ULTIMA 2.0

CS-435 Operating System

Phase 1 – Scheduler and Semaphore

By:

Drake Wood

drawood@iu.edu

James Giegerich

jgiegeri@iu.edu

2/22/19

Table of Contents

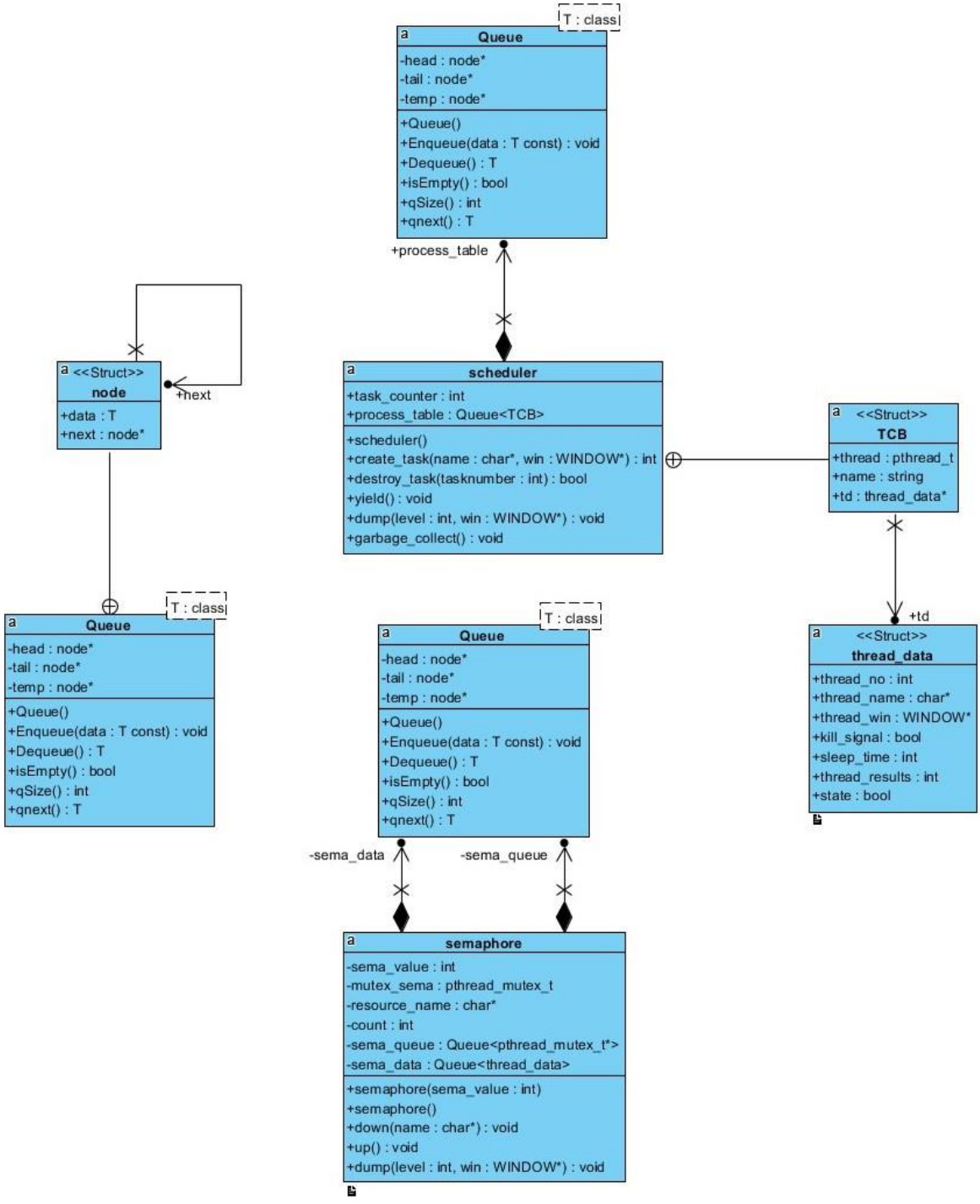
Phase Abstract

This report details the design and development of the first 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 implement resource synchronization for our simulated operating system. This needs to be used to control access to any shared resources such as printers, memory, or disks. This synchronization is done with the use of semaphores and it is imperative to support the use of any number of semaphores for future us and iterations of this program. A process table is also needed to dynamically store Task Control Blocks (TCBs). TCBs are used to preserve the state of each task, running, ready, blocked, or dead. The TCB will be used to store more information in the future also. Output must show that the scheduler, process table, and semaphore work properly. To do this dump functions for the process table and semaphore must be added. The text book Modern Operating Systems by Tanenbaum, C435 class notes, Unix manual, and some online research was used for external information.

