ECE459: Programming for Performance	Winter 2014
Lecture 5 — January 21, 2014	
Patrick Lam	*version 1*

Assignment 1 Discussion

We talked about how to do assignment 1. The task is to reassemble a picture that you fetch over the Internet using curl. You get a C implementation that uses curl to fetch the code over the network and uses libpng to read and write PNG files. It populates a buffer based on the input files and outputs a buffer combining the files' content.

Main Loop. The main loop looks like this:

main loop: for each image fragment,
  retrieve the fragment over the network;
  copy bits into our array;
Then, write all the bits in one PNG file.

You hand in (0) a fix to a resource leak in my code; (1) a pthread parallelized implementation; and (2) a nonblocking I/O implementation using the libcurl multi-handle interface.

Retrieving the files. I discussed the API for retrieving the file:

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

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

However, that wasn't enough. Before that, I had to tell curl where to put the file—it was to use the `write_cb` callback function.

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

`write_cb` puts data in `input_buffer` using a straightforward `memcpy`-based implementation. It doesn't do anything fancy, but does make sure that it doesn't overflow the buffer.

Parsing the input .PNG files. This consisted of a bunch of `libpng` magic: `libpng` will 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. I've chosen the convention that the caller must free the returned value. These conventions can trip you up and cause memory leaks if they aren't inconsistently used.

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

Writing the output .PNG file. This is simply symmetric to the read part.

Note: be sure to free everything! (We'll check.)

Using pthreads

I found this to be quite easy, but I noticed that people found all sorts of ways to do this which I hadn't anticipated. In any case, I expect that you will have to do some refactoring. I sort of on purpose made it not immediately amenable to refactoring.

You need to start some threads. Then, justify why the threads are not interfering. Time the result.

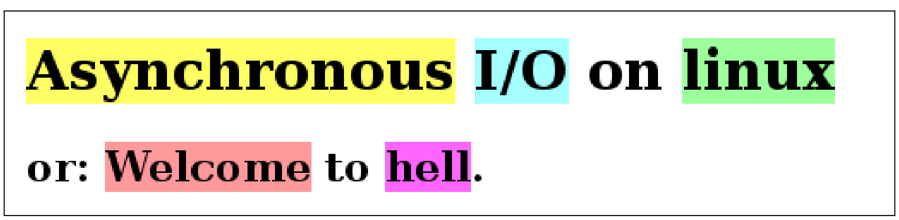
Nonblocking I/O

This part is more complicated than using threads. It is typically lower overhead (why?) and good for servers which handle lots of connections. But it is also more of a pain to program. On the other hand, you don't have to worry about shared state.

JavaScript option. As an alternate option, you were allowed to use either `node.js` or client-side JavaScript to do the nonblocking I/O. You are on your own for this option. Let me know; I'll mark those solutions myself.

Asynchronous/non-blocking I/O

Let's start with some juicy quotes.



(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.

To motivate the need for non-blocking I/O, consider some standard I/O code:

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

This isn't very performant. The problem is that the `read` call will *block*. So, your program doesn't get to use the zillions of CPU cycles that are happening while the I/O operation is occurring.

As seen previously: threads. That can be fine if you have some other code running to do work—for instance, other threads do a good job mitigating the I/O latency, perhaps doing I/O themselves. But maybe you would rather not use threads. Why not?

- potential race conditions;
- overhead due to per-thread stacks; or
- limitations due to maximum numbers of threads.

Live coding example. To illustrate the max-threads issue, we wrote `threadbomb.c`, which explored how many simultaneous threads one could start on my computer.

Non-blocking I/O. The main point of this lecture, though, is non-blocking/asynchronous I/O. The simplest example:

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

In principle, the `read` call is supposed to return instantly, whether or not results are ready. That was easy!

Well, not so much. The `O_NONBLOCK` flag actually only has the desired behaviour on sockets. The semantics of `O_NONBLOCK` is for I/O calls to not block, in the sense that they should never wait for data while there is no data available.

Unfortunately, files always have data available. Under Linux, you'd have to use `aio` calls to be able to send requests to the I/O subsystem asynchronously and not, for instance, wait for the disk to spin up. We won't talk about them, but they operate along the same lines as what we will see. They just have a different API.

Conceptual view: non-blocking I/O. Fundamentally, there are two ways to find out whether I/O is ready to be queried: polling (under UNIX, implemented via `select`, `poll`, and `epoll`) and interrupts (under UNIX, signals).

We will describe `epoll` in lecture. It is the most modern and flexible interface. Unfortunately, I didn't realize that the obvious `curl` interface does not work with `epoll` but instead with `select`. There is different syntax but the ideas are the same.

The key idea is to give `epoll` a bunch of file descriptors and wait for events to happen. In particular:

- create an epoll instance (`epoll_create1`);
- populate it with file descriptors (`epoll_ctl`); and
- wait for events (`epoll_wait`).

Let's run through these steps in order.

Creating an `epoll` instance. Just use the API:

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

The return value `epfd` is typed like a UNIX file descriptor—`int`—but doesn't represent any files; instead, use it as an identifier, to talk to `epoll`.

The parameter "`0`" represents the flags, but the only available flag is `EPOLL_CLOEXEC`. Not interesting to you.

Populating the `epoll` instance. Next, you'll want `epfd` to do something. The obvious thing is to add some `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);
```

You can also use `epoll_ctl` to modify and delete descriptors from `epfd`; read the manpage to find out how.

Waiting on an `epoll` instance. Having completed the setup, 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);
```

The given `-1` parameter means to wait potentially forever; otherwise, the parameter indicates the number of milliseconds to wait. (It is therefore "easy" to sleep for some number of milliseconds by starting an `epfd` and using `epoll_wait`; takes two function calls instead of one, but allows sub-second latency.)

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

Level-Triggered and Edge-Triggered Events

One relevant concept for these polling APIs is the concept of *level-triggered* versus *edge-triggered*. The default `epoll` behavious is level-triggered: it returns whenever data is ready. One can also specify (via `epoll_ctl`) edge-triggered behaviour: return whenever there is a change in readiness.

We had a live coding demo in lecture 6 and found that edge-triggered didn't mean what we thought it would mean. See those notes for details.

Asynchronous I/O

As mentioned above, the POSIX standard defines `aio` calls. Unlike just giving the `O_NONBLOCK` flag, using `aio` works for disk as well as sockets.

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

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

Your code submits the requests using e.g. `aio_read` and `aio_write`. If needed, wait for I/O to happen using `aio_suspend`.

Nonblocking I/O with curl. The next lecture notes give more clue about nonblocking I/O with curl. Although it doesn't work with `epoll` but rather `select`, it uses the same ideas—we'll therefore see two (three, with aio) different implementations of the same idea. Briefly, you:

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

Race Conditions

We'll next use our knowledge of three address code to analyze potential race conditions more rigourously.

Definition. 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. (But it sometimes is.) Race conditions typically arise between variables which are shared between threads.

```
#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;
}

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;
}
```

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

Example 2. Here's another example; keep the same thread definitions.

```
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;
}
```

Now do we have a data race? Why or why not?

Tracing our Example Data Race. What are the possible outputs? (Assume that initially `*x` is 1.) We'll look at the three-address code to tell.

```
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.

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.

[1]sequentially consistent; sadly, many widely-used models are wilder than this.

Here are 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

Detecting Data Races Automatically

Dynamic and static tools exist. They can help you 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 your program with debugging information (`-g` flag for `gcc`).

```
==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)
```