Lecture 9 — POSIX Threads

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

August 21, 2016

The pthread

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 Functions

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

Running Threads

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.

Creating a 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 );
```

Creating a Thread Parameters

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 Creation

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.

Threads: Passing Parameters

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;
```

Thread Attributes

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.

Exiting

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;
```

Exiting

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.

Cancellation

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.

Outliving Main

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.

Complex Example

```
#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;
  }
```

Complex Example

```
  /* 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);
}
```

Co-ordination

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` and `exec`

`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.

Signals

UNIX systems use signals to indicate events (e.g., the `Ctrl-C` on the console)
A form of event-driven programming.

Signals also are things like exceptions (division by zero, segmentation fault).

It is *synchronous* if the signal occurs as a result of the program execution (e.g., dividing by zero);

It is *asynchronous* if it comes from outside the process (e.g., the user pressing `Ctrl-C` or one process or thread sending a signal to another).

Signals are, in the end, interrupts with a certain integer ID.

Signals

By default, the kernel will handle any signal that is sent to a process with the default handler.

The behaviour of the default handler may be to ignore the signal, but some signals (segmentation fault) will result in termination of the process.

POSIX Signals

Here are some of the many signals described in the POSIX.1-1990 standard:

Signal	Comment	Value	Default Action
SIGHUP	Hangup detected	1	Terminate process
SIGINT	Keyboard interrupt (Ctrl-C)	2	Terminate process
SIGQUIT	Quit from keyboard	3	Terminate process, dump debug info
SIGILL	Illegal instruction	4	Terminate process, dump debug info
SIGKILL	Kill signal	9	Terminate process
SIGSEGV	Segmentation fault (invalid memory reference)	11	Terminate process, dump debug info
SIGTERM	Termination signal	15	Terminate process
SIGCHLD	Child stopped or terminated	20,17,18	Ignore
SIGCONT	Continue if stopped	19,18,25	Continue the process if stopped
SIGSTOP	Stop process	18,20,24	Stop process

Handling Signals

A process may inform the OS it is prepared to handle a signal itself.

Example: doing some cleanup when `Ctrl-C` is received instead of just dying.

In any event, a signal needs to be handled, even if the handling is to ignore it.

The signals `SIGKILL` and `SIGSTOP` cannot be caught, blocked, or ignored.

In a multithreaded program, to which thread should the signal be sent?

In general there are the following options:

1. Deliver the signal to the thread to which the signal applies.
2. Deliver the signal to every thread in the process.
3. Deliver the signal to certain threads in the process.
4. Assign a specific thread to receive all signals for the process.

The method that is appropriate depends on the signal and program.

Delivering Signals in UNIX

In UNIX, to deliver a signal to a process, the command is:

```
kill( pid_t pid, int signal )
```

Yes, to send a signal, even if it's not a kill signal, the command is kill.

The equivalent in POSIX pthreads is:

```
pthread_kill( pthread_t tid, int signal )
```

...where it will deliver the signal to a specific thread.

Command-Line Signals

On the command line: to send a signal, `kill` followed by a process ID.

Normally a command like `kill 24601` will send `SIGHUP` to a process.

This will, by default, kill the process.
The process has an opportunity to clean things up if it wants to.

If the process is still stuck, you can "force" kill the process with `SIGKILL`:
`kill -9 24601`.

The `-9` parameter sends signal 9 (`SIGKILL`) rather than the default 1 (`SIGHUP`).

Some users are eager to jump to `kill -9` whenever a process is stuck...

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.