ECE459: Programming for Performance	Winter 2014

Lecture 6 — January 23, 2014

Patrick Lam *version 1*

Edge-triggered vs level-triggered. We did a live coding demo and I promised more details in the notes. The example was some code (see `socket.c` in the code examples) that created a server and read from that server in either level-triggered mode or edge-triggered mode.

One would think that level-triggered mode would return from `read` whenever data was available, while edge-triggered mode would return from `read` whenever new data came in. Level-triggered does behave as one would guess: if there is data available, `read()` returns the data. However, edge-triggered mode returns whenever the state-of-readiness of the socket changes (from no-data-available to data-available). Play with it and get a sense for how it works.

Good question to think about: when is it appropriate to choose one or the other?

curl_multi

It's important to see at least one specific example of an idea. I talked about `epoll` last time and I meant that to be the specific example, but we can't quite use it without getting into socket programming, and I don't want to do that. Instead, we'll see non-blocking I/O in the specific example of the curl library, which is reasonably widely used in the Linux world.

Tragically, it's complicated to use `epoll` with `curl_multi`, and I couldn't quite figure it out. So I'll describe the `select`-based interface for `curl_multi`. A socket-based interface which works with `epoll` also exists. I won't talk about that.

The relevant steps, in any case, are:

- Create individual requests with `curl_easy_init`.
- Create a multi-handle with `curl_multi_init` and add the requests to it with `curl_multi_add_handle`.
- (for select-based interface:) put in requests & wait for results, using `curl_multi_perform`. That call generalizes `curl_easy_perform`.
- Handle completed transfers with `curl_multi_info_read`.

On the use of `curl_multi_perform`. The actual non-blocking read/write is done in `curl_multi_perform`, which returns the number of still-active handles through its parameter.

You call it in a loop, with a call to `select` above. Call `select` and then `curl_multi_perform` in a loop while there are still running transfers. You're also allowed to manipulate (delete/alter/re-add) a `curl_easy_handle` whenever a transfer finishes.

Setting up the `select`. Before you call `curl_multi_perform` and `select`, you need to set up the `select`. The curl call `curl_multi_fdset` sets up the parameters for the `select`, while `curl_multi_timeout` gives you the proper timeout to hand to `select`.

```
// zero the fd-sets
FD_ZERO(&fdread); FD_ZERO(&fdwrite); FD_ZERO(&fdexcep);
// retrieve the fds, check for error
curl_multi_fdset(multi_handle,
                 &fdread, &fdwrite, &fdexcep, &maxfd);
if (maxfd < -1) abort_("multi_fdset: couldn't wait for fds");
// retrieve the timeout
curl_multi_timeout(multi_handle, &curl_timeout);
```

In an API infelicity, you have to convert the `curl_timeout` into a `struct timeval` for use by `select`.

Calling `select`. The call itself is fairly straightforward:

```
rc = select(maxfd + 1, &fdread, &fdwrite, &fdexcep, &timeout);
if (rc == -1) abort_("[main] select error");
```

This waits for one of the file descriptors to become ready, or for the timeout to elapse (whichever happens first).

Of course, once `select` returns, you only know that something happened, but you haven't done the work yet. So you then need to call `curl_multi_perform` again to do the work.

Finally, you get the results of `curl_multi_perform` by calling `curl_multi_info_read`. It also tells you how many messages are left.

```
msg = curl_multi_info_read(multi_handle, &msgs_left);
```

The return value `msg->msg` can be either `CURLMSG_DONE` or an error. The handle `msg->easy_handle` tells you which handle finished. You may have to look that up in your collection of handles.

Cleanup. Always clean up after yourself! Use `curl_multi_cleanup` to destroy the multi-handle and `curl_easy_cleanup` to destroy each individual handle.

Example. There is a not-great example at

`http://curl.haxx.se/libcurl/c/multi-app.html`

but I'm not even sure it works verbatim. You could use it as a solution template, but you'll need to add more code—I asked you to replace completed transfers in the `curl_multi`.

