ECE459: Programming for Performance — Winter 2018

Lecture 5 — Working with Threads

Patrick Lam and Jeff Zarnett

Using Threads to Program for Performance

We'll start by seeing how to use threads on "embarrassingly parallel problems":

- mostly-independent sub-problems (little synchronization); and
- strong locality (little communication).

Later, we'll see:

- which problems are amenable to parallelization (*dependencies*)
- 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.

Here's a quick `pthreads` refresher. To compile a C or C++ program with pthreads, add the `-pthread` parameter to the compiler 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
}
```

From the man page, here's how you use `pthread_create`:

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

- **thread**: creates a handle to a thread at pointer location
- **attr**: thread attributes (NULL for defaults, more details later)
- **start_routine**: function to start execution

- **arg**: value to pass to start_routine

This function returns 0 on success and an error number otherwise (in which case the contents of *thread are undefined).

Waiting for Threads to Finish. If you want to join the threads of execution, use the `pthread_join` call. Let's improve our example.

```
#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");
  pthread_join(thread, NULL);
}
```

The main thread now waits for the newly created thread to terminate before it terminates. (C++11 requires threads to be either joined or detached when they go out of scope; we'll see the meaning of detach below.)

Here's the syntax for `pthread_join`:

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

- **thread**: wait for this thread to terminate (thread must be joinable).
- **retval**: stores exit status of thread (set by `pthread_exit`) to the location pointed by *retval. If cancelled, returns `PTHREAD_CANCELED`. `NULL` is ignored.

This function returns 0 on success, error number otherwise.

Caveat: Only call this one time per thread! Multiple calls to join on the same thread lead to undefined behaviour.

Inter-thread communication. 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, (void*)&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.

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, this is marginally acceptable:

```
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.

- Beware size mismatches between arguments: you have no guarantee that a pointer is the same size as an int, so your data may overflow. (C only guarantees that the difference between two pointers is an int.)
- Sizes of data types change between systems. For maximum portability, just use pointers you got `from malloc`.

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.

More on inter-thread synchronization. There was a comment on `pthread_join` only working if the target thread was joinable. 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. We've seen C++11 detached threads above.

```
int pthread_detach(pthread_t thread);
```

- **thread**: marks the thread as detached

This call returns 0 on success, error number otherwise.

Calling `pthread_detach` on an already detached thread results in undefined behaviour.

Finishing a thread. A thread finishes when its `start_routine` returns. But it's also possible to explicitly end a thread from within:

```
void pthread_exit(void *retval);
```

- **retval**: return value passed to function which called `pthread_join`

Alternately, returning from the thread's `start_routine` is equivalent to calling `pthread_exit`, and `start_routine`'s return value is passed back to the `pthread_join` caller. There is no C++11 equivalent.

Attributes. 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
```

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

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

Warning about detached threads. Consider the following code.

```
#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.)

Threading Challenges.

- Be aware of scheduling (you can also set affinity with pthreads on Linux).
- Make sure the libraries you use are **thread-safe**:
  - Means that the library protects its shared data (we'll see how, below).
- 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`.

References

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