

Lecture 06—curl_multi and libevent

January 23, 2014

Last Time

- A1 discussion (more on libcurl, libevent this time!)
- Asynchronous/non-blocking I/O
 - epoll; O_NONBLOCK (only works for sockets)
- What happens when you have a race condition

A1 Announcements

- `ece459-1.uwaterloo.ca` available for login.
ssh keys only, upload your key to `ecegit`.
- I've completed my sample solutions.
- New requirement in `r3`:
Part 0 = 5 marks for fixing a resource leak in my sample code. Describe it in the report.

Live coding: level-triggered vs edge-triggered

Last time, I said that you could use `epoll` in either level-triggered mode or edge-triggered mode.

Here's a demo.

When would you use one or the other?

Part I

Using curl_multi

curl_multi initialization

curl_multi: work with multiple resources at once.

How? Similar idea to epoll from last time:

1. To use curl_multi, first create the individual requests (curl_easy_init).
(Set options as needed on each handle).
2. Then, combine them with:
 - curl_multi_init();
 - curl_multi_add_handle().

curl_multi_perform: option 1, select-based interface

Main idea: put in requests and wait for results.

`curl_multi_perform` is a generalization of `curl_easy_perform` to multiple resources.

Handle completed transfers with `curl_multi_info_read`.

calling curl_multi_perform

perform interface requires use of select (not epoll).

usage (once you've curl_multi_add_handle'd):

```
curl_multi_perform(multi_handle, &still_running)
```

performs a non-blocking read/write, and
returns the number of still-active handles
(with more data to come).

Next steps after `curl_multi_perform`

do

- organize a call to `select`; and
- call `curl_multi_perform` again

while there are still running transfers.

After the `curl_multi_perform`, you can also delete, alter, and re-add an `curl_easy_handle` when a transfer finishes.

Before calling select

select needs a timeout and an fdset.
(curl provides both.)

Initializing the fdset from the multi_handle:

```
// 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");
```

Retrieving the proper timeout:

```
curl_multi_timeout(multi_handle, &curl_timeout);
```

(and then convert the long to a struct timeval).

The call to select

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

Wait for one of the fds to become ready,
or for timeout to elapse.

What next?

The call to select

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

Wait for one of the fds to become ready,
or for timeout to elapse.

What next?

Call `curl_multi_perform` again to do the work.

Knowing what happened after `curl_multi_perform`

`curl_multi_info_read` will tell you.

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

and also how many messages are left.

`msg->msg` can be `CURLMSG_DONE` or an error;
`msg->easy_handle` tells you who is done.

curl_multi cleanup

Call `curl_multi_cleanup` on the multi handle.

Then, call `curl_easy_cleanup` on each easy handle.

curl_multi_perform example

Not a great example:

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

I'm not even sure it works verbatim.

Nevertheless, you could use it as a solution template.

You'll have to add more code to replace completed transfers.

curl_multi, option 2: curl_multi_socket_action

So, I couldn't quite figure out how this works. Sorry.

Similar to the perform interface, but you have more control.
Advantage:

2 - When the application discovers action on a single socket, it calls libcurl and informs that there was action on this particular socket and libcurl can then act on that socket/transfer only and not care about any other transfers. (The previous API always had to scan through all the existing transfers.)

`http://curl.haxx.se/dev/readme-multi_socket.html`

multi_socket usage

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`.

multi_socket example

This example is even worse than the last one:

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

It contains more moving parts than we need to understand the API, and gets another library (`libevent`) involved.

Part II

More Synchronization

Mutexes Recap

Our focus is on [how to use mutexes correctly](#):

- Call `lock` on mutex `m1`. Upon return from `lock`, you have exclusive access to `m1` until you `unlock` it.
- Other calls to `lock m1` will not return until `m1` is available.

For background on selection algorithms, look at Lamport's bakery algorithm.
(Not in scope for this course.)

More on Mutexes

Can also “try-lock”: grab lock if available, else return to caller (and do something else).

Excessive use of locks can serialize programs.

- Linux kernel used to rely on a Big Kernel Lock protecting lots of resources in the 2.0 era.
- Linux 2.2 improved performance on SMPs by cutting down on the use of the BKL.

Note: in Windows, “mutex” is an inter-process communication mechanism. Windows “critical sections” are our mutexes.

Spinlocks

Functionally equivalent to mutex.

- `pthread_spinlock_t`, `pthread_spin_lock`,
`pthread_spin_trylock` and friends

Implementation difference: spinlocks will repeatedly try the lock and will not put the thread to sleep.

Good if your protected code is short.

Mutexes may be implemented as a combination between spinning and sleeping (spin for a short time, then sleep).

Read-Write Locks

Two observations:

- If there are only reads, there's no data race.
- Often, writes are relatively rare.

With mutexes/spinlocks, you have to lock the data, even for a read, since a write could happen.

But, most of the time, reads can happen in parallel, as long as there's no write.

Solution: Multiple threads can hold a read lock

`(pthread_rwlock_rdlock)`

but only one thread may hold the associated write lock

`(pthread_rwlock_wrlock);`

grabbing the write waits until current readers are done.

Semaphores

Semaphores have a value. You specify initial value.

Semaphores allow sharing of a # of instances of a resource.

Two fundamental operations: wait and post.

- wait is like lock; reserves the resource and decrements the value.
 - ▶ If value is 0, sleep until value is greater than 0.
- post is like unlock; releases the resource and increments the value.

Barriers

Allows you to ensure that (some subset of) a collection of threads all reach the barrier before finishing.

Pthreads: A barrier is a `pthread_barrier_t`.

Functions: `_init()` (parameter: how many threads the barrier should wait for) and `_destroy()`.

Also `_wait()`: similar to `pthread_join()`, but waits for the specified number of threads to arrive at the barrier

Lock-Free Algorithms

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.

Lock-free implementations are extremely complicated and must still contain certain synchronization constructs.

Semaphores Usage

```
#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);
```

- Also 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 as semaphore as a mutex?

Semaphores Usage

```
#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);
```

- Also 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 as semaphore as a mutex?

- If the initial value is 1 and you use `wait` to lock and `post` to unlock, it's equivalent to 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;
}
```

Semaphores for Signalling

Here's their 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);
}
```

Semaphores for Signalling

- ① value is initially 1.
- ② Say p2 hits its `sem_wait` first and succeeds.
- ③ value is now 0 and p2 prints “Thread 2” first.
 - If p1 happens first, it would just increase value to 2.

Semaphores for Signalling

- ① value is initially 1.
- ② Say p2 hits its `sem_wait` first and succeeds.
- ③ value is now 0 and p2 prints “Thread 2” first.
 - If p1 happens first, it 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 (and prints “Thread 1”) first.