

Lecture 9 — POSIX Threads

Jeff Zarnett

jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

June 20, 2018

The term pthread refers to a POSIX standard.

It defines thread behaviour in UNIX and UNIX-like systems.

It says how threads should behave.

This standard lets code for one UNIX-like system run easily on another.

The POSIX standard for pthreads defines something like 100 function calls.

Let us focus on a few of the important ones.

- `pthread_create`
- `pthread_exit`
- `pthread_join`
- `pthread_yield`
- `pthread_attr_init`
- `pthread_attr_destroy`
- `pthread_cancel`

So far we have examined threads at a theoretical level.

How to make something run in the background?

Well, `main` is just a function (with a special name).

The way we can start a thread is to say: run this function, but in a new thread.

The system call to create a new thread is `pthread_create`:

```
pthread_create( pthread_t *thread,  
               const pthread_attr_t *attributes,  
               void *(*start_routine)( void * ),  
               void *argument );
```

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 (can be NULL).

`start_routine` is any function that takes a single untyped pointer and returns an untyped pointer.

`arguments` is the argument passed to the `start_routine`.

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 next.

This is a matter of scheduling.

It could be either of the threads of the process, or a different process entirely.

It seems limiting to be able to put in just one parameter.

There are two ways to get around this: array or structures (`struct`).

Array: the argument provided to `pthread_create` is just a pointer to the array.

Structure: Define your own `struct`.

Defining the Parameter Structure

```
typedef struct {  
    int parameter1;  
    int parameter2;  
} parameters_t;
```

```
typedef struct {  
    int return1;  
    int return2;  
} return_t;
```

Using the Parameter Structure

The function that is to run in the new thread must expect a pointer to the arguments rather than explicit arguments:

```
void* function( void *args )  
which can then be cast to the appropriate type:  
parameters_t *arguments = (parameters_t*) args;
```

What about the thread attributes?

They can be used to query or set specific attributes of the thread, such as:

- Detached or joinable state
- Scheduling data (policy, parameters, etc...)
- Stack size
- Stack address

By default, new threads are joinable.

The use of `pthread_exit` is not the only way that a thread may be terminated.

If we want to get a return value from the thread, then we need it to exit.

Like the `wait` system call, the `pthread_join` is how we get a value:

```
pthread_create( &thread_id, NULL, function_name, &args );
```

```
// This function and the created function  
// are now running in parallel
```

```
void *void_ret;
```

```
pthread_join( thread_id, &void_ret );
```

```
return_t *returnValue = (return_t *) void_ret;
```

If a thread has no return values, it can just return `NULL`;

This will have the same effect as `pthread_exit` and send `NULL`.

If the function returns rather than calling `exit`, the thread will still be terminated.

Another way a thread might terminate: `pthread_cancel`.

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.

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */
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: a.out <integer value>\n");
        return -1;
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr, "%d must be >= 0\n", atoi(argv[1]));
        return -1;
    }
}
```



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

```
/* The thread will begin control in this function */
```

```
void *runner(void *param) {

    int i, upper = atoi(param);
    sum = 0;
    for (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: the parent thread joins the newly-spawned thread before it tries to print out the value.

If it did not join the new thread, the parent thread would print the sum early.

This is yet another example of that subject that keeps popping up...

fork results in a parent process spawning a child that is a clone of the parent.

Normally, the fork call makes a duplicate of all of the threads.

This is sensible, except for the fact that the child might call exec and replace itself with another program (throwing away all the threads).

Some UNIX systems solve this by having 2 different implementations of fork:
One that will copy all threads; and
One that will copy only the executing thread.

Cancellation (Deferred and Otherwise)

The pthread command to cancel a thread is `pthread_cancel`.

It takes one parameter (the thread identifier).

By default, a pthread is set up for deferred cancellation.

To check if the thread has been cancelled, call `pthread_testcancel` which takes no parameters.

Example: multiple file upload.