Design Document

PROJECT 2

CMPT 300 D100

Erik Schultz
eriks@sfu.ca
Ian Stewart
iastewar@sfu.ca
Yixuan Li
yixuanl@sfu.ca

25/11/2013

Erik Schultz **301034882**
Ian Stewart **301190316**
Yixuan Li **301191905**

notes for ian before submitting.

could you please do the following:

1. replace this page with the bonus marking sheet

2. resync github for the FINAL version of our source code so that we are submitting the identical hard/soft copies --- when you submit our source code digitally.

3. for PAGE 2 (the one with our names/student numbers) you need to print this page on the back of the cover page. it is the only double-sided page, the rest are all single page.

4. have a quick look through to see if you notice anything wrong. you can skip looking through the source code obviously which is the majority of the pages.

5. pat yourself on the back, great job this project :p

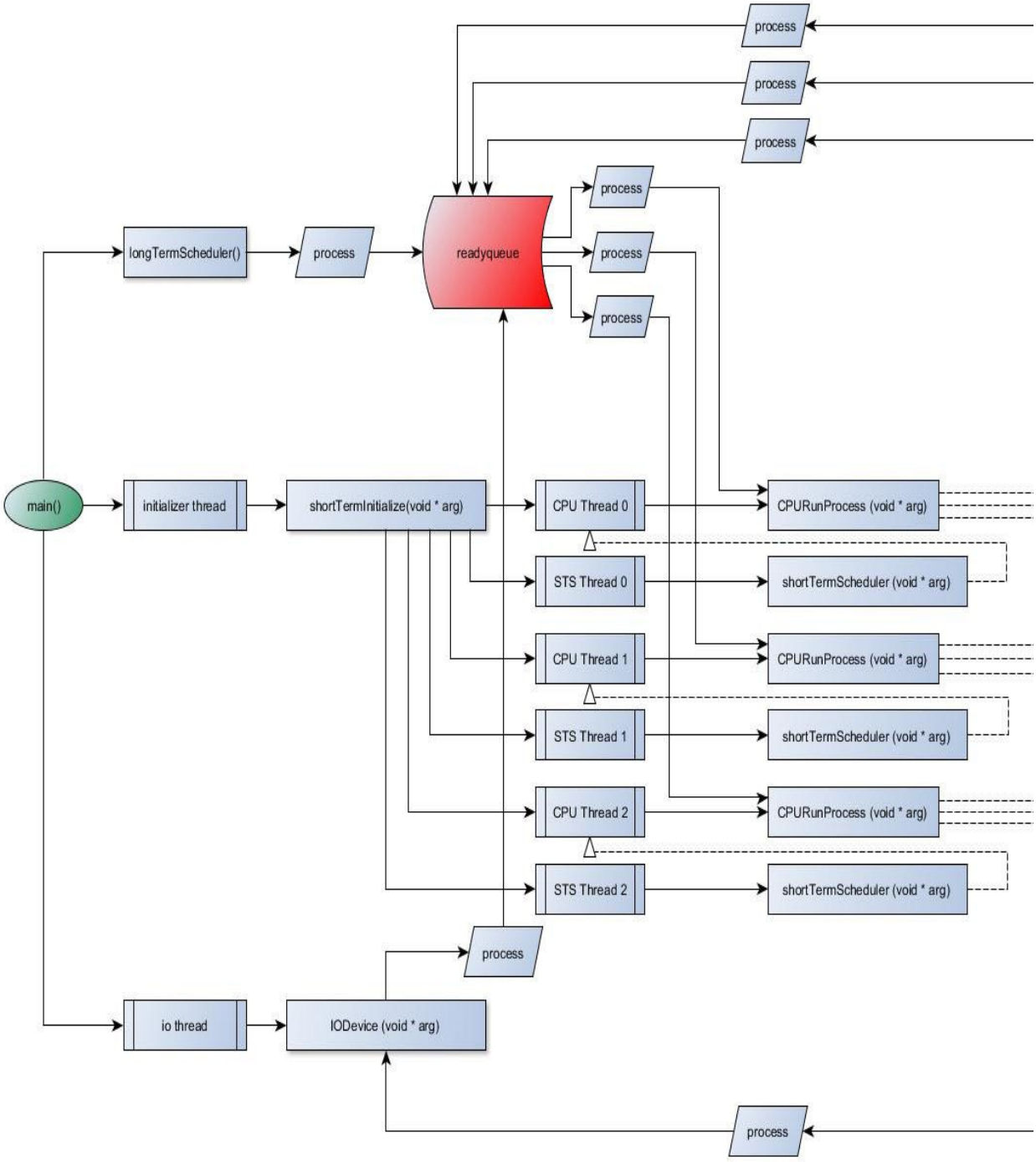
Table of Contents

Bugs / Omissions

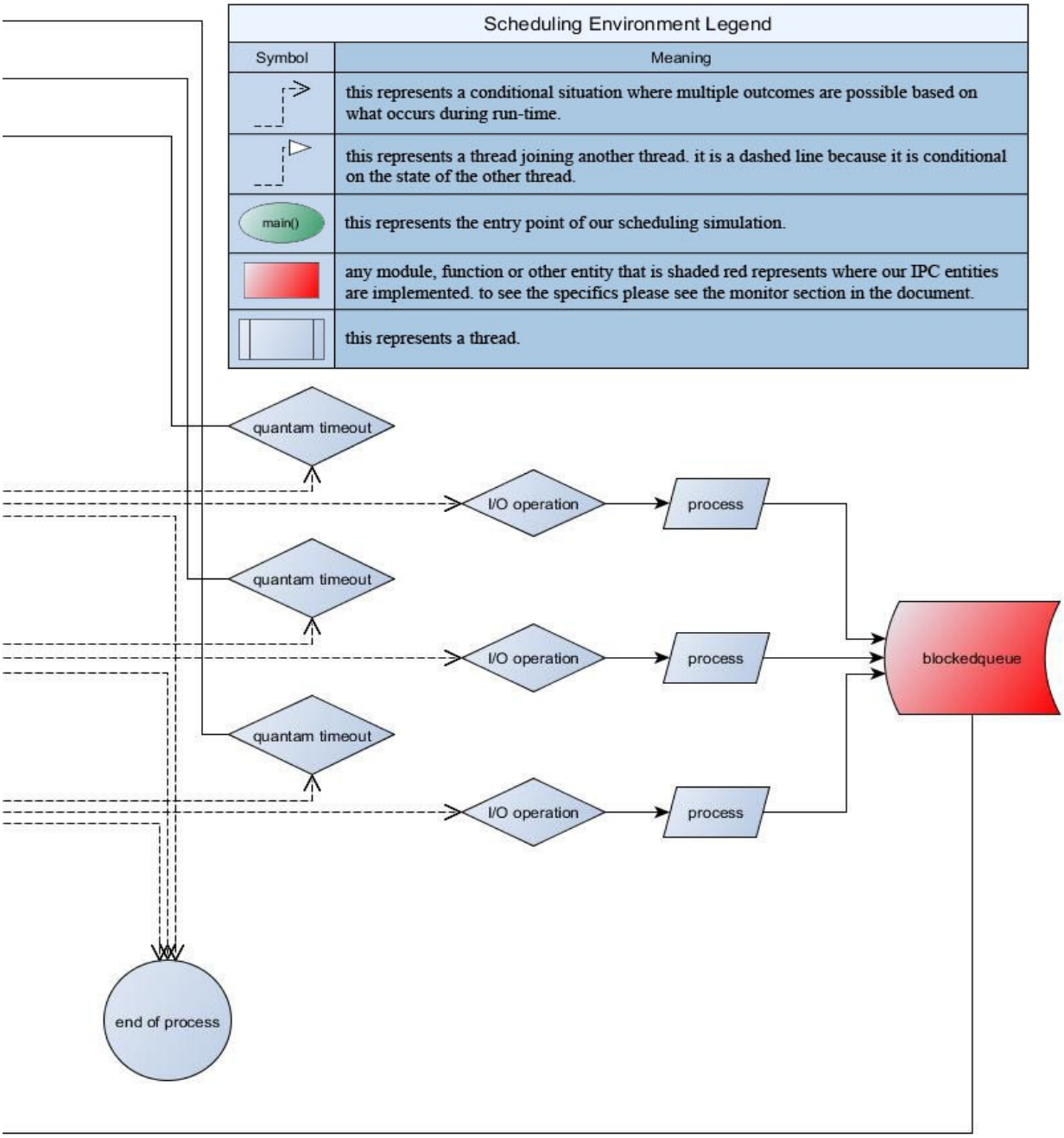
Our project is designed and works according to the specifications listed in the project 2 *bonus* criteria.

Architectural Designs

Scheduling Environment



[Left Side] ... continued on next page.


Scheduling Environment Legend
Symbol
Meaning
this represents a conditional situation where multiple outcomes are possible based on what occurs during run-time.
this represents a thread joining another thread. it is a dashed line because it is conditional on the state of the other thread.
main()
this represents the entry point of our scheduling simulation.
any module, function or other entity that is shaded red represents where our IPC entities are implemented. to see the specifics please see the monitor section in the document.
this represents a thread.
quantam timeout
quantam timeout
quantam timeout
I/O operation
I/O operation
I/O operation
process
process
process
blockedqueue
end of process

[Right Side]

Simulation Experiment

Simulation Steps:

-initialize a ready queue (from the **readyqueue** class) to hold processes (from the **process** class). This is implemented as a multi-level feedback queue (MLFQ – 3 levels).
