Introduction to I/O

CSE 421/521: Operating Systems

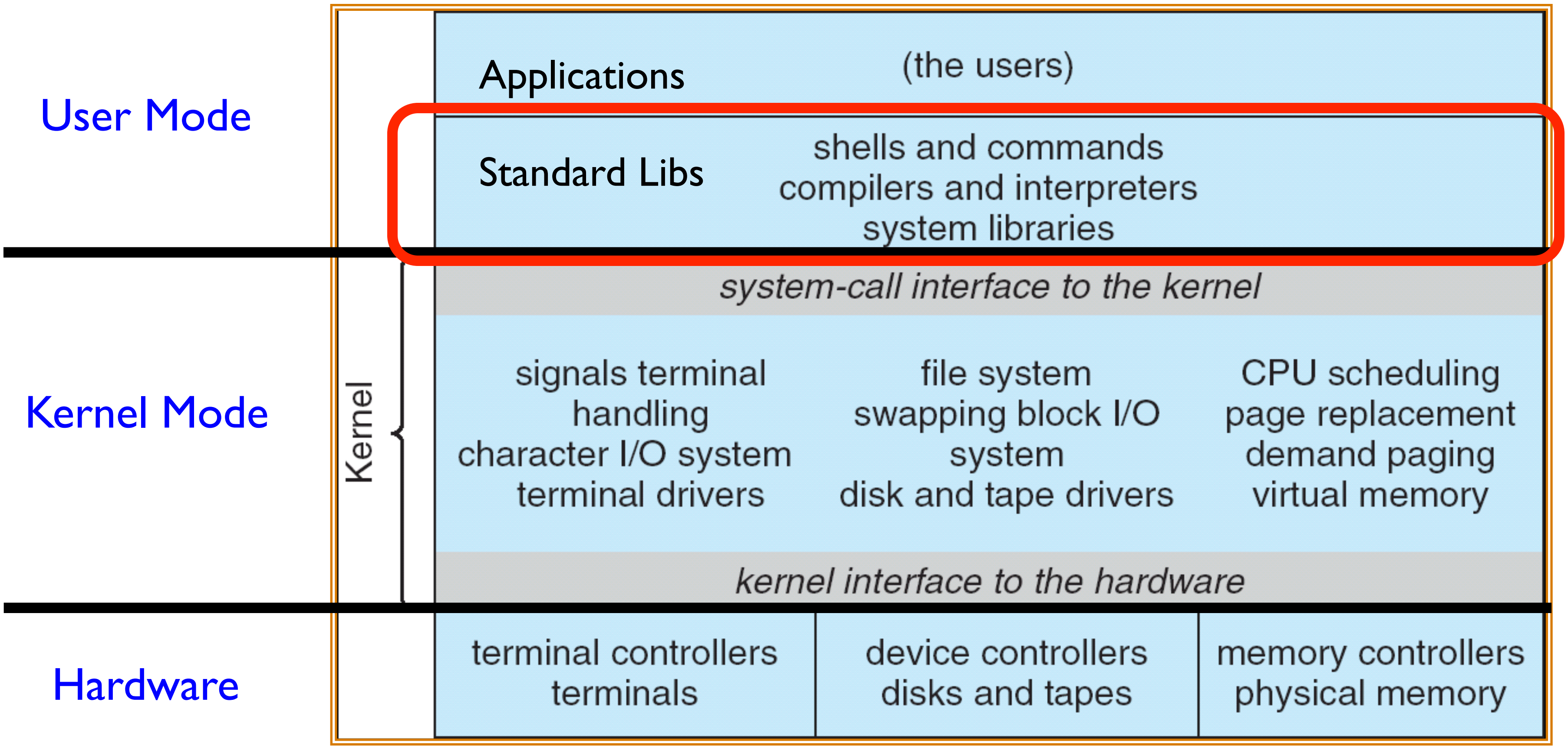
Karthik Dantu

Slides adopted from CS162 class at Berkeley, CSE 451 at U-Washington and CSE 421 by Prof Kosar at UB

Assignment #1 - Clarifications

- Grading
  - 5% - Design
  - 10% - Implementation/tests
- Git
  - Working with github to set up repos
  - Should be up by early next week
- Details of scheduling
  - In coming classes and recitations
  - Assignment given early
- Deadlines
  - 9/26 – Design document
  - 10/3 – Implementation via Autograder

Recall: UNIX System Structure

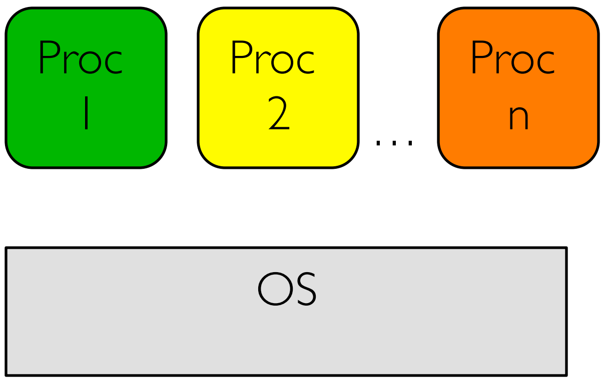


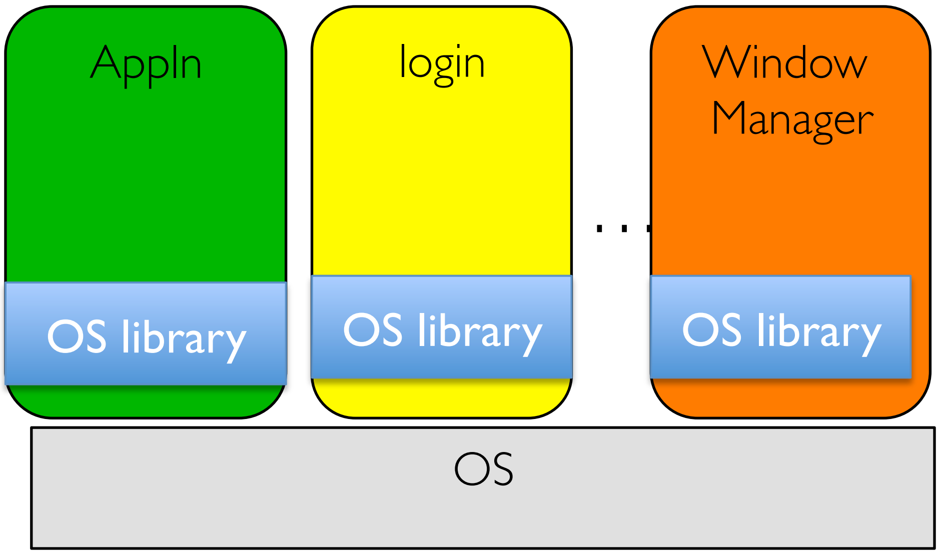
How Does the Kernel Provide Services?

- You said that applications request services from the operating system via `syscall`, but …
- I've been writing all sort of useful applications and I never ever saw a "`syscall`" !!!

