

## Lecture 7 — January 19, 2015

*Patrick Lam**version 1*

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` behaviour 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 did a live coding demo; here are more details. The example was some code (see `L07-socket.c` in the code examples) that created a server and read from that server in either level-triggered mode or edge-triggered mode.

One would think that level-triggered mode would return from `read` whenever data was available, while edge-triggered mode would return from `read` whenever new data came in. Level-triggered does behave as one would guess: if there is data available, `read()` returns the data. However, edge-triggered mode returns whenever the state-of-readiness of the socket changes (from no-data-available to data-available). Play with it and get a sense for how it works.

Good question to think about: when is it appropriate to choose one or the other?

## 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.

## curl\_multi

It's important to see at least one specific example of an idea. I talked about `epoll` last time and I meant that to be the specific example, but we can't quite use it without getting into socket programming, and I don't want to do that. Instead, we'll see non-blocking I/O in the specific example of the `curl` library, which is reasonably widely used in the Linux world.

Tragically, it's complicated to use `epoll` with `curl_multi`, and I couldn't quite figure it out. So I'll describe the `select`-based interface for `curl_multi`. A socket-based interface which works with `epoll` also exists. I won't talk about that.

The relevant steps, in any case, are:

- Create individual requests with `curl_easy_init`.
- Create a multi-handle with `curl_multi_init` and add the requests to it with `curl_multi_add_handle`.
- (for `select`-based interface:) put in requests & wait for results, using `curl_multi_perform`. That call generalizes `curl_easy_perform`.
- Handle completed transfers with `curl_multi_info_read`.

**On the use of `curl_multi_perform`.** The actual non-blocking read/write is done in `curl_multi_perform`, which returns the number of still-active handles through its parameter.

You call it in a loop, with a call to `select` above. Call `select` and then `curl_multi_perform` in a loop while there are still running transfers. You're also allowed to manipulate (delete/alter/re-add) a `curl_easy_handle` whenever a transfer finishes.

**Setting up the `select`.** Before you call `curl_multi_perform` and `select`, you need to set up the `select`. The `curl` call `curl_multi_fdset` sets up the parameters for the `select`, while `curl_multi_timeout` gives you the proper timeout to hand to `select`.

```
// 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");
// retrieve the timeout
curl_multi_timeout(multi_handle, &curl_timeout);
```

In an API infelicity, you have to convert the `curl_timeout` into a `struct timeval` for use by `select`.

**Calling `select`.** The call itself is fairly straightforward:

```
rc = select(maxfd + 1, &fdread, &fdwrite, &fdexcep, &timeout);
if (rc == -1) abort_("[main] select error");
```

This waits for one of the file descriptors to become ready, or for the timeout to elapse (whichever happens first).

Of course, once `select` returns, you only know that something happened, but you haven't done the work yet. So you then need to call `curl_multi_perform` again to do the work.

Finally, you get the results of `curl_multi_perform` by calling `curl_multi_info_read`. It also tells you how many messages are left.

```
msg = curl_multi_info_read(multi_handle, &msgs_left);
```

The return value `msg->msg` can be either `CURLMSG_DONE` or an error. The handle `msg->easy_handle` tells you which handle finished. You may have to look that up in your collection of handles.

Some gotchas (thanks Desiye Collier):

- Checking `msg->msg == CURLMSG_DONE` is not sufficient to ensure that a curl request actually happened. You also need to check `data.result`.
- (A1 hint:) To reset an individual handle in the `multi_handle`, you need to “replace” it. But you shouldn't use `curl_easy_init()` to replace the handle. In fact, you don't need a new handle at all.

**Cleanup.** Always clean up after yourself! Use `curl_multi_cleanup` to destroy the multi-handle and `curl_easy_cleanup` to destroy each individual handle. If you replace the `curl_easy_init` call by `curl_global_init` for the multithreaded case (which is a good idea), then you should use `curl_global_cleanup` to clean up.

**Example.** There is a not-great example at

```
http://curl.haxx.se/libcurl/c/multi-app.html
```

but I'm not even sure it works verbatim. You could use it as a solution template, but you'll need to add more code—I asked you to replace completed transfers in the `curl_multi`.

**A better way to use sockets.** Late-breaking news: instead of that `select()` crud, use `curl_multi_wait()`, which is just better, and easy to use. An example: <https://gist.github.com/clemensg/4960504>

**About that socket-based alternative.** There is yet another interface which would allow you to use `epoll`, but I couldn't figure it out. Sorry. The advantage, beyond using `epoll`, is that `libcurl` doesn't need to scan over all of the transfers when it receives notice that a transfer is ready. This can help when there are lots of sockets open.

From the manpage:

- Create a multi handle
- Set the socket callback with `CURLOPT_SOCKETFUNCTION`

- Set the timeout callback with `CURLOPT_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`.

There's an example, which has too many moving parts, here:

<http://curl.haxx.se/libcurl/c/hiperfifo.html>

It uses `libevent`, which I totally don't want to talk about in this class.