-initialize a blocked queue (from the **blockedqueue** class) to hold processes (from the **process** class). Processes are considered blocked if and only if they had an I/O operation while running on the CPU.
-initialize 3 threads to simulate CPUs (all threads are implemented using the **pthread** package).
-initialize 3 threads to simulate short-term-schedulers (STS) which continually attempt to **pthread_join** its matching CPU counterpart thread (based on id, only possible if not running a process). If they do join, a new process is sent to the **CPURunProcess** function (implemented in **schedulers.cpp**) from the ready queue.
-initialize a thread that continually simulates a process from the blocked queue receiving I/O and becoming unblocked. This process is then moved to the ready queue. This is done with the **IODevice** function call (implemented in **schedulers.cpp**).

Simulation Data Outputs:

-the simulation outputs the elapsed time of each simulation (this variable is named **elapsed** and calculated using **sys/time.h** facilities. This value represents milliseconds.
-the simulation outputs the number of processes run on CPUs (this number counts the number of times a process is run on a CPU with **CPURunProcess**. The simulation ends when the integer variable **TERMINATE_NOW** processes have been run (this variable counts the number of COMPLETED processes (ones that reach their end of file).

Main Program Logic

The first thing the main.cpp program does is to spawn a thread whose job it is to make three threads representing the three CPUs and three short-term schedulers. Main.cpp then spawns a thread simulating an I/O device which randomly unblocks processes from the blocked queue. It then calls the long-term scheduler. After the long-term scheduler returns, it attempts to join the initializing thread and the I/O device.

Processes are simulated with an array of instructions of random length containing randomly either CPU-bound computations or traps to I/O.

The CPUs take processes as input (usually from the ready queue) and loop through the instructions of a process. If the process calls for I/O, it is added to the blocked queue. If it is detected that a process is hogging the CPU, it is timed out and added back to the ready queue.

The three short-term schedulers attempt to join the CPU threads, and then make them again with a new process from the ready queue. If none is available, they block (in the method readyqueue::pop()).

The I/O device continually unblocks a random process from the blocked queue, and then waits.

The ready queue is where the processes stay which are ready to be given to the CPU because they have been allocated all the resources they need except the CPU. The blocked queue is where processes go which are waiting on an I/O event.

The long-term scheduler simply creates new processes and adds them to the ready queue. It terminates once it has added schedulers::TERMINATE_NOW processes to the ready queue. The I/O device and the short-term schedulers terminate once this condition has been met and additionally there are no processes left in the ready or blocked queues.

Monitor Implementation

The interface for the ready queue is this:

```
readyqueue();
~readyqueue();
void push(process * p);        // method to add a process to the ready queue
process * pop(void);           // method to get a process from the ready queue and delete it
unsigned int size(void);       // method to get the current size of the ready queue
bool empty(void);              // method to check if the queue is empty
```

The class is implemented as a monitor so that only one of its methods can be run at any time. This is achieved by using a pthread mutex with a recursive attribute. The recursive attribute allows one thread to lock the mutex more than once but requires it to unlock it the same number of times before other threads may lock it.

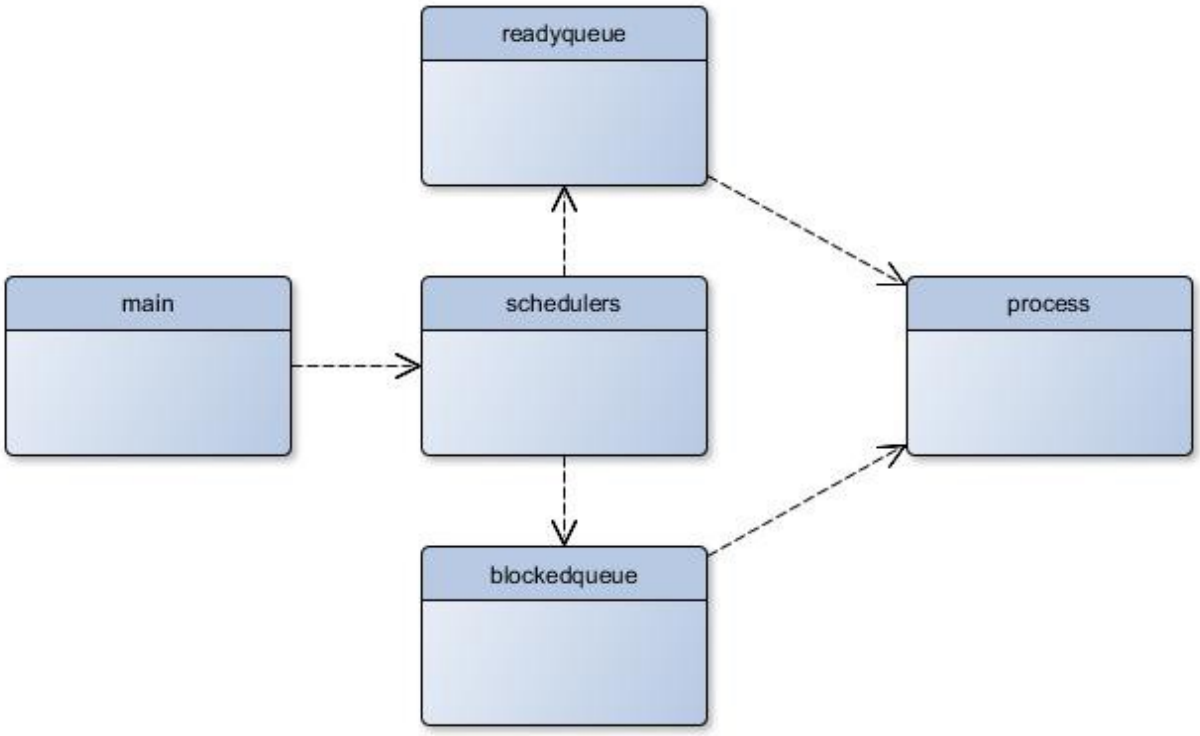
If a process calls pop() when the queue is empty, it blocks on a condition. That condition is signaled by the push(process *) method.

The blocked queue is likewise a monitor. Its interface is this:

```
blockedqueue();
~blockedqueue();
void block(process* a);              // method to block a process
process * IOFinish(int position);    // method to unblock a process
unsigned int size();                 // method to get the size of the ready queue
```

It uses a simple pthread mutex with the default attribute and no conditions.

Module Dependency Diagram



Results Discussion

Performance Metrics

The performance metric that we measured was the total run-time of the program under varying time quanta. This was measured over 8 runs per quanta and was run on the same computer since results would most likely differ on varying machines due to differences in hardware etc. We also collected the number of processes passed to the CPURunProcess function.

Simulation Results

We found that a time quantum of 8 was the best on average for our simulation of the MLFQ. (In our case the time quantum is defined as the number of instructions read before the process is timed out.) While we are unsure of why this is, our hypothesis is that it has to do with the balance between the time it takes for context switching and the fact that if the CPUs run less, the other threads in the program can run more, generating more resources.

Quantum	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Average
2	89	112	122	122	127	115	145	108	117.5
4	133	101	124	103	116	119	144	85	115.6
8	**97**	**110**	**94**	**70**	**99**	**78**	**111**	**90**	**93.6**
16	109	95	100	123	108	105	102	107	106.1
32	101	104	107	82	120	123	111	71	102.4

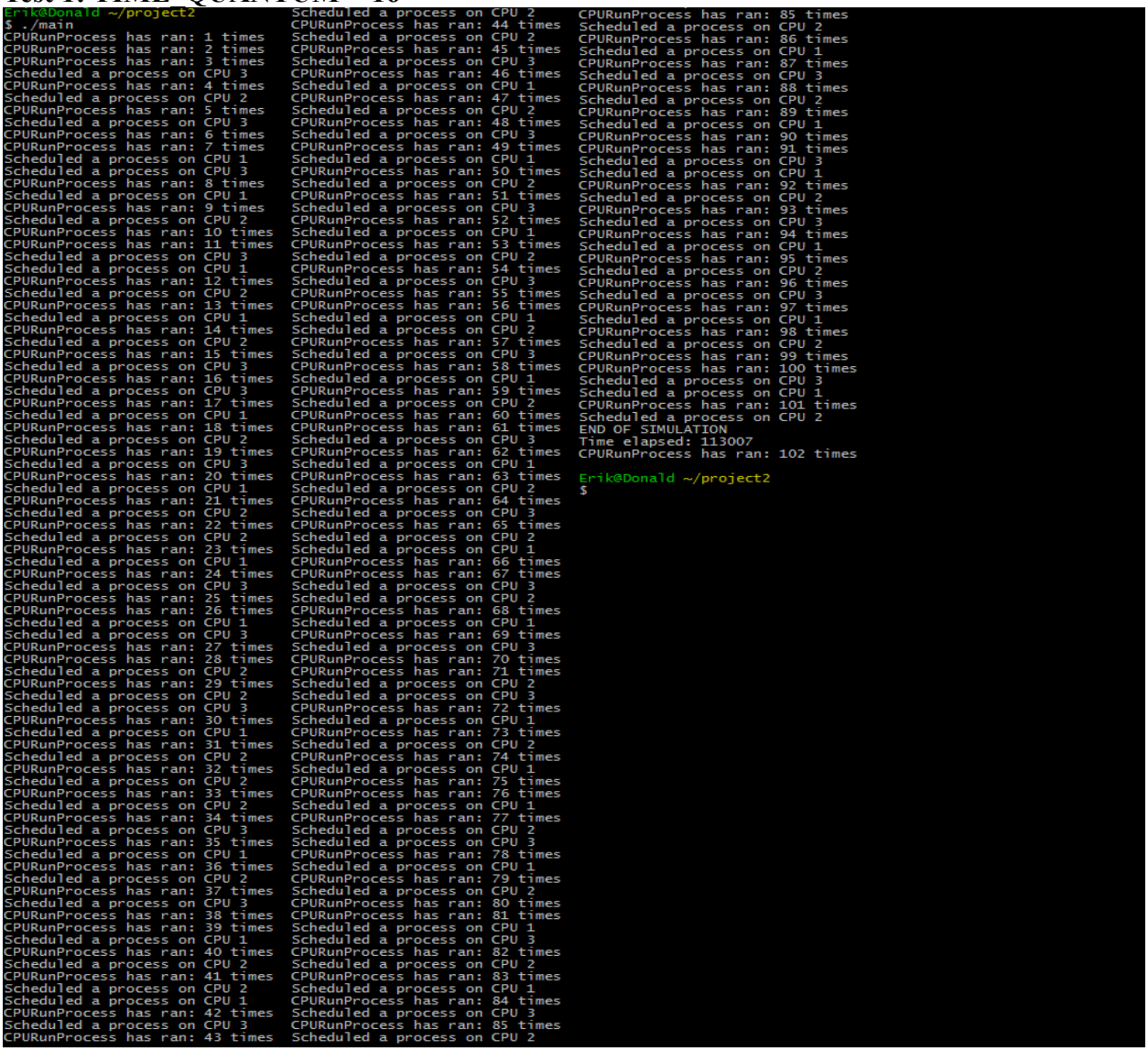
**above times are in milliseconds*

Performance Characteristics

We hypothesize that the average wait time for processes at time quantum 8 would be the least as well. The traps to I/O in the simulation are roughly 10% of the instructions, so roughly 80% of processes would not time out during that quantum. 80% was the number that in class, the professor claimed was optimal for process wait times.

Sample Annotated Output

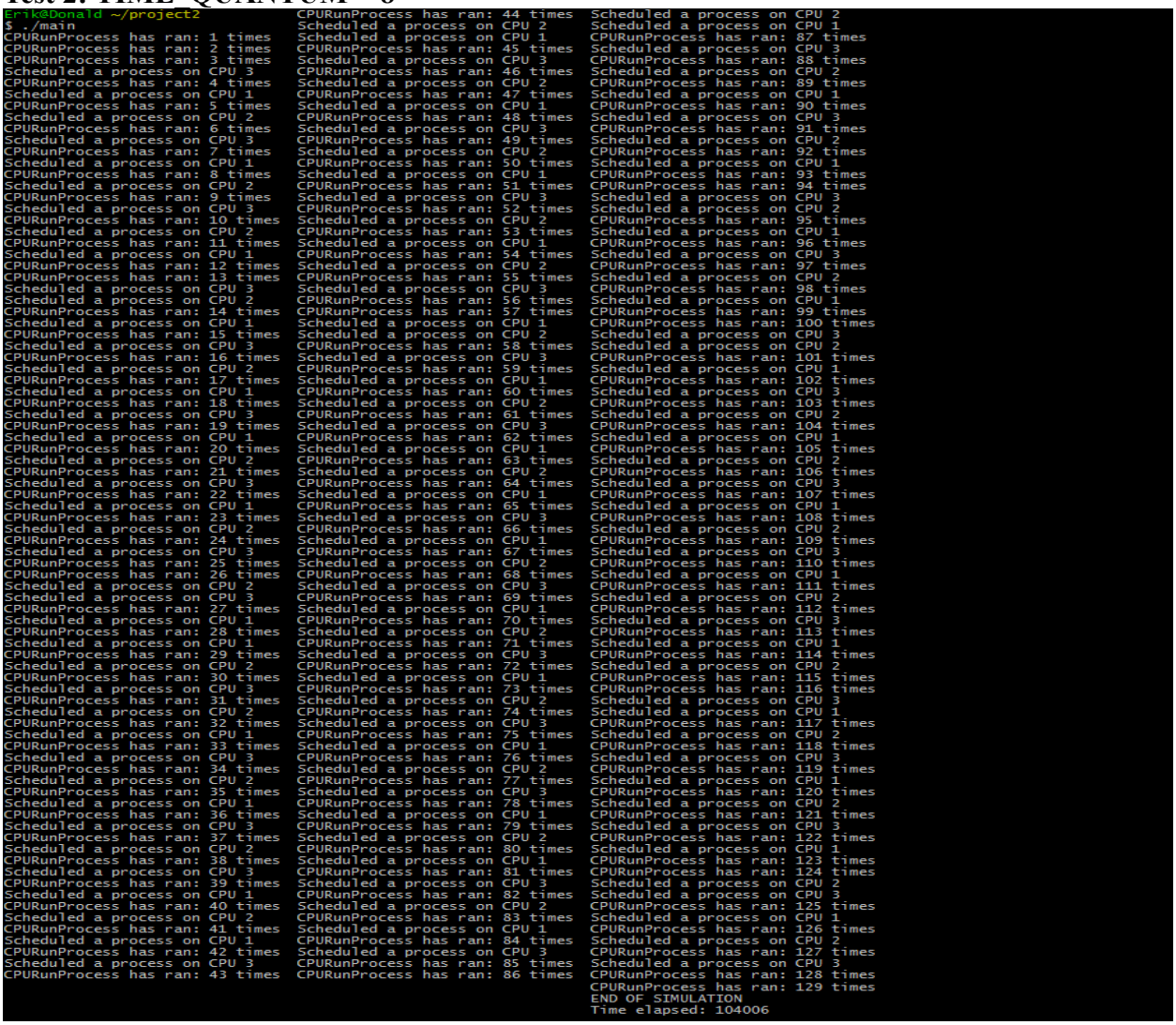
Test 1: TIME_QUANTUM = 16



Annotations:

-TIME_QUANTUM = 16
-102 processes were ran through the CPURunProcess method
-the elapsed time was 113007

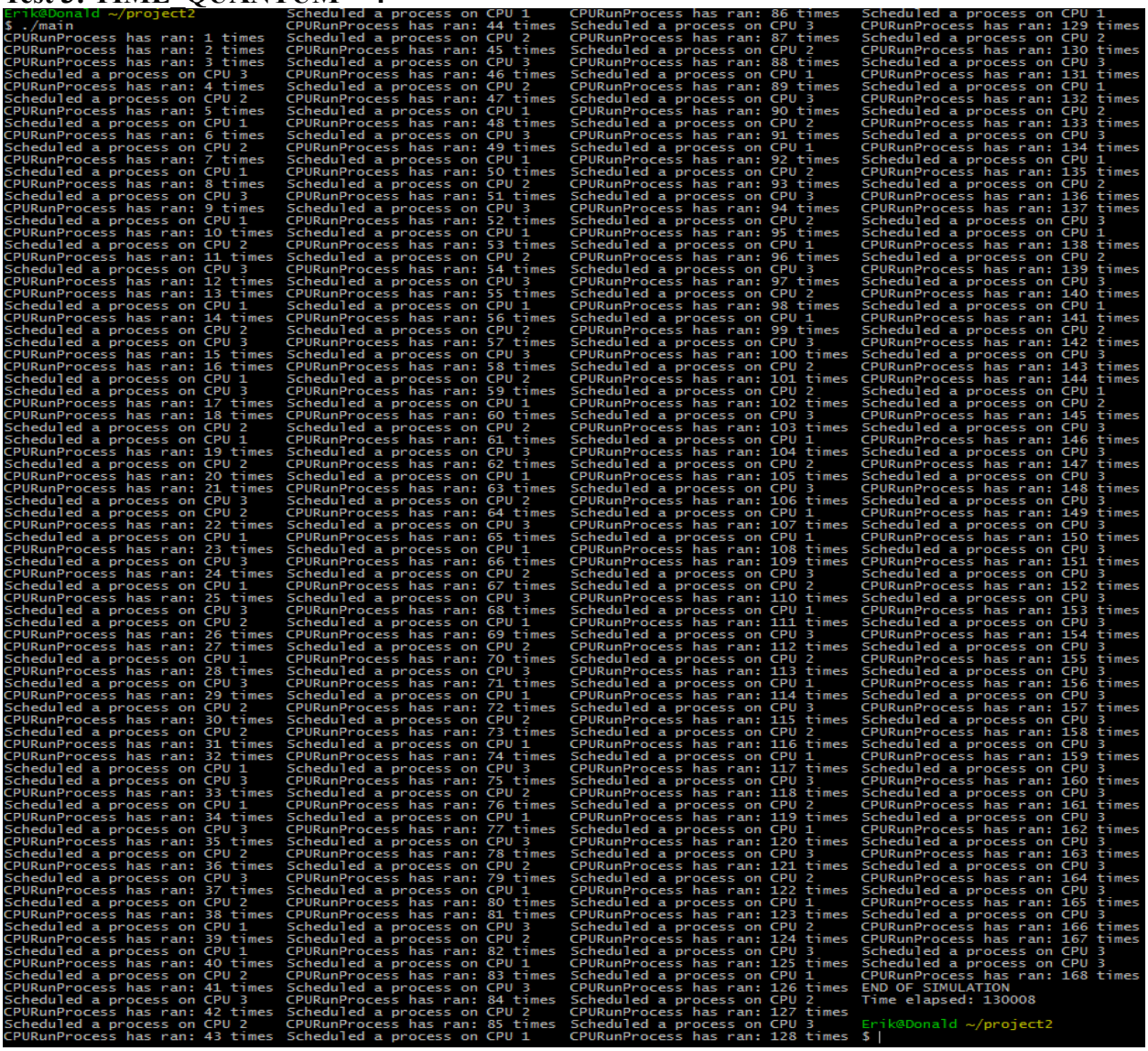
Test 2: TIME_QUANTUM = 8



Annotations:

-TIME_QUANTUM = 8
-129 processes were ran through the CPURunProcess method
-the elapsed time was 104006

Test 3: TIME_QUANTUM = 4



Annotations:

-TIME_QUANTUM = 4
-168 processes were ran through the CPURunProcess method
-the elapsed time was 130008

Source Code

```
// ----------------------------------------
// CMPT 300 Project 2
// main.cpp (file 1 of 9)
// ----------------------------------------

#include "schedulers.h"
#include <iostream>
#include <stdlib.h>
#include <pthread.h>
#include <sys/time.h>

extern pthread_mutex_t mutexNumProcesses;
extern pthread_mutex_t output;

using namespace std;

/*/ implementation overview

        *main() is used to run the simulation of processes (see process.h)
         in our multi-level feedback queue (see readyqueue.h).

        *if an I/O operation occurs and then our processes are sent
         to a 'blocked' queue (see blockedqueue.h). blocked processes
         that become unblocked are sent back to the MLFQ.

        *we have used mutexes to fake monitors for IPC situations.
/*/

int main()
{
        srand(time(0));
              // seed all the rand()s once and only once

        pthread_t initializer;
        pthread_t io;

        struct timeval start;
                        //use these to measure the time
        struct timeval end;
        long elapsed;
        gettimeofday(&start, NULL);

        pthread_create(&initializer, NULL, shortTermInitialize, NULL);
        pthread_create(&io, NULL, IODevice, NULL);

        longTermScheduler();
```

```
	pthread_join(initializer, NULL);

	pthread_join(io, NULL);


	gettimeofday(&end, NULL);
	elapsed = 1000000*(long)(end.tv_sec) + (long)(end.tv_usec) - (1000000*(long)(start.tv_sec) +
(long)(start.tv_usec));

	cout << "END OF SIMULATION" << endl;
	cout << "Time elapsed: " << elapsed << endl;

	pthread_mutex_destroy(&output);
	pthread_mutex_destroy(&mutexNumProcesses);
	return 0;
}
```

```
// ----------------------------------------
// CMPT 300 Project 2
// process.h (file 2 of 9)
// ----------------------------------------

#ifndef PROCESS_H
#define PROCESS_H

#include <iostream>
#include <stdlib.h>
#include <time.h>

/*/ class:   process
          purpose: simulates a process with an array of instructions, which are either
                                CPU-bound computation or else traps to input/output devices.
/*/

class process
{
   public:
      static const int CPU = 0;                    // defines a CPU-bound operation
      static const int IO = 1;                     // defines an IO-bound operation
      static const int END_OF_FILE = -1;     // defines the end of a process
                  int numTimeouts;                 // used for the multi-level ready queue
      int next(void);                                                  // method to retrieve the next operation
      process();
      virtual ~process();
   protected:
   private:
      int * instructions;                                        // used for process creation/randomization
      int counter;                      // ^
      int length;                       // ^
      int probability;                                           // ^
      int cpuCluster;                                                  // ^
};

#endif // PROCESS_H
```

```
// ----------------------------------------
// CMPT 300 Project 2
// process.cpp (file 3 of 9)
// ----------------------------------------

#include "process.h"

process::process()
{
   counter = 0;
          numTimeouts = 0;
          probability = 0;
   length = rand()%246+10;
   cpuCluster = 0;
   instructions = new int[length];         // array of random length between 10 and 255

   for (int i=0; i<length; i++)                        // each entry of instructions[] will be an IO or CPU
   {
      if (rand()%100 < (10 + probability) && cpuCluster == 0)
      {
         instructions[i] = IO;              // assign an IO operation
         if(probability < 21)
         {
            probability++;             // increasing probability to distribute an IO operation
         }
      }
      else
      {
         instructions[i] = CPU;           // assign a CPU operation
         probability = 0;
         if(rand()%100 < 1)                              // 1% chance for heavy CPU-bound processes
         {
            cpuCluster = rand()%30 + 1; // randomly determine how many CPU operations will follow
         }
         if(cpuCluster > 0)
         {
            cpuCluster--;
         }
      }
   }
}

process::~process()
{
   delete [] instructions;
}

// public method: process::next()
//      function: increments the program counter and returns the next instruction.
```

```
//              if it is at the end of the file, returns END_OF_FILE
int process::next()
{
   if (counter==length)
          {
      return END_OF_FILE;
   }
          else
          {
      return instructions[counter++];
   }
}
```

```
// ----------------------------------------
// CMPT 300 Project 2
// readyqueue.h (file 4 of 9)
// ----------------------------------------

#ifndef READYQUEUE_H
#define READYQUEUE_H

#include "process.h"
#include <stdlib.h>
#include <pthread.h>
#include <queue>

/*/ class:   readyqueue
        purpose: this is a multi-level queue of processes (see process.h)
                implemented as a linked list.
/*/

class readyqueue
{
private:
        std::queue<process *> queues [3];
        pthread_mutex_t myMutex;
        pthread_mutexattr_t recursive;
        pthread_cond_t emptyQ;
public:
        readyqueue();
        ~readyqueue();
        void push(process * p);                         // method to add a process to the ready queue
        process * pop(void);                        // method to get a process from the ready queue and
delete it
        unsigned int size(void);            // method to get the current size of the ready queue
        bool empty(void);               // method to check if the queue is empty
};

#endif // READYQUEUE_H
```

```
// ----------------------------------------
// CMPT 300 Project 2
// readyqueue.cpp (file 5 of 9)
// ----------------------------------------

#include "readyqueue.h"

readyqueue::readyqueue()
{
	pthread_mutexattr_init(&recursive);
	pthread_mutexattr_settype(&recursive, PTHREAD_MUTEX_RECURSIVE);
	pthread_mutex_init(&myMutex, &recursive);
	// allow the same thread to hold the same mutex more than once in a recursive function call
	// instead of blocking itself
	emptyQ = PTHREAD_COND_INITIALIZER;
}

readyqueue::~readyqueue()
{
	pthread_mutex_destroy(&myMutex);
	pthread_cond_destroy(&emptyQ);
	pthread_mutexattr_destroy(&recursive);
}

// public method: readyqueue::size()
//	function: returns the size of the ready queue
unsigned int readyqueue::size()
{
	pthread_mutex_lock(&myMutex);
	unsigned int temp = queues[0].size() + queues[1].size() + queues[2].size();
	pthread_mutex_unlock(&myMutex);
	return temp;
}

// public method: readyqueue::push(process * p)
//	function: add a process to queue
void readyqueue::push(process * p)
{
	//the level of a process in the multi-level feedback queue
	//corresponds to the number of times it has timed out.
	pthread_mutex_lock(&myMutex);
	int level = (p->numTimeouts>=2 ? 2 : p->numTimeouts);
	queues[level].push(p);
	pthread_mutex_unlock(&myMutex);
	pthread_cond_signal(&emptyQ);
}

// public method: readyqueue::pop()
//	function: add a process to queue
```

```
process * readyqueue::pop()
{
	pthread_mutex_lock(&myMutex);
	while (empty())				// wait until the queue is not empty to pop()
	{
		pthread_cond_wait(&emptyQ, &myMutex);
	}
	process * temp;
	if (!queues[0].empty())		// pop() a process from the 1st priority level
	{
		temp = queues[0].front();
		queues[0].pop();
	} else if (!queues[1].empty())  // pop() a process from the 2nd priority level
	{
		temp = queues[1].front();
		queues[1].pop();
	} else				// pop() a process from the 3rd priority level
	{
		temp = queues[2].front();
		queues[2].pop();
	}
	pthread_mutex_unlock(&myMutex);
	return temp;
}

// public method: readyqueue::empty()
//	function: returns true if nothing is in the queue
bool readyqueue::empty()
{
	pthread_mutex_lock(&myMutex);
	bool temp = size()==0;
	pthread_mutex_unlock(&myMutex);
	return temp;
}
```

```
// ----------------------------------------
// CMPT 300 Project 2
// blockedqueue.h (file 6 of 9)
// ----------------------------------------

#ifndef BLOCKEDQUEUE_H
#define BLOCKEDQUEUE_H

#include "process.h"
#include <queue>
#include <vector>
#include <pthread.h>

using namespace std;

/*/ class:   blockedqueue
          purpose: this is a single-level queue of processes (see process.h)
                implemented with a vector and a queue.
/*/

class blockedqueue
{
   public:
      void block(process* a);                                    // method to block a process
      process * IOFinish(int position);          // unblock a process
                  unsigned int size();                                    // size of the queue
                  blockedqueue();
                  ~blockedqueue();
   private:
                  pthread_mutex_t myMutex;
      queue<process*> ready;
      vector<process*> notready;
};

#endif // BLOCKEDQUEUE_H
```

```
// ----------------------------------------
// CMPT 300 Project 2
// blockedqueue.cpp (file 7 of 9)
// ----------------------------------------

#include "blockedqueue.h"

blockedqueue::blockedqueue()
{
	myMutex = PTHREAD_MUTEX_INITIALIZER;
}

blockedqueue::~blockedqueue()
{
	pthread_mutex_destroy(&myMutex);
}

// public method: blockedqueue::size()
//	function: returns the size of the blocked queue
unsigned int blockedqueue::size()
{
	pthread_mutex_lock(&myMutex);
	unsigned int temp = notready.size();
	pthread_mutex_unlock(&myMutex);
	return temp;
}

// public method: blockedqueue::block(process* a)
//	function: add a blocked process to the vector of blocked processes
void blockedqueue::block(process* a)
{
	pthread_mutex_lock(&myMutex);
   notready.push_back(a);
	pthread_mutex_unlock(&myMutex);
}

// public method: blockedqueue::IOFinish(int position)
//	function: when a process receives I/O, erase it from the vector and return it
process * blockedqueue::IOFinish(int position)
{
	pthread_mutex_lock(&myMutex);
   process* temp = notready[position];
   notready.erase(notready.begin()+position);
	pthread_mutex_unlock(&myMutex);
	return temp;
}
```

```
// ----------------------------------------
// CMPT 300 Project 2
// schedulers.h (file 8 of 9)
// ----------------------------------------

#ifndef SCHEDULERS_H
#define SCHEDULERS_H

#include "process.h"
#include "readyqueue.h"
#include "blockedqueue.h"
#include <iostream>
#include <unistd.h>
#include <stdlib.h>
#include <math.h>

/*/ files:    schedulers.h, schedulers.cpp
          purpose:  encapsulate our scheduler implementation into a seperate file.
          function: create processes in our MLFQ, simulate running a process on a CPU,
                  simulate I/O blocking, simulate a process becoming unblocked and
                              returning to the MLFQ.
/*/

void longTermScheduler ();                // creates processes to be run and pushes the to the ready queue
void * shortTermInitialize (void * arg);    // initializes CPUs and STSs
void * shortTermScheduler (void * arg);     // continually check if threads are done or not
void * CPURunProcess (void * arg);                          // simulates running a process from the ready
queue
void * IODevice (void * arg);                           // simulates unblocking processes in the blocked queue
const int TIME_QUANTUM = 16;
const int MAX_MULTIPROGRAM = 16;
const int TERMINATE_NOW = 5;                // arbitrary number of processes to be run in our
simulation so
                                    // it doesn't run indefinitely.
const int CONTEXT_SWITCH = 100;                                 // microseconds it takes for a context
switch

#endif // SCHEDULERS_H
```

```
// ----------------------------------------
// CMPT 300 Project 2
// schedulers.cpp (file 9 of 9)
// ----------------------------------------

#include "schedulers.h"

readyqueue r;
blockedqueue b;
pthread_t CPUthreads [3];
pthread_t schedulerThreads [3];
pthread_t io;
pthread_mutex_t mutexNumProcesses = PTHREAD_MUTEX_INITIALIZER; // don't increment
process::numProcesses twice to the same value
pthread_mutex_t output = PTHREAD_MUTEX_INITIALIZER; // only output one thing at a time or it
will be garbled
int numCPUProcesses = 0; // number of processes that pass through the CPUs
int numLTSProcesses = 0; // number of processes that pass through the long-term scheduler

// function: longTermScheduler()
//  purpose: creates processes and pushes them to the ready queue
void longTermScheduler()
{
	while (numLTSProcesses <= TERMINATE_NOW)                    // only simulate up
to TERMINATE_NOW
	{
		if (r.size()>=MAX_MULTIPROGRAM)
		{
			// we don't want too many processes in the ready queue
			// the LTS will not schedule more than MAX_MULTIPROGRAM processes in
the queue
			// but if stuff migrates there from the blocked queue then it may end up having
more.
			pthread_yield();

			continue;
		}
		process * p = new process();
		r.push(p);
		numLTSProcesses++;
	}
}

// function: shortTermInitialize(void * arg)
//  purpose: initializes our simulation CPUs and STSs
void * shortTermInitialize(void * arg)
{
	for (int i=0; i<3; i++)
	{
```

```
                pthread_create(&CPUthreads[i], NULL, CPURunProcess, (void *)new process());       //
run 3 CPU threads
        }
        for (int i=0; i<3; i++)
        {   // We want all the CPURunProcesses to be created before the short term schedulers
                // so that they each at least have a chance to run
                pthread_create(&schedulerThreads[i], NULL, shortTermScheduler, (void *)&i);  // run 3
STS threads for each CPU
        }
}

// function: shortTermScheduler (void * arg)
//  purpose: continually check whether each thread is done or not
void * shortTermScheduler (void * arg)
{
        int i = *(int *)arg;
        while (true)
        {
                pthread_join(CPUthreads[i], NULL);
                                // try to join the thread
                if (numLTSProcesses <= TERMINATE_NOW || r.size()!=0 || b.size()!=0)
                {
                        pthread_create(&CPUthreads[i], NULL, CPURunProcess, (void *)r.pop());       //
send a process to a CPU
                        pthread_mutex_lock(&output);
                        cout << "Scheduled a process on CPU " << i << endl;
                        pthread_mutex_unlock(&output);
                }
                else {
                        break;
                }
                pthread_yield();
        }
        return 0;
}

// function: CPURunProcess (void * arg)
//  purpose: simulates a process running on a CPU. iterates through process' instructions[]
void * CPURunProcess (void * arg)
{
        process * p = (process *)arg;
        pthread_mutex_lock(&mutexNumProcesses);
        numCPUProcesses++; // this is a critical section as discussed in class
        pthread_mutex_lock(&output);
        cout << "CPURunProcess has ran: " << numCPUProcesses << " times" << endl;
        pthread_mutex_unlock(&output);
        pthread_mutex_unlock(&mutexNumProcesses);
        int counter = 1;
        int next = p->next();
```

```
	while (next!=process::END_OF_FILE)
	{
		if (next==process::IO)
		{
			// this simulates a trap to IO. We want to block the process and resume it later
			b.block(p);
			usleep(CONTEXT_SWITCH); // simulates a context switch
			return 0;
		}
		counter++;
		if (counter==(pow(2.0, (double)(p->numTimeouts))*TIME_QUANTUM))
		{	// a process from level 1 gets TIME_QUANTUM instructions,
			// one from level 2 gets twice that, level three twice that, etc.

			// this simulates a timing out of the process. We want to add it back to the ready
queue.
			if (p->numTimeouts < 3)
			{
				p->numTimeouts++;
			}
			r.push(p);
			usleep(CONTEXT_SWITCH);
			return 0;
		}
		next = p->next();
		pthread_yield(); // maybe someone else wants a chance
	}
	// we reach this point in the code if the process has reached the end of its file
	return 0;
}

// function: IODevice (void * arg)
//  purpose: simulates I/O operations unblocking processes from the blocked queue
void * IODevice (void * arg)
{
	while (numLTSProcesses <= TERMINATE_NOW || r.size()!=0 || b.size()!=0)
	{
		while (b.size()==0)
		{
			usleep(1000);
			pthread_yield();
		}
		r.push(b.IOFinish(rand()%b.size()));
		usleep(rand()%1000);
	}
	return 0;
}
```