ECE459: Programming for Performance Winter 2020

Lecture 6 — Working with Threads

Patrick Lam and Jeff Zarnett *2019-10-26*

Using Threads to Program for Performance

We'll start by seeing how to use threads on "embarrassingly parallel problems", which have mostly-independent sub-problems (little synchronization) and strong locality (little communication). Later, we'll see which problems are amenable to parallelization (*dependencies*) and alternative parallelization patterns (right now, just use one thread per sub-problem).

About pthreads. pthreads stands for POSIX threads. It's available on most systems, including pthreads Win32 (which I don't recommend). Use Linux, and our provided server, for this course. C++11 also includes threads in its specification.

For your convenience, the function signatures of the relevant pthread calls is shown below. We'll assume you used them to at least some extent in a previous context. If this is new to you (for whatever reason), then look at the `pthreads.pdf` document in the course repository, where a quick summary of this material is provided [Bar14].

```
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 )
```

To compile a C or C++program with threads, add the `-pthread` parameter to the compiler commandline. Old compilers might complain because they default to a too-old C. Starting with `gcc` 5, the GCC C compiler will always default to at least C11. `g++` 5 defaults to C++98, while 6 and up default to at least C++14. To ensure C99 support even with old GCCs, add `-std=c99` to the commandline. To ensure C++11 support in GCC, use `-std=C++11`.

Starting a new thread. You can start a thread with `pthread_create()` or by creating a `std::thread`:

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

void* run(void*) {
  printf("In run\n");
}

int main() {
  pthread_t thread;
  pthread_create(&thread, NULL, run, NULL);
  printf("In main\n");
}
```

```
#include <thread>
#include <iostream>

void run() {
  std::cout << "In run\n";
}

int main() {
  std::thread t1(run);
  std::cout << "In main\n";
  t1.join(); // see below
}
```

Attributes. In previous courses, you were probably told that the default attributes would be fine. And generally they are... if it is a first concurrency course! But in a more advanced context, we may need to pay attention to the attributes. The most salient thread attribute that was previously discussed was about whether a thread was detached (i.e., not joinable).

Beyond being detached/joinable, threads have additional attributes. (Note, also, that even though being joinable rather than detached is the default on Linux, it's not necessarily the default everywhere). Here's a list.

- Detached or joinable state
- Scheduling inheritance
- Scheduling policy
- Scheduling parameters
- Scheduling contention scope
- Stack size
- Stack address
- Stack guard (overflow) size

Basically, you create and destroy attributes objects with `pthread_attr_init` and `pthread_attr_destroy` respectively. You can pass attributes objects to `pthread_create`. For instance,

```
size_t stacksize;
pthread_attr_t attributes;
pthread_attr_init(&attributes);
pthread_attr_getstacksize(&attributes, &stacksize);
printf("Stack size = %i\n", stacksize);
pthread_attr_destroy(&attributes);
```

Running this on a laptop produces:

```
jon@riker examples master % ./stack_size
Stack size = 8388608
```

Obviously, stack size is not the only attribute you can check. The pthread library contains a lengthy listing of all the get-this and get-that functions that define how to read the value of a particular attribute. If you are curious! And along with get-functions there are naturally set-functions as well.

Once you have a thread attribute object, you can set the thread state to joinable:

```
pthread_attr_setdetachstate(&attributes, PTHREAD_CREATE_JOINABLE);
```

Things Fall Apart

Okay, so you should be at least somewhat familiar with basic thread usage. But unfortunately there are some number of pitfalls that we need to avoid. These may have a negative impact on the correctness or the performance of your program.

Stack Allocation. Recall that the `pthread_create` call allows you to pass data to the new thread. Let's see how we might do that...

```
int i;
for (i = 0; i < 10; ++i) {
  pthread_create(&thread[i], NULL, run, &i);
}
```

Wrong! This is a *terrible* idea. Why?

1. The value of `i` will probably change before the thread executes.
2. The memory for `i` may be out of scope, and therefore invalid by the time the thread executes.

The worst part is that in a simple program where you just write most of the pthread launch logic in main, then `i` will not go out of scope and it will seem like everything is fine... Instead we need to do this:

```
int i;
int* arg;
for (i = 0; i < 10; ++i) {
  arg = malloc( sizeof( int ) );
  *arg = i;
  pthread_create(&thread[i], NULL, run, arg);
}
```

And then `arg` needs to be deallocated at some point by the thread.

On the other hand, you can pull off something similar with C++11 threads:

```
int i;
for (i = 0; i < 10; ++i) {
  std::thread t(run, i);
  t.detach();
}
```

This is OK because we pass `i` by value, which doesn't work for pthreads.

In pthreads-land, you might be able to get away with this.