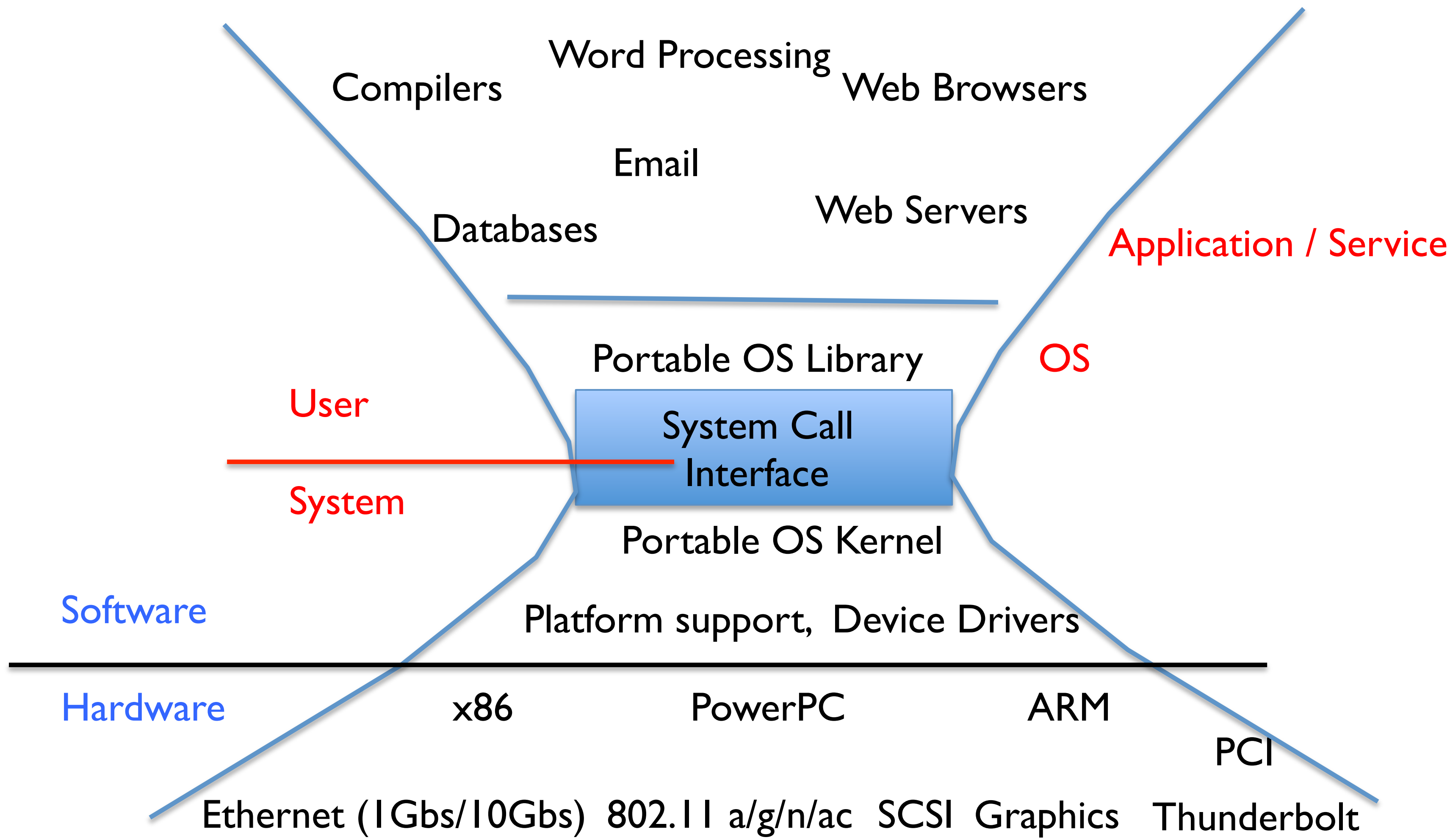
- That's right.
- It was buried in the programming language runtime library (e.g., `libc.a`)
- … Layering

OS Run-Time Library





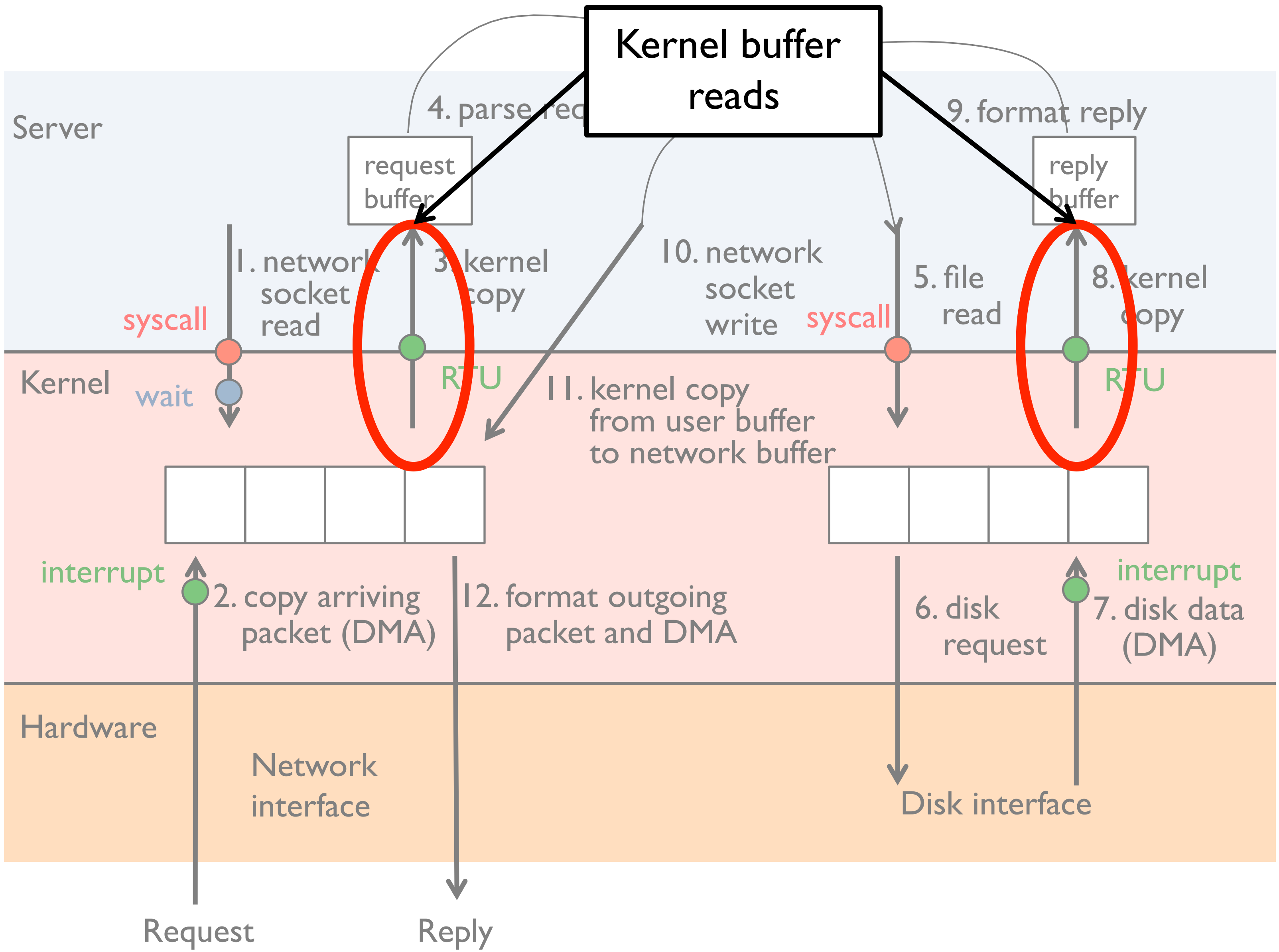
A Kind of Narrow Waist



Key Unix I/O Design Concepts

- Uniformity
  - file operations, device I/O, and interprocess communication through open, read/write, close
  - Allows simple composition of programs
    - `find | grep | wc …`
- Open before use
  - Provides opportunity for access control and arbitration
  - Sets up the underlying machinery, i.e., data structures
- Byte-oriented
  - Even if blocks are transferred, addressing is in bytes
- Kernel buffered reads
  - Streaming and block devices looks the same
  - read blocks process, yielding processor to other task

