CIS 452 Lab: Week #4
Multi-threaded Programming: pthreads

Overview

The purpose of this lab is to introduce multi-threaded programming: the fundamental mechanisms for thread creation, execution, management, communication and termination. It examines the first essential element of any thread system, execution context (the remaining two elements being scheduling and synchronization).

The documentation refers to the POSIX **pthreads** API, a standardized run-time library that natively implements threads on Linux. See the online documentation (i.e. `man` pages) for additional details on the syntax and operation of the library calls described in this lab.

Hand-in:

- A word document containing the answer to the numbered questions. (The questions should also be included either in bold or italicized).
- Any requested screenshots (uploaded as separate files)
- Program source code (no zip files)

Thread Creation and Execution

A traditional process in a UNIX system is simply a single-threaded program. It has a single thread (or path) of execution, known as the initial thread, that begins by executing **`main()`**.

Like processes, all threads (with the exception of the initial one executing `main()`) are spawned from an existing thread. The syntax of the library routine that creates a thread is given below:

```
int pthread_create (pthread_t* tid       // thread id (returned from create routine)
                    const pthread_attr_t* attr,   // optional attributes
                    void* (*start)(void*),        // address of function to execute
                    void* arg);                   // argument passed to thread
```

This call creates a new thread running the user-defined function named **`start`**. The `start()` function is passed a single argument **`arg`**, of type **`void*`**, and returns a value of the same type. Any optional attributes (e.g., scheduling policy) may be set at creation time. The identifier of the newly created thread is returned in **`tid`**. After completion of this function call, the newly created thread will begin executing the function `start()`, using the argument(s) pointed to by `arg`. As usual, the return value of the function call is a status indicator; where a non-zero value means the function failed.

The sample program below demonstrates simple thread creation. Default values are used for the attributes. The function does not take any arguments. Note the use of error-checking for all function calls.

sampleProgramOne

```
#include <pthread.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

void* doGreeting(void* arg);

int main()
{
  printf("Running the program\n");
  pthread t thread1;  // thread ID holder
  int threadStatus;         // captures any error code

  // create and start a thread executing the "doGreeting()" function
  threadStatus = pthread_create (&thread1, NULL,  doGreeting, NULL);
  if (threadStatus != 0) {
    fprintf (stderr, "Thread create error %d: %s\n", threadStatus, strerror(threadStatus));
    exit (1);
  }
  return 0;
}

void* doGreeting (void* arg)
{
  sleep(1);
  printf ("Thread version of Hello, world.\n");
  return arg;
}
```

Try the following operations and answer the numbered questions:

Start up a Terminal session, which provides you with the UNIX command-line interface. Compile and run sampleProgramOne. You will need to link in the pthreads library (i.e. you must compile with the **`-pthread`** compiler option). Observe the results.

Try inserting a 2-second `sleep()` into the `main()` function (after thread creation); compile and re-run.

1. Describe/explain your observations, i.e., what must have happened in the original, unmodified program? (4 points)

Thread Suspension and Termination

Similar to Unix processes, threads have the equivalent of the `wait()` and `exit()` system calls, in this case called `join()` and `exit()`. The calls are used to block threads and terminate threads, respectively.

To instruct a thread to block while waiting for another thread to complete, use the **`pthread_join()`** function. This function is also the mechanism used to get a return value from a thread. Note that any thread can join on (and hence wait for) any other thread. The function prototype:

```
int pthread_join(pthread_t thread1, void **value_ptr)
```

This function specifies that the calling thread should block and wait for `thread1` to complete execution. The value returned by `thread1` will be accessible via the argument `value_ptr`. In addition to explicitly joining, threads may also use semaphores, conditional waits and other synchronization mechanisms to keep track of when other threads exit.

Sometimes parent threads have ongoing work to perform, for example, functioning as a dispatcher. Instead of waiting for a child thread to complete, a parent can specify that it does not require a return value or any explicit synchronization with a child thread. To do this, the parent thread uses the **`pthread_detach()`** function. A child thread can also specify that it does not need to join with any other threads by detaching itself. After the call to detach, there is no thread waiting for the child - it executes independently until termination. The prototype for this function is as follows:

```
int pthread_detach(pthread_t thread1)
```

with `thread1` representing the identity of the detached thread.

The **`pthread_exit()`** function causes the calling thread to terminate. Resources are recovered, and a value is returned to the joined thread (if one exists). A thread may explicitly call `pthread_exit()`, or it may simply terminate, usually by returning from its start function. Although it is not strictly enforced that you use either `join()` or `detach()`, it is good practice because non-detached threads which have exited but have not been joined are equivalent to zombie processes (i.e. their resources cannot be fully recovered).

The sample program below is a functioning multi-threaded program that uses the library functions described above. It may not behave exactly as you might expect.

sampleProgramTwo

```c
#include <pthread.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>


void* doGreeting(void* myArgument);

int threadCounter = 0;

int main()
{
    pthread_t thread1, thread2;  // thread ID's
    void *result1, *result2;     // return values
    int threadStatus1, threadStatus2; // store the result of the thread creation
    int joinStatus1, joinStatus2; // store the result of the thread join


    threadStatus1 = pthread create(&thread1, NULL, doGreeting, "hi");
    if(threadStatus1 != 0){
      fprintf (stderr, "Thread create error %d: %s\n", threadStatus1, strerror(threadStatus1));
      exit (1);
    }

    threadStatus2 = pthread_create(&thread2, NULL, doGreeting, "bye");
    if(threadStatus2 != 0) {
      fprintf(stderr, "Thread create error %d: %s\n", threadStatus2, strerror(threadStatus2));
      exit (1);
    }

    // join with the threads (wait for them to terminate);  get their return vals
    joinStatus1 = pthread_join(thread1, &result1);
    if(joinStatus1 != 0) {
        fprintf(stderr, "Join error %d: %s\n", joinStatus1, strerror(joinStatus1));
        exit (1);
    }

    joinStatus2 = pthread_join(thread2, &result2);
    if(joinStatus2 != 0) {
      fprintf(stderr, "Join error %d: %s\n", joinStatus2, strerror(joinStatus2));
      exit (1);
    }

    printf("Thread one returned: [%s]\n", (char *)result1);
    printf("Thread two returned: [%s]\n", (char *)result2);

    return 0;
}

// function called by the threads, will print out a different message based on the argument
passed to it
void* doGreeting(void* myArgument)
{
  printf("String passed = %s\n", (char*)myArgument);
  threadCounter = threadCounter + 1;
  printf("ThreadCounter = %d\n", threadCounter);

  // Print out a message based on the argument passed to us
  for (int loop = 0;  loop < 10;  loop++) {
    if (strcmp((char*)myArgument, "hi")==0){
      printf("Hello\n");
    }
    else{
      printf("Good Bye\n");
    }
  }

  if (strcmp((char*)myArgument, "hi") == 0){
    myArgument = "Hello is done";
```

```
  }
  else{
    myArgument = "Good Bye is done";
  }
  return myArgument;
}
```

Try the following operations and answer the numbered questions:

Compile and run sampleProgramTwo. Run sampleProgramTwo multiple times.

2. What does sampleProgramTwo output? If you run it a repeated number of times does the output vary? Why? (6 points)

Insert a one-second `sleep()` at the beginning of the loop in the doGreeting() function. Compile and run the modified program.

3. Report your results again. Explain why they are different from the results seen in question 2. (4 points)

Thread Communication

There are two methods used by threaded programs to communicate. The first method uses shared memory. But sometimes it is desirable to communicate thread-specific information to each individual thread. For this purpose, we can use the second method of communication: via the argument value (the `void *`) passed to a thread's start function. The following sample program uses shared memory for communication; it also demonstrates the mechanism of thread-specific arguments to pass each thread unique information.

sampleProgramThree

```
#include <pthread.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

void* doGreeting(void* arg);


// global (shared and specific) data
int sharedData = 5;
char sampleArray[2] = {'a','b'};

int main()
{
  pthread_t thread1, thread2;
  void *result1, *result2;
  int threadStatus1, threadStatus2;
  int joinStatus1, joinStatus2;

  threadStatus1 = pthread_create (&thread1, NULL,  doGreeting, &sampleArray[0]);
```

```
  if (threadStatus1 != 0){
    fprintf (stderr, "Thread create error %d: %s\n", threadStatus1, strerror(threadStatus1));
    exit (1);
  }

  threadStatus2 = pthread_create(&thread2, NULL, doGreeting, &sampleArray[1]);
  if (threadStatus2 != 0) {
    fprintf (stderr, "thread create error %d: %s\n", threadStatus2, strerror(threadStatus2));
    exit (1);
  }

  printf ("Parent sees %d\n", sharedData);
  sharedData++;

  joinStatus1 = pthread_join(thread1, &result1);
  if (joinStatus1 != 0) {
    fprintf (stderr, "Join error %d: %s\n", joinStatus1, strerror(joinStatus1));
    exit (1);
  }

  joinStatus2 = pthread join(thread2, &result2);
  if (joinStatus2 != 0) {
    fprintf (stderr, "Join error %d: %s\n", joinStatus2, strerror(joinStatus2));
    exit (1);
  }
  printf ("Parent sees %d\n", sharedData);

  return 0;
}

void* doGreeting(void* myArgument)
{
  char *myPtr = (char *)myArgument;

  printf ("Child receiving %c initially sees %d\n", *myPtr, sharedData);
  sleep(1);
  sharedData++;
  printf ("Child receiving %c now sees %d\n", *myPtr, sharedData);
  return NULL;
}
```

Try the following operations and answer the numbered questions:

4. 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. (6 points)
5. Explain in your own words how the thread-specific (not shared) data is communicated to the child threads. (4 points)

Lab Programming Assignment (Blocking Multi-threaded Server) (30 points)

A multi-threaded fileserver receives a file access request from a client and dispatches a worker thread to satisfy the request and resumes accepting new requests from other clients. The worker threads proceed to service their assigned request (potentially blocking while waiting for the disk). This mini-programming assignment simulates the thread execution manifested by a multi-threaded fileserver process.

Develop a multi-threaded program with the following specifications:

Dispatch thread:

- Input a string from the user (simulating the name of a file to access)
- Spawn a child thread and communicate to it the filename requested/entered by the user
- Immediately repeat the input/spawn sequence (i.e., accept a new file request)
- Ensure your dispatcher is capable of properly handling rapidly entered filenames (i.e., the user rapidly entering the filenames in succession)

Worker threads:

- Obtain the simulated filename from the dispatcher
- Sleep for a certain amount of time, simulating the time spent performing a file access:
    - with 80% probability, sleep for 1 second. This simulates the scenario that the Worker thread has found the desired file in the disk cache and serves it up quickly.
    - with 20% probability, sleep for 7-10 seconds (randomly). This simulates the scenario that the worker thread has *not* found the requested file in the disk cache and must block while it is read in from the hard drive.
- Wake up, print a diagnostic message that includes the name of the file accessed, then terminate

It's ok if the resulting display looks somewhat "messy"; that shows that true concurrency is occurring.

Your program should continue to execute until terminated by the user (^C). At that point, your program should print out basic statistics:

- Total number of file requests received

When terminated, your program should cleanup as appropriate and shutdown gracefully.

In addition to being correct, your program should be efficient and should execute in parallel.