Please print in pen:
Waterloo Student ID Number:

WatIAM/Quest Login Userid:

UNIVERSITY OF WATERLOO

Examination
Midterm
Winter 2017
ECE 459

Times: Saturday 2017-02-11 at 08:30 to 09:30
Duration: 1 hour (60 minutes)
Exam ID: 3413893
Sections: ECE 459 LEC 001
Instructors: Jeffrey Zarnett

Open Book

Candidates may bring any reasonable aids. Open book, open notes. Calculators without communication capability permitted.

Instructions:

1. This exam is open book, open notes.
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 interpreted as 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 four (4) questions. 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. Do not fail this city.
9. After reading and understanding the instructions, sign your name in the space provided below.

Signature

Marking Scheme (For Examiner Use Only):

Question	Mark	Weight
1		10
2		10
3		12
4		18
Total		**50**

Please initial:

Question 1: Short Answer [10 marks]

Answer these questions using at most three sentences. Each is worth 2 marks.

(a) Here's a function prototype: `void swap(int* restrict a, int* restrict b)`. Write a call to `swap` which does not respect the restrict qualifier, including necessary context.

(b) Define and give an example of a loop-carried dependency.

(c) Using the effective access time formula below, calculate what the hit rate h must be to achieve an effective access time of 16 cycles if the cache access time is 9 cycles and the memory access time is 300 cycles. If your answer is in decimal form, round to 2 decimal places.

$$\text{Effective Access Time} = h \times t_c + (1 - h) \times t_m$$

(d) Explain *bandwidth* and *latency* using either a bank-branch or grocery-store analogy.

(e) Can these two statements be run in parallel? Explain.

```
int y = f(x);
int z = g(x);
```


Please initial:

Question 2: OpenMP [10 marks]

The program below is a small variation on the program `search`, originally a solution to an ECE 254 exam problem, which we have used in a number of examples (such as to demonstrate pthreads and valgrind). Assume the appropriate `#include` directives are present (but not shown). There is also large integer array called `array` defined and initialized, and entries in the array are unique (no duplicate values). You may also assume a standard of C99.

```
1  #define NUM_THREADS 8
2  #define ARRAY_SIZE 12000
3
4  typedef struct parameter {
5      int start;
6      int end;
7      int sv;
8  } parameter_t;
9
10 void *search( void *void_arg ) {
11     int i;
12     parameter_t *arg = (parameter_t *) void_arg;
13     int *result = malloc( sizeof( int ) );
14     *result = -1; // Default value
15
16     for ( i = arg->start; i < arg->end; ++i ) {
17         if ( array[i] == arg->sv ) {
18             *result = i;
19             break;
20         }
21     }
22     free( void_arg );
23     pthread_exit(result);
24 }
```

```
26 int main( int argc, char** argv ) {
27     int search_val;
28
29     if ( argc < 2 ) {
30         printf("A search value is required.\n");
31         return -1;
32     }
33     search_val = atoi( argv[1] );
34
35     pthread_t threads[NUM_THREADS];
36     void* returnValue;
37     for ( int i = 0; i < NUM_THREADS; ++i ) {
38         parameter_t* params = malloc( sizeof ( parameter_t ) );
39         params->start = i * (ARRAY_SIZE / NUM_THREADS);
40         int end = (i + 1) * (ARRAY_SIZE / NUM_THREADS);
41         if ( i == (NUM_THREADS - 1) ) {
42             end = ARRAY_SIZE;
43         }
44         params->end = end;
45         params->sv = search_val;
46
47         pthread_create(&threads[i], NULL, search, params);
48     }
49
50     for ( int i = 0; i < NUM_THREADS; ++i ) {
51         pthread_join( threads[i], &returnValue );
52         int *rv = (int *) returnValue;
53         if ( -1 != *rv ) {
54             printf("Found at %d by thread %d\n", *rv, i);
55         }
56         free( returnValue );
57     }
58
59     pthread_exit(0);
60 }
```

Now, you will rewrite this code using OpenMP instead of pthreads, so that it is equivalent in functionality (searches the array in parallel) and produces the same output. You are allowed to change the signature of the search function. You do not have to put in any `#include` directives. You should also let OpenMP choose the number of threads. As before, the large integer array called `array` is defined and initialized without any duplicate values in it, and the C standard is C99.

Please initial:

Question 3: Norse Mythology [12 marks]

In this question you take the job of Heimdall: he guards the rainbow bridge to Valhalla. Only the worthy may pass. The following program has been submitted to you for your judgement and you will do what the tools Valgrind and Helgrind do, only better. In addition to reporting memory leaks and concurrency problems you will also explain how to fix the problems you identify. Consider the code below:

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <sys/types.h>
4  #include <pthread.h>
5
6  #define NUM_CPUS 8
7
8  int index = 0;
9  int* results;
10 int max_val;
11
12 int compare (const void * a, const void * b) {
13    return ( *(int*)b - *(int*)a );
14 }
15
16 int is_prime( int num ) {
17     if (num <= 1) {
18         return 0;
19     } else if (num <= 3) {
20         return 1;
21     } else if (num % 2 == 0 || num % 3 == 0) {
22         return 0;
23     }
24     for (int i = 5; i*i <= num; i+=6 ) {
25         if (num % i == 0 || num % (i + 2) == 0) {
26             return 0;
27         }
28     }
29     return 1;
30 }
31
32 void *find_primes( void *void_arg ) {
33     int *start = (int *) void_arg;
34     int *count = malloc( sizeof( int ) );
35     *count = 0;
36
37     for ( int i = *start; i < max_val; i += NUM_CPUS){
38         if ( 1 == is_prime( i ) ) {
39             results[index] = i;
40             ++index;
41             ++(*count);
42         }
43     pthread_exit( count );
44 }
```

