>
Message
+Source_Task_Id : int
+Destination_Task_Id : int
+Message_Arrival_Time : time_t
+Message_Size : int
+Message_Text : char*
ipc
+Message_Send(message : Message*) : int
+Message_Receive(Task_id : int, message : Message*) : int
+Message_Count(Task_id : int) : int
+Message_Count() : int
+Message_Print(Task_id : int, Win : WINDOW*) : void
+Message_DeleteAll(Task_id : int) : int
+ipc_Message_Dump(Win : WINDOW*) : void
<<Struct>>
TCB
+thread : pthread_t
+name : string
+thread_no : int
+thread_win : WINDOW*
+kill_signal : bool
+sleep_time : int
+thread_results : int
+state : int
+pause_state : int
+memhandle : int
+pdumpwin : WINDOW*
+sdumpwin : WINDOW*
+conwin : WINDOW*
+logwin : WINDOW*
+messwin : WINDOW*
+memwin : WINDOW*
+input : char*
+mailbox : Queue<Message*>
mem_mgr
+head : MemNode*
+tail : MemNode*
+Mem_Core : unsigned char[1024]
+create_node(value : int) : void
+mem_mgr(size : int, default_initial_value : char)
+MemAlloc(size : int, owner : string) : int
+Mem_Free(memory_handle : int) : int
+Mem_Read(memory_handle : int, ch : char*) : int
+Mem_Write(memory_handle : int, ch : char) : int
+Mem_Read(memory_handle : int, offset_from_beg : int, text_size : int, text : char*) : int
+Mem_Write(memory_handle : int, offset_from_beg : int, text_size : int, text : char*) : int
+First_Fit(size : int) : int
+Mem_Left() : int
+Mem_Largest() : int
+Mem_Smallest() : int
+Mem_Coalesce() : int
+Mem_Dump(starting_from : int, num_bytes : int, win : WINDOW*) : int
+Core_Dump(win : WINDOW*) : int
+Mem_Usage(win : WINDOW*) : int
<<Struct>>
node
+data : T
+next : node*
+next
T : class
Queue
-head : node*
-tail : node*
-temp : node*
+Queue()
+Enqueue(data : T const) : void
+Dequeue() : T
+isEmpty() : bool
+qSize() : int
+qnext() : T
+mailbox
+process_table
-sema_data
GLOBAL
+char_to_binary(value : unsigned char) : string
+main() : int
+random_file_write() : void
+random_file_read() : void
+random_file_read_write_1() : void
+random_file_read_write_2() : void
<<Struct>>
MemNode
+base : int
+limit : int
+handle : int
+size : int
+current_location : int
+owner : string
+status : bool
+linked : int
+start : bool
+next : MemNode*
+next
semaphore
-sema_value : int
-resource_name : string
-count : int
-sema_data : Queue<TCB*>
+semaphore(sema_value : int, sema_name : char*)
+semaphore()
+down() : void
+up() : void
+dump(level : int, win : WINDOW*) : void
scheduler
+task_counter : int
+process_table : Queue<TCB*>
+run() : void
+scheduler()
+create_task(name : char*, win : WINDOW*, pdumpwin : WINDOW*) : int
+create_ui_task(pdumpwin : WINDOW*, sdumpwin : WINDOW*, conwin : WINDOW*, logwin : WINDOW*, messwin : WINDOW*, memwin : WINDOW*) : int
+destroy_task(tasknumber : int) : bool
+yield() : void
+dump(level : int, win : WINDOW*) : void
+garbage_collect() : void
+pause() : void
+unpause() : void

Source Code

Main.cpp

```
/*==================================================================================|
|   Assignment:   Ultima 2.0 Phase 3
|    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:  4/08/2019
|     Due Date:     4/08/2019
|==================================================================================|
|  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"
#include "memory.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");
semaphore sema_memory(1, (char *)"memory");
scheduler sched; //creates scheduler
ipc IPC; // creates ipc
mem_mgr Mem_Mgr(1,1); // creates mem manager - size, initial value - dont do anything right now
```

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

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

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

	WINDOW * Heading_Win = newwin(5, 80, 0, 2);
	box(Heading_Win, 0,0);
	mvwprintw(Heading_Win, 1, 26, "ULTIMA 2.0 Phase 3 (Spring 2019)");
	mvwprintw(Heading_Win, 3, 28, "Drake Wood		James Giegerich");
	wrefresh(Heading_Win);

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

	WINDOW * Log_Win = create_window(15, 60, 20, 2);
	write_window(Log_Win, 1, 18, "...........Log...........\n");

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

	WINDOW * Console_Win = create_window(15, 20, 20, 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, 5, 2);
	WINDOW * W2 = create_window(15, 25, 5, 30);
	WINDOW * W3 = create_window(15, 25, 5, 57);

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

	WINDOW * Process_Table = create_window(9, 80, 0, 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, 9, 83);
		write_window(Sema_Dump, 1, 5, "       SEMAPHORE DUMP \n ------------------------------------\n");

//--------------------------------------------------------------------------------
//Creating a memory dump window
//--------------------------------------------------------------------------------

		WINDOW * Mem_Dump = create_window(22, 80, 35, 2);
		write_window(Mem_Dump, 1, 5, "                MEMORY DUMP \n -----------------------------------------------------
--\n");

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

		WINDOW * Message_Dump = create_window(17, 80, 25, 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.
                                                                          // If no input is ready,
getch returns ERR

//-------------------------------------------------------------------------------
//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,
Mem_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 3
|    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:   4/08/2019
|==================================================================================|
|  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.  See the expected
                                                                            // Output section (below)
for suggestions.
```

Semaphore.cpp

```
/*================================================================================|
|   Assignment:   Ultima 2.0 Phase 3
|    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:  4/08/2019
|     Due Date:   4/08/2019
|================================================================================|
|  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 3
|    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:     4/08/2019
|==================================================================================|
|  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 3
|    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:  4/08/2019
|     Due Date:   4/08/2019
|==================================================================================|
| Description: Contains the definitions for the functions outlined in window.h
*==================================================================================*/

#include "window.h"
#include "semaphore.h"
#include "ipc.h"
#include "time.h"
#include "memory.h"
#include <cstring>

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;
extern mem_mgr Mem_Mgr;

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;
        char write;
        char read = '=';
        unsigned seed= time(0);
   srand(seed);



        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);

                //writes a random piece of data to memory at the beginning of yield cycle
                if(yield_quantum == 1){
                    write = '0' + rand()%77;
                    Mem_Mgr.Mem_Write(tcb->memhandle,write); // write to memory once per yield cycle
                }


            }
            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
                    // reads 1 ch of memory and prints at the end of yield cycle
                    if(Mem_Mgr.Mem_Read(tcb->memhandle,&read)){
                        sprintf(buff," Reading from memory... \n    %c\n", read);
                        write_window(Win, buff);
                    }
                    write_window(Win, " I'm yielding...\n");
                    sched.yield();
                    //sleep(1);
                }
            }

        } // end while
        Mem_Mgr.Mem_Free(tcb->memhandle);
    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;
    WINDOW * memwin = tcb->memwin;
    char* text= "This example shows the overloaded write/read function";
    int textsize = strlen(text);
    char* read = "reading didnt work";
    char buff[256];

        while (tcb->state != 3){ // not dead
        while(tcb->state == 2) { // running
            // updates process table when thread gets cpu time, removed for now
            //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 and Coalesce

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

                Mem_Mgr.Mem_Coalesce();

                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);

                //core dump
                Mem_Mgr.Core_Dump(memwin);
                sprintf(buff, " memory largest: %d smallest: %d left: %d \n",
Mem_Mgr.Mem_Largest(), Mem_Mgr.Mem_Smallest(), Mem_Mgr.Mem_Left());
                write_window(logwin, buff);

                // overloaded read does not work
                /*
                if(Mem_Mgr.Mem_Read(tcb->memhandle,9, 20, &read)){
                        sprintf(buff," Reading from memory... \n    %s\n", read);
                        write_window(logwin, buff);
                    }
                */

                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 and mem usage

                display_help(conwin);

                Mem_Mgr.Mem_Usage(memwin);

                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);


                Mem_Mgr.Mem_Write(tcb->memhandle, 9, textsize, text);

                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");
                    Mem_Mgr.Mem_Free(tcb->memhandle);
```

```
                    // free this memory if thread is dying
                    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 3
|    File Name:    queue.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:      4/08/2019
|==================================================================================|
|  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 3
|    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:     4/08/2019
|==================================================================================|
|  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);
}
```

Scheduler.h

```
/*===============================================================================|
|   Assignment:   Ultima 2.0 Phase 3
|    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:     4/08/2019
|===============================================================================|
|  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;
                int memhandle;                          // Memory handle
                Queue<ipc::Message*> mailbox;
                WINDOW *pdumpwin; // these are for the main input
                WINDOW *sdumpwin;
                WINDOW *conwin;
                WINDOW *logwin;
                WINDOW *messwin;
                WINDOW *memwin;
                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, WINDOW * memwin);
        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 3
|   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:             4/08/2019
|    Due Date:      4/08/2019
|=====================================================================================|
|  Description: Contains the definitions for the functions outlined in scheduler.h
*=====================================================================================*/

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

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

extern mem_mgr Mem_Mgr;
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 memhandle;
		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;
			memhandle = tcb->memhandle;


			if ( tcb->state == READY){
```

```
                        sprintf(buff, " Task #%d status: Ready\t\tMemory Handle: %d\n", procnum1,
memhandle);
                }
                else if (tcb->state == RUNNING){
                        sprintf(buff, " Task #%d status: Running\tMemory Handle: %d\n", procnum1,
memhandle);
                }
                else if (tcb->state == BLOCKED){
                        sprintf(buff, " Task #%d status: Blocked\t\tMemory Handle: %d\n", procnum1,
memhandle);
                }
                else if (tcb->state == DEAD){
                        sprintf(buff, " Task #%d status: Dead\t\tMemory Handle: %d\n", procnum1,
memhandle);
                }// 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;

        tcb->memhandle = Mem_Mgr.MemAlloc(128, tcb->name);
        if (tcb->memhandle == -1){
                tcb->state = BLOCKED; // block if mem alloc fails
                //will cause  seg fault
        }


        // 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, WINDOW * memwin){
	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->memwin = memwin;

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

	// get memory
	char buff[256];
	tcb->memhandle = Mem_Mgr.MemAlloc(128, tcb->name);  // ask for 128 bits
	if (tcb->memhandle == -1){
		tcb->state = BLOCKED; // block if mem alloc fails
		// will cause seg fault
	}

	// 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 3
|    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:      4/08/2019
|===============================================================================|
|  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-idÃ¢â,¬â„¢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 3
|   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:      4/08/2019
|==================================================================================|
|  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-idÃ¢â‚¬â„¢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);
        }
}
```

Memory.h

```
/*==============================================================================|
|   Assignment:   Ultima 2.0 Phase 3
|    File Name:    memory.h
|  Dependencies: string 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:  4/08/2019
|     Due Date:     4/08/2019
|==============================================================================|
|  Description: This is the header file which defines the class mem_mgr.
                                        -MemNode is the structure that contains information for each block of
memory.
                                        -create_node creates a node with given size, must be called multiple times.
                                        -mem_mgr constructor for the core memory.
                                        -MemAlloc Called by a task to ask for memory to be assigned.
                                        -Mem_Free Clears memory when task is dead.
                                        -Mem_Read Returns memory and can be overloaded.
                                        -Mem_Write Writes to memory and can be overloaded.
                                        -First_Fit Returns the mem handle for the first fit for requested size.
                                        -Mem_Left Returns how much memory is left.
                                        -Mem_Largest Returns the largest piece of free memory left.
                                        -Mem_Smallest Returns the smalles piece of free memory left.
                                        -Mem_Coalesce Combines free memory nodes into a larger block.
                                        -Mem_Dump Dumps a specific memory segment.
                                        -Core_Dump Dumps entire core memory.
                                        -Mem_Usage Reports memory usage.
*==============================================================================*/

#ifndef MEMORY_H
#define MEMORY_H
#include <string>
#include "window.h"


class mem_mgr{
          public:
                    struct MemNode{
                              int base;
                              int limit;
                              int handle;
                              int size;
                              int current_location;
                              std::string owner;
                              MemNode * next;
                              bool status; // 0 is a hole, 1 is a process
                              int linked; // 0 is unlinked/end of link, 1+ is linked to the node with that handle going
forward
```

```
                        bool start; // 0 is not start of a link, 1 is start of a link
                };
        MemNode * head;
        MemNode * tail;

        void create_node(int value);  // creates and initializes Memory Node.
        mem_mgr(int size, char default_initial_value); // allocate 1024 unsigned chars and initialize the entire
memory with . dots
        int MemAlloc(int size, std::string owner); // returns a unique integer memory_handle or -1 if not enough
memory is available. set the current_location for this memory segment (beginning of the allocated area
        int Mem_Free(int memory_handle); // place #'s in the memory freed, return -1 if errors occur
        int Mem_Read(int memory_handle, char *ch); // read a character from current location in memory and
bring it back in ch, return a -1 if at end of bounds, keep track of the current location or the location next char to be
read.
        int Mem_Write(int memory_handle, char ch); // write a character to the current location in memory,
return a -1 if at end of bounds.
                                                                                                  // overloaded
multi-byte read and write
        int Mem_Read(int memory_handle, int offset_from_beg, int text_size, char *text);
        int Mem_Write(int memory_handle, int offset_from_beg, int text_size, char *text);
        int First_Fit(int size);// given a desired size will return the first node/link handle that can provide enough
space.

        unsigned char Mem_Core[1024];

