CIS 452
Lab 4 Report

Ashley Hendrickson
Muna Gigowski
Fall 2019

Thread Creation and Execution

Question One

Describe/explain your observations, i.e. what must have happened in the original, unmodified program?

When first running the program, before adding the sleep(2) line of code, the program would indicate that it ran successfully since no error messages were displayed, but nothing would show up on the command line at all, almost as if nothing had even ran. After adding the sleep(2) line however, when running the program it would pause for a moment, then print "Thread version of Hello, World." In looking at the original code, it makes sense that this would happen because in the do_greeting function that the newly created thread was assigned to execute, there was a sleep(1) line that came before the actually printing out of the greeting, meaning that the thread would need to wait one second before printing. However, the other thread that was running did not have a sleep() call at all, so after it created the new thread and evaluated the if conditional it immediately moved on to the return line, which exited the program before the other thread was able to finish its execution and print out to the screen. When we added the sleep(2) to the main function though, it caused the thread running that code to wait until the other thread did it printing, before returning and exiting the program.

Thread Suspension and Termination

Question Two

Report your results, particularly the observed formatting.

In running the program the first time, the output was as follows:

World
World
World
World
World
World
World
World
World
World
Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello

Question Three

Report your results again. Explain why they are different from the results seen in question 2.

In running the program the second time, after adding the sleep(1) line, the output was as follows:

World
Hello World
Hello Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World

The results are different this time because, in the first run without the sleep(), one thread entered the do_greeting2() function before the other, which caused it to start looping and printing out its value before the other thread got a chance to catch up and begin looping and printing out its value. Once we added the sleep() line at the top of the for loop, it caused each thread to wait for 1 second before each subsequent iteration of the loop - giving the other thread time to catch up and finish its previous iteration. Although this did not completely fix the issue of synchronization between the two threads, it definitely helped coordinate their printing.

Question Four

Based on your observations: does pthreads under Linux use the 1-to-1 or the many-to-one thread mapping model? Justify your answer.

It uses the one-to-one thread mapping model, because when using the command ps -L -u gigowskm while sampleProgram2 is executing, it individually displays all three of the threads that are running within that program separately, meaning that the kernel is aware of all 3 individual threads (opposed to the many-to-one model where the thread are abstracted because they are all routed into one single kernel thread).

Thread Communication

Question Five

Compile the sample program and run it multiple times (you may see some variation between runs). Choose one particular sample run. Describe, trace, and explain the output of the program.

The output of the program was as follows:

Child receiving a initially sees 5
Child receiving b initially sees 5
Parent sees 5
Child receiving a now sees 7
Child receiving b now sees 8
Parent sees 8

This output can be explained by following the process of execution within sampleProgram3, starting after both child threads have been created:

1. The first thread enters do_greeting3() and hits the print line with the initial value given to the global variable "sharedData," which was initialized to 5. This thread prints out the value to be 5. The thread then sleeps for 1 second.
2. The second thread enters do_greeting3() and hits the print line, also printing out "sharedData" with its initial value of 5. The thread then sleeps for 1 second.
3. While both child threads are sleeping, after creating both threads and assigning them to do_greeting3() in the main function, the parent thread hits its print line and prints out "sharedData" with the initial value of 5. It then hits the next line, incrementing sharedData by one, making it have the value of 6. The parent thread then calls pthread_join() twice, once for each child thread, causing it to wait to continue execution until both children have finished.
4. Back in do_greeting3(), the first child thread is done sleeping and increments sharedData to 7, then hits the second print line in the function, which causes it to display the new value of 7 for sharedData. It then finishes and returns.
5. The second child thread also finishes sleeping and increments sharedData again, but since it hit after the first child thread incremented sharedData, it had the value of 7 to start - meaning that after it incremented the final value of sharedData was 8. The second child then prints that value out to the screen before finishing and returning.
6. Now that both children finished, the parent begins executing again and prints the final value of sharedData again, which was 8. It then returns in main, exiting the program.

Question Six

Explain in your own words how the thread-specific (not shared) data is communicated to the child threads.

Thread-specific data is communicated between child threads via the use of shared global variables, or shared memory locations that each thread writes to, effectively communicating data to one another.

Lab Programming Assignment (Blocking Multi-threaded Server)

Source Code

```
#include<iostream>
#include<thread>
#include <pthread.h>
#include <string>
#include <stdlib.h>
#include<mutex>
#include <errno.h>
#include<unistd.h>
#include <signal.h>

#define NUM_THREADS 100

using namespace std;
mutex mtx;
double totalAccessTime;
int numTimesAccessed;
int counter = 0;
pthread_t threads[NUM_THREADS];
string fileName;

void *returnFile(void* arg);
void getFile();
void my_handler(int num);

int main(){

        //set up sigHandler to receive ^C signal and call custom signal
handler function
        struct sigaction sigIntHandler;
        sigIntHandler.sa_handler = my_handler;
    sigemptyset(&sigIntHandler.sa_mask);
    sigIntHandler.sa_flags = 0;

    sigaction(SIGINT, &sigIntHandler, NULL);

        while(1)
                getFile();
}
```

```
void getFile(){

        cout <<"Enter in a file name .... \n";
        cin >> fileName;
        int status;

        //Creates thread and assigns it to execute the returnFile function
with fileName as an argument
        if ((status = pthread_create (&threads[counter], NULL, returnFile,
&fileName)) != 0) {
         cerr << "thread create error: " << endl;
         exit (1);
    }
}

void *returnFile(void* arg){

        string fname = *reinterpret_cast<std::string*>(arg);

        if(rand()%11>7){
                int sleepTime=7+rand()%4;
                mtx.lock();
                totalAccessTime += sleepTime;
                ++numTimesAccessed;
                ++counter;
                mtx.unlock();
                sleep(sleepTime);
                cout <<"retrieved file " << fname << " from database
successfully" << endl;
        }
        else{
                sleep(1);
                mtx.lock();
                ++totalAccessTime;
                ++numTimesAccessed;
                ++counter;
                mtx.unlock();
                cout <<"retrieved file " << fname  << " from database
successfully" << endl;
        }

        //Once the current thread has finished its work, it needs to detech
from the parent thread and then properly end itself
        pthread_detach(pthread_self());
        pthread_exit(NULL);
}

//When user enters ^C, print final stats before exiting the program
void my_handler(int num) {

        cout <<" Total number of file requests received: " +
std::to_string(numTimesAccessed) + "\n";
        cout <<"Average file access time: " + std::to_string((totalAccessTime
/ numTimesAccessed)) + "\n";
        exit(0);
}
```

Sample Output