About that socket-based alternative. There is yet another interface which would allow you to use `epoll`, but I couldn't figure it out. Sorry. The advantage, beyond using `epoll`, is that `libcurl` doesn't need to scan over all of the transfers when it receives notice that a transfer is ready. This can help when there are lots of sockets open.

From the manpage:

- Create a multi handle

- Set the socket callback with `CURLMOPT_SOCKETFUNCTION`
- Set the timeout callback with `CURLMOPT_TIMERFUNCTION`, to get to know what timeout value to use when waiting for socket activities.
- Add easy handles with `curl_multi_add_handle()`
- Provide some means to manage the sockets libcurl is using, so you can check them for activity. This can be done through your application code, or by way of an external library such as libevent or glib.
- Call `curl_multi_socket_action(..., CURL_SOCKET_TIMEOUT, 0, ...)` to kickstart everything. To get one or more callbacks called.
- Wait for activity on any of libcurl's sockets, use the timeout value your callback has been told.
- When activity is detected, call `curl_multi_socket_action()` for the socket(s) that got action. If no activity is detected and the timeout expires, call `curl_multi_socket_action(3)` with `CURL_SOCKET_TIMEOUT`.

There's an example, which has too many moving parts, here:

`http://curl.haxx.se/libcurl/c/hiperfifo.html`

It uses `libevent`, which I totally don't want to talk about in this class.

More synchronization primitives

We'll proceed in order of complexity.

Recap: Mutexes. Recall that our goal in this course is to be able to use mutexes correctly. You should have seen how to implement them in an operating systems course. Here's how to use them.

- Call `lock` on mutex ℓ_1. Upon return from `lock`, your thread has exclusive access to ℓ_1 until it `unlock`s it.
- Other calls to `lock` ℓ_1 will not return until `m1` is available.

For background on implementing mutual exclusion, see Lamport's bakery algorithm. Implementation details are not in scope for this course.

Key idea: locks protect resources; only one thread can hold a lock at a time. A second thread trying to obtain the lock (i.e. *contending* for the lock) has to wait, or *block*, until the first thread releases the lock. So only one thread has access to the protected resource at a time. The code between the lock acquisition and release is known as the *critical region*.

Some mutex implementations also provide a "try-lock" primitive, which grabs the lock if it's available, or returns control to the thread if it's not, thus enabling the thread to do something else. (Kind of like non-blocking I/O!)

Excessive use of locks can serialize programs. Consider two resources A and B protected by a single lock ℓ. Then a thread that's just interested in B still has acquire ℓ, which requires it to wait for any other thread working with A. (The Linux kernel used to rely on a Big Kernel Lock protecting lots of resources in the 2.0 era, and Linux 2.2 improved performance on SMPs by cutting down on the use of the BKL.)

Note: in Windows, the term "mutex" refers to an inter-process communication mechanism. "Critical sections" are the mutexes we're talking about above.

Spinlocks. Spinlocks are a variant of mutexes, where the waiting thread repeatedly tries to acquire the lock instead of sleeping. Use spinlocks when you expect critical sections to finish quickly[1]. Spinning for a long time consumes lots of CPU resources. Many lock implementations use both sleeping and spinlocks: spin for a bit, then sleep for longer. At some point, we saw a live coding example comparing spinlocks to normal mutexes.

Reader/Writer Locks. Recall that data races only happen when one of the concurrent accesses is a write. So, if you have read-only ("immutable") data, as often occurs in functional programs, you don't need to protect access to that data. For instance, your program might have an initialization phase, where you write some data, and then a query phase, where you use multiple threads to read the data.

Unfortunately, sometimes your data is not read-only. It might, for instance, be rarely updated. Locking the data every time would be inefficient. The answer is to instead use a *reader/writer* lock. Multiple threads can hold the lock in read mode, but only one thread can hold the lock in write mode, and it will block until all the readers are done reading.

```
int readData(int c1, int c2) {              // glib usage example
  g_static_rw_lock_reader_lock (&rwlock);
  int result = data[c1] + data[c2];
  g_static_rw_lock_reader_unlock (&rwlock);
}

void writeData(int c1, int c2, int value) {
  g_static_rw_lock_writer_lock (&rwlock);
  data[c1] += value; data[c2] -= value;
  g_static_rw_lock_writer_unlock (&rwlock);
}
```

Semaphores/condition variables. While semaphores can keep track of a counter and can implement mutexes, you should use them to support signalling between threads or processes.

In pthreads, semaphores can also be used for inter-process communication, while condition variables are like Java's `wait()`/`notify()`.

Barriers. This synchronization primitive allows you to make sure that a collection of threads all reach the barrier before finishing. In pthreads, each thread should call `pthread_barrier_wait()`,

[1]For more information on spinlocks in the Linux kernel, see `http://lkml.org/lkml/2003/6/14/146`.

which will proceed when enough threads have reached the barrier. Enough means a number you specify upon barrier creation.

Lock-Free Code. We'll talk more about this in a few weeks. Modern CPUs support atomic operations, such as compare-and-swap, which enable experts to write lock-free code. A recent research result [McK11, AGH+11] states the requirements for correct implementations: basically, such implementations must contain certain synchronization constructs.

Semaphores

As you learned in previous courses, semaphores have a `value` and can be used for signalling between threads. When you create a semaphore, you specify an initial value for that semaphore. Here's how they work.

- The `value` can be understood to represent the number of resources available.
- A semaphore has two fundamental operations: `wait` and `post`.
- `wait` reserves one instance of the protected resource, if currently available—that is, if `value` is currently above 0. If `value` is 0, then `wait` suspends the thread until some other thread makes the resource available.
- `post` releases one instance of the protected resource, incrementing `value`.

Semaphore Usage. Here are the relevant API calls.

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
```

This API is a lot like the mutex API:

- must link with `-pthread` (or `-lrt` on Solaris);
- all functions return `0` on success;
- same usage as mutexes in terms of passing pointers.

How could you use a `semaphore` as a `mutex`?

Semaphores for Signalling. Here's an example from the book. How would you make this always print "Thread 1" then "Thread 2" using semaphores?

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

void* p1 (void* arg) { printf("Thread 1\n"); }

void* p2 (void* arg) { printf("Thread 2\n"); }

int main(int argc, char *argv[])
{
    pthread_t thread[2];
    pthread_create(&thread[0], NULL, p1, NULL);
    pthread_create(&thread[1], NULL, p2, NULL);
    pthread_join(thread[0], NULL);
    pthread_join(thread[1], NULL);
    return EXIT_SUCCESS;
}
```

Proposed Solution. Is it actually correct?

```
sem_t sem;
void* p1 (void* arg) {
  printf("Thread 1\n");
  sem_post(&sem);
}
void* p2 (void* arg) {
  sem_wait(&sem);
  printf("Thread 2\n");
}

int main(int argc, char *argv[])
{
    pthread_t thread[2];
    sem_init(&sem, 0, /* value: */ 1);
    pthread_create(&thread[0], NULL, p1, NULL);
    pthread_create(&thread[1], NULL, p2, NULL);
    pthread_join(thread[0], NULL);
    pthread_join(thread[1], NULL);
    sem_destroy(&sem);
}
```

Well, let's reason through it.

- `value` is initially 1.
- Say `p2` hits its `sem_wait` first and succeeds.
- `value` is now 0 and `p2` prints "Thread 2" first.
- It would be OK if `p1` happened first. That would just increase `value` to 2.

Fix: set the initial `value` to 0. Then, if `p2` hits its `sem_wait` first, it will not print until `p1` posts, which is after `p1` prints "Thread 1".

References

[AGH+11] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 487–498, New York, NY, USA, 2011. ACM.

[McK11] Paul McKenney. Concurrent code and expensive instructions. Linux Weekly News, `http://lwn.net/Articles/423994/`, January 2011.