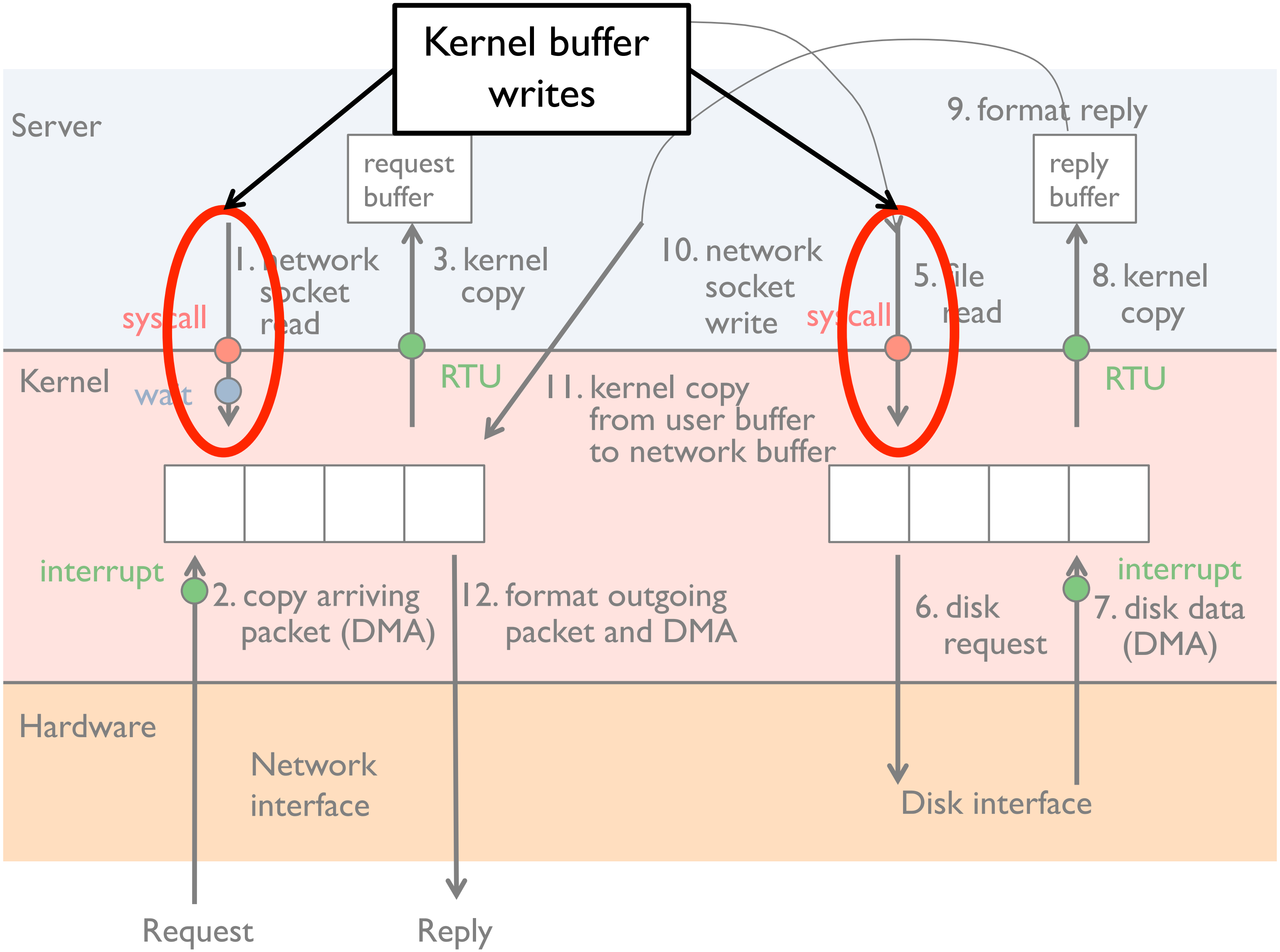
Putting it together: web server



Key Unix I/O Design Concepts

- Uniformity
  - file operations, device I/O, and interprocess communication through open, read/write, close
  - Allows simple composition of programs
    - `find | grep | wc …`
- Open before use
  - Provides opportunity for access control and arbitration
  - Sets up the underlying machinery, i.e., data structures
- Byte-oriented
  - Even if blocks are transferred, addressing is in bytes
- Kernel buffered reads
  - Streaming and block devices looks the same
  - read blocks process, yielding processor to other task
- Kernel buffered writes
  - Completion of out-going transfer decoupled from the application, allowing it to continue

Putting it together: web server



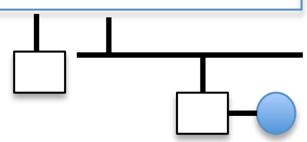
Key Unix I/O Design Concepts

- Uniformity
  - file operations, device I/O, and interprocess communication through open, read/write, close
  - Allows simple composition of programs
    - `find | grep | wc …`
- Open before use
  - Provides opportunity for access control and arbitration
  - Sets up the underlying machinery, i.e., data structures
- Byte-oriented
  - Even if blocks are transferred, addressing is in bytes
- Kernel buffered reads
  - Streaming and block devices looks the same
  - read blocks process, yielding processor to other task
- Kernel buffered writes
  - Completion of out-going transfer decoupled from the application, allowing it to continue
- Explicit close

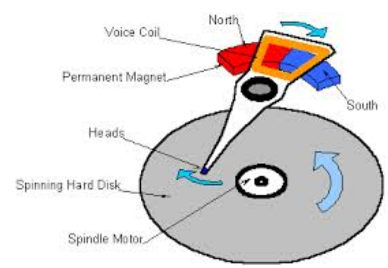
I/O & Storage Layers

Application / Service

Layer	
High Level I/O	streams
Low Level I/O	handles
Syscall	registers
File System	descriptors
I/O Driver	Commands and Data Transfers



Disks, Flash, Controllers, DMA

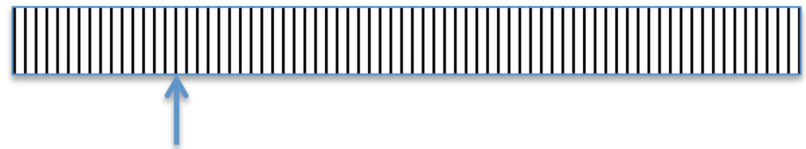


The File System Abstraction

- High-level idea
  - Files live in hierarchical namespace of filenames
- File
  - Named collection of data in a file system
  - File data
    - Text, binary, linearized objects
  - File Metadata: information about the file
    - Size, Modification Time, Owner, Security info
    - Basis for access control
- Directory
  - "Folder" containing files & Directories
  - Hierachical (graphical) naming
    - Path through the directory graph
    - Uniquely identifies a file or directory
      - `/home/ff/cs162/public_html/fa17/index.html`
  - Links and Volumes (later)

C High-Level File API – Streams (review)

- Operate on “streams” - sequence of bytes, whether text or data, with a position



```
#include <stdio.h>
FILE *fopen( const char *filename, const char *mode );
int fclose( FILE *fp );
```

Mode Text	Binary	Descriptions
r	rb	Open existing file for reading
w	wb	Open for writing; created if does not exist
a	ab	Open for appending; created if does not exist
r+	rb+	Open existing file for reading & writing.
w+	wb+	Open for reading & writing; truncated to zero if exists, create otherwise
a+	ab+	Open for reading & writing. Created if does not exist. Read from beginning, write as append

Don’t forget to flush

Connecting Processes, Filesystem, and Users

- Process has a ‘current working directory’
- Absolute Paths
  - `/home/kdantu/cs421`
- Relative paths
  - `index.html`, `./index.html` - current WD
  - `../index.html` - parent of current WD
  - `~`, `~kdantu` - home directory

C API Standard Streams

- Three predefined streams are opened implicitly when a program is executed
  - `FILE *stdin` – normal source of input, can be redirected
  - `FILE *stdout` – normal source of output, can be redirected
  - `FILE *stderr` – diagnostics and errors, can be redirected

- `STDIN` / `STDOUT` enable composition in Unix
  - Recall: Use of pipe symbols connects `STDOUT` and `STDIN`
    - `find | grep | wc …`

C high level File API – Stream Ops

```
#include <stdio.h>
// character oriented
int fputc( int c, FILE *fp );              // rtn c or EOF on err
int fputs( const char *s, FILE *fp );   // rtn >0 or EOF

int fgetc( FILE * fp );
char *fgets( char *buf, int n, FILE *fp );
```

```
DESCRIPTION
     The fgets() function reads at most one less than the number of characters
     specified by size from the given stream and stores them in the string
     str.  Reading stops when a newline character is found, at end-of-file or
     error.  The newline, if any, is retained.  If any characters are read and
     there is no error, a `\0' character is appended to end the string.
```

C high level File API – Stream Ops

```
#include <stdio.h>
// character oriented
int fputc( int c, FILE *fp );          // rtn c or EOF on err
int fputs( const char *s, FILE *fp );  // rtn >0 or EOF

int fgetc( FILE * fp );
char *fgets( char *buf, int n, FILE *fp );

// block oriented
size_t fread(void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);

size_t fwrite(const void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);
```

C high level File API – Stream Ops

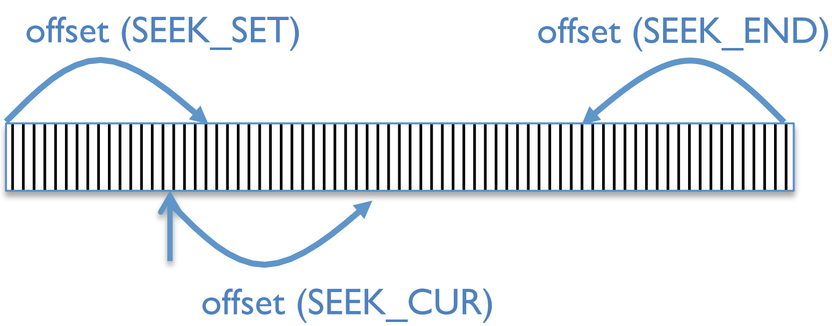
```
#include <stdio.h>
// character oriented
int fputc( int c, FILE *fp );    // rtn c or EOF on err
int fputs( const char *s, FILE *fp );   // rtn >0 or EOF

int fgetc( FILE * fp );
char *fgets( char *buf, int n, FILE *fp );

// block oriented
size_t fread(void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);

size_t fwrite(const void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);

// formatted
int fprintf(FILE *restrict stream, const char *restrict format,
       ...);
int fscanf(FILE *restrict stream, const char *restrict format,
       ...);
```

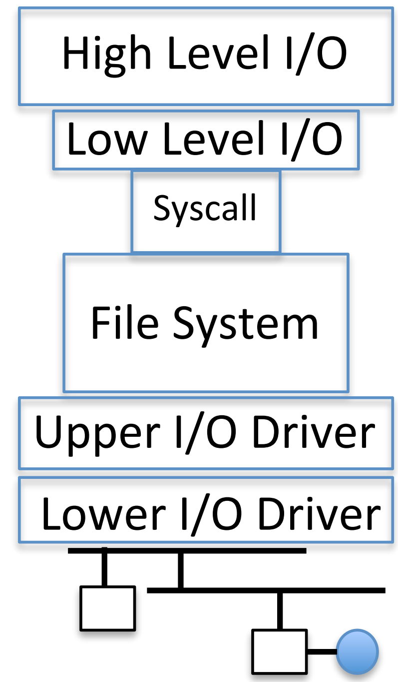
Example Code

```
#include <stdio.h>

#define BUFLEN 256
FILE *outfile;
char mybuf[BUFLEN];

int storetofile() {
  char *instring;

  outfile = fopen("/usr/homes/testing/tokens", "w+");
  if (!outfile)
    return (-1);    // Error!
  while (1) {
    instring = fgets(mybuf, BUFLEN, stdin); // catches overrun!

    // Check for error or end of file (^D)
    if (!instring || strlen(instring)==0) break;

    // Write string to output file, exit on error
    if (fputs(instring, outfile)< 0) break;
  }
  fclose(outfile);  // Flushes from userspace
}
```

C Stream API positioning

```
int fseek(FILE *stream, long int offset, int whence);
long int ftell (FILE *stream)
void rewind (FILE *stream)
```





- Preserves high level abstraction of uniform stream of objects
- Adds buffering for performance

What’s below the surface ??

Application / Service

High Level I/O — streams

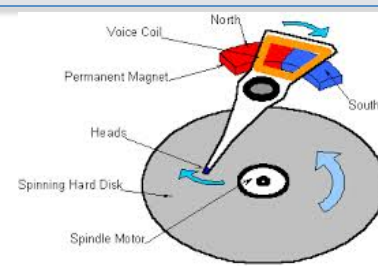
Low Level I/O — handles

Syscall — registers

File System — descriptors

I/O Driver — Commands and Data Transfers

Disks, Flash, Controllers, DMA



C Low level I/O

- Operations on File Descriptors – as OS object representing the state of a file
  - User has a “handle” on the descriptor

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>

int open (const char *filename, int flags [, mode_t mode])
int creat (const char *filename, mode_t mode)
int close (int filedes)
```

Bit vector of:

- Access modes (Rd, Wr, …)
- Open Flags (Create, …)
- Operating modes (Appends, …)

Bit vector of Permission Bits:

- User|Group|Other X R|W|X

http://www.gnu.org/software/libc/manual/html_node/Opening-and-Closing-Files.html

C Low Level: standard descriptors

```
#include <unistd.h>

STDIN_FILENO -  macro has value 0
STDOUT_FILENO - macro has value 1
STDERR_FILENO - macro has value 2

int fileno (FILE *stream)

FILE * fdopen (int filedes, const char *opentype)
```

- Crossing levels: File descriptors vs. streams
- Don’t mix them!

C Low Level Operations

