Lecture 06—A1; Race Conditions; More Synchronization; Async I/O

January 21, 2014

Roadmap

Past: Modern Hardware, Threads

Now: A1 discussion, non-blocking I/O;

Next: Race Conditions, Locking

Last Time

- Processes vs threads.
- Creating, joining and exiting POSIX threads. Remember, they are 1:1 with kernel threads and can run in parallel on multiple CPUs.
- Difference between joinable and detached threads.

Part I

Assignment 1

Your Task

Re-assemble the picture:

  

What I provide

Serial C code:

- uses curl to fetch the image over the network;
- uses libpng to stitch together the image.

Plus, a web API to provide images to you.

What you hand in

Part 1: pthreads parallelized implementation.

- really easy!
- (also, analyze your speedups in the report.)

Part 2: nonblocking I/O implementation.

- more challenging;
- lecture today will help.

Also Parts 0, 3: analysis & discussion.

Tour of the code

main loop: while still missing some fragments,

- retrieve a fragment over the network;
- copy bits into our array;

Then, write all the bits in one PNG file.

Notable bits I: retrieving the file

Here's how I retrieve the file:

```
curl_easy_setopt(curl, CURLOPT_URL, url);

// do curl request; check for errors
res = curl_easy_perform(curl);
```

But wait! I had to tell curl where to put the file:

```
struct bufdata bd;
bd.buf = input_buffer;
curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, write_cb);
curl_easy_setopt(curl, CURLOPT_WRITEDATA, &bd);
```

My `write_cb` callback function puts data in `input_buffer` (straightforward `memcpy`-based implementation).

Notable bits II: parsing the fragments

Bunch of `libpng` magic:

`libpng` wants to put the image data in a `png_bytep *` array, where each element points to a row of pixels.

My `read_png_file` function allocates the data; caller must free.

Then, `paint_destination` fills in the output array, pasting together the fragments.

Notable bits III: writing the output

Well, not that notable. Symmetric to read.
Note: be sure to free everything! (We'll check.)

Part (a): using pthreads

You might need to refactor the code to parallelize it well.

Start some threads.

Justify why the threads are not interfering. Time the result.

Part (b): nonblocking I/O

Main subject of this lecture.
Will be more complicated than using threads!

Part (b)': JavaScript

As an alternate option, you may use either `node.js` or client-side JavaScript to do the nonblocking I/O.

Let me know if you want to do this. You are on your own, though.

Part II

Asynchronous/non-blocking I/O

Juicy Quotes

Asynchronous I/O on linux

or: Welcome to hell.

(mirrored at `compgeom.com/~piyush/teach/4531_06/project/hell.html`)

"Asynchronous I/O, for example, is often infuriating."
— Robert Love. *Linux System Programming, 2nd ed,* page 215.

Why non-blocking I/O?

Consider some I/O:

```
fd = open(...);
read(...);
close(fd);
```

Not very performant—under what conditions do we lose out?

Mitigating I/O impact

So far: can use threads to mitigate latency.
What are the disadvantages?

Mitigating I/O impact

So far: can use threads to mitigate latency.
What are the disadvantages?

- race conditions
- overhead/max # of thread limitations

Live coding: forkbomb Patrick's laptop!

(well, threadbomb anyway)

An Alternative to Threads

Asynchronous/nonblocking I/O.

```
fd = open(..., O_NONBLOCK);
read(...); // returns instantly!
close(fd);
```

...

(credit: Yskyflyer, Wikimedia Commons)

Not Quite So Easy: Live Demo

Doesn't work on files—they're always ready. Only e.g. sockets.

Other Outstanding Problem with Nonblocking I/O

How do you know when I/O is ready to be queried?

Other Outstanding Problem with Nonblocking I/O

How do you know when I/O is ready to be queried?

- polling (`select`, `poll`, `epoll`)
- interrupts (signals)

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.

We'll see an example next time.

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 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).

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