```
46 int main( int argc, char** argv ) {
47     if (argc < 2) {
48         printf("A maximum value is required.\n");
49         return -1;
50     }
51     max_val = atoi( argv[1] ); /* Read input as max val */
52     results = malloc( max_val * sizeof( int ) );
53     if (results == NULL) {
54         return -1;
55     }
56
57     pthread_t threads[NUM_CPUS];
58     int global_sum = 0;
59
60     for ( int i = 0; i < NUM_CPUS; ++i ) {
61         int* start = malloc( sizeof( int ) );
62         *start = i;
63         pthread_create(&threads[i], NULL, find_primes, start);
64     }
65
66     void* temp_sum;
67     for ( int i = 0; i < NUM_CPUS; ++i ) {
68         pthread_join( threads[i], &temp_sum );
69         int *thread_sum = (int *) temp_sum;
70         global_sum += *thread_sum;
71     }
72     printf("Found %d total primes less than %d.\n",
73         global_sum, max_val);
74
75     qsort(results, global_sum, sizeof(int), compare);
76
77     for (int i = 0; i < global_sum; ++i ) {
78         printf( "%d\n", results[i]);
79     }
80
81     pthread_exit(0);
82 }
```

(1) Memory Leaks: Identify any memory leaks below (anything Valgrind would complain about with the options `--leak-check=full` and `--show-leak-kinds=all`). For each memory leak: Indicate whether they are "definitely lost", "indirectly lost", or "still reachable" (you may ignore possibly lost). State the line on which the memory is allocated and explain where the appropriate deallocation statement(s) should go.

(2) Concurrency Problems: Identify any concurrency problems with this code (anything Helgrind would complain about on its own; the `--read-var-info=yes` option just helps you track things down). For each concurrency problem: state the nature of the concurrency problem, the potential effects, and explain how you would solve the problem (no code is necessary but it can be used to illustrate).

Please initial:

Question 4: `printf("You're a wizard, %s!\n", firstname);` [18 marks]

At Hogwarts School of Witchcraft and Wizardry, undesirable jobs such as cooking and cleaning are done by House Elves, magical creatures who apparently are not covered under any sort of employment standards or workers rights legislation. Each day an elf has to do two undesirable tasks before it may go home: (1) Cook for a feast, and (2) fetch supplies from storage. They complete both of these tasks using magic. Due to the laws of magic ("don't cross the streams!") only one elf can be doing each of those tasks at a time. In the meantime, however, elves are not allowed to stand around and be lazy. If they are unable to do either of those tasks, they should clean the school.

At the beginning of the day (program), Dobby (the main program) summons his fellow elves (starts their threads). At any given moment, multiple elves may be working. Dobby is not allowed to dismiss an elf until that elf has finished both mandatory tasks, but Dobby must dismiss an elf as soon as the elf has completed his/her work; Dobby can't wait until all the elves are done to start sending them home. When Dobby has nothing to do, he should sleep (be blocked), because he is a much better negotiator than any of the other elves at the school and his contract allows that. The other elves should never get blocked or spin lock (they can always be cleaning). When an elf is done, Dobby should say (print to the console), "You are dismissed, elf N." where N is the identifier of that elf.

The skeleton program shown below is not complete; it does not have any of the synchronization in it. The `#include` directives are not shown but you may assume the exist. If you add anything that requires a new `#include` directive you do not have to write it in; assume it will be added for you. Your code should not have any memory leaks or race conditions that cause program errors (benign races are okay). Remember to deallocate any allocated memory and to destroy/clean up anything intialized.

Complete the code shown on the next page to implement the functionality described above.

Assume the following function prototypes match to functions that implement the appropriate functionality (even though the implementations are not shown). The parameter `id` is the numerical identifier of the elf doing the task.

```
void cook_feast( int id );
void get_supplies( int id );
void clean_school( );
```

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_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 )
```

Remember that `NULL` can be given for the attributes.

Please initial:

```
#define NUM_ELVES 25

/* Global Variables Area */




void cook_feast( int id );
void get_supplies( int id );
void clean_school( );

void* elf( void* arg ) {
    int *id = (int*) arg;
    bool cooked = false;
    bool supplies = false;




    while (!cooked || !supplies) {

        if ( !cooked ) {


            cook_feast( *id );


            cooked = true;


        }

        if ( !supplies ) {


            get_supplies( *id );


            supplies = true;


        }



        clean_school( );


    }







    free( id );
    pthread_exit( NULL );
}

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









    pthread_t threads[NUM_ELVES];


    for (int i = 0; i < NUM_ELVES; ++i) {
        int* id = malloc( sizeof (int) );
        *id = i;
        pthread_create(&threads[i], NULL, elf, id);
    }

    /* Wait for each elf to be done and then dismiss them */
```

```
    return 0;
}
```