Design Diagram



Source Code

Main.cpp

```
/*============================================================================|
|   Assignment:       Ultima 2.0 Phase 1
|      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:    12/16/2019
|  Last Updated:       12/25/2019
|       Due Date:  12/25/2019
|============================================================================|
|  Description: Contains the main function that interacts with the scheduler, semaphore, queue, and window classes.
|               Multiple curses windows and 3 pthreads are created which are then passed to other functions to handle.
|               The rest of main contains an input loop for the user to control what happens while the program is running.
|               Details of the way functions are used and their parameters as well as
|               computation details are given in the class descriptions.
*============================================================================*/


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

```
#include "semaphore.h"
#include "window.h"

using namespace std;
semaphore sema(1);  // creates semaphore

//---------------------------------------------------------------------------
//-----------------------------MAIN------------------------------------------
//---------------------------------------------------------------------------

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

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

    WINDOW * Heading_Win = newwin(12, 80, 3, 2);
    box(Heading_Win, 0,0);

    char semabuff[32];
    sprintf(semabuff, " heading window");       // Giving semaphore a resource name
    sema.down(semabuff);
    mvwprintw(Heading_Win, 2, 28, "ULTIMA 2.0 (Spring 2019)");
    mvwprintw(Heading_Win, 4, 2, "Starting ULTIMA 2.0.....");
    mvwprintw(Heading_Win, 5, 2, "Starting Thread 1....");
    mvwprintw(Heading_Win, 6, 2, "Starting Thread 2....");
    mvwprintw(Heading_Win, 7, 2, "Starting Thread 3....");
    mvwprintw(Heading_Win, 9, 2, "Press 'q' or Ctrl-C to exit the program...");
    wrefresh(Heading_Win);
    sema.up();
```

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

    WINDOW * Log_Win = create_window(10, 60, 30, 2);
    write_window(Log_Win, 1, 18, ".....Log.....\n");

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

    WINDOW * Console_Win = create_window(10, 20, 30, 62);
    write_window(Console_Win, 1, 1, " ....Console....\n");
    write_window(Console_Win, 2, 1, "Ultima # ");
    write_window(Log_Win, " Main program started\n" );

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

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

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

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

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

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

    WINDOW * Process_Table = create_window(8, 40, 3, 83);
    write_window(Process_Table, 1, 5, "      PROCESS TABLE DUMP \n ---------------------------------------\n");

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

    WINDOW * Sema_Dump = create_window(7, 40, 11, 83);
    write_window(Sema_Dump, 1, 5, "        SEMAPHORE DUMP \n ---------------------------------------\n");

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

    sprintf(semabuff, " user console"); // giving semaphore a resource name
    sema.down(semabuff);
    cbreak();                           // Set up keyboard I/O processing
    noecho();                           // disable line buffering
    nodelay(Console_Win, true);         // disable automatic echo of characters read by getch(), wgetch()
    sema.up();                          // nodelay causes getch to be a non-blocking call.
    char buff[256];                     // If no input is ready, getch returns ERR
    int input = -1;
    int CPU_Quantum = 1;

//---------------------------------------------------------------------------
//Input loop for user console
//---------------------------------------------------------------------------

    while(input != 'q')
    {
       sched.dump(1, Process_Table);   // updates dump windows
       sema.dump(1, Sema_Dump);        // every loop
```

```
sprintf(semabuff, " input loop");   // giving semaphore a resource name
sema.down(semabuff);
input = wgetch(Console_Win);
sema.up();

switch(input)
{
   case '1':
   case '2':
   case '3':
      sprintf(buff, " %c\n", input);
      write_window(Console_Win, buff);
      sprintf(buff, " Attempting to kill task %c\n", input);
      write_window(Log_Win, buff);

      if (sched.destroy_task(input - '0')){       //killed
         sprintf(buff, " Task %c killed\n", input);
         write_window(Log_Win, buff);
      }else{                                      //already dead
         sprintf(buff, " Task %c already dead\n", input);       // giving semaphore a resource name
         write_window(Log_Win, buff);
      } // end else

      sprintf(semabuff, " clear console");
      sema.down(semabuff);
      wclear(Console_Win);
      sema.up();

      write_window(Console_Win, 1, 1, "Ultima # ");
      break;
   case 'c':                  // CLEAR
      sprintf(semabuff, " user case c");
      sema.down(semabuff);
```

```
    refresh();                  // Clear the entire screen (in case it is corrupted)
    wclear(Console_Win);    // Clear the Console window
    sema.up();

    write_window(Console_Win, 1, 1, "Ultima # ");
    break;
case 'h':                   // HELP
    display_help(Console_Win);
    write_window(Console_Win, 8, 1, "Ultima # ");
    break;
case 'g':
    sched.garbage_collect();
    break;
case 'q':                   // QUIT
    write_window(Log_Win," Quiting the main program....\n" );
    for (int i = 1; i < 4; i++){
    if (sched.destroy_task(i)){     // Killed
       sprintf(buff, " Task %d killed\n", i);
       write_window(Log_Win, buff);
    }else{                          // Already dead
       sprintf(buff, " Task %d already dead\n", i);
       write_window(Log_Win, buff);
    } // end else
} // end if
    break;
case ERR:   // If wgetch() return ERR, that means no keys were pressed
    break;
default:
    sprintf(buff, " %c\n", input);
    write_window(Console_Win, buff);
    write_window(Console_Win, " -Invalid Command\n");
    write_window(Log_Win, buff);
    write_window(Log_Win, " -Invalid Command\n");
    write_window(Console_Win, " Ultima # ");
```

```
            break;
        } // end switch
        sleep(1);
        CPU_Quantum++;
    } // end while

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

Semaphore.h

