Please print in pen:
Waterloo Student ID Number:

WatIAM/Quest Login Userid:

UNIVERSITY OF WATERLOO

Examination
Midterm
Winter 2019
ECE 459

Times: Wednesday 2019-02-27 at 17:45 to 18:45 (5:45 to 6:45PM)
Duration: 1 hour (60 minutes)
Exam ID: 4026714
Sections: ECE 459 LEC 001,002
Instructors: Jeff Zarnett, Patrick Lam

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 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 four (4) questions. Not all are equally difficult.
5. The exam lasts **60** minutes and there are 45 marks.
6. Verify that your name and student ID number is on the cover page.
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		12
2		5
3		13
4		15
Total		**45**

Please initial:

1 Short Answer [12 Marks total]

Answer these questions using at most three sentences. Each question is worth 3 points.

(a) Consider the following code excerpt [1]:

```
void printStatus(char *status, int taskNum, int threadNum) {
    #pragma omp critical(printStatus)
    {
        for (int i = 0; i < taskNum; i++) printf("␣␣␣␣");
        printf("␣%s%i␣\n", status, threadNum);
    }
}

void task(int taskNum) {
    // "r"un task
    printStatus("r", taskNum, omp_get_thread_num());
    sleep(1);

    // "s"leeping task that can yield
    printStatus("s", taskNum, omp_get_thread_num());
    // wallClockTime() does what you think...
    double idleStartTime = wallClockTime();
    while (wallClockTime() < idleStartTime + 1) {
       #pragma omp taskyield
    }

    // "c"ontinue task
    printStatus("c", taskNum, omp_get_thread_num());
    sleep(1);
}

int main(int argc, char* argv[]) {
    #pragma omp parallel
    #pragma omp single nowait
    {
        for (int i = 0; i < NTASKS; i++) printf("␣%02d␣", i);
        printf("\n");

        for (int i = 0; i < NTASKS; i++) {
            #pragma omp task untied
            task(i);
        }
    }

    return 0;
}
```

Here is one possible output from the program with `OMP_NUM_THREADS=3`.

```
 00  01  02  03  04
 r1
     r2
         r0
 s1
     s2
         s0
 c1
     c2
         c0
                 r0
             r2
                 s0
             s2
                 c0
             c2
```

This output conforms to the specification of `untied`, but doesn't demonstrate the effect of the `untied` clause. Show an alternate output by writing down what would be printed to console below. Your output must be possible only if the `untied` clause is present but not if it is absent.

[1] credit: Jeorsch at Stackoverflow, `https://stackoverflow.com/questions/47658571/how-to-yield-resume-openmp-untied-tasks-correctly`, accessed 16 Feb 2019.

Please initial:

(b) You are thinking about changing some parts of a code base to use readers-writers locks instead of regular mutex locks for a particular critical section. What facts would you need to decide if this change is worth it?

(c) Suppose that hardware designers tell you the size of level 1 data cache, but refuse to tell you the cache replacement strategy. Explain how you could determine this.

(d) Can proper use of thread cancellation handlers improve performance? Explain your answer.

2 Fun Without Locks [5 marks]

Consider this code (credit `https://www.airs.com/blog/archives/79`):

```
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
static int acquires_count = 0;

int trylock() {
    int res = pthread_mutex_trylock( &mutex );
    if (res == 0) {
        ++acquires_count;
    }
    return res;
}
```

In the past, gcc has compiled this code to a comparison (which puts its result into the CPU's carry flag), an add (with carry), and an unconditional store—that is, the generated code has no branches. So, `acquires_count` is always written to. If you run this code in parallel with itself, `acquires_count` can get the wrong answer. Show an execution trace where this happens.

Please initial:

3 OpenMP Tasks [13 Marks]

Professor Derek Wright needs your help; the ECE department is still working on the accreditation report. Professor Aagaard has assembled a database that contains accreditation data. (Meanwhile, Professor Lam [im]patiently waits to use the data for the SE report that he'll have to write.) This data includes information about how many AUs a course has in a given term. These can change over time, as the curriculum is updated based on student feedback (and accreditation needs), so the term information is important. Professor Wright has a program that lets him find out the total number of AUs for a given set of courses by querying the database. A sequence of course-term pairs is provided as input, and the total AUs is printed out.

