Lecture 04 - Pthreads and Simple Locks

ECE 459: Programming for Performance

Jon Eyolfson

University of Waterloo

January 11, 2012

Background

- Recall the difference between a processes and threads
- Threads are basically a light-weight process that piggy-back on processes' address space
- Traditionally (pre Linux 2.6) you had to use `fork` (for processes) and `clone` (for threads)

History

- `clone` had a number of issues with POSIX compliance
  - Poor support for signal handling, scheduling, and inter-process synchronization primitives

- Mostly used `fork` in the past, which creates a new process
  - Drawbacks?
  - Benefits?

`fork` is Safer and More Secure Than Threads

- Each process has it's own virtual address space
  - Memory pages are not copied, they are copy-on-write
  - Therefore, uses less memory than you would expect
- Buffer overruns or other security holes do not expose other processes
- If a process crashes, the others can continue
- **Example:** In Chrome, each tab is a seperate process

Threads are Easier and Faster

- Interprocess communication (IPC) is harder and slower than interthread communication
  - Need to use operating system utilities (pipes, semaphores, shared memory, etc) instead of thread library
- Much higher startup, shutdown and synchronization cost
- Pthreads fix the issues of `clone` and provides a uniform interface for most systems **(focus of Assignment 1)**

Appropriate Time to Use Processes

If your application follows these guidelines:

- Mostly independent with little or no communication
- The startup and shutdown costs are negligible to overall runtime
- Want to be safer against bugs and security holes

For performance reasons, along with ease and consistency we'll use Pthreads (the same concepts apply to both)

Quick fork Usage

```
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 is the return value, the parent gets the pid of the child, the child gets 0

Threads Offer a Speedup of 6.5

Here's a benchmark between `fork` and Pthreads on my 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 offer much lower overhead

Assumptions

First we'll see how to use threads on "embarrassingly parallel problems"

- Made up of mostly independent sub-problems (little synchronization)
- Strong locality (little communication)

Later we'll see

- What 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 a 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 a should carry a warning:

- Be careful between size mismatches between the arguments, pointers are 4 bytes on a 32-bit machine and 8 bytes on a 64-bit machine, your data may overflow
- Sizes of variables also change between systems, for maximum portability just use pointers `through 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 that 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 of calling `pthread_exit` with that return value

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

Other Thread Utilities

```
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`

Attributes

By default, threads are *joinable* on Linux, but a more portable way is to set thread attributes. There 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 my 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 Thread 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 Thread Solution

```c
#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

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 it's shared data
- Reentrant code is also safe, it means a program can have more than one thread executing concurrently
- **Example:** In Assignment 1, we'll use `rand_r` instead of `rand`

Mutual Exclusion

- 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

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 this 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 a `pthread_mutex_trylock` if needed

Example Problem

Recall 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);
}
```

volatile Keyword

- Used to notify the compiler that the variable may change between lines of code

```
int i = 0;

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

volatile prevents this beening optimized to

```
int i = 0;

while (true) {
  ...
```

- Variable will not actually be volatile in the critical section and only prevents useful optimizations
- Usually wrong unless there is a very **very** good reason for it