```
/*=============================================================================|
|   Assignment:       Ultima 2.0 Phase 1
|      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:    12/16/2019
| Last Updated:       12/25/2019
|       Due Date:  12/25/2019
|=============================================================================|
| 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"

class semaphore {
   int sema_value;        // 0 or 1 in the case of a binary semaphore
   Queue <pthread_mutex_t*>sema_queue;
   Queue <thread_data>sema_data;
   pthread_mutex_t mutex_sema;
   char* resource_name; // The name of the resource being managed
   int count;

   public:
   semaphore(int sema_value);
   ~semaphore();

   void down(char* name);                  // 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 1
|      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:    12/16/2019
|  Last Updated:       12/25/2019
|       Due Date:  12/25/2019
|============================================================================|
|  Description: Contains the definitions for the functions outlined in semaphore.h
*============================================================================*/

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

extern semaphore sema;

void semaphore::down(char* name){                   // Lock resource
      pthread_mutex_lock(&mutex_sema);
      resource_name = name;
      --sema_value;

      if (sema_value < 0){
         pthread_mutex_t *mutex = new pthread_mutex_t;
         assert(!pthread_mutex_init(mutex, NULL));
         sema_queue.Enqueue(mutex);

         pthread_mutex_unlock(&mutex_sema);
```

```
        pthread_mutex_lock(mutex);


        pthread_mutex_lock(mutex);
        pthread_mutex_destroy(mutex);
        delete mutex;
      } // end if
      else
        pthread_mutex_unlock(&mutex_sema);
} // end of down

void semaphore::up(){                                  // Release the resource
      pthread_mutex_lock(&mutex_sema);
      ++sema_value;
      if (!sema_queue.isEmpty()){
        pthread_mutex_t *mutex = sema_queue.Dequeue();
        pthread_mutex_unlock(mutex);
      } // end if
      pthread_mutex_unlock(&mutex_sema);
} // end of up

void semaphore::dump(int level, WINDOW * win){
   char semabuff[32];
   sprintf(semabuff, " sema dump");                    // giving semaphore a resource name
   sema.down(semabuff);
   wclear(win);
   sema.up();
   char buff[256];

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

   sprintf(buff, " Resource: %s \n", resource_name); // Print the resource name of the current semaphore
   write_window(win, buff);
```

```
   sprintf(buff, " Sema Value: \t%d \n", sema_value); // Print sema value
   write_window(win, buff);

write_window(win, " Sema Queue: ");
            for (int i = sema_value; i < 1; i++)
                        write_window(win, " td->");
            if (sema_value >= 1)
                                    write_window(win, "empty\n");
} // end of dump

semaphore::semaphore(int sema_value){
   count = 0;
   this->sema_value = sema_value;
   assert(!pthread_mutex_init(&mutex_sema, NULL));
}

semaphore::~semaphore(){
   pthread_mutex_destroy(&mutex_sema);
}
```

Window.h

```
/*============================================================================|
|   Assignment:       Ultima 2.0 Phase 1
|      File Name:  window.h
| Dependencies:     none
|        Authors:     Drake Wood, James Giegerich
|      Language:   C++
|       Compiler:  G++
|           Class:    C435 - Operating Systems
|       Instructor:    Dr. Hakimzadeh
|  Date Created:    12/16/2019
|  Last Updated:       12/25/2019
|       Due Date:  12/25/2019
|============================================================================|
|  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);

#endif
```

Window.cpp

```
/*============================================================================|
|   Assignment:       Ultima 2.0 Phase 1
|      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:    12/16/2019
|  Last Updated:       12/25/2019
|       Due Date:  12/25/2019
|============================================================================|
|  Description: Contains the definitions for the functions outlined in window.h
*============================================================================*/


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


extern semaphore sema;


WINDOW *create_window(int height, int width, int starty, int startx)
{
   char semabuff[32];
   sprintf(semabuff, " create window");       // giving semaphore a resource name
   sema.down(semabuff);
   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.up();


    return Win;
} // end of create_window

void write_window(WINDOW * Win, const char* text)
{
    char semabuff[32];
    sprintf(semabuff, " write window");     // Giving semaphore a resource name
    sema.down(semabuff);
    wprintw(Win, text);
    box(Win, 0 , 0);
    wrefresh(Win);              // Draw the window
    sema.up();
 } // end of write_window

void write_window(WINDOW * Win, int y, int x, const char* text)
{
    char semabuff[32];
    sprintf(semabuff, " write window with y x");        // Giving semaphore a resource name
    sema.down(semabuff);
    mvwprintw(Win, y, x, text);
    box(Win, 0 , 0);
    wrefresh(Win);              // Draw the window
    sema.up();
} // end of write_window

void display_help(WINDOW * Win)
{
    char semabuff[32];
    sprintf(semabuff, " display help");     // Giving semaphore a resource name
    sema.down(semabuff);
    wclear(Win);
```