Input is parsed using a function `course_term* get_next( )` which returns a pointer to a structure of type `course_term` which is comprised of the course number and the term (see the definition of the structure below). If we've reached the end of input, this function returns `NULL`. The implementation of this function is not shown, but assume it works correctly. This function cannot be run in parallel.

```
typedef struct {
  char course[8]; /* string representation, e.g. ECE459 */
  int term; /* integer, e.g., 1191 for Winter 2019 */
} course_term;
```

To find the number of AUs, there is the function `double query_db( course_term * ct )` . This queries the database for the number of AUs for the specified course in the specified term which is returned as a `double`. Again, the implementation is not shown, but assume it works correctly.

```
typedef struct node {
  pthread_t t;
  struct node * next;
} node;

node* head;
double total = 0.0;
pthread_mutex_t lock;

void* query( void* arg ) {
  course_term* ct = ( course_term* ) arg;
  double aus = query_db( ct );
  pthread_mutex_lock( &lock );
  total += aus;
  pthread_mutex_unlock( &lock );
  free( ct );
  pthread_exit( NULL ); /* 0 is equivalent */
}
```

```
int main( int argc, char** argv ) {
  pthread_mutex_init( &lock, NULL );
  head = NULL;

  while( 1 ) {
    course_term* next = get_next( );
    if ( next == NULL ) {
      break;
    }
    node* n = malloc( sizeof( node ));
    n->next = head;
    head = n;
    pthread_create( &n->t, NULL, query, next );
  }

  node* n = head;
  while( n != NULL ) {
   pthread_join( n->t, NULL );
   node* old = n;
   n = n->next;
   free( old );
  }
  printf( "Total AUs: %g\n", total );

  pthread_mutex_destroy( &lock );
  return 0;
}
```

In the space below, rewrite the code above to use **OpenMP Tasks**. Your modified program should be free of race conditions, not have any memory leaks, and produce the same output as the original program. Compiling this without OpenMP enabled should result in a serial version of the program being correctly generated. The new code should not contain any pthread function calls. Note that you no longer need the `node` struct to track threads.

```
course_term* get_next( );
double query_db( course_term* ct );
double total = 0.0d;
int main( int argc, char** argv ) {









  printf( "Total AUs: %g\n.", total );
  return 0;
}
```


Please initial:

4 Asynchronous I/O [15 marks]

You have been asked to design a program that processes a group of files. You can use asynchronous I/O to partially parallelize this: start the read for file $n+1$ and process file n in the meantime. This doesn't work for the first file, so a blocking read takes place below in the starter code. The maximum size of any file we will read is `MAX_SIZE`, so always use this size as the length of a read (even though you may read less, that's okay). You need two buffers: one for the file being processed and one where the next read is taking place.

A list of files to read will be provided as arguments on the commandline to the program. Remember, `argc` tells you the total count of arguments (including the file name of the executable) and `argv` is an array of `char*` ("strings") of the file names. `argv[0]` is the name of the executable, but every subsequent entry in the array is a file name.

After a file is read into memory, you should create and enqueue an asynchronous I/O request using POSIX `aio` to read the next one. Then you can process the current file's data using `void process( char* buffer )`. If processing is finished but the asynchronous read is not, use `sleep(1)` as many times as necessary. Recall that `EINPROGRESS` is returned from `aio_error( aiocb * cb )` if the operation is still in progress.

For your convenience, the AIO structure (of which you only need to set the first 4 fields):

```
struct aiocb {
  int aio_fildes;               /* File descriptor */
  off_t aio_offset;             /* Offset for I/O */
  volatile void* aio_buf;       /* Buffer */
  size_t aio_nbytes;            /* Number of bytes to transfer */
  int aio_reqprio;              /* Request priority */
  struct sigevent aio_sigevent; /* Signal Info */
  int aio_lio_opcode;           /* Operation for List I/O */
};
```

Complete the code below to implement the desired behaviour. Your code should not leak any memory or have any race conditions. Assume that errors won't occur and therefore you do not need to check for them (i.e., memory allocation, enqueuing an asynchronous read, actually reading the data, et cetera, will always succeed). Be sure to close files when reading is finished.

```
void process( char* buffer ); /* Implementation not shown */

int main( int argc, char** argv ) {
  char* buffer1 = malloc( MAX_SIZE * sizeof( char ));
  char* buffer2 = malloc( MAX_SIZE * sizeof( char ));

  int fd = open( argv[1], O_RDONLY );
  memset( buffer1, 0, MAX_SIZE * sizeof( char ));
  read( fd, buffer1, MAX_SIZE );
  close( fd );

  for ( int i = 2; i < argc; i++ ) {









  }



  return 0;
}
```

Please initial: