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)