```
ssize_t read (int filedes, void *buffer, size_t maxsize)
 - returns bytes read, 0 => EOF, -1 => error
ssize_t write (int filedes, const void *buffer, size_t size)
 - returns bytes written

off_t lseek (int filedes, off_t offset, int whence)

int fsync (int fildes) – wait for i/o to finish
void sync (void) – wait for ALL to finish
```

- When write returns, data is on its way to disk and can be read, but it may not actually be permanent!

And lots more !

- TTYs versus files
- Memory mapped files
- File Locking
- Asynchronous I/O
- Generic I/O Control Operations
- Duplicating descriptors

```
int dup2 (int old, int new)
int dup (int old)
```

Another example: lowio-std.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

#define BUFSIZE 1024

int main(int argc, char *argv[])
{
  char buf[BUFSIZE];
  ssize_t writelen = write(STDOUT_FILENO, "I am a process.\n", 16);

  ssize_t readlen  = read(STDIN_FILENO, buf, BUFSIZE);

  ssize_t strlen   = snprintf(buf, BUFSIZE,"Got %zd chars\n", readlen);

  writelen = strlen < BUFSIZE ? strlen : BUFSIZE;
  write(STDOUT_FILENO, buf, writelen);

  exit(0);
}
```

What’s below the surface ??

Application / Service

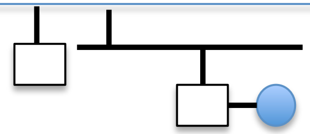
High Level I/O

Low Level I/O

Syscall

File System

I/O Driver



streams

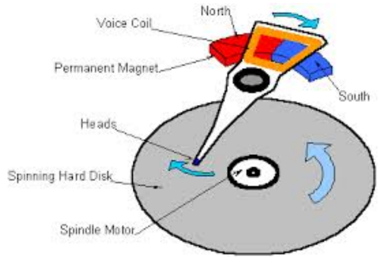
handles

registers

descriptors

Commands and Data Transfers

Disks, Flash, Controllers, DMA



Recall: SYSCALL

syscalls.kernelgrok.com

BCal UCB CS162 cullermayeno Wikipedia Yahoo! News Popular Imported From Safari

Linux Syscall Reference

Show 10 entries Search:

#	Name	Registers						Definition
		eax	ebx	ecx	edx	esi	edi	
0	sys_restart_syscall	0x00	-	-	-	-	-	kernel/signal.c:2058
1	sys_exit	0x01	int error_code	-	-	-	-	kernel/exit.c:1046
2	sys_fork	0x02	struct pt_regs *	-	-	-	-	arch/alpha/kernel/entry.S:716
3	sys_read	0x03	unsigned int fd	char __user *buf	size_t count	-	-	fs/read_write.c:391
4	sys_write	0x04	unsigned int fd	const char __user *buf	size_t count	-	-	fs/read_write.c:408
5	sys_open	0x05	const char __user *filename	int flags	int mode	-	-	fs/open.c:900
6	sys_close	0x06	unsigned int fd	-	-	-	-	fs/open.c:969
7	sys_waitpid	0x07	pid_t pid	int __user *stat_addr	int options	-	-	kernel/exit.c:1771
8	sys_creat	0x08	const char __user *pathname	int mode	-	-	-	fs/open.c:933
9	sys_link	0x09	const char __user *oldname	const char __user *newname	-	-	-	fs/namei.c:2520

Showing 1 to 10 of 338 entries

First Previous 1 2 3 4 5 Next Last

Generated from Linux kernel 2.6.35.4 using Exuberant Ctags, Python, and DataTables.
Project on GitHub. Hosted on GitHub Pages.

- Low level lib parameters are set up in registers and syscall instruction is issued
  - A type of synchronous exception that enters well-defined entry points into kernel

What's below the surface ??

File descriptor number
 - an int

File Descriptors
 - a struct with all the info about the files

Application / Service

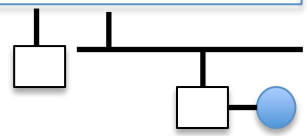
High Level I/O

Low Level I/O

Syscall

File System

I/O Driver



streams

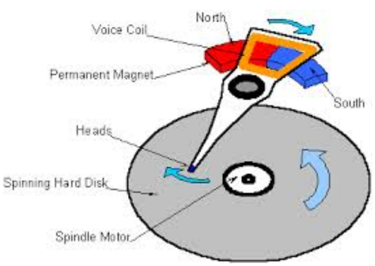
handles

registers

descriptors

Commands and Data Transfers

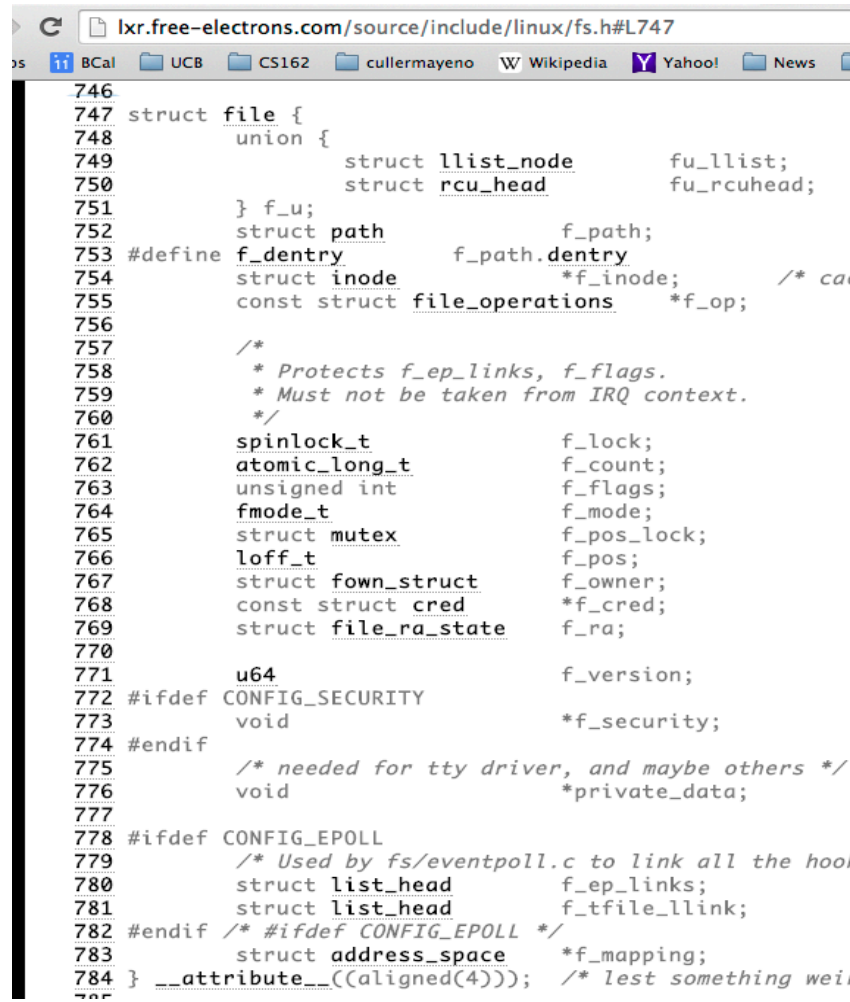
Disks, Flash, Controllers, DMA



Internal OS File Descriptor

- Internal Data Structure describing everything about the file
  - Where it resides
  - Its status
  - How to access it