        //private:
        int Mem_Left(); // return the amount of core memory left in the OS
        int Mem_Largest(); // return the size of the largest available memory segment
        int Mem_Smallest(); // return the size of the smallest available memory segment
        int Mem_Coalesce(); // combine two or more contiguous blocks of free space and place . dots in the
coalesced memory.
        int Mem_Dump(int starting_from, int num_bytes, WINDOW * win); // dump the contents of memory
        int Core_Dump(WINDOW * win);
        int Mem_Usage(WINDOW * win);
};

#endi
```

Memory.cpp

```
/*==================================================================================|
|    Assignment:   Ultima 2.0 Phase 3
|     File Name:   memory.cpp
|  Dependencies:   memory.h semaphore.h cstring
|       Authors:   Drake Wood, James Giegerich
|      Language:   C++
|      Compiler:   G++
|         Class:   C435 - Operating Systems
|    Instructor:   Dr. Hakimzadeh
|  Date Created:   2/16/2019
|  Last Updated:   4/08/2019
|      Due Date:   4/08/2019
|==================================================================================|
|  Description: Contains the definitions for the functions outlined in memory.h
*==================================================================================*/

#include "memory.h"
#include "semaphore.h"
#include <cstring>

extern semaphore sema_memory;
extern semaphore sema_screen;

mem_mgr::mem_mgr(int size, char default_initial_value){// allocate 1024 unsigned chars and initialize the entire
memory with . dots

    for (int i = 1; i < 9; i++){
        create_node(i);
    }

    for (int i = 0; i < 1024; i++){
        Mem_Core[i] = '.';
    }
}

void mem_mgr::create_node(int value){

    MemNode * temp = new MemNode;
    temp->handle = value;
    temp->limit = temp->handle * 128 - 1;
    temp->base = temp->limit - 127;
    temp->owner = "none";
    temp->status = 0;
    temp->size = 128;
    if (value != 8){
        temp->linked = value + 1; // linked to the next node
    }else{
        temp->linked = 0; // last node wont have anything linked to it
    }
    if (value = 1){
        temp->start = 1; // first node will start the linked nodes for the unused space
    }else{
        temp->start = 0; // // other nodes will not be a starting point
    }

    temp->current_location = temp->base; // set the CL to the first point of the node

    if (head == NULL){
        head = temp;
        tail = temp;
        temp = NULL;
    }
    else {
        tail->next = temp;
        tail = temp;
    }
}

int mem_mgr::MemAlloc(int size, std::string owner){// returns a unique integer memory_handle or -1 if not
enough memory is available. set the current_location for this memory segment (beginning of the allocated area
    int handle = First_Fit(size);
    sema_memory.down(); // calling this after first fit because it uses sema_memory also
    int count = 128; // temp->size should go here

    MemNode * temp = this->head;
    while (temp->handle != handle){ // get us to the right node
        temp = temp->next;
    }

    temp->status = 1; //set status to allocated
    temp->owner = owner; // assign the owner
    temp->current_location = temp->base; // se the CL to the first point of the node.
    temp->start = 1;

    while (size > count){ // get and link additional nodes if we need more space, we know these are available
from first_fit()
        temp = temp->next;
        temp->status = 1; //set status to allocated
        temp->owner = owner; // assign the owner
        temp->current_location = temp->base; // se the CL to the first point of the node.
        count += 128; // temp->size should go here
    }

    if (temp->linked){ // if this last node is linked then we need to unlink and set the next node to the start of
a link
                temp->linked = 0;
                temp = temp->next;
                temp->start = 1;
            }
    sema_memory.up();
    return handle;
}
```

```
int mem_mgr::First_Fit(int size){ // given a desired size will return the first node/link handle that can provide
enough space.
        sema_memory.down();
        int handle;
        int tempcount = 0;
        MemNode* temp = this->head;
        while(temp){
                tempcount = 0;
                if (temp->status == 0){ // find a hole while is also the first node
                        handle = temp->handle; // record handle in case this one ends up being big enough
                        tempcount = tempcount + 128; // keep track of size // temp->size should go here

                        while (temp->linked){
                                temp = temp->next;
                                tempcount = tempcount + 128; // find the full size of the hole // temp->size
should go here
                        }
                }
                if (tempcount >= size){ // if the hole is large enough return the handle
                        sema_memory.up();
                        return handle;
                }
                temp = temp->next; // check next node if needed.
        }
        sema_memory.up();
        return -1;
}

int mem_mgr::Mem_Free(int memory_handle){// place #'s in the memory freed, return -1 if errors occur
                                                                                        // only use this on the first
node/handle in a link
sema_memory.down();
        int runagain = 0;
        MemNode* temp = this->head;

        while (temp->handle != memory_handle){ // get us to the right node
                temp = temp->next;
        }



        if (temp->start == 1){ // must be used on start of memory
                do{
                        for(int i = temp->base; i < temp->limit + 1; i++){
                                Mem_Core[i] = '#'; // clear out all left over data
                        }
                        // reset attributes
                        temp->status = 0;
                        temp->owner = "none";
                        temp->current_location = temp->base;

                        if(temp->linked){
                                temp = temp->next; // move to next node and run again
                                runagain = 1;
                }else{
                    runagain = 0; // end
                }

            }while(runagain); // make sure  to clear out all linked nodes as well

            sema_memory.up();
            return 1;
        }
        sema_memory.up();
        return -1;

}

// does not move back to previous linked node
int mem_mgr::Mem_Read(int memory_handle, char *ch){// read a character from current location in memory and
bring it back in ch, return a -1 if at end of bounds, keep track of the current location or the location next char to be
read.
        sema_memory.down();
        MemNode* temp = this->head;
        while (temp->handle != memory_handle){ // get us to the right node
            temp = temp->next;
        }

        while((temp->current_location > temp->limit) && (temp->linked > 0)){
                    temp = temp->next; // if the head is full but has a linked node then move to
that
                }

        if (temp->current_location > temp->base){
            *ch = Mem_Core[temp->current_location -1]; // finds the most recent ch written
            temp->current_location = temp->current_location - 1;        // move current location

        }else{
            sema_memory.up();
            return -1;
        }

        sema_memory.up();
        return 1;
}


//does not check if the node is full and if it is linked to another so it can write more.
int mem_mgr::Mem_Write(int memory_handle, char ch){     // write a character to the current location in
memory, return a -1 if at end of bounds.
        sema_memory.down();
        MemNode* temp = this->head;
        while(temp){
            if (temp->handle == memory_handle){

                while((temp->current_location > temp->limit) && (temp->linked > 0)){
```

```
                temp = temp->next; // if the head is full but has a linked node then move to
that
            }

            if (temp->current_location <= temp->limit){ // if == to limit current location will end up 1
past limit
                Mem_Core[temp->current_location] = ch;

                temp->current_location = temp->current_location + 1;
                sema_memory.up();
                return 1;
            }else{
                sema_memory.up();
                return -1;
            }
        }
        else{
            temp = temp->next;
        }
    }
    sema_memory.up();
    return -1;
}

// overloaded multi-byte read and write
//****************************not currently functioning-causes seg fault******************************
int mem_mgr::Mem_Read(int memory_handle, int offset_from_beg, int text_size, char *text){
    sema_memory.down();
    int cl = 0; //current location for offset
    MemNode* temp = this->head;

    while(temp){
        if (temp->handle == memory_handle){
            cl = temp->base + offset_from_beg;

            for (int i = 0; i < text_size; i++){
                text[i] = Mem_Core[cl+i];
            }
            sema_memory.up();
        return 1;

        }
        else{
            temp = temp->next;
        }
    }
    sema_memory.up();
    return -1;
}

int mem_mgr::Mem_Write(int memory_handle, int offset_from_beg, int text_size, char *text){
    sema_memory.down();
    int cl = 0; //current location for offset
    MemNode* temp = this->head;

    while(temp){
        if (temp->handle == memory_handle){ // find the right handle
            cl = temp->base + offset_from_beg;

            for (int i = 0; i < text_size; i++){
                Mem_Core[cl+i] = text[i];
            }
            sema_memory.up();
        return 1;

        }
        else{
            temp = temp->next;
        }
    }
    sema_memory.up();
    return -1;
}

int mem_mgr::Mem_Left(){// return the amount of core memory left in the OS
    sema_memory.down();
    int counter = 0;
    MemNode* temp = this->head;
    while(temp){
        if (temp->status == 0){
            counter = counter + 128; // temp->size should go here
        }
        temp = temp->next;
    }
    sema_memory.up();
    return counter;
}

int mem_mgr::Mem_Largest(){// return the size of the largest available memory segment
    sema_memory.down();
    int counter = 0;
    int tempcount = 0;
    MemNode* temp = this->head;
    while(temp){
        tempcount = 0;
        if (temp->status == 0){
            tempcount = tempcount + 128; // temp->size should go here
            while (temp->linked){
                temp = temp->next;
                tempcount = tempcount + 128; // temp->size should go here
            }

            if (tempcount > counter){
                counter = tempcount;
            }
        }
```

```
                temp = temp->next;
        }
        sema_memory.up();
        return counter;
}

int mem_mgr::Mem_Smallest(){// return the size of the smallest available memory segment
        sema_memory.down();
        int counter = 1024;
        int tempcount = 0;
        MemNode* temp = this->head;
        while(temp){
                tempcount = 0;
                if (temp->status == 0){
                        tempcount = tempcount + 128; // temp->size should go here
                        while (temp->linked){
                                temp = temp->next;
                                tempcount = tempcount + 128; // temp->size should go here
                        }
                        if (tempcount < counter){
                        counter = tempcount;
                        }
                }

                temp = temp->next;
        }
        sema_memory.up();
        return counter;
}


int mem_mgr::Mem_Coalesce(){ // combine two or more contiguous blocks of free space and place . dots in the
coalesced memory.
        sema_memory.down();
        int array[9];

        MemNode* temp = this->head;

        for(int i = 1; i < 9; i++){
                if (temp->status == 0){
                        array[i]=1;
                        for(int j = temp->base; j < temp->limit + 1; j++){
                                Mem_Core[j] = '.'; // clear out all left over data
                        }
                }
                temp = temp->next;
        }

        temp = this->head;

        for(int i = 1; i < 9; i++){
                if (array[i] == 1){
                        temp->start = 1;
```

```
            while(array[i+1] == 1){
                temp->linked = temp->handle+1;
                temp = temp->next;
                i++;
            }
        }
        temp = temp->next;
    }

    sema_memory.up();
    return 1;
}

int mem_mgr::Mem_Dump(int starting_from, int num_bytes, WINDOW * win){// dump the contents of memory
for specific location
    sema_memory.down();
    char buff[256];
    int end = starting_from + num_bytes;

    sprintf(buff, " Memory dump of %d bytes starting at address:%d \n", num_bytes, starting_from);
    write_window(win, buff);

    for (int i = starting_from; i < end + 1; i++)
        {
            sprintf(buff, "%c", Mem_Core[i]);
            write_window(win, buff);
        }
        sema_memory.up();
        return 1;
}
int mem_mgr::Mem_Usage(WINDOW * win){
    sema_memory.down();

    MemNode* temp = this->head;
    char buff[256];
    sprintf(buff, "  Status  | Mem Handle | Start Loc | End Loc | Size |  Cur Loc  |  TID  \n");
    write_window(win,buff);
    char * status;
    const char * name;
    int handle, base, limit, size,  current_location;
    int start;
    int status1;
    while(temp){
        start = temp->start;
        if(start == 1){
            status1 = temp->status;
            if (status1){
                status = "Used";
            }else{
                status = "Free";
            }
            name = temp->owner.c_str();
            handle = temp->handle;
```

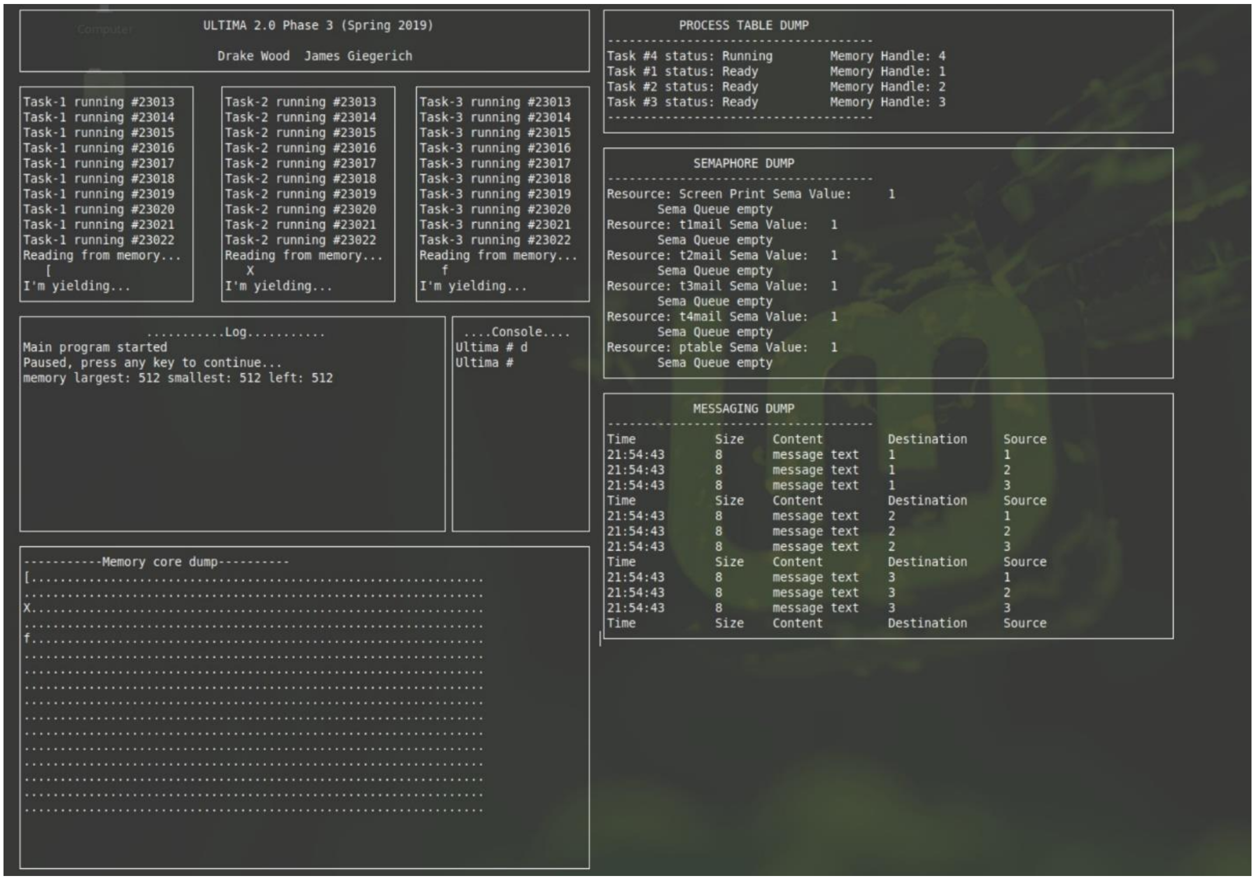
```
                        base = temp->base;
                        limit = temp->limit;
                        size = temp->size;
                        current_location = temp->current_location;
                        sprintf(buff, "  %s\t\t%d\t\t%d\t%d\t%d\t%d\t%s \n",status, handle, base, limit, size,
current_location, name);
                        write_window(win,buff);
                }
                temp = temp->next;
        }

