ECE459: Programming for Performance Winter 2021

Lecture X – A Crash Course on Threads

Jeff Zarnett *2020-09-28*

POSIX Threads

The term `pthread` refers to the POSIX standard (also known as the IEEE 1003.1c standard) that defines thread behaviour in UNIX and UNIX-like systems (Linux, Mac OS X, Solaris...). This is a specification document that says how threads should behave. This standard lets code for one UNIX-like system (e.g., Solaris) run easily on another (e.g., Linux). The POSIX standard for pthreads defines something like 100 function calls, but we need not examine all of them.

- `pthread_create` – Create a new thread.
- `pthread_exit` – Terminate the calling thread.
- `pthread_join` – Wait for a specific thread to exit. The caller cannot proceed until the thread it is waiting for calls `pthread_exit`. Note that it is an error to join a thread that has already been joined.
- `pthread_detach` – If we want to make it so that a thread cannot be joined, then we can make it a "detached" thread with this function.
- `pthread_yield` – Release the CPU and let another thread run. As they all belong to the same program, we expect that threads want to co-operate rather then compete for CPU time and threads can make decisions about when it would be ideal to let some other thread run instead.
- `pthread_attr_init` – Create and initialize a thread's attributes. The attributes contain things like the priority of the thread. ("After you, sir." "Oh no, after you.")
- `pthread_attr_destroy` – Remove a thread's attributes. Free up the memory holding the thread's attributes. This does not terminate the threads.
- `pthread_cancel` – Signal cancellation to a thread; this can be asynchronous or deferred, depending on the thread's attributes.
- `pthread_testcancel` – A thread can check to see if it has been cancelled. If that is the case, this function terminates the calling thread.

This list of functions gives us an overview of the toolkit we have, but we need to elaborate with some examples to fully understand how they work.

Creating a New Thread. When we want to start a new thread, we have to say what that new thread is supposed to do. The function signature for `pthread_create` looks like:

```
pthread_create( pthread_t *thread, const pthread_attr_t * attr, void *(*start_routine)( void * ), void *arg );
```

Where: `thread` is a pointer to a `pthread` identifier and will be assigned a value when the thread is created. The attributes `attr` may contain various characteristics (but you may supply `NULL` if you want the defaults). The third parameter is the function to run, but it requires a little more explanation.The last parameter, `arguments` is the argument passed to the `start_routine`. But that second last one is weird.

The `start_routine` parameter is the name of any function that takes a single untyped pointer and returns an untyped pointer. That is, the function signature has to match those two conditions. The name of the function (and the name of the argument) can be anything you like. See the example below:

```
void* do_something( void* start_params )
```

After the new thread has been created, the process has two threads in it. The OS makes no guarantee about which thread will be executing after the new one is created; this is a matter of scheduling. It could be either of the threads of the process, both of them at the same time, or a different process entirely.

Our experience with C-like languages suggests it is normal to have a single return value from a function, but usually we can have multiple input parameters. It seems limiting to be able to put in just one. There are two ways to get around this: with an array or with structures. In the case of the array, the argument provided to `pthread_create` is just a pointer to the array. This is also, incidentally, how you can get multiple return values out of a function in Java or C# (`public Object[] foo()`), but I don't recommend it as a good programming practice. The other way to do it is to use the `struct`, defining a structure for the parameter type and one for the return type.

The function that is to run in the new thread must expect a pointer to the arguments and then it will need to be cast to the appropriate (actual) type:

```
void* function( void * void_arg ) {
  parameters_t *arguments = (parameters_t*) args;
  /* continue after this */
}
```

This does imply that the caller of the `pthread_create` function has to know what kind of argument is expected in the function being called. That is fairly normal; we do have to know what the arguments mean when we pass them in to any function, but in this case we don't have the "hints" that the types provide.

What about the thread attributes? They can be used to set whether a thread is detached or joinable, scheduling policy, etc. By default, new threads are usually joinable (that is to say, that some other thread can call `pthread_join` on them). As noted before, it is a logical error to attempt multiple joins on the same thread. To prevent a thread from ever being joined, it can be created in the detached state (or the method `pthread_detach` can be called on a joinable thread). Trying to join a detached thread is also a logical error [Bar14]; testing tends to show that if you join with NULL you are ignored, but if you try to collect a value, your program will crash. For virtually all scenarios that we will consider in this course the default values will be fine.

Once we do that, the new thread we created is running. It does whatever its code does, so everything proceeds as expected, until of course the thread gets to the end. Usually, it will terminate with `pthread_exit`. The use of `pthread_exit` is not the only way that a thread may be terminated. Sometimes we want the thread to persist (hang around), but if we want to get a return value from the thread, then we need it to exit.

Returning Values. If a thread has no return values, it can just `return NULL;` which will have the same effect as `pthread_exit` and send `NULL` back to the thread that has joined it. If the function that is called as a task returns normally rather than calling the exit routine, the thread will still be terminated.

Another way a thread might terminate is if the `pthread_cancel` function is called with it as the target. As before, if the termination is deferred rather than asynchronous, the thread is responsible for cleaning up after itself before it stops.