- Pointer:
  `struct file *file`




```
struct file {
        union {
                struct llist_node       fu_llist;
                struct rcu_head         fu_rcuhead;
        } f_u;
        struct path             f_path;
#define f_dentry        f_path.dentry
        struct inode            *f_inode;       /* cacl
        const struct file_operations    *f_op;

        /*
         * Protects f_ep_links, f_flags.
         * Must not be taken from IRQ context.
         */
        spinlock_t              f_lock;
        atomic_long_t           f_count;
        unsigned int            f_flags;
        fmode_t                 f_mode;
        struct mutex            f_pos_lock;
        loff_t                  f_pos;
        struct fown_struct      f_owner;
        const struct cred       *f_cred;
        struct file_ra_state    f_ra;

        u64                     f_version;
#ifdef CONFIG_SECURITY
        void                    *f_security;
#endif
        /* needed for tty driver, and maybe others */
        void                    *private_data;

#ifdef CONFIG_EPOLL
        /* Used by fs/eventpoll.c to link all the hook
        struct list_head        f_ep_links;
        struct list_head        f_tfile_llink;
#endif /* #ifdef CONFIG_EPOLL */
        struct address_space    *f_mapping;
} __attribute__((aligned(4)));  /* lest something weir
```

File System: from syscall to driver

In fs/read_write.c

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
  ssize_t ret;
  if (!(file->f_mode & FMODE_READ)) return -EBADF;
  if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
    return -EINVAL;
  if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
  ret = rw_verify_area(READ, file, pos, count);
  if (ret >= 0) {
    count = ret;
    if (file->f_op->read)
      ret = file->f_op->read(file, buf, count, pos);
    else
      ret = do_sync_read(file, buf, count, pos);
    if (ret > 0) {
      fsnotify_access(file->f_path.dentry);
      add_rchar(current, ret);
    }
    inc_syscr(current);
  }
  return ret;
}
```

Lower Level Driver

- Associated with particular hardware device
- Registers / Unregisters itself with the kernel
- Handler functions for each of the file operations

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*flock) (struct file *, int, struct file_lock *);
    [...]
};
```

Recall: Device Drivers

- Device Driver: Device-specific code in the kernel that interacts directly with the device hardware
  - Supports a standard, internal interface
  - Same kernel I/O system can interact easily with different device drivers
  - Special device-specific configuration supported with the `ioctl()` system call
- Device Drivers typically divided into two pieces:
  - Top half: accessed in call path from system calls
    - implements a set of standard, cross-device calls like `open()`, `close()`, `read()`, `write()`, `ioctl()`, `strategy()`
    - This is the kernel's interface to the device driver
    - Top half will *start* I/O to device, may put thread to sleep until finished
  - Bottom half: run as interrupt routine
    - Gets input or transfers next block of output
    - May wake sleeping threads if I/O now complete

Life Cycle of An I/O Request

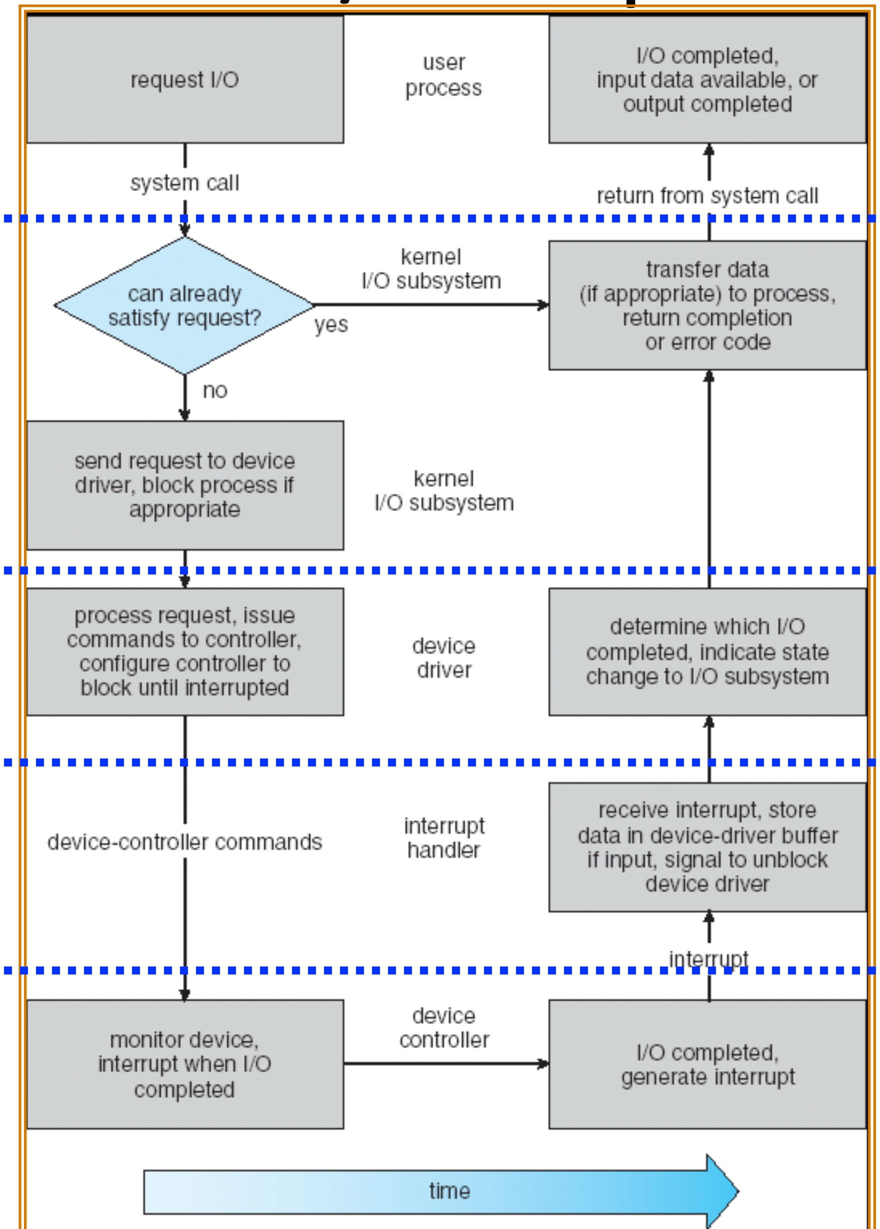
User Program

Kernel I/O Subsystem

Device Driver Top Half

Device Driver Bottom Half

Device Hardware



So what happens when you fgetc?

Application / Service

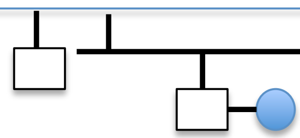
High Level I/O — streams

Low Level I/O — handles

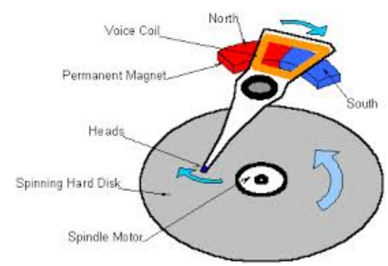
Syscall — registers

File System — descriptors

I/O Driver — Commands and Data Transfers



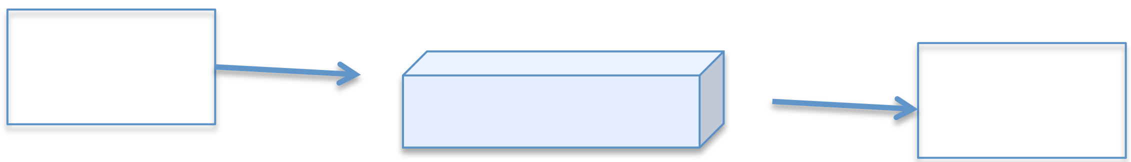
Disks, Flash, Controllers, DMA



Communication between processes

- Can we view files as communication channels?

```
write(wfd, wbuf, wlen);
```

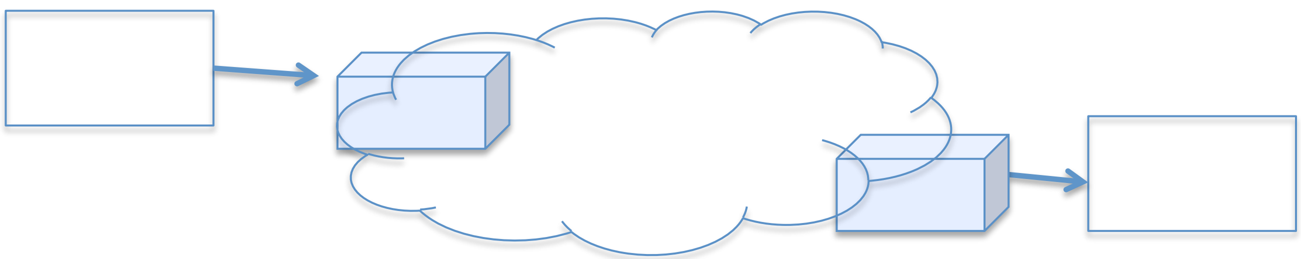


```
n = read(rfd,rbuf,rmax);
```

- Producer and Consumer of a file may be distinct processes
  - May be separated in time (or not)
- However, what if data written once and consumed once?
  - Don't we want something more like a queue?
  - Can still look like File I/O!