```
   sema.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, "h= Help Screen");
   write_window(Win, 6, 1, "q= Quit");
   write_window(Win, 7, 1, "g= Garbage Collect");
 } // end of display_help

void *perform_simple_output(void *arguments)
{
   thread_data *td = (thread_data *) arguments;   // Extract the thread arguments: (method 1)
   int thread_no = td->thread_no;                 // Cast arguments in to thread_data
   WINDOW * Win = td->thread_win;
   int CPU_Quantum =0;
   char buff[256];

   while(!td->kill_signal) {                      // While thread is alive print output
      sprintf(buff, " Task-%d running #%d\n", thread_no, CPU_Quantum++);
      write_window(Win, buff);
   } // end while
   write_window(Win, " I'm dying...\n");
} // end of perform_simple_output
```

Queue.h

```
/*============================================================================|
|   Assignment:       Ultima 2.0 Phase 1
|      File Name:  queue.h
| Dependencies:     none
|        Authors:     Drake Wood, James Giegerich
|      Language:   C++
|       Compiler:  G++
|           Class:    C435 - Operating Systems
|       Instructor:    Dr. Hakimzadeh
|  Date Created:    12/16/2019
|  Last Updated:       12/25/2019
|       Due Date:  12/25/2019
|============================================================================|
|  Description: This is the header file which defines the queue template class used by the semaphore
|             and scheduler classes.
*============================================================================*/

#ifndef QUEUE_H
#define QUEUE_H

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

template <class T>
```

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

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



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

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

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

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

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

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

```
        tail->next = NULL;
        tail->data = data;
    } // end if
    else {
    tail->next = new node;
    tail->next->data = data;
    tail->next->next = NULL;
    tail = tail->next;
    } // end else
} // end of Enqueue

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

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

#endif
```

Queue.cpp

```
/*============================================================================|
|   Assignment:       Ultima 2.0 Phase 1
|      File Name:  queue.cpp
| Dependencies:     queue.h
|         Authors:     Drake Wood, James Giegerich
|      Language:   C++
|       Compiler:  G++
|           Class:    C435 - Operating Systems
|       Instructor:    Dr. Hakimzadeh
|  Date Created:    12/16/2019
|  Last Updated:       12/25/2019
|       Due Date:  12/25/2019
|============================================================================|
|  Description: Contains the definitions for the functions outlined in queue.h
*============================================================================*/


#include "queue.h"


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


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


   if (isEmpty()){
```

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

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

// Dequeues node and returns data stored
```

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

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

Scheduler.h

```
/*============================================================================|
|   Assignment:       Ultima 2.0 Phase 1
|      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:    12/16/2019
|  Last Updated:       12/25/2019
|       Due Date:  12/25/2019
|============================================================================|
|  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>

struct thread_data{
    int thread_no;          // Thread number
    char* thread_name;      // Thread name
    WINDOW *thread_win; // Thread's window
    bool kill_signal;           // Kill signal flag live/kill
    int sleep_time;         // Sleep time
    int thread_results;       // Results
    bool state;                 // State running/blocked
};

class scheduler {
  public:
  struct TCB {
    pthread_t thread;
    std::string name;
    thread_data *td;
  };

  scheduler();
  int task_counter;
  Queue <TCB>process_table;
  int create_task(char* name, WINDOW *win);   // Create appropriate data structures and calls coroutine()
  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
};                                                // 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 1
|      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:    12/16/2019
|  Last Updated:       12/25/2019
|       Due Date:  12/25/2019
|==========================================================================|
|  Description: Contains the definitions for the functions outlined in scheduler.h
*==========================================================================*/


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


extern semaphore sema;


void scheduler::dump(int level, WINDOW * win) {
   char semabuff[32];
   sprintf(semabuff, " scheduler dump");      // giving semaphore a resource name
   sema.down(semabuff);
   wclear(win);
   sema.up();
   write_window(win, 1, 5, "      PROCESS TABLE DUMP \n ------------------------------------\n");
   char buff[256];
   int procnum1;
   int size = process_table.qSize();
```

```
   if (size == 0)                // Check if anything is in the queue
      write_window(win, " No tasks currently running...\n");

   for(int i = 0 ; i < size ; i++){       // Search for a dead task
      TCB tcb = process_table.Dequeue();
      process_table.Enqueue(tcb);
      procnum1 = tcb.td->thread_no;

      if ( tcb.td->kill_signal == false){
         sprintf(buff, " Task #%d status Alive\n", procnum1);
      }else{
         sprintf(buff, " Task #%d status Dead\n", procnum1);
      } // end else
      write_window(win, buff);
   } // end for
   write_window(win, " ------------------------------------\n");
}// end of dump

void scheduler::garbage_collect() {           // Delete those with  dead status
   int size = process_table.qSize();
   for(int i = 0 ; i < size ; i++){
      TCB tcb = process_table.Dequeue();
      if (tcb.td->kill_signal == false){
         process_table.Enqueue(tcb);
      } // end if
   } // end for
} // end of garbage_collect

int scheduler::create_task(char* name, WINDOW *win){
   int result_code;
   TCB tcb;
   tcb.td = new thread_data;
```

```
   this->task_counter++;
   tcb.td->thread_win = win;
   write_window(tcb.td->thread_win, 13, 1, "Starting Thread.....\n");
   tcb.td->thread_no = this->task_counter;
   tcb.td->kill_signal = false;
   tcb.td->sleep_time = 1+ rand() % 3;
   tcb.td->thread_results = 0;
   tcb.name = name;
   tcb.td->state = true;

   // create thread running simple output in its own window
   result_code = pthread_create(&tcb.thread, NULL, perform_simple_output, tcb.td);
   assert(!result_code);          // if there is any problems with result code. display it and end program.

   process_table.Enqueue(tcb);    // add to process table
} // end of create_task

scheduler::scheduler(){
   task_counter = 0;
}

bool scheduler::destroy_task(int tasknumber) {
   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.td->thread_no == tasknumber){
            if (tcb.td->kill_signal == false){        // other processes are added back to Q
               tcb.td->kill_signal =true;
               return 1;
            }else {    //already killed.
               return 0;
```

```
            } // end else
          } // end if
        } // end if
    } // end for
} // end of destroy_task

void scheduler::yield() {
    // unable to produce a working yield function at this time
    // Semaphores currently handle the scheduling
}
```

