Advanced I/O

W4118 Operating Systems I

columbia-os.github.io

Credits to Jae

Nonblocking I/O

Two ways to make “slow” system calls nonblocking:

- call **`open()`** with **`O_NONBLOCK`**
- call **`fcntl()`** to turn on **`O_NONBLOCK`** file status flag
  - file status flag is part of the file table entry

Nonblocking slow system call returns -1 with errno set to EAGAIN if it would have blocked

Why do that?

Modern Nonblocking I/O: `io_uring`

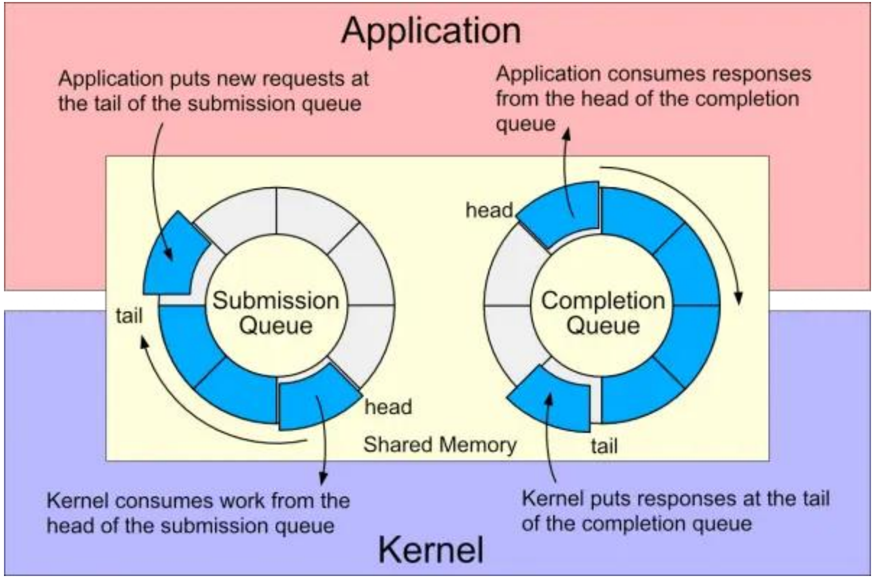
Polling for completions requires going into the kernel using a system call.

How can you avoid that?

Modern Nonblocking I/O: `io_uring`

Polling for completions requires going into the kernel using a system call.

How can you avoid that?


Application
Application puts new requests at the tail of the submission queue
Application consumes responses from the head of the completion queue
head
tail
Submission Queue
head
Completion Queue
Shared Memory
tail
Kernel consumes work from the head of the submission queue
Kernel puts responses at the tail of the completion queue
Kernel

Created by Donal Hunter

I/O Multiplexing

Network example: How can we monitor two connections simultaneously?

I/O Multiplexing

Network example: How can we monitor two connections simultaneously?

1. Nonblocking reads alternating between the two connections
2. Kernel I/O multiplexing

I/O Multiplexing

`select()` API for I/O multiplexing

```
#include <sys/select.h>

int select(int maxfdp1, // max fd plus 1, or simply pass FD_SETSIZE

           fd_set *restrict readfds,   // see if they're ready for reading
           fd_set *restrict writefds,  // see if they're ready for writing
           fd_set *restrict exceptfds, // see if exceptional condition occurred
                                       // ex) urgent out-of-band data in TCP

           struct timeval *restrict tvptr); // timeout

        // Returns: count of ready descriptors, 0 on timeout,-1 on error

int FD_ISSET(int fd, fd_set *fdset);

        // Returns: nonzero if fd is in set, 0 otherwise

void FD_CLR(int fd, fd_set *fdset);
void FD_SET(int fd, fd_set *fdset);
void FD_ZERO(fd_set *fdset);
```

Better I/O Multiplexing

`poll()` API for I/O multiplexing

```
#include <sys/select.h>

int poll(struct pollfd fds[], // the fds to monitor

           nfds_t nfds,   // number of fds to monitor
           int timeout)  // timeout in milliseconds
        // Returns: count of ready descriptors, 0 on timeout,-1 on error

struct pollfd {
      int fd;   // the fd to monitor of fds to monitor
      short events; // the events of interest, POLLIN for data to read, POLLOUT for for data to write
      short revents; // the events that actually occurred
}
```

Why is the `poll()` API considered better than select()?

What’s still a problem? `epoll()` to the rescue