CISC 372: **Midterm Exam 2** Practice

Name: ______________________________

Instructions. Write your name in the indicated space.
You may use one 8.5 × 11" sheet of paper with anything written on both sides.
No points will be subtracted for an incorrect answer, so it is to your advantage to answer all questions.
There should be plenty of room to work out the problems, but if you want scrap paper, raise your hand.

1. Phread basics

1. Using Pthreads, write C code that (1) launches 100 threads, each running the function `f`, then (2) waits for all the threads to terminate. Declare any variables used. You can assume that `f` will not use its argument, and the values returned by `f` are not needed.

Note: Write only code to do exactly what is requested. Do not write additional code. Do not define `f`. Do not write a complete program. Do not write `#include <pthread.h>`.

______ **2.** The operating system usually associates to each thread its own ...

(A) heap
(B) security information
(C) private address space
(D) call stack
(E) set of file descriptors

______ **3.** Suppose two threads can access a shared variable `x` and execute concurrently as follows:
Thread 1: `y=x+1;`
Thread 2: `z=2*x;`
Is there a data race?

(A) Yes
(B) No
(C) not enough information given to decide

______ **4.** Suppose two threads can access a shared array `a` (with length at least 3) and execute concurrently as follows:
Thread 1: `a[1] = 23;`
Thread 2: `a[2] = 46;`
Is there a data race?

(A) Yes
(B) No
(C) not enough information given to decide

______ **5.** Suppose two threads can access a shared variable `x` and execute concurrently as follows:
Thread 1: `x=1;`
Thread 2: `x=1;`
Is there a data race?

(A) Yes
(B) No
(C) not enough information given to decide

______ **6.** Consider the C declaration `int (*f[])();`. What is the type of `f`?

(A) array-of-pointer-to-function-returning-int
(B) pointer-to-function-returning-array-of-int
(C) function-returning-array-of-pointer-to-int
(D) array-of-function-returning-pointer-to-int
(E) function-returning-pointer-to-array-of-int

______ **7.** Which of the following results from a call to `pthread_cond_signal(c)`:

(A) the calling thread is added to the wait set of `c`
(B) the thread receiving the signal regains the mutex lock
(C) the threads receiving the signal know that the condition upon which they are waiting is now true
(D) all threads are removed from the wait set of `c`
(E) if the wait set of `c` was nonempty, at least one thread is removed from the wait set

2. Condition Variables

8. The following program creates two threads, one running `f`, the other, `g`. Function `f` repeats the following, forever: it waits for `n` to be even, then divides it by 2. Function `g` repeats the following forever: it waits for `n` to be odd, then multiplies it by 3 and adds 1.
Fill in the definition of `f`. (The code for `g` should be similar and you don't have to fill that in.)

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
int n;
pthread_mutex_t mutex;
pthread_cond_t cond;

void *f(void *arg) {

}

void *g(void *arg) {...}

int main(int argc, char *argv[]) {
  n = atoi(argv[1]);
  pthread_mutex_init(&mutex, NULL);
  pthread_cond_init(&cond, NULL);
  pthread_t t1, t2;
  pthread_create(&t1, NULL, f, NULL);
  pthread_create(&t2, NULL, g, NULL);
}
```

3. Concurrency Flags

9. Fill in the missing parts in the following pseudocode to create a barrier between the two threads. Use the standard flag operations *flag_init*, *flag_destroy*, *flag_lower*, and *flag_raise*.

```
flag_t f1, f2;

Initialization code: // fill in some code to initialize f1, f2






Thread 1:
  int i=0;
  while (i<n) {
    i=i+1;
    // insert code below to create barrier:







  }

Thread 2:
  int j=0;
  while (j<n) {
    j=j+1;
    // insert code below to create barrier:







  }

Destruction code: // fill in below
```

4. OpenMP

10. Insert OpenMP annotations to parallelize the following code with maximal concurrency, without affecting the output. *Hint*: one parallel region will do. Maximize overlap, but look out for data races.