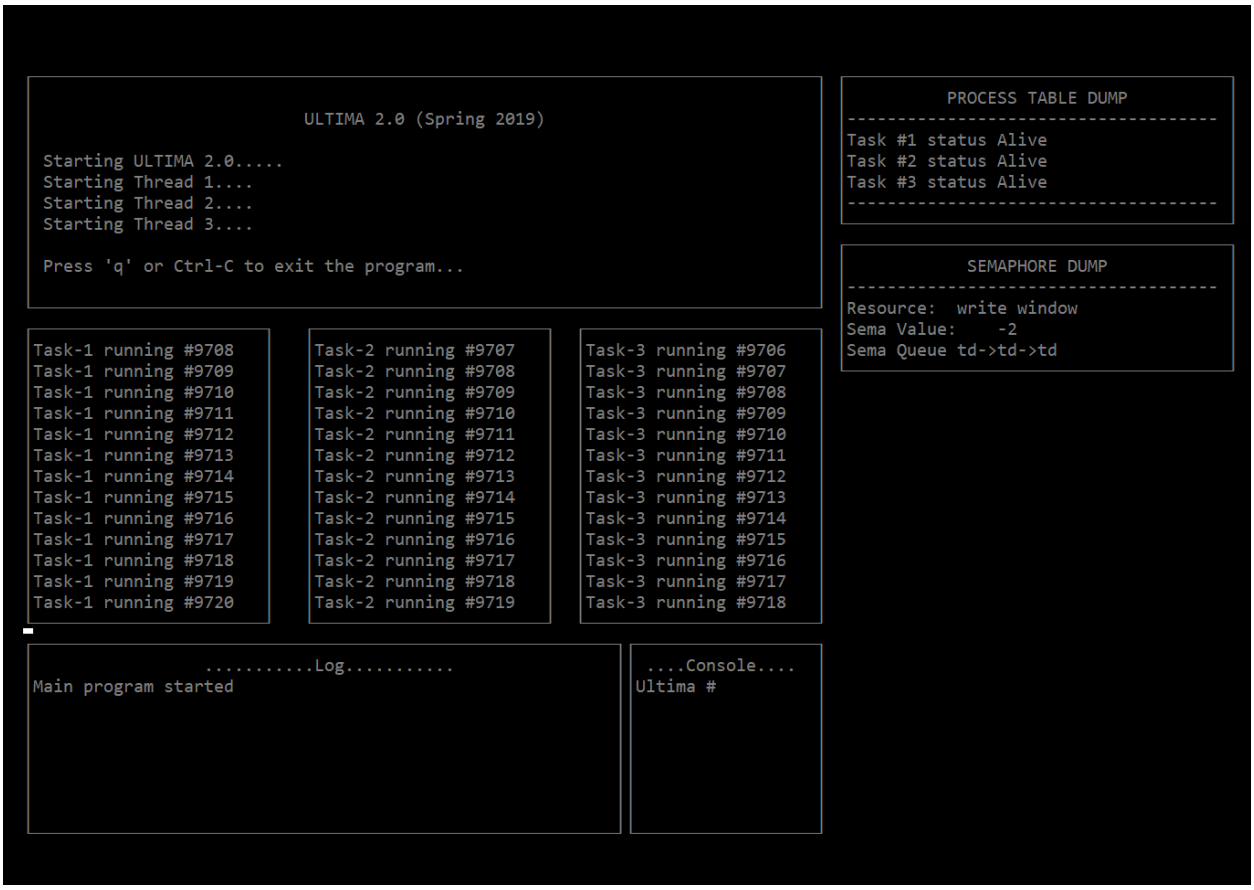
Makefile

```
# 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

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

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

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

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

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

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

clean:

   rm *.o
```