A thread may also be terminated indirectly: if the entire process is terminated or if `main` finishes first (without calling `pthread_exit` itself). Indeed, `main` can use `pthread_exit` as the last thing that it does. Without that, `main` will not wait for other, unjoined threads to finish and they will all get suddenly terminated. If `main` calls `pthread_exit` then it will be blocked until the threads it has spawned have finished [Bar14].

Collecting Returned Values. Like the `wait` system call, the `pthread_join` is how we get a value out of the spawned thread:

```
pthread_join( pthread_t thread, void** retval );
```

The first parameter specifies the thread that you want to join. The second parameter is... wait... two stars? What we are looking for is a pointer to a void pointer. That is, we are going to supply a pointer that the join function will update to be pointing to the value returned by that function. Typically we supply the address of a pointer. This will be hopefully clearer in the example:

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

void * run( void * argument ) {
  char* a = (char*) argument;
  printf("Provided argument is %s!\n", a);
  int * return_val = malloc( sizeof( int ));
  *return_val = 99;
  pthread_exit( return_val );
}

int main( int argc, char** argv ) {
  if (argc != 2) {
      printf("Invalid args.\n");
      return -1;
  }
  pthread_t t;
  void* vr;

  pthread_create( &t, NULL, run, argv[1] );
  pthread_join( t, &vr );
  int* r = (int*) vr;
  printf("The other thread returned %d.\n", *r);
  free( vr );
  pthread_exit( 0 );
}
```

Thread Cancellation. Thread cancellation is exactly what it sounds like: a running thread will be terminated before it has finished its work. Once the user presses the cancel button on the file upload, we want to stop the upload task that was in progress. The thread that we are going to cancel is called the *target* (because we shoot targets, I guess) and there are two ways a thread might get cancelled [SGG13]:

1. **Asynchronous Cancellation:** One thread immediately terminates the target.
2. **Deferred Cancellation:** The target is informed that it is cancelled; the target is responsible for checking regularly if it is terminated, allowing it to clean itself up properly.

The pthread attributes can be used to set the cancellation type before it is created. A thread can declare its own cancellation type through the use of the function:

```
pthread_setcanceltype( int type, int *oldtype )
```

The first parameter is the new state we'd like this thread to take on, which would be one of the constants `PTHREAD_CANCEL_DEFERRED` or `PTHREAD_CANCEL_ASYNCHRONOUS`. The second parameter will be updated to point to what the previous state was (although we might not care).

In deferred cancellation, a thread is responsible for checking if it has been cancelled, and if so, and stopping its activity and cleaning up (closing open files, etc.) before it terminates. It's possible, though generally poor programming practice (and very difficult), to never check for cancellation.

Given that a thread can effectively ignore a cancellation if it is the deferred cancellation type, why would we ever choose that over asynchronous cancellation? Suppose the thread we are cancelling has some resources. If the thread is terminated in a disorderly fashion, the operating system may not reclaim all resources from that thread. Thus a resource may appear to be in use even though it is not, denying that resource to other threads and processes that may want to use it [SGG13].

The pthread command to cancel a thread is `pthread_cancel` and it takes one parameter (the thread identifier). By default, a pthread is set up for deferred cancellation. In the function that runs as a thread, to check if the thread has been cancelled, the function call is `pthread_testcancel` which takes no parameters.

Suppose your background task is to upload a bunch of files, consecutively. It is good programming practice to check `pthread_testcancel` at the start or end of each iteration of the loop, and if cancellation has been signalled, clean up open files and network connections, and then `pthread_exit`. Thus, if the thread has been told to cancel, it will do as it is told within a fairly short period of time.

It is noteworthy that a large number of functions are *cancellation points*; that is, the POSIX specification requires there is an implicit check for cancellation when calling one of those functions. There is an even larger number of functions that are "potential cancellation points", where the specification says that they could be cancellation points (but maybe aren't). You'll have to check the spec to see if that is the case for a specific function if there is a scenario where unexpected cancellation is a problem.

Now's not a good time! With the presence of cancellation points or asynchronous cancellation, sometimes a thread can be terminated before it has cleaned up some resources. This is undesirable. One way that we can guard against this is to register cleanup handlers for that thread. If, say, our thread allocated some memory, it would be wise to register a cleanup handler that deallocates that memory in case the thread should die unceremoniously. The function signatures are:

```
pthread_cleanup_push( void (*routine)(void*), void *argument ); /* Register cleanup handler, with argument */
pthread_cleanup_pop( int execute ); /* Run if execute is non-zero */
```

To add a cleanup handler, the push function is used. Its two arguments are the function that is supposed to run, and a pointer to the argument that cleanup function will need.

The push function always needs to be paired with the pop function at the same level in your program (where level is defined by the curly braces). You should think of them as being like the opening curly brace at the start of a statement and the closing curly brace at the end; they have to be correctly matched up. The pop function takes one argument: whether it should run or not. If the thread is cancelled, the cleanup function will run; if it continues to the pop function, then you get to choose whether it runs or not.

Consider the following code:

