Lecture 07—Async I/O; Race Conditions; More Synchronization

January 19, 2015

Roadmap

Past: Modern Hardware, Threads;

Now: Non-blocking I/O;

Next: Race Conditions, Locking.

Last Time

- Assignment 1 walkthrough.
- Concept behind non-blocking I/O.

Part I

Async I/O with epoll

Using `epoll`

Key idea: give `epoll` a bunch of file descriptors;
wait for events to happen.

Steps:

1. create an instance (`epoll_create1`);
2. populate it with file descriptors (`epoll_ctl`);
3. wait for events (`epoll_wait`).

Creating an epoll instance

```
int epfd = epoll_create1(0);
```

efpd doesn't represent any files; use it to talk to epoll.

0 represents the flags (only flag: EPOLL_CLOEXEC).

Populating the `epoll` instance

To add `fd` to the set of descriptors watched by `epfd`:

```
struct epoll_event event;
int ret;
event.data.fd = fd;
event.events = EPOLLIN | EPOLLOUT;
ret = epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &event);
```

Can also modify and delete descriptors from `epfd`.

Waiting on an `epoll` instance

Now we're ready to wait for events on any file descriptor in `epfd`.

```
#define MAX_EVENTS 64

struct epoll_event events[MAX_EVENTS];
int nr_events;

nr_events = epoll_wait(epfd, events, MAX_EVENTS, -1);
```

-1: wait potentially forever; otherwise, milliseconds to wait.

Upon return from `epoll_wait`, we have `nr_events` events ready.

Level-Triggered and Edge-Triggered Events

Default `epoll` behaviour is level-triggered:
return whenever data is ready.

Can also specify (via `epoll_ctl`) edge-triggered behaviour:
return whenever there is a change in readiness.

Live Coding: Level-Triggered vs Edge-Triggered

Asynchronous I/O

POSIX standard defines aio calls.

These work for disk as well as sockets.

Key idea: you specify the action to occur when I/O is ready:

- nothing;
- start a new thread;
- raise a signal

Submit the requests using e.g. `aio_read` and `aio_write`.

Can wait for I/O to happen using `aio_suspend`.

Nonblocking I/O with curl

Similar idea to `epoll`:

- build up a set of descriptors;
- invoke the transfers and wait for them to finish;
- see how things went.

Part II

Using curl_multi

curl_multi initialization

curl_multi: work with multiple resources at once.

How? Similar idea to `epoll`:

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 III

Race Conditions

Race Conditions

- A race occurs when you have two concurrent accesses to the same memory location, at least one of which is a **write**.

When there's a race, the final state may not be the same as running one access to completion and then the other.

Race conditions arise between variables which are shared between threads.

Example Data Race (Part 1)

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

void* run1(void* arg)
{
    int* x = (int*) arg;
    *x += 1;
}

void* run2(void* arg)
{
    int* x = (int*) arg;
    *x += 2;
}
```

Example Data Race (Part 2)

```
int main(int argc, char *argv[])
{
    int* x = malloc(sizeof(int));
    *x = 1;
    pthread_t t1, t2;
    pthread_create(&t1, NULL, &run1, x);
    pthread_join(t1, NULL);
    pthread_create(&t2, NULL, &run2, x);
    pthread_join(t2, NULL);
    printf("%d\n", *x);
    free(x);
    return EXIT_SUCCESS;
}
```

Do we have a data race? Why or why not?

Example Data Race (Part 2)

```
int main(int argc, char *argv[])
{
    int* x = malloc(sizeof(int));
    *x = 1;
    pthread_t t1, t2;
    pthread_create(&t1, NULL, &run1, x);
    pthread_join(t1, NULL);
    pthread_create(&t2, NULL, &run2, x);
    pthread_join(t2, NULL);
    printf("%d\n", *x);
    free(x);
    return EXIT_SUCCESS;
}
```

Do we have a data race? Why or why not?

- No, we don't. Only one thread is active at a time.

Example Data Race (Part 2B)

```
int main(int argc, char *argv[])
{
    int* x = malloc(sizeof(int));
    *x = 1;
    pthread_t t1, t2;
    pthread_create(&t1, NULL, &run1, x);
    pthread_create(&t2, NULL, &run2, x);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("%d\n", *x);
    free(x);
    return EXIT_SUCCESS;
}
```

Do we have a data race now? Why or why not?

Example Data Race (Part 2B)

```
int main(int argc, char *argv[])
{
    int* x = malloc(sizeof(int));
    *x = 1;
    pthread_t t1, t2;
    pthread_create(&t1, NULL, &run1, x);
    pthread_create(&t2, NULL, &run2, x);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("%d\n", *x);
    free(x);
    return EXIT_SUCCESS;
}
```

Do we have a data race now? Why or why not?

- Yes, we do. We have 2 threads concurrently accessing the same data.

Tracing our Example Data Race

What are the possible outputs? (initially `*x` is 1).

```
1  run1                          run2
2  D.1 = *x;                     D.1 = *x;
3  D.2 = D.1 + 1;                D.2 = D.1 + 2
4  *x = D.2;                     *x = D.2;
```

- Memory reads and writes are key in data races.

Outcome of Example Data Race

- Let's call the read and write from `run1` R1 and W1; R2 and W2 from `run2`.
- Assuming a sane[1] memory model, R_n must precede W_n.

All possible orderings:

Order				`*x`
R1	W1	R2	W2	4
R1	R2	W1	W2	3
R1	R2	W2	W1	2
R2	W2	R1	W1	4
R2	R1	W2	W1	2
R2	R1	W1	W2	3

[1] sequentially consistent

Detecting Data Races Automatically

Dynamic and static tools can help find data races in your program.

- `helgrind` is one such tool. It runs your program and analyzes it (and causes a large slowdown).

Run with `valgrind --tool=helgrind <prog>`.

It will warn you of possible data races along with locations.

For useful debugging information, compile with debugging information (`-g` flag for `gcc`).

Helgrind Output for Example

```
==5036== Possible data race during read of size 4 at
         0x53F2040 by thread #3
==5036== Locks held: none
==5036==    at 0x400710: run2 (in datarace.c:14)
...
==5036==
==5036== This conflicts with a previous write of size 4 by
         thread #2
==5036== Locks held: none
==5036==    at 0x400700: run1 (in datarace.c:8)
...
==5036==
==5036== Address 0x53F2040 is 0 bytes inside a block of size
         4 alloc'd
...
==5036==    by 0x4005AE: main (in datarace.c:19)
```

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);
}
```

Part IV

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 datarace.
- 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.