Lecture 04—Pthreads and Simple Locks

ECE 459: Programming for Performance

January 16, 2014

Last time: threads, etc

Parallelism vs Concurrency;

Processes vs Threads;

Threading Models (live demo);

Multiprocessing.

Processes: fork Usage Example (OS refresher)

```
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, which starts execution 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.

Live coding: threads vs fork

Results: Threads Offer a Speedup of 6.5 over `fork`

Here's a benchmark between `fork` and Pthreads on a laptop, creating and destroying 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`.

Assumptions

First, we'll see 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 can be parallelized (dependencies)
- alternative parallelization patterns
  (right now, just use one thread per sub-problem)

POSIX Threads

- Available on most systems
- Windows has Pthreads Win32, but I wouldn't use it; use Linux for this course
- API available by `#include <pthread.h>`
- Compile with pthread flag (`gcc -pthread prog.c -o prog`)

Creating Threads

```
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

returns 0 on success, error number otherwise
(contents of *thread are undefined)

Creating Threads—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");
}
```

Simply creates a thread and terminates
(usage isn't really right, as we'll see.)

Waiting for Threads

```
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.

returns 0 on success, error number otherwise.

Only call this one time per thread! Multiple calls on the same thread leads to undefined behaviour.

Waiting for Threads—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);
}
```

This now waits for the newly created thread to terminate.

Passing Data to Threads...Wrongly

Consider this snippet:

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

This is a terrible idea. Why?

Passing Data to Threads...Wrongly

Consider this snippet:

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

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

Passing Data to Threads

What about:

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

...

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

This is suggested in the book, but should carry a warning:

Passing Data to Threads

What about:

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

...

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

This is suggested in the book, but should carry a warning:

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

Detached Threads

Joinable threads (the default) wait for someone to call `pthread_join` before they release their resources.

Detached threads release their resources when they terminate, without being joined.

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

thread: marks the thread as detached

returns 0 on success, error number otherwise.

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

Thread Termination

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

retval: return value passed to function that calls `pthread_join`

start_routine returning is equivalent to calling `pthread_exit` with that return value;

`pthread_exit` is called implicitly when the `start_routine` of a thread returns.

Attributes

By default, threads are *joinable* on Linux, but a more portable way to know what you're getting is to set thread attributes. You can change:

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

Attributes—Example

```
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
```

Setting a thread state to joinable:

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

Detached Threads: Warning!

```
#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?

Detached Threads: Solution to Problem

```
#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
}
```

Make the final call `pthread_exit` if you have any detached threads.

Three Useful Routines

```
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.

Use `pthread_equal` if you're comparing 2 threads.

If you want to run a section of code once, you need `pthread_once` (it's well-named). It will run only once per `once_control`.

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.
- glibc reentrant functions are also safe: a program can have more than one thread calling these functions concurrently.
- **Example:** `rand_r` versus `rand`.

Mutual Exclusion

Mutexes are the most basic type of synchronization.

- 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.

Live Coding Example: Mutual Exclusion

Creating Mutexes—Example

```
pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t m2;

pthread_mutex_init(&m2, NULL);
...
pthread_mutex_destroy(&m1);
pthread_mutex_destroy(&m2);
```

- Two ways to initialize mutexes: statically and dynamically
- If you want to include attributes, you need to use the dynamic version

Mutex Attributes

- **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.

Using Mutexes: Example

```
// 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.
- Also you can use `pthread_mutex_trylock`, if needed.

Data Race Example

Recall that dataraces occur when two concurrent actions access the same variable and at least one of them is a **write**

```
...
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?

Example Problem Solution

```
...
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);
}
```