```
#include <math.h>
#include <stdio.h>
#define N 100000
double a[N], b[N], c[N];

int main() {
  int i;







  for (i=0; i<N; i++)
    a[i] = sin(i*1.271);






  for (i=0; i<N; i++)
    b[i] = cos(i*.376);






  for (i=1; i<N; i++)
    b[i] += b[i-1];






  for (i=0; i<N; i++) {
    c[i] = a[i]*b[i];
    if (i>0)
      c[i] += a[i-1]*0.01;
  }





  for (i=0; i<N; i++)
    printf("%f\n", c[i]);
}
```

______ **11.** Consider the following code. Assume the request for 2 threads is granted.

```
#include <stdio.h>
int main() {
#pragma omp parallel num_threads(2)
  {
    #pragma omp for nowait
    for (int i=0; i<2; i++) printf("%d", i);
    #pragma omp for
    for (int i=2; i<4; i++) printf("%d", i);
  }
  printf("\n");
}
```

Which of the following could be the output? Choose the best answer.

(A) 0213
(B) 3102
(C) 3210
(D) all of the above are possible
(E) anything could be output because the code has undefined behavior

______ **12.** Consider the following code. Assume the request for 2 threads is granted.

```
#include <stdio.h>
int main() {
  int i=1;
#pragma omp parallel num_threads(2) shared(i)
  {
    i=i+1;
    printf("%d", i);
  }
}
```

Which of the following best describes the possible outputs?

(A) 23 only
(B) 32 only
(C) 33 only
(D) 23, 32, or 33
(E) anything could be output because the code has undefined behavior

______ **13.** Does the following exhibit correct use of OpenMP?

```
#include <stdio.h>
#define n 4
int a[n], b[n];
int main() {
  b[0] = a[0];
#pragma omp parallel for shared (a,b)
  for (int i=1; i<n; i++) {
    b[i] = b[i-1] + a[i];
  }
}
```

(A) Correct
(B) Incorrect

______ **14.** Does the following exhibit correct use of OpenMP?

```
#include <stdio.h>
#define n 4
int a[n], b[n];
int main() {
  for (int i=0; i<n; i++) b[i] = i;
#pragma omp parallel for shared (a,b)
  for (int i=0; i<n; i++) {
    for (int j=0; j<n; j++) { a[i] = a[i] + b[j]; }
  }
  for (int i=0; i<n; i++) printf("%d ", a[i]);
  printf("\n");
}
```

(A) Correct
(B) Incorrect

Appendix A. Pthreads Reference

A.1. **Types.**

`pthread_t` `pthread_attr_t` `pthread_mutex_t` `pthread_mutexattr_t` `pthread_cond_t` `pthread_condattr_t`

A.2. **Functions.**

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void*), void *arg)
int pthread_join(pthread_t thread, void **value_ptr)
int pthread_mutex_init(pthread_mutex_t * mutex, pthread_mutexattr_t * attr)
int pthread_mutex_destroy(pthread_mutex_t * mutex)
int pthread_mutex_lock(pthread_mutex_t * mutex)
int pthread_mutex_unlock(pthread_mutex_t * mutex)
int pthread_cond_init(pthread_cond_t * cond, pthread_condattr_t *attr)
int pthread_cond_destroy(pthread_cond_t *cond)
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)
int pthread_cond_signal(pthread_cond_t *cond)
int pthread_cond_broadcast(pthread_cond_t *cond)
```

Appendix B. OpenMP Reference

B.1. **Types.**

`omp_lock_t`

B.2. **Functions.**

```
int omp_get_num_threads();
int omp_get_thread_num();
void omp_init_lock(omp_lock_t *lock);
void omp_destroy_lock(omp_lock_t *lock);
void omp_set_lock(omp_lock_t *lock);
void omp_unset_lock(omp_lock_t *lock);
```

B.3. **Pragmas.**

```
#pragma omp parallel
  num_threads(n)
  shared(x,y,...)
  private(x,y,...)
  default(shared|private|none)
  firstprivate(x,y,...)
#pragma omp for
  reduction(op:x,y,...)
  shared(x,y,...)
  private(x,y,...)
  ordered
  schedule(static,chunksize)
  schedule(dynamic,chunksize)
  schedule(guided,chunksize)
  nowait
  collapse(n)
#pragma omp sections
#pragma omp single
#pragma omp section
#pragma omp barrier
#pragma omp ordered
#pragma omp critical (name)
#pragma omp atomic
#pragma omp master
#pragma omp threadprivate(...)
```