        sema_memory.up();
}
int mem_mgr::Core_Dump( WINDOW * win){  // dumps entire contents of memory to the screen
        sema_screen.down();
        wclear(win);
        sema_screen.up();

        sema_memory.down();
        char buff[256];
        int count = 0;
        write_window(win, "\n -----------Memory core dump---------- \n");
        for (int i = 0; i < 16; i++){
                for (int j = 1; j < 65; j++){
                        buff[j] = Mem_Core[count];
                        count++;
                }
                buff[0] = ' ';
                buff[65] = '\n';
                write_window(win, buff);
        }
        sema_memory.up();
}
```

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 memory.o

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


memory.o: memory.h memory.cpp semaphore.h window.h

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


clean:

        rm *.
```

Output



```
ULTIMA 2.0 Phase 3 (Spring 2019)

Drake Wood  James Giegerich
```

```
Task-1 running #23013
Task-1 running #23014
Task-1 running #23015
Task-1 running #23016
Task-1 running #23017
Task-1 running #23018
Task-1 running #23019
Task-1 running #23020
Task-1 running #23021
Task-1 running #23022
Reading from memory...
   [
I'm yielding...
```

```
Task-2 running #23013
Task-2 running #23014
Task-2 running #23015
Task-2 running #23016
Task-2 running #23017
Task-2 running #23018
Task-2 running #23019
Task-2 running #23020
Task-2 running #23021
Task-2 running #23022
Reading from memory...
   X
I'm yielding...
```

```
Task-3 running #23013
Task-3 running #23014
Task-3 running #23015
Task-3 running #23016
Task-3 running #23017
Task-3 running #23018
Task-3 running #23019
Task-3 running #23020
Task-3 running #23021
Task-3 running #23022
Reading from memory...
   f
I'm yielding...
```

```
           ...........Log...........
Main program started
Paused, press any key to continue...
memory largest: 512 smallest: 512 left: 512
```

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

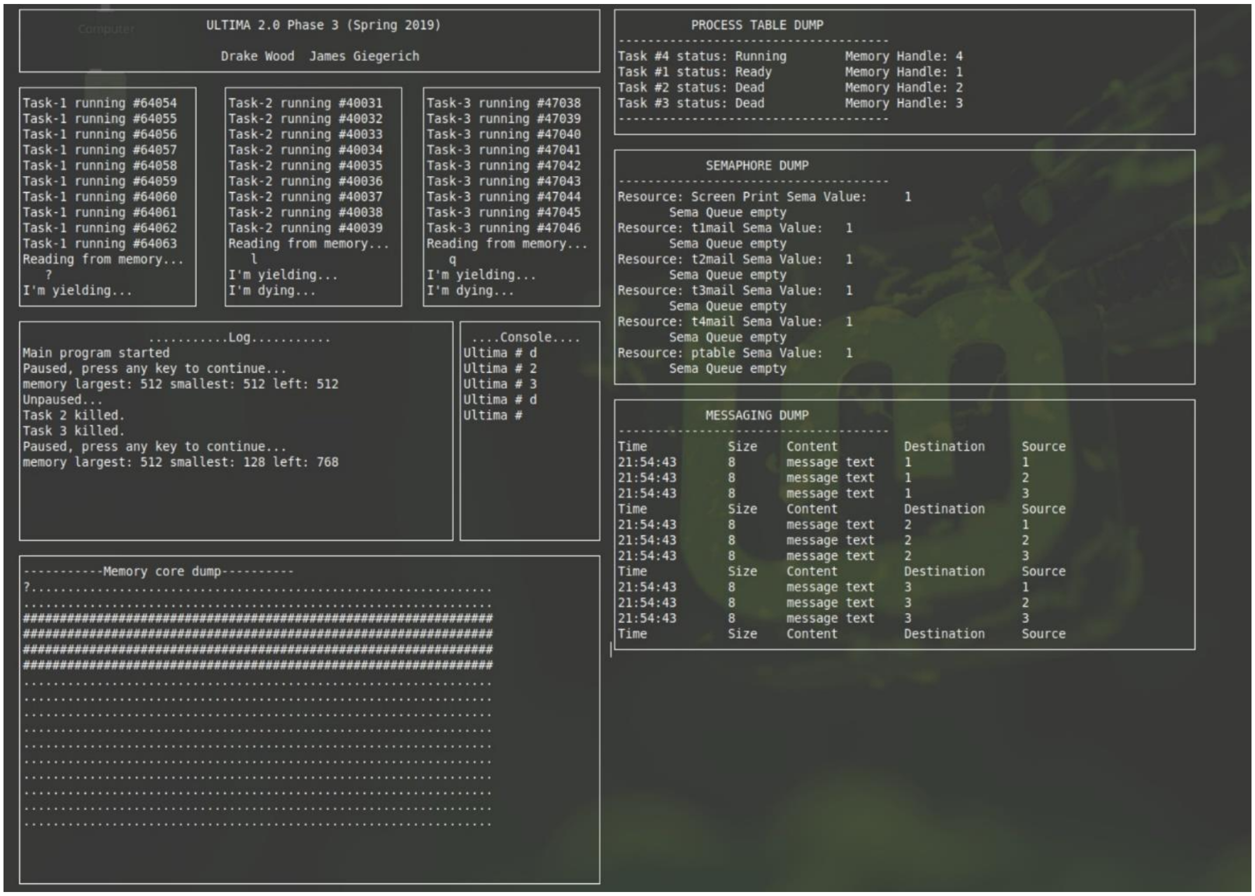
```
----------Memory core dump---------
[...............................................................
................................................................
X...............................................................
................................................................
f...............................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
```

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

```
          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
21:54:43      8       message text    1               1
21:54:43      8       message text    1               2
21:54:43      8       message text    1               3
Time          Size    Content         Destination     Source
21:54:43      8       message text    2               1
21:54:43      8       message text    2               2
21:54:43      8       message text    2               3
Time          Size    Content         Destination     Source
21:54:43      8       message text    3               1
21:54:43      8       message text    3               2
21:54:43      8       message text    3               3
Time          Size    Content         Destination     Source
```

Here we are displaying the write and read, each of the tasks writes a random bit to their memory at the beginning of their cycle and reads it into their window at the end. We can see in the Memory core dump that the memory contains the same thing that was read back. In the log window the largest, smallest and memory left are displayed, 4 128bit blocks are taken by the threads and 1 512 bit block remains.



```
ULTIMA 2.0 Phase 3 (Spring 2019)

  Drake Wood  James Giegerich
```

```
Task-1 running #64054
Task-1 running #64055
Task-1 running #64056
Task-1 running #64057
Task-1 running #64058
Task-1 running #64059
Task-1 running #64060
Task-1 running #64061
Task-1 running #64062
Task-1 running #64063
Reading from memory...
   ?
I'm yielding...
```

```
Task-2 running #40031
Task-2 running #40032
Task-2 running #40033
Task-2 running #40034
Task-2 running #40035
Task-2 running #40036
Task-2 running #40037
Task-2 running #40038
Task-2 running #40039
Reading from memory...
   l
I'm yielding...
I'm dying...
```

```
Task-3 running #47038
Task-3 running #47039
Task-3 running #47040
Task-3 running #47041
Task-3 running #47042
Task-3 running #47043
Task-3 running #47044
Task-3 running #47045
Task-3 running #47046
Reading from memory...
   q
I'm yielding...
I'm dying...
```

```
                ...........Log...........
Main program started
Paused, press any key to continue...
memory largest: 512 smallest: 512 left: 512
Unpaused...
Task 2 killed.
Task 3 killed.
Paused, press any key to continue...
memory largest: 512 smallest: 128 left: 768
```

```
 ....Console....
Ultima # d
Ultima # 2
Ultima # 3
Ultima # d
Ultima #
```

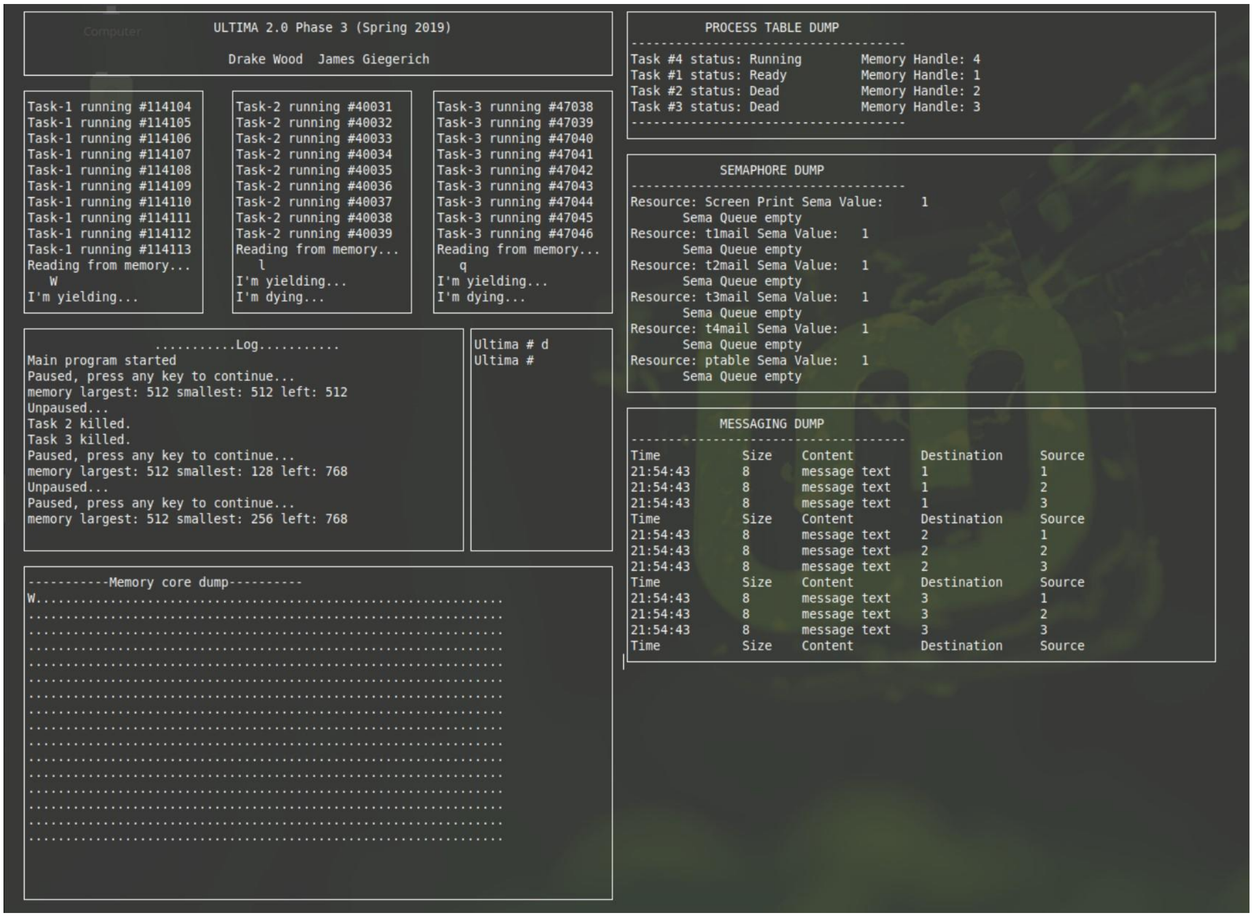
```
----------Memory core dump----------
?...............................................................
................................................................
################################################################
################################################################
################################################################
################################################################
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
```

```
          PROCESS TABLE DUMP
------------------------------------
Task #4 status: Running        Memory Handle: 4
Task #1 status: Ready          Memory Handle: 1
Task #2 status: Dead           Memory Handle: 2
Task #3 status: Dead           Memory Handle: 3
------------------------------------
```

```
            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
21:54:43      8       message text    1               1
21:54:43      8       message text    1               2
21:54:43      8       message text    1               3
Time          Size    Content         Destination     Source
21:54:43      8       message text    2               1
21:54:43      8       message text    2               2
21:54:43      8       message text    2               3
Time          Size    Content         Destination     Source
21:54:43      8       message text    3               1
21:54:43      8       message text    3               2
21:54:43      8       message text    3               3
Time          Size    Content         Destination     Source
```

