a02-celliott.md

CISS 370: Operating Systems

Assignment 2

Due: February 10, 2021

Christian Elliott

1. Download *threadHello.c*, compile it, and run it several times. What happens when you run it? Do you get the same result if you run it multiple times? What if you are also running some other demanding processes (e.g., compiling a big program, playing a Flash game on a website, or watching streaming video) when you run this program?

- The program creates 10 processes and runs them in order. However, because they are separate processes, they complete at unpredictable intervals. Therefore, you do not get the same result running the program multiple times. If you were to run the program during the operation of other process intensive programs, its behaviour would be even more unpredictable as the processes will be queued along with other running programs. It is also important to note: Because of the second for loop containing the line `exitValue = thread_join(threads[i]);` , each process must have returned before the line `printf("Main thread done.\n");` is executed.

2. For the *threadHello* program in Figure 4.6, suppose that we delete the second for loop so that the main routine simply creates NTHREADS threads and then prints "Main thread done." What are the possible outputs of the program now. Hint: Fewer than NTHREADS+1 lines may be printed in some runs. Why?

- This relates to the last part of my first answer: `thread_join()` waits for a thread to finish. Therefore, when we remove this, the main thread no longer waits on the sub-threads to complete. Therefore, it's entirely possible for the main thread to reach the final `printf` and `return 0;` before all the children threads are completed. Thus, not all the print lines may be executed by the threads created by the thread array.

3. Write a program that has two threads. Make the first thread a simple loop that continuously increments a counter and prints a period (".") whenever the value of that counter is divisible by 10,000,000. Make the second thread repeatedly wait for the user to input a line of text and then print "Thank you for your input." On your system, does the first thread makes rapid progress? Does the second thread respond quickly?

- For this question, it does not specify we should use a `thread_join()` to wait for the completion of the threads, so I didn't use it. Here is my code:

```c
#include <stdio.h>
#include "thread_lib/thread.h"

static thread_t thread1;
static thread_t thread2;

static void t1(int n);
static void t2();

int main(int argc, char **argv) {
    thread_create(&thread1, &t1, 0);
    thread_create(&thread2, &t2, 0);

    printf("Exiting main ....\n");
    return 0;
```

```
}

void t1(int n) {
    printf("In t1\n");
    while (1) {
        printf("In t1 while loop\n");
        ++n;
        if (n % 10000000 == 0) {
            printf(".");
        }
    }
    thread_exit(n);
}

void t2(int n) {
    printf("In t2\n");
    while (1) {
        printf("In t2 while loop\n");
        char inp[10];
        fgets(inp, 10, stdin);
        printf("Thank you for your input.");
    }
    thread_exit(n);
}
```

- So, when I compile and run my program, its behavior is unpredictable. One time it spent around 20 iterations in the t1 while loop, and another time it went straight into the t2 function.
- Fixed my code:

```
#include <stdio.h>
#include "thread_lib/thread.h"

static thread_t thread1;
static thread_t thread2;

static void t1(int n);
static void t2();

int main(int argc, char **argv) {
    thread_create(&thread1, &t1, 0);
    thread_create(&thread2, &t2, 0);

    thread_join(thread1);
    thread_join(thread2);
    printf("Exiting main ....\n");
    return 0;
}

void t1(int n) {
    printf("In t1\n");
    while (1) {
        ++n;
        if (n % 10000000 == 0) {
            printf(".");
        }
    }
    thread_exit(n);
}

void t2(int n) {
    printf("In t2\n");
    while (1) {
```

```
            printf("Give me some text: ");
            char inp[10];
            fgets(inp, 10, stdin);
            printf("Thank you for your input.\n");
        }
        thread_exit(n);
    }
```

- The program executes both t1 and t2, however, t2 waits on characters via the stdin stream, which hides the stdout stream. While t2 waits, t1 is continuing, and therefore incrementing n. After I give char(s) as input to t2, we briefly see the periods printed out representing the amount of iterations t1 has completed in its while loop, during the time it takes to give input to t2 and press enter.

4. For the *threadHello* program in Figure 4.6, the procedure go() has the parameter *np* and the local variable *n*. Are these variables per-thread or shared state? Where does the compiler store these variables' states?

- np, the input param, is a global variable as it exists outside of the thread creation. However, the local variable, n, would exist inside the stack space allocated for the thread.

5. For the *threadHello* program in Figure 4.6, the procedure main() has local variables such as *i* and *exitValue*. Are these variables per-thread or shared state? Where does the compiler store these variables?

- *i* and *exitValue* are shared state, and their value would be accessible to each thread as the data shared to threads. However, if a thread were to modify their value their behavior would be unpredictable.