Please print in pen:
Waterloo Student ID Number:

WatIAM/Quest Login Userid:

UNIVERSITY OF WATERLOO

Examination
Midterm
Winter 2018
ECE 459

Times: Monday 2018-02-12 at 20:45 to 21:45 (8:45 to 9:45PM)
Duration: 1 hour (60 minutes)
Exam ID: 3706412
Sections: ECE 459 LEC 001,002
Instructors: Jeff Zarnett, Patrick Lam

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, calculators with no communication capability permitted.
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. There are five (5) questions. Not all are equally difficult.
4. The exam lasts **60** minutes and there are 100 marks.
5. Verify that your name and student ID number is on the cover page.
6. 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.
7. Do not fail this city.
8. 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		25
2		20
3		15
4		25
5		15
Total		**100**

Please initial:

1 Short Answer [25 marks total]

Answer these questions using at most three sentences. Each question is worth 2.5 points.

(a) Your main program starts this thread 8 times and joins them. Write down two potential values for `counter` after the threads all `join()`.

```
void * run (void * arg) {
  for (int i = 0; i < 10000; ++i) {
    ++counter;
  }
  return 0;
}
```

(b) You are riding your bicycle 10 km at a speed of 20 km/h. Your path includes 10 red lights, each with a cycle time of 50 seconds green, 10 seconds red. (You should expect that if you arrive at a red light, your wait time will be 5 seconds.) Assume that acceleration and braking are instantaneous. What is your expected trip time?

(c) Give a simple real-life scenario in which there is a WAW (Write-After-Write) dependency.

(d) (Order-of-magnitude estimation:) How many bytes can you write to disk in a second in a machine with an SSD?

(e) Why is this code wrong? Explain how you would fix it.

```
int main(int argc, char * argv[]) {
  pthread_t threads[10];
  for( int i = 0; i < 10; ++i ) {
    struct params p;
    p.x = 0;
    p.y = 0;
    p.z = 0;
    pthread_create(&threads[i], NULL, run, &p);
    pthread_detach(t1);
  }
  pthread_exit( 0 );
}
```

```
void * run (void * arg) {
  // do something correct with parameters
}
```


Please initial:

(f) Explain the semantics of the OpenMP directive `#pragma omp single`.

(g) You have a new CPU and would like to measure a branch mispredict rate (and you, rightly, distrust the advertising material of the manufacturer). You have at hand the specification documents that say the cost of a mispredicted branch is x cycles. Devise a test plan to work out what the mispredict rate is. Assume you don't have a tool like Cachegrind to do it for you, but you do have a way to count how many cycles a given set of instructions takes.

(h) Weather forecasting works like this: given observations of the current weather state (fit to a grid), models use fluid dynamics and thermodynamics equations to predict future weather states for each grid point. Forecasters use a number of weather models and combine the results. Explain how Gustafson's Law allows us get better weather forecasts, and also how Amdahl's Law limits weather forecasting.

(i) What is one problem with keeping a bunch of joinable threads around indefinitely?

(j) Consider the code below. Assume the implementations of functions `do_work()` and `print_result` are correct. What problem will Valgrind (memcheck) analysis reveal about this code and how do you fix it?

```
void compute() {
  struct point * p = malloc(sizeof(struct point *));
  p->x = 0;
  p->y = 0;
  p->z = 0;
  p->a = 0.0;

  do_work( p );
  print_result( p );
  free( p );
}
```

```
struct point {
  int x;
  int y;
  int z;
  double a;
};
```


Please initial:

2 Concurrency and Parallelism

2.1 Overhead Costs [10 marks]

You are given access to a computer. You know the number of cores it has (call it N) and you are sure that nobody else is using it right now. Describe a set of experiments to determine the cost of switching between threads. You can write programs and you have access to a command like UNIX `time` which displays elapsed (wall clock), system, and user times. You can also measure the number of thread switches in a process execution. Assume that user times do not include thread switching time. Your experimental protocol should explain what you would expect to see for the user-level and kernel-level threading models.

2.2 Autoparallelization [10 marks]

Correcting a Loop. [5 marks] Consider the following loop which potentially could be parallelized by the automatic parallelization routines of a sufficiently smart compiler. But unfortunately, as it is, it cannot be automatically parallelized. Explain what the problem(s) is (are) and show a corrected version.

```
void add_arrays( int * c, int * a, int * b, int * length ) {
  for( int i = 0; i < *length; ++i ) {
    c[i] = a[i] + b[i];
  }
}
```

Function Call. [5 marks] Autoparallelization routines will decline to parallelize a loop that has an an arbitrary function call in it. A colleague proposes a rule for the compiler that says: a loop containing a call to a function `f` that takes arbitrary paramaters should be allowed, if the compiler can be certain that `f` does not modify any of the parameters (or the data that they point to, if they are pointers). So the function call `bar( a, b )` is allowed if the function does not write to `a`, `b`, `*a`, or `*b`. Does this modification work? Justify your answer.

Please initial:

3 Callback-Based Asynchronous I/O [15 marks]

Consider the following list traversal code.

```
struct node {
  struct node * next;
  int data;
};

void traverse_list(struct node * n) {
  send_msg(START, 0);
  while (n != NULL) {
    send_msg(VISIT, n->data);
    n = n->next;
  }
  send_msg(END, 0);
}
```

This code sends messages, using `send_msg`, indicating its progress through the list.

Your task is to write pseudocode for callback-based code that handles the messages sent by `traverse_list`, printing out each message as it is received, and enforcing that the messages arrive in the correct order. As seen in the `traverse_list` implementation, the order must respect the regular expression `START VISIT* END`; that is, a START followed by 0 or more VISITs and then an END.

- The entry point to your program is function `main()`, which needs to set up a callback to handle the `START` method and then call the magic function `wait_for_events()`, which waits for events and calls any handlers you've set up.
- To set up a handler for a message, call function `add_cb(MSG, handler)`. Note that `handler` is a function pointer.
- To remove all handlers for a message, call function `clear_cb(MSG)`.
- A handler should take a `MSG` and a `void *` parameter, and should return `void`.
- If a message arrives out of order, the program should call `abort()`. You may assume that the program calls `abort()` if an event occurs for which there is no handler.
- You may use `print()` to print a message. Don't worry about syntax for `print()`.
- Clean up after yourself.

Please initial:

4 Harry Potter and the OpenMP Tasks [25 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 jobs for the elves to do, and the elves do them. When all work is done, the program may terminate.

Dobby is the *master* task. At the start of the day, Dobby prepares jobs to do by calling `struct job * prep_tasks()`. This function returns a pointer to an array of `struct job`, each of which is a job to do. The array is always of length `NUM_TASKS`. For each task, Dobby recruits a new worker elf (OpenMP Task) to perform the job. The work performs that job by calling `do_work(struct job todo)`. When the job is finished, the worker elf should update the total number of jobs completed (global variable `completed`). When all jobs are completed, Dobby can deallocate the memory of the list of jobs and the day is finished (program exits).

Complete the code below to implement the functionality described above using OpenMP directives. You may assume all relevant `#include` and `#define` directives are present.

```
int completed = 0;

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









  return 0;
}
```


Please initial:

5 Lock Convoy Costs [15 marks]

Consider threads A and B, which are both running and contending on a lock ℓ. Thread Z is also running and not contending on ℓ. All three threads have the same priority and this does not change during execution. Your system has a single core.

Assume a thread's timeslice is at most 500k cycles. A context switch takes 10k cycles and does not count in any thread's timeslice. Threads A and B run for 400k cycles before requesting ℓ, and compute under the lock for 700k cycles. They then start over with the 400k cycle computation, etc. Assume that it takes 0 cycles to obtain the lock. Assume thread Z is constantly computing. The thread scheduler is fair—lock queues are first-in-first-out.

You may choose any schedule that allows each of A and B to acquire and release ℓ twice. Show your schedule (including lock acquisitions and context switches). How many cycles will the processor spend (a) in context switches; (b) in thread Z; (c) in each of threads A and B?

Please initial: