

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: until all image fragment fetched,  
    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; (2) a nonblocking I/O implementation using the libcurl multi-handle interface; and (3) an Amdahl's Law etc. discussion.

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);
```

(The server then gives you back an arbitrary image fragment and you have to loop until you get them all.) 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.

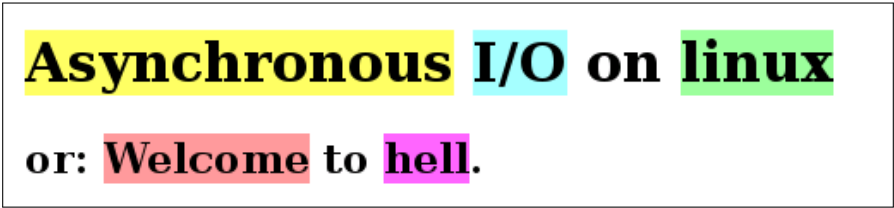
Nonblocking I/O

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.



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.

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).