Please print in pen:
Waterloo Student ID Number:

WatIAM/Quest Login Userid:

UNIVERSITY OF WATERLOO

Examination
Midterm
Fall 2017
ECE 254

Times: Wednesday 2017-10-25 at 17:45 to 18:45 (5:45 to 6:45PM)
Duration: 1 hour (60 minutes)
Exam ID: 3587180
Sections: ECE 254 LEC 001,002
Instructors: Carlos Moreno, Jeff Zarnett

Special Materials

Candidates may bring only the listed aids.

· Calculator - Non-Programmable

Instructions:

1. No aids are permitted except non-programmable calculators with no persistent memory.
2. Turn off all communication devices. Communication devices must be stored with your personal items for the duration of the exam. Taking a communication device to a washroom break during this examination is not allowed and will be considered an academic offence.
3. Place all bags at the front or side of the examination room, or beneath your table, so they are inaccessible.
4. There are two (2) questions, with multiple parts. Not all are equally difficult.
5. The exam lasts 60 minutes and there are 50 marks.
6. Verify that your name and student ID number is on the cover page and that your examination code appears on the bottom of each page of the examination booklet.
7. If you feel like you need to ask a question, know that the most likely answer is “Read the Question”. No questions are permitted. If you find that a question requires clarification, proceed by clearly stating any reasonable assumptions necessary to complete the question. If your assumptions are reasonable, they will be taken into account during grading.
8. A reference sheet is attached as the last page of the examination.
9. Do not fail this city.
10. After reading and understanding the instructions, sign your name in the space provided below.

Signature

Marking Scheme (For Examiner Use Only):

Question	Mark	Weight	Question	Mark	Weight
1a		10	2a		14
1b		10	2b		4
1c		5	2c		4
			2d		3
Total					**50**

© 2017 University of Waterloo
Please initial:

Question 1: Processes and Threads [25 marks total]

1A: Parallelization with Threads [10 marks]

Consider the following code that computes the value of pi (π) using a monte carlo method: n random points are generated and we count the number of those points x that are inside the unit circle. As we generate only positive random numbers, x / n, the ratio of points, represents only one quadrant and must be multiplied by 4 to give an estimate for pi. The C `rand()` function does not parallelize well; instead assume there is a generator function `unsigned int random( unsigned int i )` that generates a random number between 0 and the unsigned integer maximum (4294967296). The implementation of `random` is not shown, but is safe for use in multithreaded programs.

```
#include <stdlib.h>
#include <stdio.h>

#define INT_MAX 4294967296

int compute( int i ) {
  double x = (double)random( i ) / INT_MAX;
  double y = (double)random( i ^ random( i )) / INT_MAX;
  if ( (x*x + y*y) <= 1) {
    return 1;
  }
  return 0;
}
```

In this question, you will use this code to calculate pi. You must be sure that your program deallocates any resources that are allocated and is free of race conditions or other concurrency problems. Remember that what you write in `main` needs to match up with the specification for the `run` function and vice-versa.

Part 1: Thread wrapper. (5 marks) Assume that the `run` function is provided with a pointer to an integer containing the number of iterations to run. It should return a pointer to the number of points x that were within the first quadrant of the circle for its iterations. Complete the `run` function below.

Part 2. Write `main`. (5 marks) In this part of the question, you will write `main` which needs to (1) create the appropriate number of threads, (2) collect the return value from each thread, and (3) compute the value of pi and print it out. Complete `main` below to implement the desired behaviour.

```
void* run( void* argument ) {




}
```

```
int main( int argc, char** argv ) {

  if (argc < 3) {
    printf("Too few arguments provided.\n");
    return -1;
  }

  int num_threads = atoi( argv[1] );
  if (num_threads < 1) {
    printf("Invalid number of threads.\n");
    return -1;
  }

  int num_iterations = atoi( argv[2] );
  if ( num_iterations < 1 ) {
    printf("Invalid number of iterations.\n");
    return -1;
  }









}
```

1B. Operating System Toolkit [10 marks]

For each of the following mechanisms, explain how an operating system makes use of such mechanism, including what the benefit is (2 marks each):

(a) Hardware interrupts

(b) Timer interrupts

(c) Traps

(d) Execution mode (user mode vs. kernel or privileged mode)

(e) The test-and-set instruction

1C. Process Switching [5 marks]

The operating system (OS) routinely gives up control of the processor (e.g., in a uni-processor architecture whenever a process is dispatched for execution). However, the OS relies on the hardware to guarantee that it will regain control of the processor eventually.

(a) Explain what is (are) the hardware mechanism(s) that provide(s) this assurance for the OS. Your explanation should include the cases of processes that are CPU-bound (i.e., intensive computations without I/O) as well as processes that execute I/O operations, under reasonable/standard assumptions on how I/O operations are done.

(b) In addition to the hardware mechanism(s) that provide(s) the guarantee, there are also software mechanisms that facilitate OS operation (i.e., the OS regaining control of the CPU). Name and explain one such software mechanism.

© 2017 University of Waterloo
Please initial:

Question 2: Concurrency and Synchronization [25 marks total]

2A. Harry Potter and the pthread House Elves [14 marks]

At Hogwarts School of Witchcraft and Wizardry, undesirable tasks are done by House Elves, magical creatures who apparently are not covered under any sort of employment standards or workers rights legislation. They work as a team under the direction of Dobby, who is a free elf. Dobby creates a list of items for the elves to do, and the elves do them. This cycle never ends, unfortunately for all involved: Dobby and the elves must keep working.

When the list is empty, Dobby posts 20 tasks to do by calling `post_tasks()`. After that, Dobby sleeps (is blocked). Worker elves take tasks using the function `take_task()` (which cannot be safely run in parallel) and then do the work by calling `do_work()` (which can be done in parallel). If the list of tasks is empty, an elf cannot call `take_task()` and must instead wake up Dobby, who will then post the next 20 tasks before going to sleep again; meanwhile the elf is blocked until the next group of tasks is ready. Note that Dobby and the elves are responsible for managing the count of tasks available.

Complete the code below to implement the functionality described above. You may assume all relevant `#include` directives are present and that all threads are started and terminated elsewhere in the code (these are not shown).

Global variables:

```
int tasks;
pthread_mutex_t mutex;
sem_t empty_list;
sem_t full_list;
```

Initialize global variables in `init` and perform cleanup of anything allocated or initialized in `cleanup` (4 marks):

```
void init() {




}
```

```
void cleanup() {




}
```

Complete the C functions below for the dobby and the house elves (8 marks):

```
void* dobby( void * ignore ) {




}
```

```
void* house_elf( void * ignore ) {




}
```


© 2017 University of Waterloo
Please initial:

2B. The Dining Philosophers [4 marks]

Write a pseudocode description, in the style we have used in lectures, of the behaviour of each of the philosophers in the basic dining philosophers problem (the one where the philosophers can get stuck). There are `n` philosophers and `n` chopsticks. Assume that the following are created and defined:
`id` – the id of each philosopher (from 0 to n-1)
`chopsticks` – a semaphore array of capacity `n`, and
`eat()` – a function that makes the philosopher eat when they have both chopsticks.

2C. Use of Semaphores [4 marks]

A fellow student suggests that general semaphores are really just integers that track the current state and therefore it is possible to replace a semaphore type `sem_t` in the program with a simple `int` that can be incremented and decremented using normal C statements. Give two (2) reasons why using an integer is not equivalent to using a semaphore. Explain.

2D. Semaphore Initial Values [3 marks]

Recall that when a semaphore is created, its initial value is important. The choice of initial value depends on the particular application. For each of the following values, name one application or one synchronization pattern that requires the semaphore to be initialized to that value. In all cases, briefly explain why.

(a) 0

(b) 1

(c) 10

© 2017 University of Waterloo
Please initial:

Reference Sheet

Assume always the C99 standard (e.g., you can declare an integer in the same line as the for statement).

Memory is allocated in C with `malloc()` and to get the size of memory you want to allocate, there is `sizeof`, normally used in conjunction with `malloc`. Example: `int* p = malloc( sizeof( int ) );`

Memory is deallocated using `free`. Example: `free( p );`

An argument can be converted to an integer using the function `int atoi( char* arg )`.

Printing is done using `printf` with formatting. `%d` prints integers; `%lu` prints unsigned longs; `%f` prints double-precision floating point numbers. A newline is created with `\n`.

Some UNIX functions you may need:

```
pid_t fork( )
pid_t wait( int* status )
pid_t waitpid( pid_t pid, int status )
int kill( pid_t pid, int signal ) /* returns 0 returned if signal sent, -1 if an error */

int open(const char *filename, int flags);  /* Returns a file descriptor if successful, -1 on error */
ssize_t read(int file_descriptor, void *buffer, size_t count); /* Returns number of bytes read */
ssize_t write(int file_descriptor, const void *buffer, size_t count); /* Returns number of bytes written */
int rename(const char *old_filename, const char *new_filename); /* Returns 0 on success */
int close(int file_descriptor);
```

When opening a file the following flags may be used for the `flags` parameter (and can be combined with bitwise OR, the `|` operator):

Value	Meaning
`O_RDONLY`	Open the file read-only
`O_WRONLY`	Open the file write-only
`O_RDWR`	Open the file for both reading and writing
`O_APPEND`	Append information to the end of the file
`O_TRUNC`	Initially clear all data from the file
`O_CREAT`	Create the file
`O_EXCL`	If used with `O_CREAT`, the caller MUST create the file; if the file exists it will fail

For your convenience, a quick table of the various pthread and semaphore functions we have discussed:

```
pthread_create( pthread_t *thread, const pthread_attr_t *attributes,
                void *(*start_routine)( void * ), void *argument )
pthread_join( pthread_t thread, void **returnValue )
pthread_detach( pthread_t thread )
pthread_cancel( pthread_t thread )
pthread_testcancel( ) /* If the thread is cancelled, this function does not return (thread terminated) */
pthread_exit( void *value )
pthread_mutex_init( pthread_mutex_t *mutex, pthread_mutexattr_t *attributes )
pthread_mutex_lock( pthread_mutex_t *mutex )
pthread_mutex_trylock( pthread_mutex_t *mutex ) /* Returns 0 on success */
pthread_mutex_unlock( pthread_mutex_t *mutex )
pthread_mutex_destroy( pthread_mutex_t *mutex )

sem_init( sem_t* semaphore, int shared, int initial_value); /* 0 for shared OK */
sem_destroy( sem_t* semaphore )
sem_wait( sem_t* semaphore )
sem_post( sem_t* semaphore )
```


© 2017 University of Waterloo
Please initial: