ECE254: Operating Systems and Systems Programming Fall 2015

Lecture 9 — POSIX Threads (the pthread)

Jeff Zarnett

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. Let us focus on a few of the important ones and we can see they are, for the most part, very similar to what we saw with parent and child processes [Tan08]:

- `pthread_create` – Create a new thread. This works a lot like `fork`.
- `pthread_exit` – Terminate the calling thread. This is like `exit` in that it ends execution and returns a value.
- `pthread_join` – Wait for a specific thread to exit. This is like `wait`: 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_yield` – Release the CPU and let another thread run. There is no analogous call for processes[1] because every process is expected to act as if it is the only process in the system. In the case of threads, however, since 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.

So far we have examined threads at a theoretical level, but have not actually considered how to make something run in the background. Noting that `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` and its use looks like [HZM14]:

```
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 (but you may supply `NULL` if you want the defaults). The `start_routine` is any function that takes a single untyped pointer and returns an untyped pointer. Finally, the last parameter, `arguments` is the argument passed to the `start_routine`.

[1] At least not in UNIX. In Mac OS 9, however...

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, 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 it seems limiting to be able to put in just one parameter. There are two ways to get around this: with an array or with structures (`struct`). 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` as below [HZM14], defining a structure for the parameter type and one for the return type. In the example, all four variables are integers, but they could be of any type.

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

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

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 [Bar14]:

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

The first item in that list indicates if a thread is joinable or detached. 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].

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. Like the `wait` system call, the `pthread_join` is how we get a value out of the spawned thread [HZM14]:

```
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;` 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].

Let us examine a slightly more complex example that invokes more of the pthread system calls. The code sample below from [SGG13] provides an example of a multithreaded C program that uses pthreads to calculation the summation of a non-negative integer in a second thread.

```
#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 ti; /* 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(&tid, &attr, runner, argv[1]); /* wait for the thread to exit */
  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 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. This is yet another example of that subject that keeps popping up: co-ordination...

The `fork` and `exec` System Calls

As discussed earlier, `fork` results in a parent process spawning a child that is a clone of the parent. We would normally expect that the `fork` call makes a duplicate of all of the threads of the parent process. This is sensible, except for the fact that the child might call `exec` and replace itself with another program (throwing away all the threads). Therefore the work that was done to duplicate all the threads of the program is wasted. Some UNIX systems approach this by having two different implementations of `fork`, one that will copy all threads and one that will copy only the executing thread [SGG13].

Signals

UNIX systems use signals to indicate events (e.g., the `Ctrl-C` on the console), which is a form of event-driven programming. Signals also are things like exceptions (division by zero, segmentation fault), etc. A signal may be *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.

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.

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

Alternatively, a process may inform the operating system it is prepared to handle the signal itself (such as doing some cleanup when the `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. Note that the signals `SIGKILL` and `SIGSTOP` cannot be caught, blocked, or ignored.

In a multithreaded program, the difficulty is: to which thread should the signal be sent? In general there are the following options [SGG13]:

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.

Which method for distributing the signal is appropriate will depend on the signal. Synchronous signals will be sent to the thread that triggered the signal, but the process could handle asynchronous signals in any of the four ways. UNIX threads allow a thread to say what signals it will accept. If delivered to a process, the signal will be handled by the first thread found that will accept the signal. That thread can redistribute the signal to other threads if necessary.

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.

On the command line, the command to send a signal is also `kill` followed by a process ID. Normally a command like `kill 24601` will send `SIGHUP` to a process, which 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 by sending `SIGKILL` with the command `kill -9 24601`. The `-9` parameter says to send signal 9 (`SIGKILL`) rather than the default 1 (`SIGHUP`). Some users are eager to jump to `kill -9` whenever a process is stuck, but it's usually worthwhile to attempt a less severe killing (`SIGHUP` or `Ctrl-C`) so that the process can at least try to clean up.

Cancellation

We have already examined the concept of thread cancellation and the difference between asynchronous and deferred cancellation.

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.

References

[Bar14] Blaise Barney. POSIX Threads Programming, 2014. Online; accessed 1-March-2015.

[HZM14] Douglas Wilhelm Harder, Jeff Zarnett, and Vajih Montaghami. *A Practical Introduction to Real-Time Systems for Undergraduate Engineering*. 2014. Online; version 0.14.12.22.

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

[Tan08] Andrew S. Tanenbaum. *Modern Operating Systems, 3rd Edition*. Prentice Hall, 2008.