```
void* do_work( void* argument ) {
  struct job * j = malloc( sizeof( struct job ) );
  /* Do something useful with this structure */
  /* Actual work to do not shown */
  free( j );
  pthread_exit( NULL );
```

Suppose that the thread is cancelled during the block operating on `j` and it is set up for asynchronous cancellation. This means that the code will never get to the `free()` call, which means that the memory allocated at the beginning is leaked! We can remedy this with application of a cleanup handler:

```
void cleanup( void* mem ) {
  free( mem );
}

void* do_work( void* argument ) {
  struct job * j = malloc( sizeof( struct job ) );
  pthread_cleanup_push( cleanup, j );
  /* Do something useful with this structure */
  /* Actual work to do not shown */
  free( j );
  pthread_cleanup_pop( 0 ); /* Don't run */
  pthread_exit( NULL );
```

Attributes and Using Memory to Pass Data. The earlier example used the return value of a thread. Sometimes, of course, we don't want to do that. One of the advantages of the use of threads is that data can be passed between threads using memory directly. In this case, because there is no return value that we are about, we can use `NULL` in the call to join. This example also shows how to initialize and the attributes, although it doesn't override any of the defaults.

```
#include <pthread.h>
#include <stdio.h>

int sum; /* Shared Data */

void *runner(void *param);

int main( int argc, char **argv ) {

  pthread_t tid; /* the thread identifier */
  pthread_attr_t attr; /* set of thread attributes */

  if ( argc != 2 ) {
    fprintf(stderr,"usage: %s <integer value>\n", argv[0]);
    return -1;
  }
  if ( atoi( argv[1] ) < 0 ) {
    fprintf(stderr, "%d must be >= 0\n", atoi(argv[1]));
    return -1;
  }

  /* set the default attributes */
  pthread_attr_init( &attr );
  /* create the thread */
  pthread_create( &tid, &attr, runner, argv[1] );
  pthread_join( tid, NULL );
  printf( "sum = %d\n", sum );
  pthread_exit( NULL );
}

void *runner( void *param ) {
  int upper = atoi( param );
  sum = 0;
  for ( int i = 1; i <= upper; i++ ) {
    sum += i;
  }
  pthread_exit( 0 );
}
```

In this example, both threads are sharing the global variable `sum`. We have some form of co-ordination here because the parent thread will join the newly-spawned thread (i.e., wait until it is finished) before it tries to print out the value. If it did not join the spawned thread, the parent thread would print out the sum early.

Count to 10... Let's do a different take on that program:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int sum = 0;

void* runner( void *param ) {
    int upper = atoi( param );
    for (int i = 1; i <= upper; i++ ) {
        sum += i;
    }
    pthread_exit( 0 );
}

int main( int argc, char** argv ) {

    pthread_t tid[3];
```

```
    if ( argc != 2 ) {
        printf("An␣integer␣value␣is␣required␣as␣an␣argument.\n");
        return -1;
    }
    if ( atoi( argv[1]) < 0 ) {
        printf( "%d␣must␣be␣>=␣0.\n", atoi(argv[1]) );
    }

    for ( int i = 0; i < 3; ++i ) {
        pthread_create( &tid[i], NULL, runner, argv[1] );
    }
    for ( int j = 0; j < 3; ++j ) {
        pthread_join( tid[j], NULL );
    }
    printf( "sum␣=␣%d.\n", sum );

    pthread_exit( 0 );
}
```

What's going wrong here? For very small values of the argument, nothing, but for a large number we get some strange and inconsistent results. Why? There are three threads that are modifying `sum`.

Summary: Relevant pthread Signatures

```
pthread_create( pthread_t *thread, const pthread_attr_t *attributes,
                void *(*start_routine)( void * ), void *argument )
pthread_join( pthread_t thread, void **return_value )
pthread_detach( pthread_t thread )
pthread_cancel( pthread_t thread )
pthread_testcancel( ) /* If the thread is cancelled, this function does not return (thread terminated) */
pthread_exit( void *value )
pthread_mutex_init( pthread_mutex_t *mutex, pthread_mutexattr_t *attributes )
pthread_mutex_lock( pthread_mutex_t *mutex )
pthread_mutex_trylock( pthread_mutex_t *mutex ) /* Returns 0 on success */
pthread_mutex_unlock( pthread_mutex_t *mutex )
pthread_mutex_destroy( pthread_mutex_t *mutex )
pthread_cleanup_push( void (*routine)(void*), void *argument ); /* Register cleanup handler, with argument */
pthread_cleanup_pop( int execute ); /* Run if execute is non-zero */

sem_init( sem_t* semaphore, int shared, int initial_value); /* 0 for shared OK */
sem_destroy( sem_t* semaphore )
sem_wait( sem_t* semaphore )
sem_post( sem_t* semaphore )
```

References

[Bar14] Blaise Barney. POSIX Threads Programming, 2014. Online; accessed 1-March-2015. URL: `https://computing.llnl.gov/tutorials/pthreads/`.

[SGG13] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts (9th Edition)*. John Wiley & Sons, 2013.