CSSE 332 - Operating Systems

Introduction to Multi-threading

Name: Solution Key

Question 1. (5 points) What are the main differences between a *thread* and a *process*? In your answer, list out the pieces of memory and the processor that are shared between threads and those that are not.

Solution: A thread is generally a lightweight process. Threads share the code, globals, and heap memory regions, while each thread will have its own stack and its own set of registers. Generally, context switching with threads is faster.

Question 2. (5 points) Consider the following function that we'd like to run in its own separate thread:

```c
void *thread_function(void *arg) {
  // argument contains an id given to the thread
  int my_id = *(int*)arg;

  printf("Hello from thread %d\n", my_id);
  return NULL;
}
```

In the box below, write a piece of code that would create and run this function in a new thread. Make sure to include the code that waits for the created thread to return.

Solution:

```c
int i = 0;
pthread_t th;

pthread_create(&th, 0, thread_function, &i);

// do some work
pthread_join(th, 0);
```

Question 3. (5 points) Consider the following piece of code, how many threads would we end up with when it executes?

```
pthread_t threads[5];
for(int i = 0; i < 5; i++) {
  pthread_create(&threads[i], 0, some_function, 0);
}
```

Solution: We create 5 new threads, but we also have the main thread, so we end up with 6 threads.

Question 4. (5 points) Consider the following snippet of code:

```
void thread_fn(void *arg) {
  int *id = *(int*)arg;
  printf("Thread %d executing\n", id);
  return 0;
}

// somewhere else
pthread_t threads[5];
for(int i = 0; i < 5; i++) {
  pthread_create(&threads[i], 0, thread_fn, &i);
}
```

In the box below, write down for each thread, the output that will be printed on the screen. For example, write something like `Thread 1:  Thread 0 executing` and so on.

Solution: We cannot tell. The loop variable `i` is on main's stack but it also will be destroyed after the loop ends. When a thread runs, `i` might have changed its value. Even worse, `i` might no longer exist, so we might get a segmentation fault!

Question 5. (5 points) Consider the following snippet of code:

```
void thread_fn(void *arg) {
  int id = *(int*)arg;
  printf("Thread %d is executing \n", id);
  return 0;
}

// somewhere else
pthread_t threads[5];
int ids[5];
for(int i = 0; i < 5; i++) {
  ids[i] = i+1;
  pthread_create(&threads[i], NULL, thread_fn, &ids[i]);
}

for(int i = 0; i < 5; i++) {
  pthread_join(threads[i], 0);
}
printf("All the threads have returned\n");
```

In the box below, write the order in which the print outs will show up on the console when this code executes.

Solution: We cannot tell since the scheduler picks which threads will run at which point in time. Without manual synchronization, we cannot tell which thread will run at which time.