Communication Across the world looks like file IO

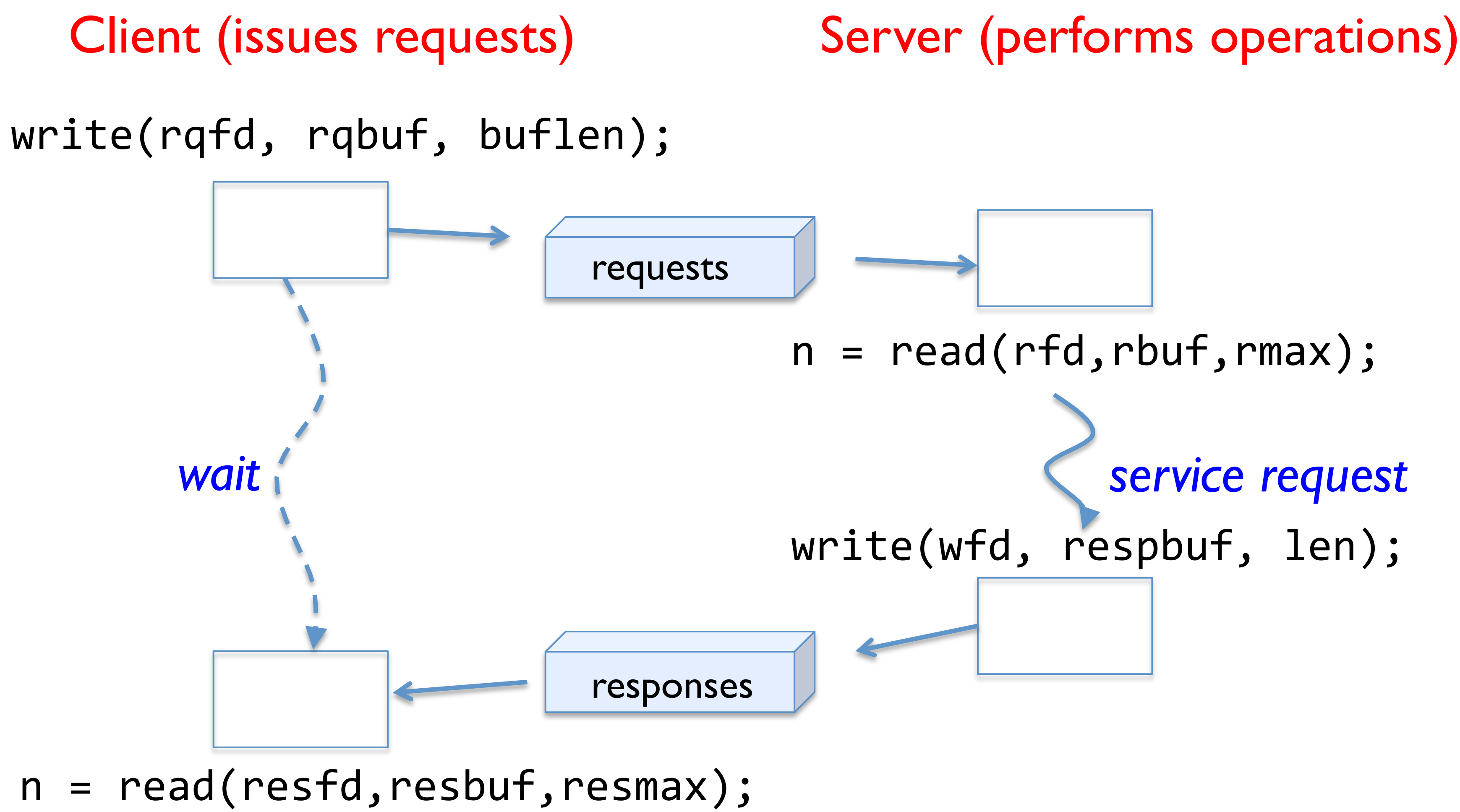
```
write(wfd, wbuf, wlen);
```



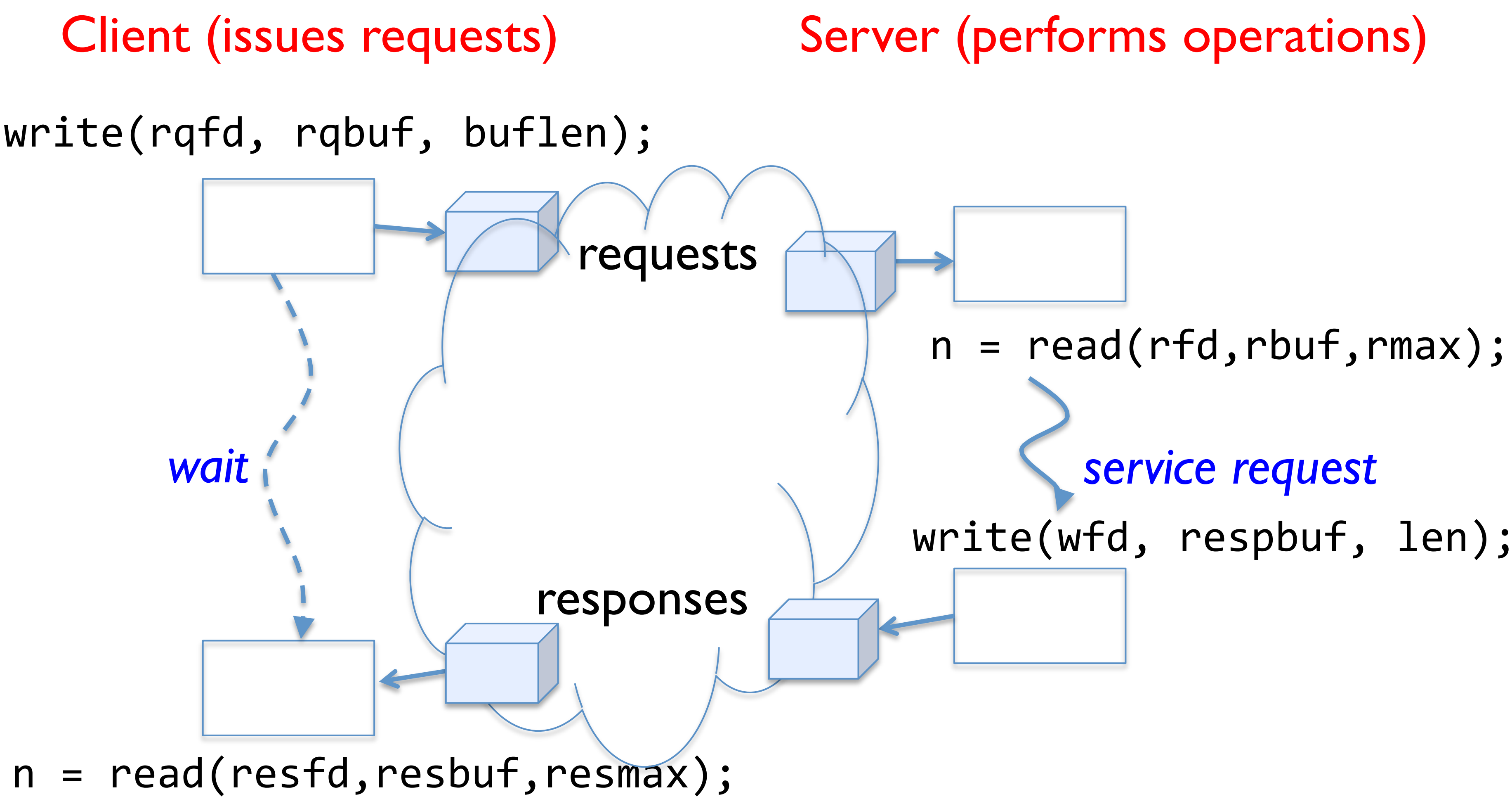
```
n = read(rfd,rbuf,rmax);
```

- Connected queues over the Internet
  - But what's the analog of open?
  - What is the namespace?
  - How are they connected in time?

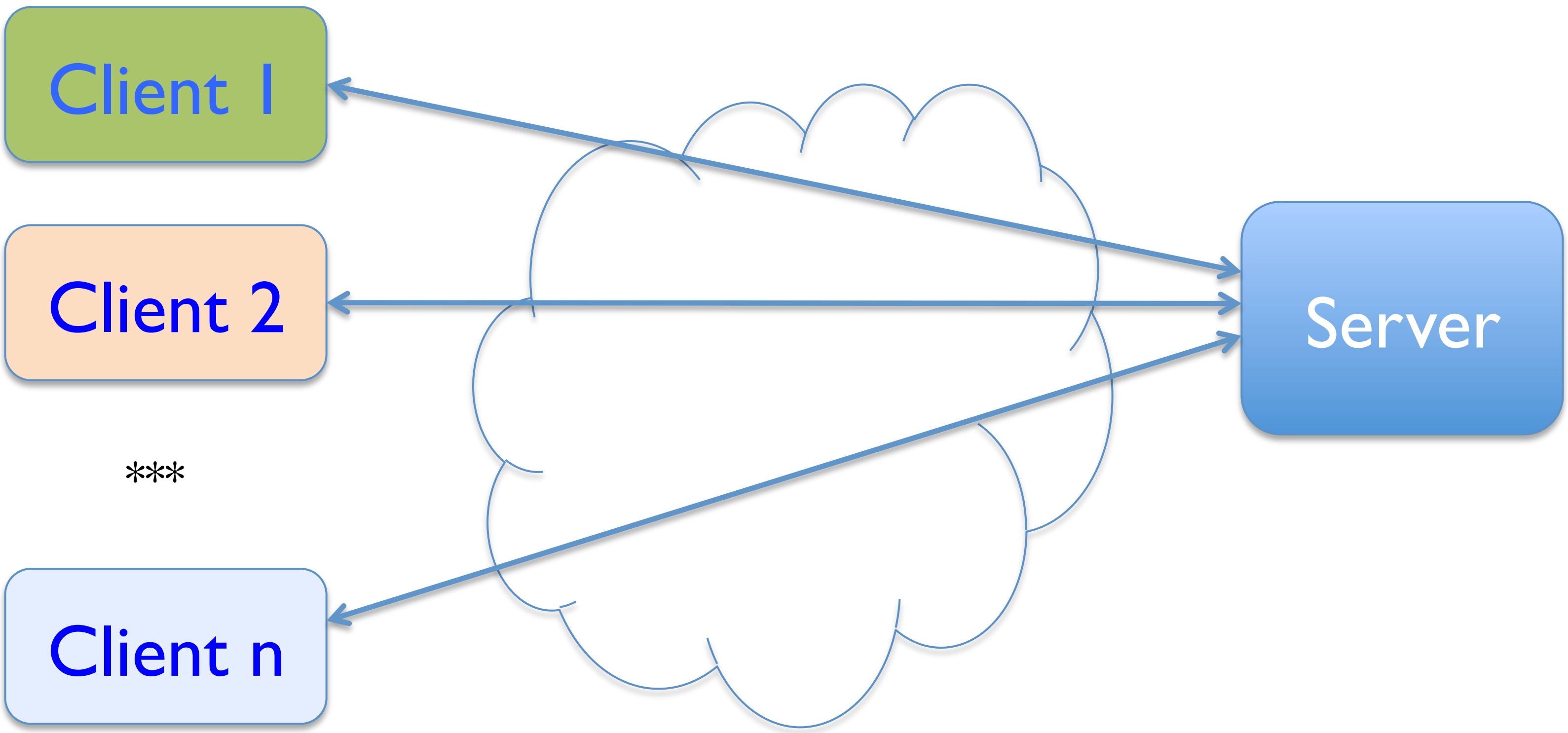
Request Response Protocol



Request Response Protocol



Client-Server Models



- File servers, web, FTP, Databases, …
- Many clients accessing a common server

Conclusion (I)

- System Call Interface is "narrow waist" between user programs and kernel

- Streaming IO: modeled as a stream of bytes
  - Most streaming I/O functions start with "f" (like "`fread`")
  - Data buffered automatically by C-library functions

- Low-level I/O:
  - File descriptors are integers
  - Low-level I/O supported directly at system call level

- `STDIN` / `STDOUT` enable composition in Unix
  - Use of pipe symbols connects `STDOUT` and `STDIN`
    - `find | grep | wc …`

Conclusion (II)

- Device Driver: Device-specific code in the kernel that interacts directly with the device hardware
  - Supports a standard, internal interface
  - Same kernel I/O system can interact easily with different device drivers

- File abstraction works for inter-processes communication (local or Internet)