This output shows what happens when a task is killed, and its memory is freed. Task 1 still has its memory while 2 and 3 both have been cleared. 512 is still the largest block available but now the smallest is 128 and there is 768 total free.



```
ULTIMA 2.0 Phase 3 (Spring 2019)

Drake Wood  James Giegerich
```

```
Task-1 running #114104
Task-1 running #114105
Task-1 running #114106
Task-1 running #114107
Task-1 running #114108
Task-1 running #114109
Task-1 running #114110
Task-1 running #114111
Task-1 running #114112
Task-1 running #114113
Reading from memory...
   W
I'm yielding...
```

```
Task-2 running #40031
Task-2 running #40032
Task-2 running #40033
Task-2 running #40034
Task-2 running #40035
Task-2 running #40036
Task-2 running #40037
Task-2 running #40038
Task-2 running #40039
Reading from memory...
   l
I'm yielding...
I'm dying...
```

```
Task-3 running #47038
Task-3 running #47039
Task-3 running #47040
Task-3 running #47041
Task-3 running #47042
Task-3 running #47043
Task-3 running #47044
Task-3 running #47045
Task-3 running #47046
Reading from memory...
   q
I'm yielding...
I'm dying...
```

```
...........Log...........
Main program started
Paused, press any key to continue...
memory largest: 512 smallest: 512 left: 512
Unpaused...
Task 2 killed.
Task 3 killed.
Paused, press any key to continue...
memory largest: 512 smallest: 128 left: 768
Unpaused...
Paused, press any key to continue...
memory largest: 512 smallest: 256 left: 768
```

```
Ultima # d
Ultima #
```

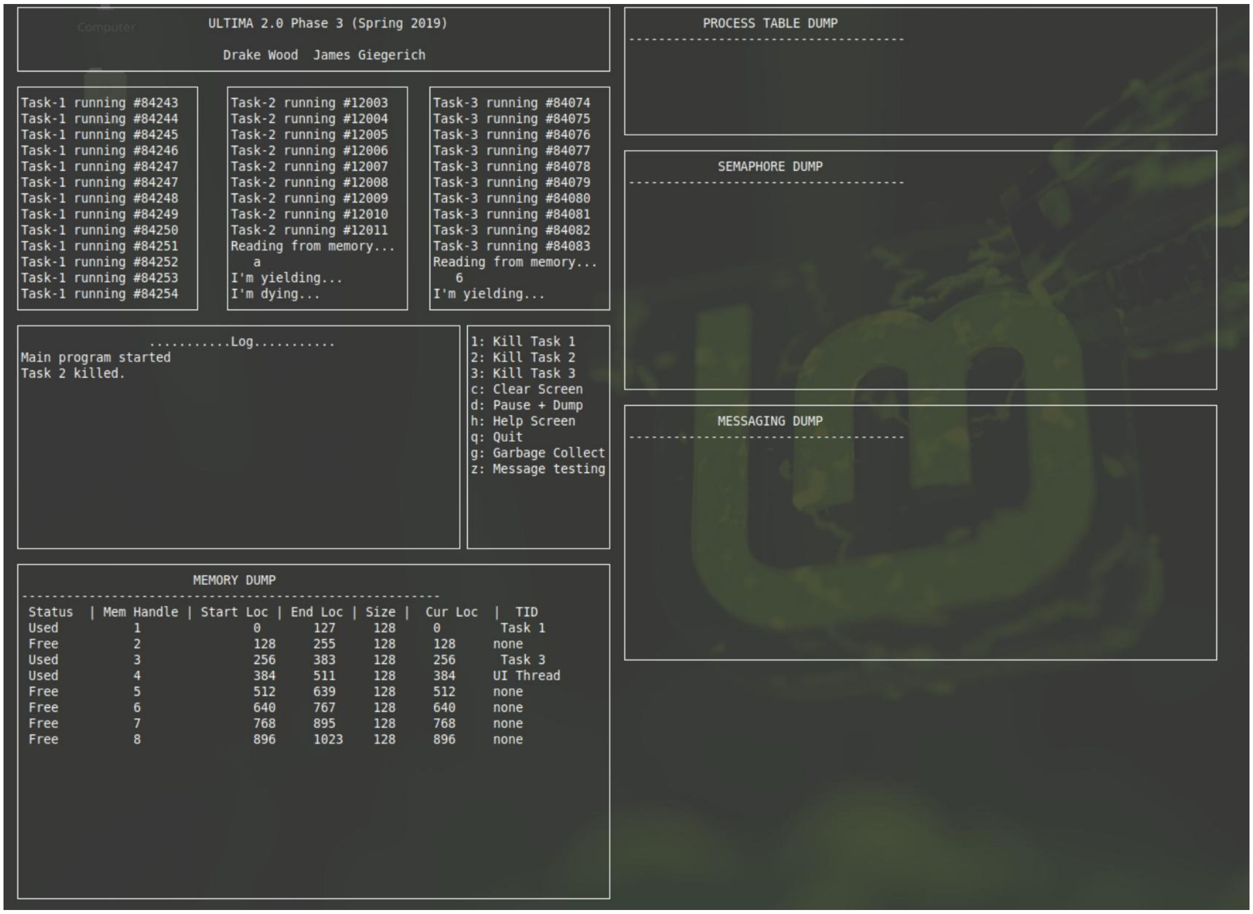
```
----------Memory core dump----------
W...............................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
```

```
         PROCESS TABLE DUMP
------------------------------------
Task #4 status: Running        Memory Handle: 4
Task #1 status: Ready          Memory Handle: 1
Task #2 status: Dead           Memory Handle: 2
Task #3 status: Dead           Memory Handle: 3
------------------------------------
```

```
          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
21:54:43      8       message text    1               1
21:54:43      8       message text    1               2
21:54:43      8       message text    1               3
Time          Size    Content         Destination     Source
21:54:43      8       message text    2               1
21:54:43      8       message text    2               2
21:54:43      8       message text    2               3
Time          Size    Content         Destination     Source
21:54:43      8       message text    3               1
21:54:43      8       message text    3               2
21:54:43      8       message text    3               3
Time          Size    Content         Destination     Source
```

This is directly after the last output and coalesce has been called. The memory has been reset back to dots for task 2 and 3. 512 is still the largest memory block because task 4, the UI thread is dividing 2 holes, the 512 an the smallest 256 with a total of 768 remaining.



```
              ULTIMA 2.0 Phase 3 (Spring 2019)

                 Drake Wood  James Giegerich
```

```
Task-1 running #84243
Task-1 running #84244
Task-1 running #84245
Task-1 running #84246
Task-1 running #84247
Task-1 running #84247
Task-1 running #84248
Task-1 running #84249
Task-1 running #84250
Task-1 running #84251
Task-1 running #84252
Task-1 running #84253
Task-1 running #84254
```

```
Task-2 running #12003
Task-2 running #12004
Task-2 running #12005
Task-2 running #12006
Task-2 running #12007
Task-2 running #12008
Task-2 running #12009
Task-2 running #12010
Task-2 running #12011
Reading from memory...
   a
I'm yielding...
I'm dying...
```

```
Task-3 running #84074
Task-3 running #84075
Task-3 running #84076
Task-3 running #84077
Task-3 running #84078
Task-3 running #84079
Task-3 running #84080
Task-3 running #84081
Task-3 running #84082
Task-3 running #84083
Reading from memory...
   6
I'm yielding...
```

```
                 ...........Log...........
Main program started
Task 2 killed.
```

```
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
```

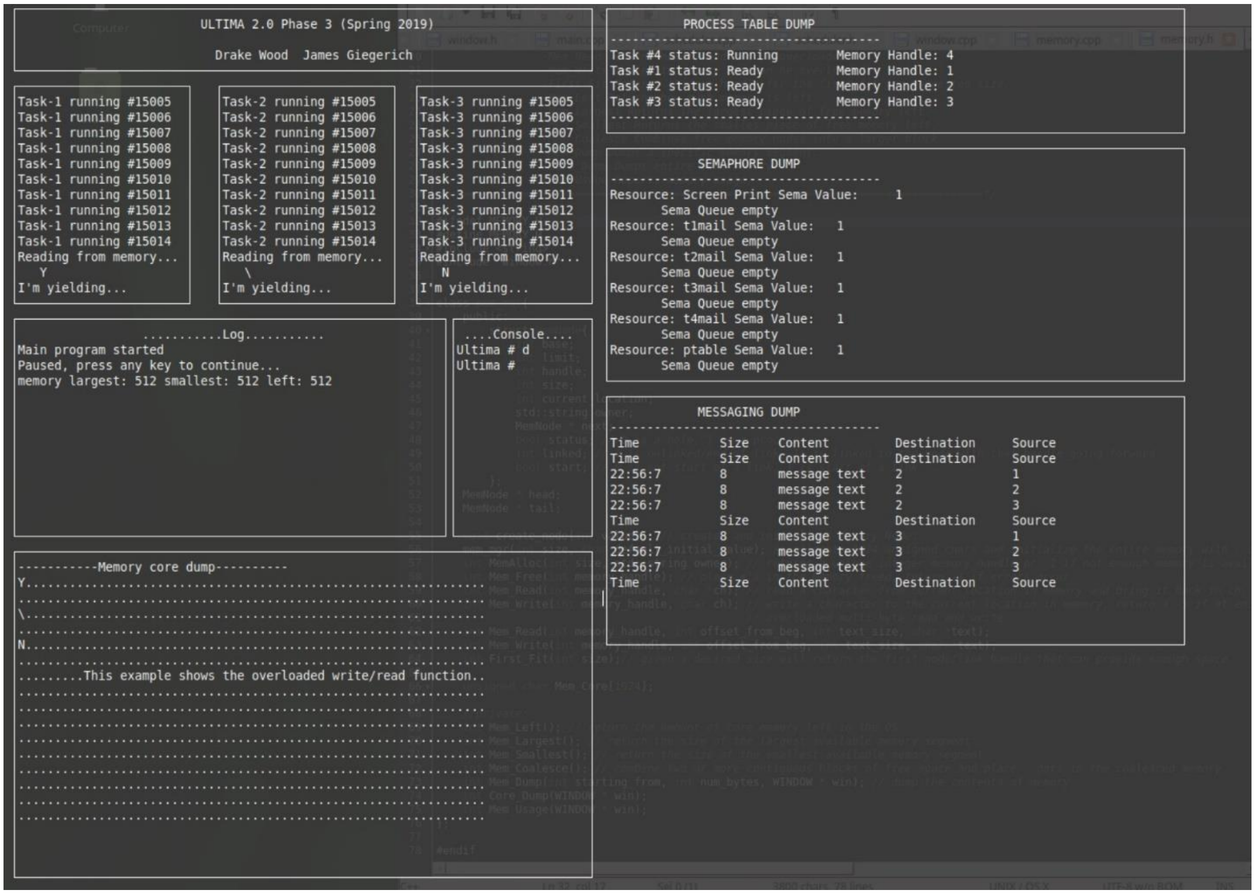
```
                       MEMORY DUMP
--------------------------------------------------------
 Status  | Mem Handle | Start Loc | End Loc | Size |  Cur Loc  |  TID
 Used         1              0       127     128     0        Task 1
 Free         2              128     255     128     128     none
 Used         3              256     383     128     256      Task 3
 Used         4              384     511     128     384     UI Thread
 Free         5              512     639     128     512     none
 Free         6              640     767     128     640     none
 Free         7              768     895     128     768     none
 Free         8              896     1023    128     896     none
```

```
          PROCESS TABLE DUMP
-------------------------------------
```

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

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

This output has the memory dump that displays each blocks status, we can see that task 2 has been killed and its memory was freed and is a hole between task 1 and 3.



```
ULTIMA 2.0 Phase 3 (Spring 2019)

Drake Wood  James Giegerich
```

```
Task-1 running #15005
Task-1 running #15006
Task-1 running #15007
Task-1 running #15008
Task-1 running #15009
Task-1 running #15010
Task-1 running #15011
Task-1 running #15012
Task-1 running #15013
Task-1 running #15014
Reading from memory...
   Y
I'm yielding...
```

```
Task-2 running #15005
Task-2 running #15006
Task-2 running #15007
Task-2 running #15008
Task-2 running #15009
Task-2 running #15010
Task-2 running #15011
Task-2 running #15012
Task-2 running #15013
Task-2 running #15014
Reading from memory...
   \
I'm yielding...
```

```
Task-3 running #15005
Task-3 running #15006
Task-3 running #15007
Task-3 running #15008
Task-3 running #15009
Task-3 running #15010
Task-3 running #15011
Task-3 running #15012
Task-3 running #15013
Task-3 running #15014
Reading from memory...
   N
I'm yielding...
```

```
           ...........Log...........
Main program started
Paused, press any key to continue...
memory largest: 512 smallest: 512 left: 512
```

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

```
----------Memory core dump----------
Y...............................................................
................................................................
\...............................................................
................................................................
N...............................................................
................................................................
.........This example shows the overloaded write/read function..
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
```

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

```
           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
Time           Size    Content         Destination     Source
22:56:7        8       message text    2               1
22:56:7        8       message text    2               2
22:56:7        8       message text    2               3
Time           Size    Content         Destination     Source
22:56:7        8       message text    3               1
22:56:7        8       message text    3               2
22:56:7        8       message text    3               3
Time           Size    Content         Destination     Source
```

This last output shows the overloaded write with a 9-offset coming from the UI thread with handle 4. We were unable to successfully implement the overloaded read function so there is no output for that function.