ECE459: Programming for Performance Winter 2013

Lecture 4 — January 17, 2013

Patrick Lam *version 1*

Processes versus Threads

The first design decision that you need to solve when parallelizing programs is whether you should use threads or processes.

- Threads are basically light-weight processes which piggy-back on processes' address space.

Traditionally (pre-Linux 2.6) you had to use `fork` (for processes) and `clone` (for threads). But `clone` is not POSIX compliant, and its man page says that it's Linux-specific—FreeBSD uses `rfork()`. (POSIX is the standard for Unix-like operating systems).

When processes are better. `fork` is safer and more secure than threads.

1. Each process has its own virtual address space:
   - Memory pages are not copied, they are copy-on-write. Therefore, processes use less memory than you would expect.
2. Buffer overruns or other security holes do not expose other processes.
3. If a process crashes, the others can continue.

Example: In the Chrome browser, each tab is a separate process. Scott McCloud explained this: `http://uncivilsociety.org/2008/09/google-chrome-comic-by-scott-m.html`.

When threads are better. Threads are easier and faster.

1. Interprocess communication (IPC) is more complicated and slower than interthread communication; must use operating system utilities (pipes, semaphores, shared memory, etc) instead of thread library (or just memory reads and writes).
2. Processes have much higher startup, shutdown and synchronization costs than threads.
3. Pthreads fix the issues of `clone` and provide a uniform interface for most systems. (You'll work with them in Assignment 1.)

How to choose? If your application is like this:

- mostly independent tasks, with little or no communication;
- task startup and shutdown costs are negligible compared to overall runtime; and
- want to be safer against bugs and security holes,

then processes are the way to go. If it's the opposite of this, then use threads.

For performance reasons, along with ease and consistency across systems, we'll use threads, and Pthreads in particular.

Creating and Using Processes

Let's start with a simple usage example showing how to use the basic `fork()` system call.

```
pid = fork();
if (pid < 0) {
        fork_error_function();
} else if (pid == 0) {
        child_function();
} else {
        parent_function();
}
```

`fork` produces a second copy of the calling process; both run concurrently after the call. The only difference between the copies is the return value: the parent gets the pid of the child, while the child gets 0.

Overhead of Processes. The common wisdom is that processes are expensive, threads are cheap. Let's verify this with a benchmark on a laptop which creates and destroys 50,000 threads:

```
jon@riker examples master % time ./create_fork
0.18s user 4.14s system 34% cpu 12.484 total
jon@riker examples master % time ./create_pthread
0.73s user 1.29s system 107% cpu 1.887 total
```

Clearly Pthreads incur much lower overhead than `fork`. Pthreads offer a speedup of 6.5 over processes in terms of startup and teardown costs.

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.

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

Starting a new thread. You can start a thread with the call `pthread_create()` as follows:

```
#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");
}
```

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.

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

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`.

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.

```
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.

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
}
```

Three useful Pthread calls. You may find these helpful.

```
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;
```

- `pthread_self` returns the handle of the currently running thread.
- `pthread_equal` compares 2 threads for equality.
- `pthread_once` runs a block of code just once, even across threads, based on a (presumably-global) control variable. This is useful for initialization code.

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, in Assignment 1, we'll use `rand_r`, not `rand`.

Synchronization

You'll need some sort of synchronization to get sane results from multithreaded programs. We'll start by talking about how to use mutual exclusion in Pthreads.

Mutual Exclusion. Mutexes are the most basic type of synchronization. As a reminder:

- Only one thread can access code protected by a mutex at a time.
- All other threads must wait until the mutex is free before they can execute the protected code.

Here's an example of using mutexes:

```
pthread_mutex_t m1_static = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t m2_dynamic;

pthread_mutex_init(&m2_dynamic, NULL);
...
pthread_mutex_destroy(&m1_static);
pthread_mutex_destroy(&m2_dynamic);
```

You can initialize mutexes statically (as with `m1_static`) or dynamically (`m2_dynamic`). If you want to include attributes, you need to use the dynamic version.

Mutex Attributes. Both threads and mutexes use the notion of attributes. We won't talk about mutex attributes in any detail, but here are the three standard ones.

- **Protocol**: specifies the protocol used to prevent priority inversions for a mutex.
- **Prioceiling**: specifies the priority ceiling of a mutex.

- **Process-shared**: specifies the process sharing of a mutex.

You can specify a mutex as *process shared* so that you can access it between processes. In that case, you need to use shared memory and `mmap`, which we won't get into.

Mutex Example. Let's see how this looks in practice. It is fairly simple:

```
// code
pthread_mutex_lock(&m1);
// protected code
pthread_mutex_unlock(&m1);
// more code
```

- Everything within the `lock` and `unlock` is protected.
- Be careful to avoid deadlocks if you are using multiple mutexes (always acquire locks in the same order across threads).
- Another useful primitive is `pthread_mutex_trylock`. We may come back to this later.

Data Races

Why are we bothering with locks? Data races. A data race occurs when two concurrent actions access the same variable and at least one of them is a **write**. (This shows up on Assignment 1!)

```
...
static int counter = 0;

void* run(void* arg) {
    for (int i = 0; i < 100; ++i) {
        ++counter;
    }
}

int main(int argc, char *argv[])
{
    // Create 8 threads
    // Join 8 threads
    printf("counter = %i\n", counter);
}
```

Is there a datarace in this example? If so, how would we fix it?

```
...
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
static int counter = 0;

void* run(void* arg) {
    for (int i = 0; i < 100; ++i) {
        pthread_mutex_lock(&mutex);
        ++counter;
        pthread_mutex_unlock(&mutex);
    }
}

int main(int argc, char *argv[])
```

```
{
    // Create 8 threads
    // Join 8 threads
    pthread_mutex_destroy(&mutex);
    printf("counter = %i\n", counter);
}
```

The `volatile` keyword

This qualifier notifies the compiler that a variable may be changed by "external forces". It therefore ensures that the compiled code does an actual read from a variable every time a read appears (i.e. the compiler can't optimize away the read). It does not prevent re-ordering nor does it protect against races.

Here's an example.

```
int i = 0;

while (i != 255) {
  ...
```

`volatile` prevents this from being optimized to:

```
int i = 0;

while (true) {
  ...
```

Note that the variable will not actually be `volatile` in the critical section; most of the time, it only prevents useful optimizations. `volatile` is usually wrong unless there is a *very* good reason for it.