epoll: asynchronous I/O on Linux

Pierre-Marie de Rodat

December 8, 2011

Plan

1. Synchronous/asynchronous

2. epoll vs select and poll

3. Asynchronous usage for synchronous API: gevent

Synchronous I/O

- A system call for I/O blocks until something can be returned.
- Quite easy to use.
- But when you want to do more than a single I/O in your program, complexity greatly increases with threads, forks, etc.
- Traditional I/O programming style in Unix.

```
def serve(servsock):
    while run:
        clientsock = accept(servsock)
        handle_client(clientsock)
    close(servsock)
```

```
def handle_client(csock):
    buffer = []
    size = -1
    while not size:
        size = read(csock, buffer, 512);
        write(csock, buffer, size);
    close(csock)
```

Asynchronous I/O

- A system call returns immediately, sometimes returning that it couldn’t make the job.
- The system will send some kind of notification when completed.
- Induces *event programming*.

```
def serve(servsock):
    csocks = []
    register(servsock)
    while run and (event, sock = get_events()):
        if sock == servsock and event == NEW_CLIENT:
            csocks.add(accept(servsock))
        elif sock != servsock:
            if event == INPUT_DATA: ...
            elif event == OUTPUT_DATA: ...
            elif event == CLOSE: csocks.remove(sock)
```

Plan

1. Synchronous/asynchronous

2. epoll vs select and poll

3. Asynchronous usage for synchronous API: gevent

select and poll

```
int select(int nfds, fd_set *restrict readfds,
    fd_set *restrict writefds, fd_set *restrict errorfds,
    struct timeval *restrict timeout);

int poll(struct pollfd fds[], nfds_t nfds, int timeout);
```

- Both require the kernel to read the whole set of file descriptors at each loop iteration.
- It is worse for `select` since the program needs to build the file descriptor set at each iteration!
- Too long when dealing with *many* file descriptors.

The epoll API

```
int epoll_create(int size);
int epoll_ctl(
    int epfd, int op, int fd,
    struct epoll_event *event);
int epoll_wait(int epfd, struct epoll_event *events,
    int maxevents, int timeout);
```

- First, create an epoll object (returns a file descriptor).
- Register file descriptors to watch wait for events, handle them, register other fds, etc.
- Then close the epoll object.

Level-Triggered

- Default mode for registered events.
- Each time `epoll_wait` is called, the kernel yields events for ready file descriptors.
- If you do not handle one event, the next call to `epoll_wait` will return this event again.

Edge-Triggered

- A call to `read` or `write` should return `EAGAIN` *before* waiting for an event.
- Events that are not handled will not be yielded by the next call to `epoll_wait`.
- Before being returned, events are combined: when a file descriptor is available for reading, you shoud read it until it returns `EAGAIN`.
- So, there is a possibility of starvation due to small bugs.
- Should be (a little bit) more efficient, but your program should be well tested!

Plan

1. Synchronous/asynchronous

2. epoll vs select and poll

3. Asynchronous usage for synchronous API: gevent

Aim

- Take advantage of asynchronous I/O to handle many I/O in a single thread.
- Enable one to write synchronous-like code:
  - Code seems to run alone.
  - It looks simpler!

Example: gevent = coroutine + libev

- Simulating multiple threads can be done with coroutines.
- Switching between coroutines is done by syscalls.
- Syscalls looks blocking; `accept`, `read` and `write` actually are non-blocking and wrapped to do clever switching.
- Gevent is a Python library that implements this (it uses greenlet and libev).

```
def syscall(operation, fd, data):
    data, should_block = make_real_syscall(
        operation, fd, data)
    if should_block:
        current_coroutine.register(
            fd, operation)
        data = switch()
    return data

def serve(servsock):
    while run:
        clientsock = accept(servsock)
        Coroutine(handle_client, args=[clientsock, ])
    close(servsock)
```

```
def handle_client(csock):
    buffer = []
    size = -1
    while not size:
        size = read(csock, buffer, 512);
        write(csock, buffer, size);
    close(csock)
```