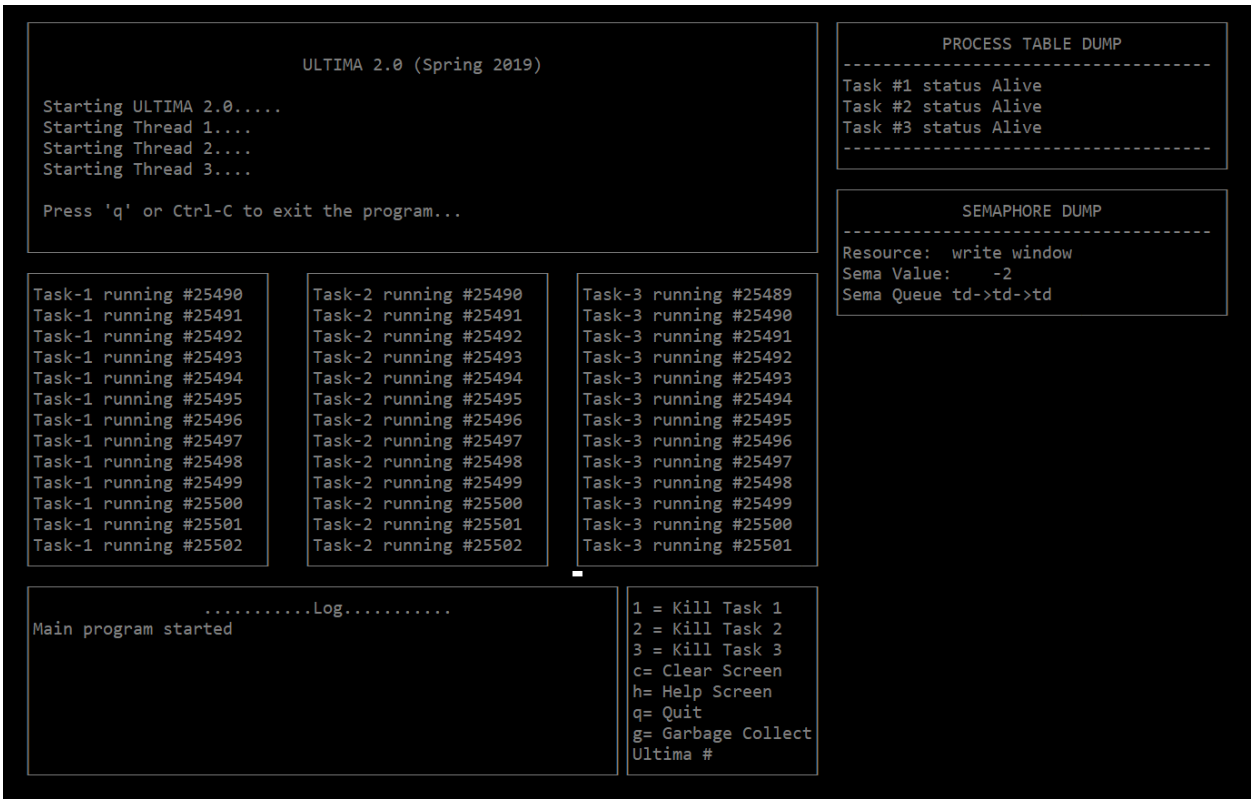
Output

The test plan for this phase was producing all possible input combinations to check for errors that may occur in any situation. We know the system is working because we can show the three processes running and perform any input we choose. The user can kill any and all threads in any order, garbage collect to delete the dead ones, or end the program without error. While the program does function correctly overall there were two areas that were not completed as assigned. First the yield function in the scheduler class does not do anything. We were not able to find a way to control the processes outside of the use of a semaphore directly for resource control. The second is the semaphore dump does not display the contents of the semaphore queue. Below are screenshots of our program with various inputs and outputs being tested.