```
int i;
for (i = 0; i < 10; ++i)
  pthread_create(&thread[i], NULL, &run, (void*)i);

...

void* run(void* arg) {
  int id = (int)arg;
```

It's not ideal, though. We're abusing the pointer by storing a number in it. Which might work, but you do have to be careful. Beware size mismatches between arguments: you have no guarantee that a pointer is the same size as an int, so your data may overflow. And, sizes of data types change between systems. For maximum portability, just use pointers you got from `malloc`.

Libraries Make sure the libraries you use are **thread-safe**: the library protects its shared data (i.e., can be called by more than one thread concurrently). We'll get into this in some more detail later.

How do you know if a function is thread safe? If you wrote it yourself or have the source code, you could take a look. But for an arbitrary library the best way to know is...read the documentation. No! Anything but that!!!

The glibc reentrant functions are also safe: a program can have more than one thread calling these functions concurrently. For example, use `rand_r`, not `rand`.

Catch You Later... Joinable threads (which is the default on Linux) wait for someone to call `pthread_join` before they release their resources (e.g. thread stacks). On the other hand, you can also create *detached* threads, which release resources when they terminate, without being joined.

It is good practice to detach a thread using `pthread_detach` if you have no intention of joining it ever. Otherwise the memory and resources allocated to this thread might hang around for the whole execution of your program. Be polite and clean these up sooner!

Calling `pthread_detach` on an already detached thread results in undefined behaviour. If we try to join a thread that's detached, it's also undefined behaviour. In a live coding demo I found that the program didn't crash if the join call used `NULL` as the second argument (i.e., we were not trying to collect any value), but things got weird quickly if we did try to get a return value out of it. Better not to do this.

The idiomatic way of returning data from threads in C++11 appears to be using futures. `std::async` provides support for this:

```
#include <thread>
#include <iostream>
#include <future>

int run() {
  return 42;
}

int main() {
  std::future<int> t1_retval = std::async(std::launch::async, run);
  std::cout << t1_retval.get();
}
```

This launches your thread for you. The `get()` call waits until the answer is ready and returns it to you.

Consider also the following small danger of detached threads:

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

void* run(void*) {
  printf("In run\n");
}

int main() {
  pthread_t thread;
  pthread_create(&thread, NULL, &run, NULL);
  pthread_detach(thread);
  printf("In main\n");
}
```

When I run it, it just prints "In main". Why?

Solution. Use `pthread_exit` to quit if you have any detached threads.

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

void* run(void*) {
  printf("In run\n");
}

int main() {
  pthread_t thread;
  pthread_create(&thread, NULL, &run, NULL);
  pthread_detach(thread);
  printf("In main\n");
  pthread_exit(NULL); // This waits for all detached
                      // threads to terminate
}
```

(There is no C++11 equivalent.)

Finishing a thread. The typical way to end a thread is via the `pthread_exit` call with a pointer to the return value (if any is desired). On some systems, though, this results in reported memory leaks. Well, nobody says the authors of the thread libraries are perfect. Returning from the thread's `start_routine` via a `return` statement is equivalent to calling `pthread_exit`, and `start_routine`'s return value is passed back to the `pthread_join` caller. That may work better for you. There is no C++11 equivalent.

Now's not a good time! You may also recall that threads can be terminated using cancellation, which may be asynchronous (immediate death) or synchronous (a thread terminates when it gets to a cancellation point). 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 );
```

Advanced pthread calls

Here's a couple more advanced pthread function calls that are available for our use [Bar14]:

```
pthread_t pthread_self( void );
int pthread_equal( pthread_t t1, pthread_t t2 );
int pthread_once(pthread_once_t* once_control, void (*init_routine)(void));
pthread_once_t once_control = PTHREAD_ONCE_INIT;
```

The call to `pthread_self` is a way to get a reference to the `pthread_t` structure of the currently operating thread while it's running! This is distinct from the idea of calling `gettid()` which tells you the kernel thread number of a given thread. But even so, having the number would not be helpful as pretty much all the pthread function calls operate on the `pthread_t` structure.

Unfortunately, the regular equality operator (`==`) can't be reliably used on a `pthread_t` structure, so there exists a separate equality test. Strangely, though, it does the opposite of what you might expect in Java or similar: if the two thread IDs are equal, `pthread_equal`returns a nonzero value; otherwise 0. So here 0 means non-equal. Sigh.

The last function and macro are about things you want to happen once but only once, no matter how many threads are started. The control variable is presumably global, and the first time that a thread calls the `pthread_once` function, then the `init_routine` function runs. On all later invocations, it does not. On return from `pthread_once`, it is guaranteed that the initialization routine has finished executing.

A quick example of the init routine, then:

```
sem_t sem;
pthread_mutex_t lock;
pthread_once_t once_control = PTHREAD_ONCE_INIT;

void init_func( ) {
  sem_init( &sem, 0, 0 );
  pthread_mutex_init( &lock, NULL );
}

void* thread1( void * arg ) {
  pthread_once( once_control, init_func );
  /* Do stuff */
}

void* thread2( void* arg ) {
  pthread_once( once_control, init_func );
  /* Do stuff */
}
```

The function `init_func` will execute only once, no matter whether thread 1 or thread 2 is created and executed first. In previous cases we've usually done initialization in `main()` or some designated `init` function that's called from `main()`. That makes sense in a trivial program (the kind we use in an example in class or on an exam), but that does not scale to a real program...

References

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