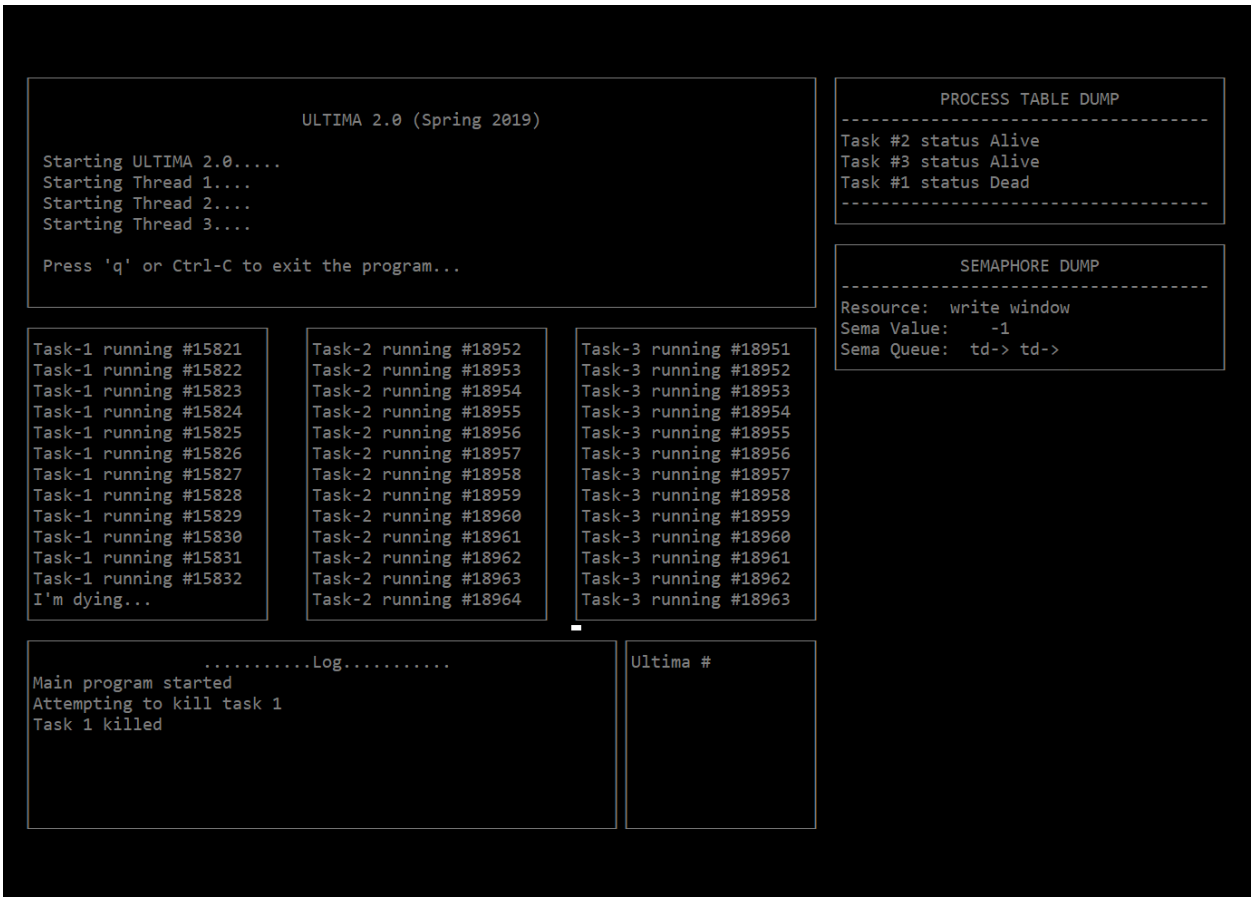
First, we see Ultima during runtime as all three threads are running:



Next, the help window:

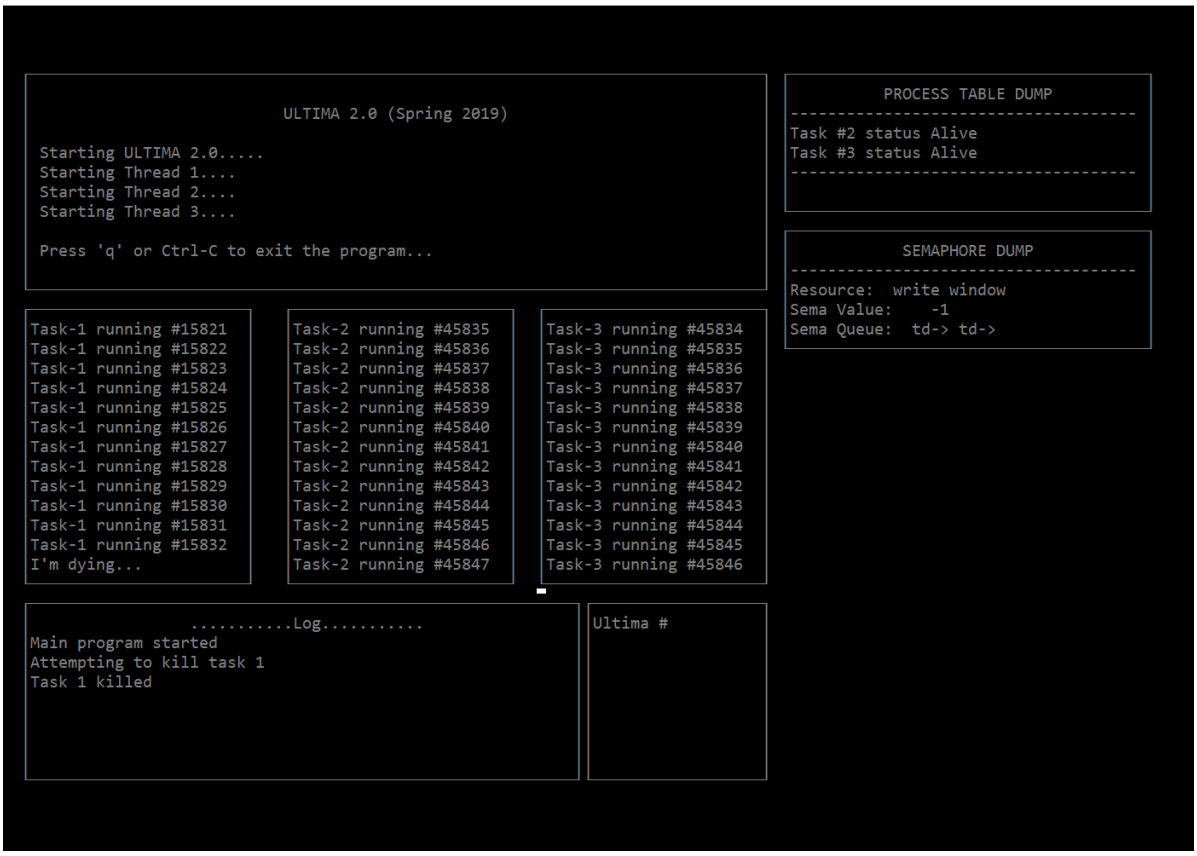


Then, task 1 is killed:

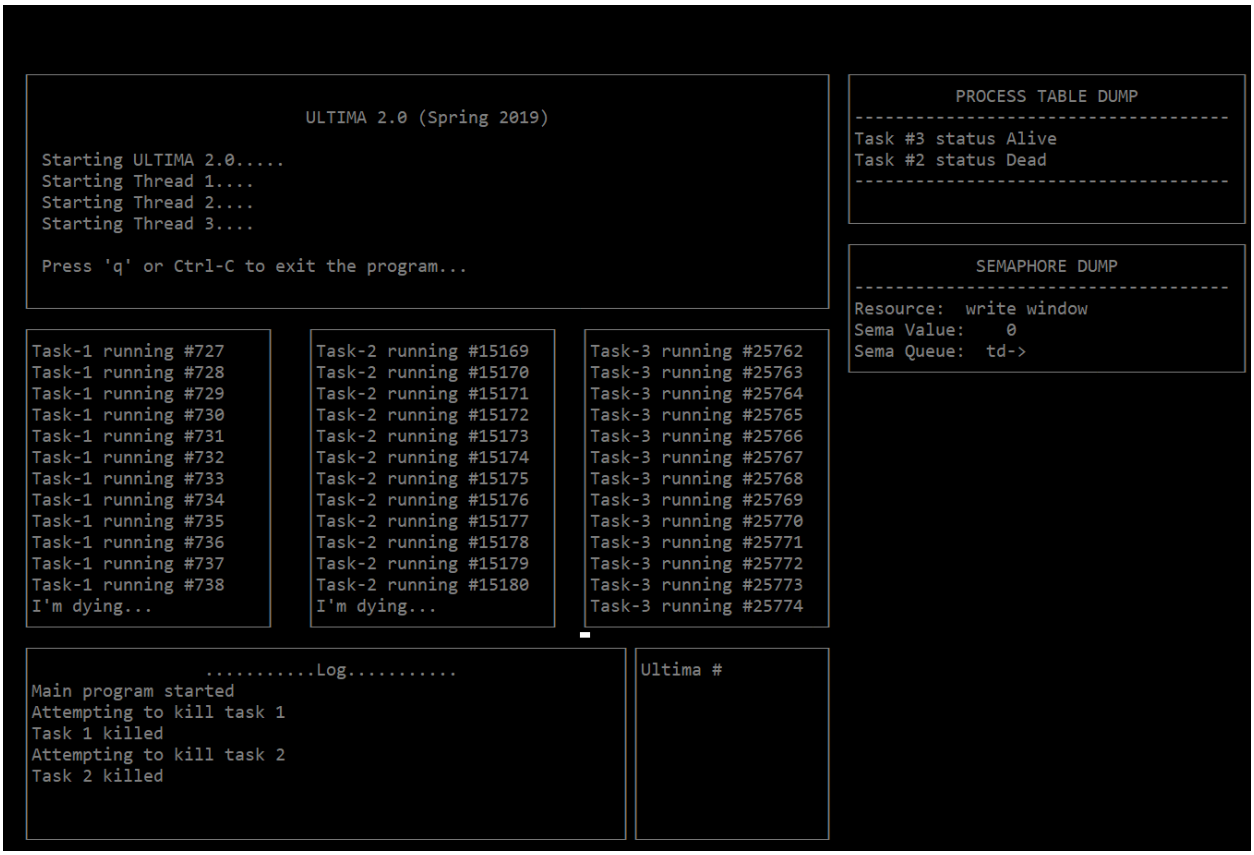


The status in the process table switches, and the semaphore dump updates Sema Value and the size of the Sema Queue.

Now, scheduler::garbage_collect() is called to remove the thread, as seen in the process table:



Thread 2 is now killed, as the dump windows update accordingly:



The garbage collect again, and then thread 3 is killed:

And then garbage collect one last time: