CS162
Operating Systems and Systems Programming
Lecture 4
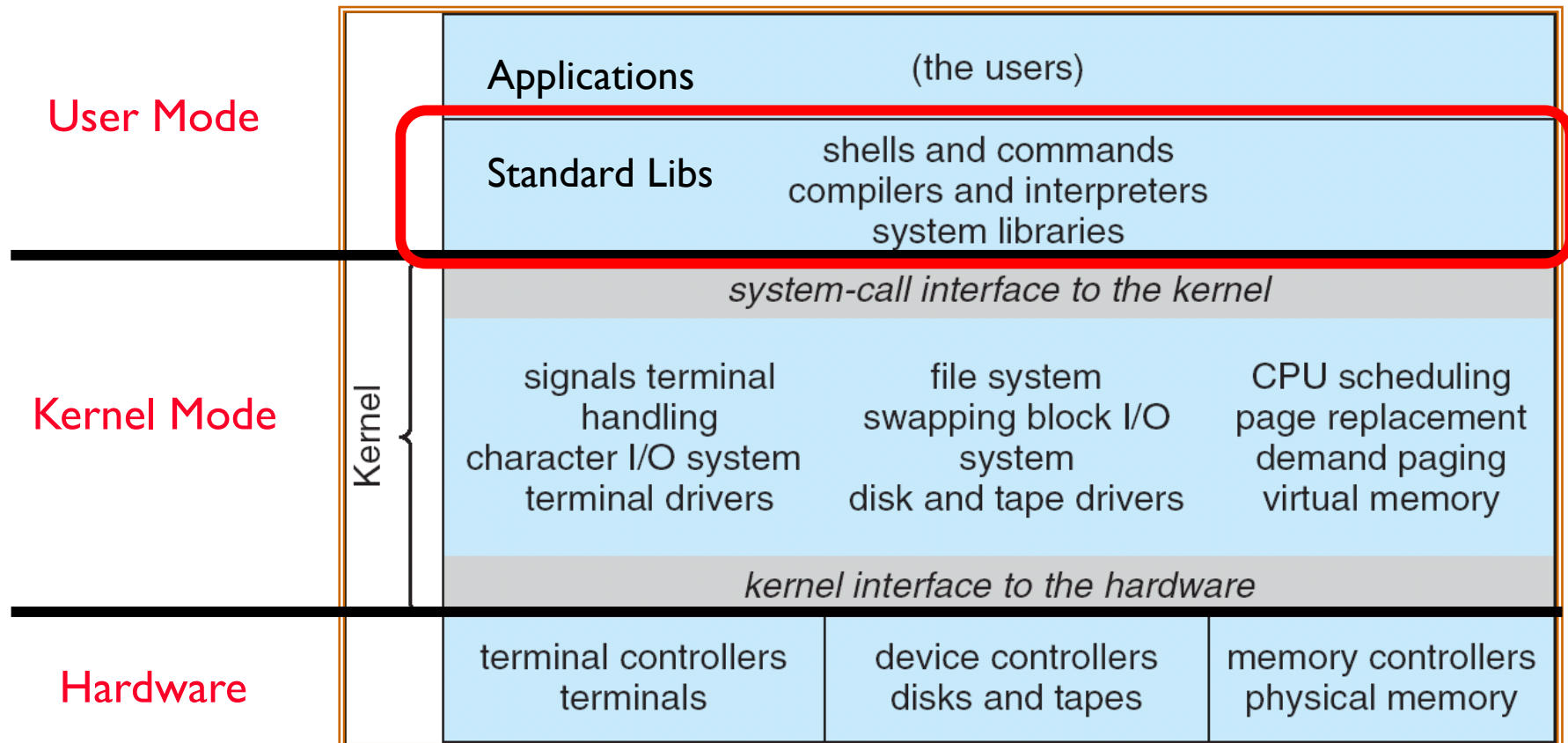
Introduction to I/O, Sockets, Networking

September 5th, 2018

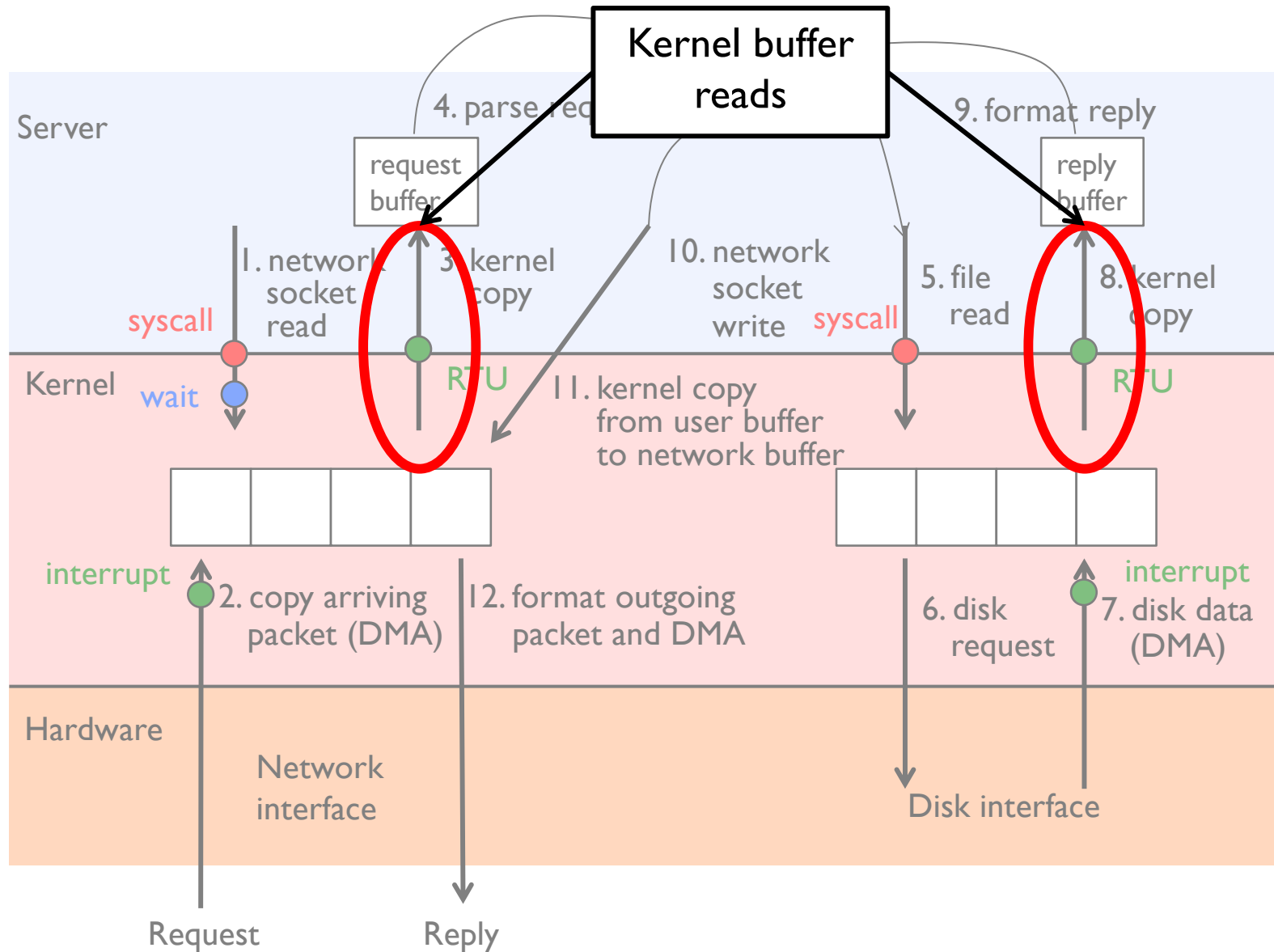Prof. Ion Stoica

http://cs162.eecs.Berkeley.edu

Recall: UNIX System Structure

User Mode	**Applications** (the users)
	Standard Libs shells and commands compilers and interpreters system libraries

system-call interface to the kernel

Kernel Mode	signals terminal handling character I/O system terminal drivers	file system swapping block I/O system disk and tape drivers	CPU scheduling page replacement demand paging virtual memory

kernel interface to the hardware

Hardware terminal controllers terminals	device controllers disks and tapes	memory controllers physical memory

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
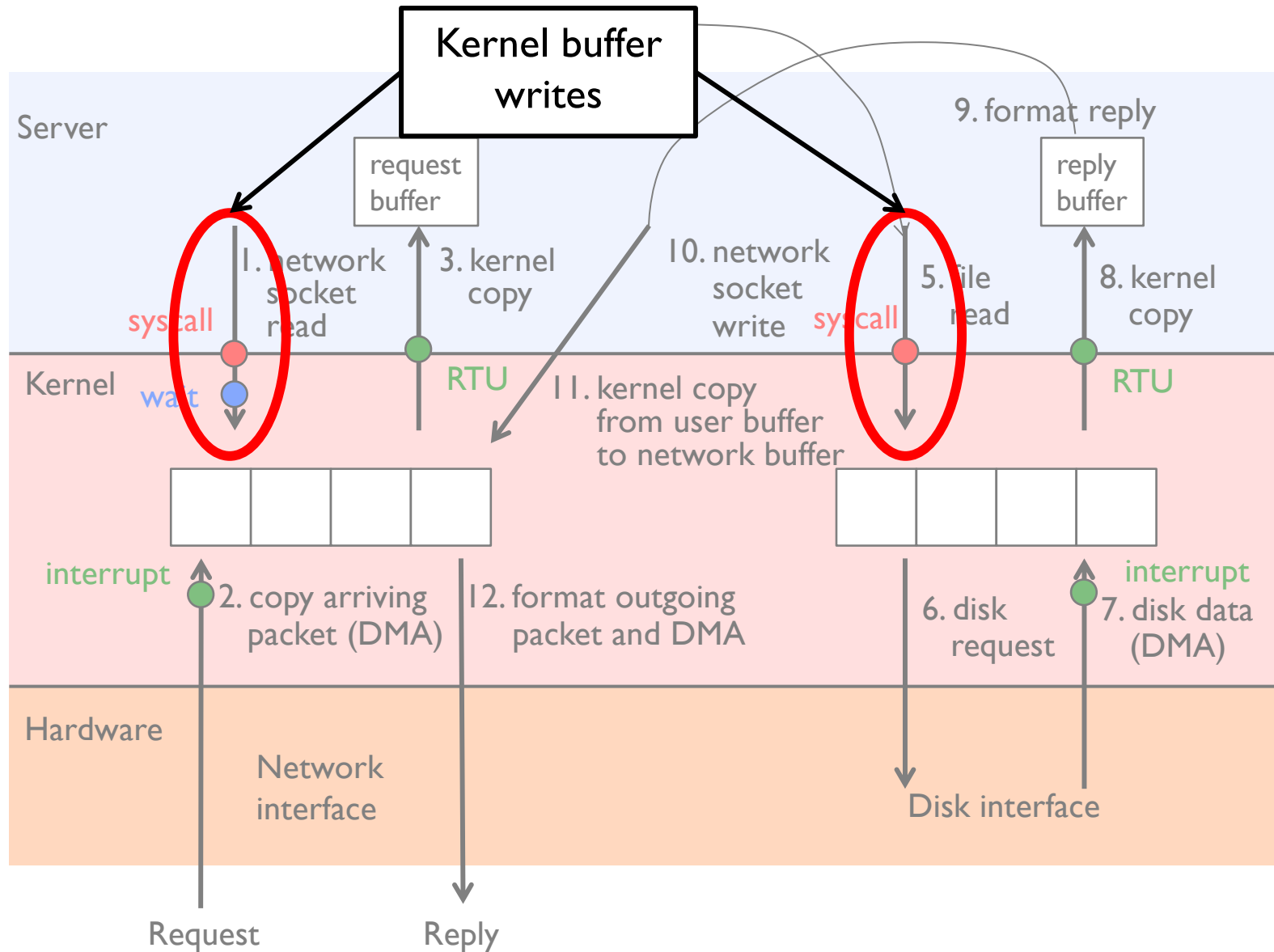 - read blocks process, yielding processor to other task

Putting it together: web server

Server

4. parse req

request buffer

Kernel buffer reads

9. format reply

reply buffer

syscall

1. network socket read

3. kernel copy

10. network socket write

syscall

5. file read

8. kernel copy

Kernel

wait

RTU

11. kernel copy from user buffer to network buffer

RTU

interrupt

2. copy arriving packet (DMA)

12. format outgoing packet and DMA

interrupt

6. disk request

7. disk data (DMA)

Hardware

Network interface

Disk interface

Request

Reply

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

Kernel buffer writes

Server

request buffer

9. format reply

reply buffer

1. network socket read

3. kernel copy

10. network socket write

5. file read

8. kernel copy

syscall

syscall

Kernel

wait

RTU

11. kernel copy from user buffer to network buffer

RTU

interrupt

2. copy arriving packet (DMA)

12. format outgoing packet and DMA

6. disk request

interrupt

7. disk data (DMA)

Hardware

Network interface

Disk interface

Request

Reply

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
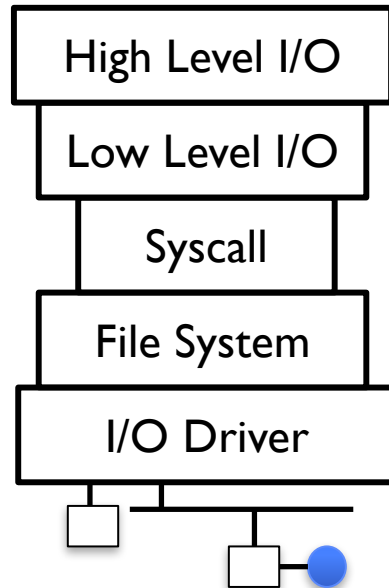
High Level I/O — streams

Low Level I/O — handles

Syscall — registers

File System — descriptors

I/O Driver — Commands and Data Transfers

Disks, Flash, Controllers, DMA
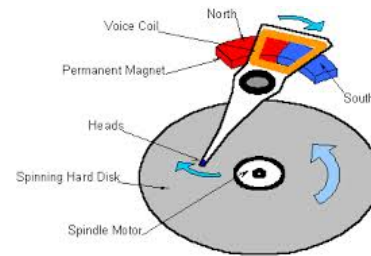
The File System Abstraction

- 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/fa18/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/ff/cs162`

- Relative paths
    - `index.html`, `./index.html`   - current WD
    - `../index.html`   - parent of current WD
    - `~`, `~cs162`   - 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 <u>size</u> from the given <u>stream</u> and stores them in the string <u>str</u>. 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
}
```
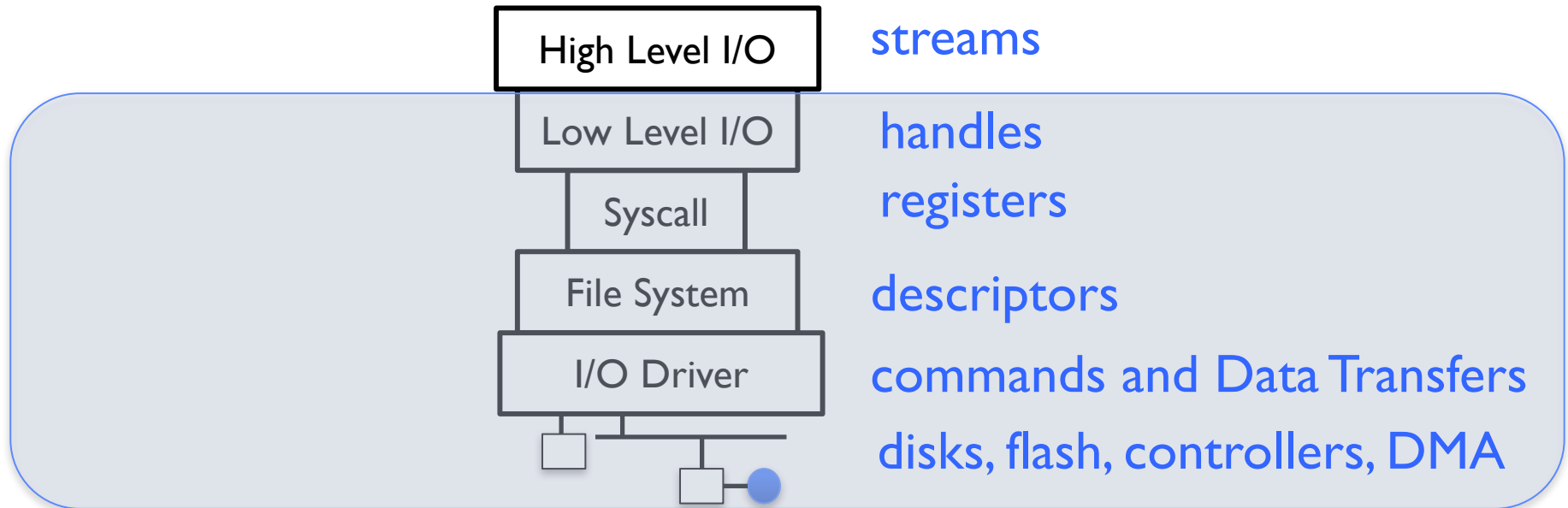
C Stream API positioning

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

offset (SEEK_SET)   offset (SEEK_END)

whence

offset (SEEK_CUR)

High Level I/O

Low Level I/O

Syscall

File System

I/O Driver
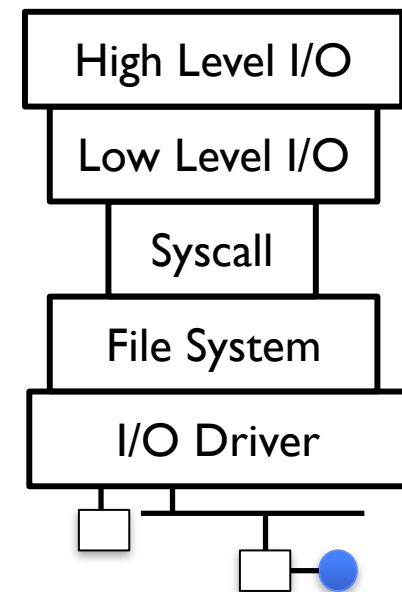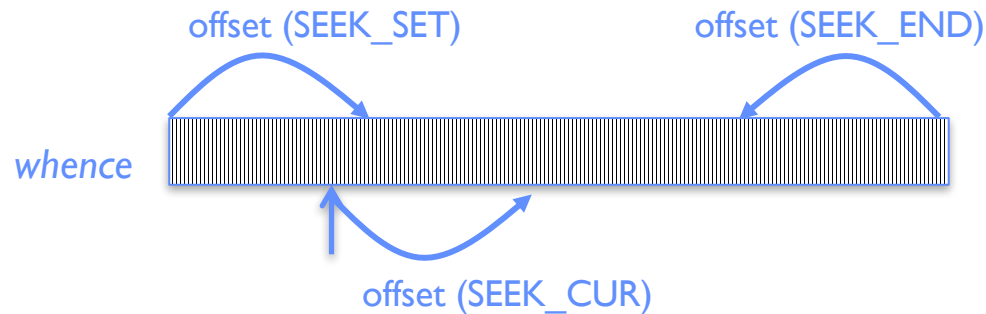
- Preserves high level abstraction of uniform stream of objects
- Adds buffering for performance

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
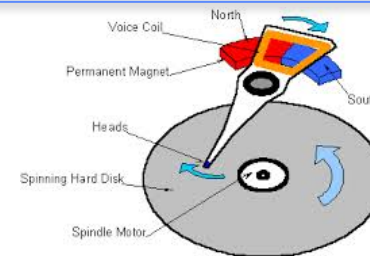
C Low level I/O

- 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

```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);
}
```

Administrivia

- Waitlist was closed last Friday

- Recommendation: Read assigned readings *before* lecture

- Group sign up this week
  - Get finding groups ASAP – deadline Friday 9/7 at 11:59PM
  - 4 people in a group!

- TA *preference* signup form due Monday 9/10 at 11:59PM
  - Everyone in a group must have the same TA!
    » Preference given to same section
  - Participation: Get to know your TA!

BREAK
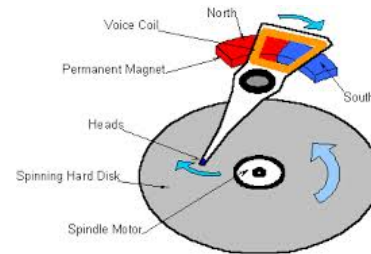
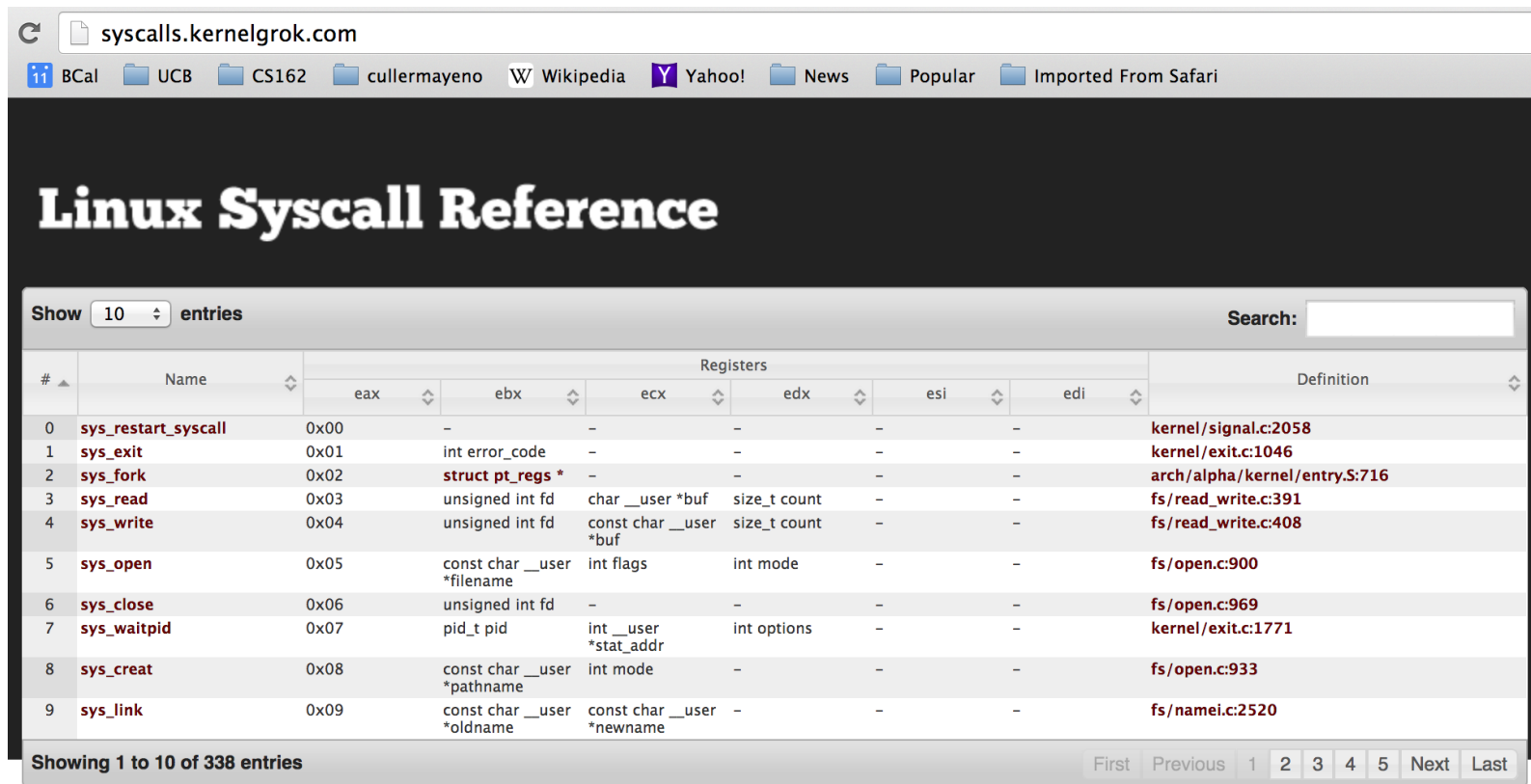# What's below the surface ??
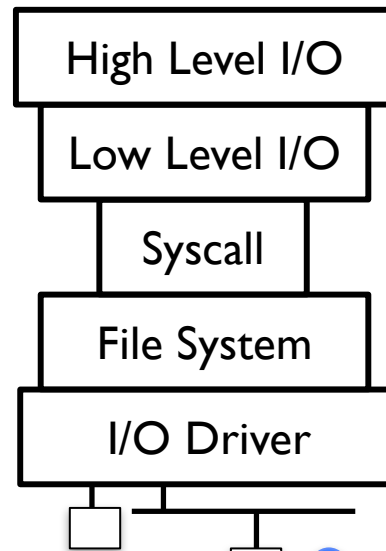
Application / Service

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

Voice Coil
North
Permanent Magnet
South
Heads
Spinning Hard Disk
Spindle Motor

Recall: SYSCALL

Linux Syscall Reference

# ▲	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

Application / Service

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

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

Disks, Flash, Controllers, DMA
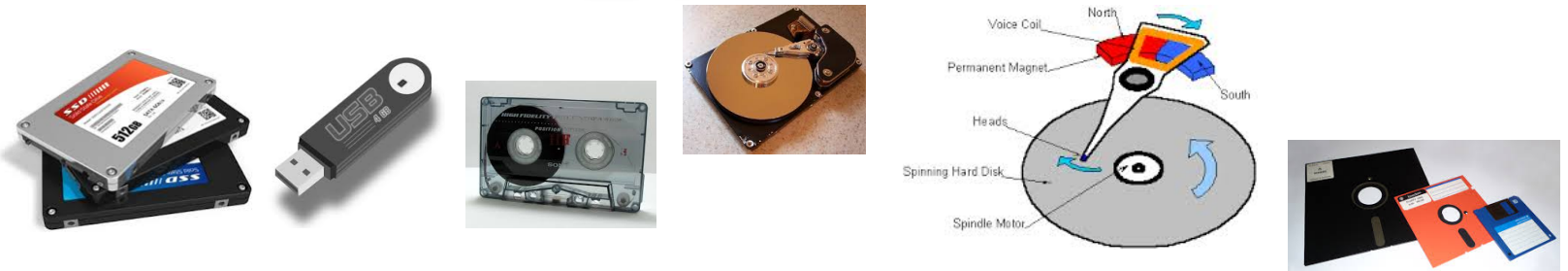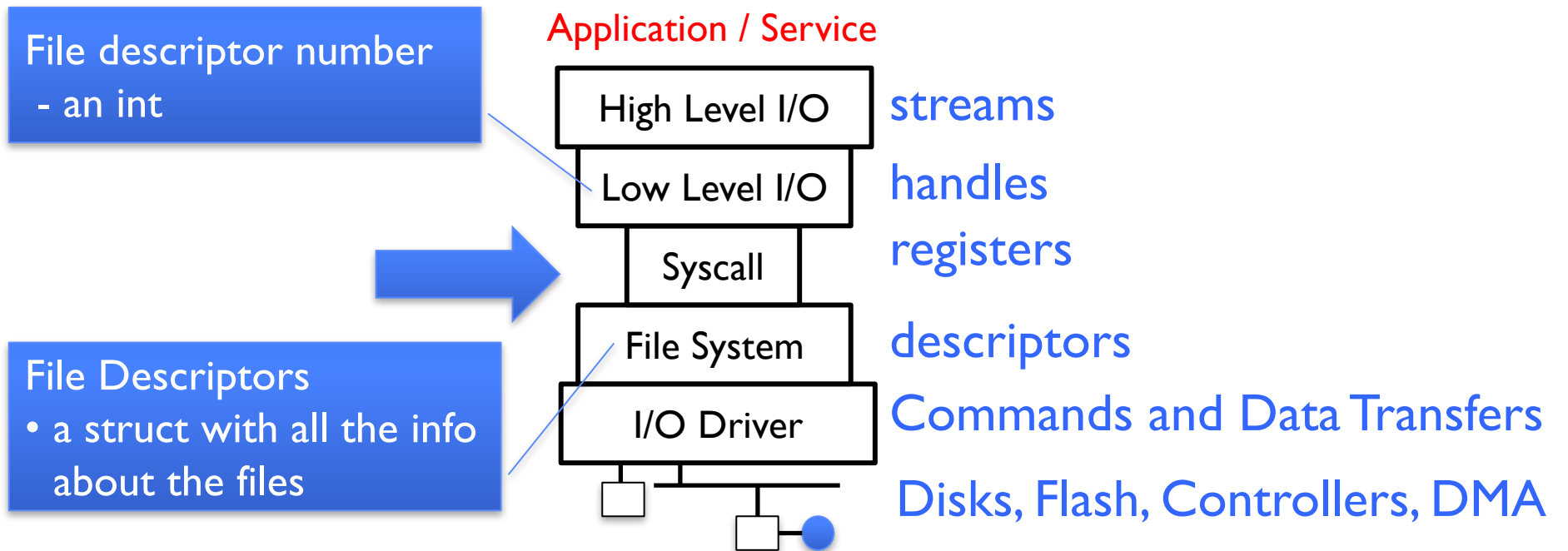
Internal OS File Descriptor

- Internal Data Structure describing everything about the file

  – Where it resides

  – Its status

  – How to access it

- Pointer:
  struct file *file

```
  746
  747  struct file {
  748          union {
  749                          struct llist_node       fu_llist;
  750                          struct rcu_head         fu_rcuhead;
  751          } f_u;
  752          struct path                     f_path;
  753  #define f_dentry        f_path.dentry
  754          struct inode                    *f_inode;       /* cacl
  755          const struct file_operations    *f_op;
  756
  757          /*
  758           * Protects f_ep_links, f_flags.
  759           * Must not be taken from IRQ context.
  760           */
  761          spinlock_t                      f_lock;
  762          atomic_long_t                   f_count;
  763          unsigned int                    f_flags;
  764          fmode_t                         f_mode;
  765          struct mutex                    f_pos_lock;
  766          loff_t                          f_pos;
  767          struct fown_struct              f_owner;
  768          const struct cred               *f_cred;
  769          struct file_ra_state            f_ra;
  770
  771          u64                             f_version;
  772  #ifdef CONFIG_SECURITY
  773          void                            *f_security;
  774  #endif
  775          /* needed for tty driver, and maybe others */
  776          void                            *private_data;
  777
  778  #ifdef CONFIG_EPOLL
  779          /* Used by fs/eventpoll.c to link all the hook:
  780          struct list_head                f_ep_links;
  781          struct list_head                f_tfile_llink;
  782  #endif /* #ifdef CONFIG_EPOLL */
  783          struct address_space            *f_mapping;
  784  } __attribute__((aligned(4)));   /* lest something weir
```

File System: from syscall to driver

In `fs/read_write.c`

```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
  if (!file->f_op || (!file->f_op->read &&
    return -EINVAL;
  if (unlikely(!access_ok(VERIFY_WRITE, bu
  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;
}
```

- Read up to "count" bytes from "file" starting from "pos" into "buf".
- Return error or number of bytes red.

File System: from syscall to driver

In `fs/read_write.c`

```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))) ret
  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;
}
```

Make sure we are allowed to read this file

File System: from syscall to driver

In `fs/read_write.c`

```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;
}
```

Check if file has read methods

File System: from syscall to driver

In fs/read_write.c

```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, coun
    else
      ret = do_sync_read(file, buf, count, p
    if (ret > 0) {
      fsnotify_access(file->f_path.dentry);
      add_rchar(current, ret);
    }
    inc_syscr(current);
  }
  return ret;
}
```

- Check whether we can write to buf (e.g., buf is in the user space range)
- unlikely(): hint to branch prediction this condition is unlikely

File System: from syscall to driver

In `fs/read_write.c`

```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, po
    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;
}
```

Check whether we read from a valid range in the file.

File System: from syscall to driver

In `fs/read_write.c`

```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;
}
```

If driver provide a read function (f_op->read) use it; otherwise use do_sync_read()

File System: from syscall to driver

In `fs/read_write.c`

```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;
}
```

Notify the parent of this file that the file was red (see http://www.fieldses.org/~bfields/kernel/vfs.txt)

File System: from syscall to driver

In `fs/read_write.c`

```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, po
    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;
}
```

Update the number of bytes red by "current" task (for scheduling purposes)

File System: from syscall to driver

In fs/read_write.c

```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;
}
```

Update the number of read syscalls by "current" task (for scheduling purposes)
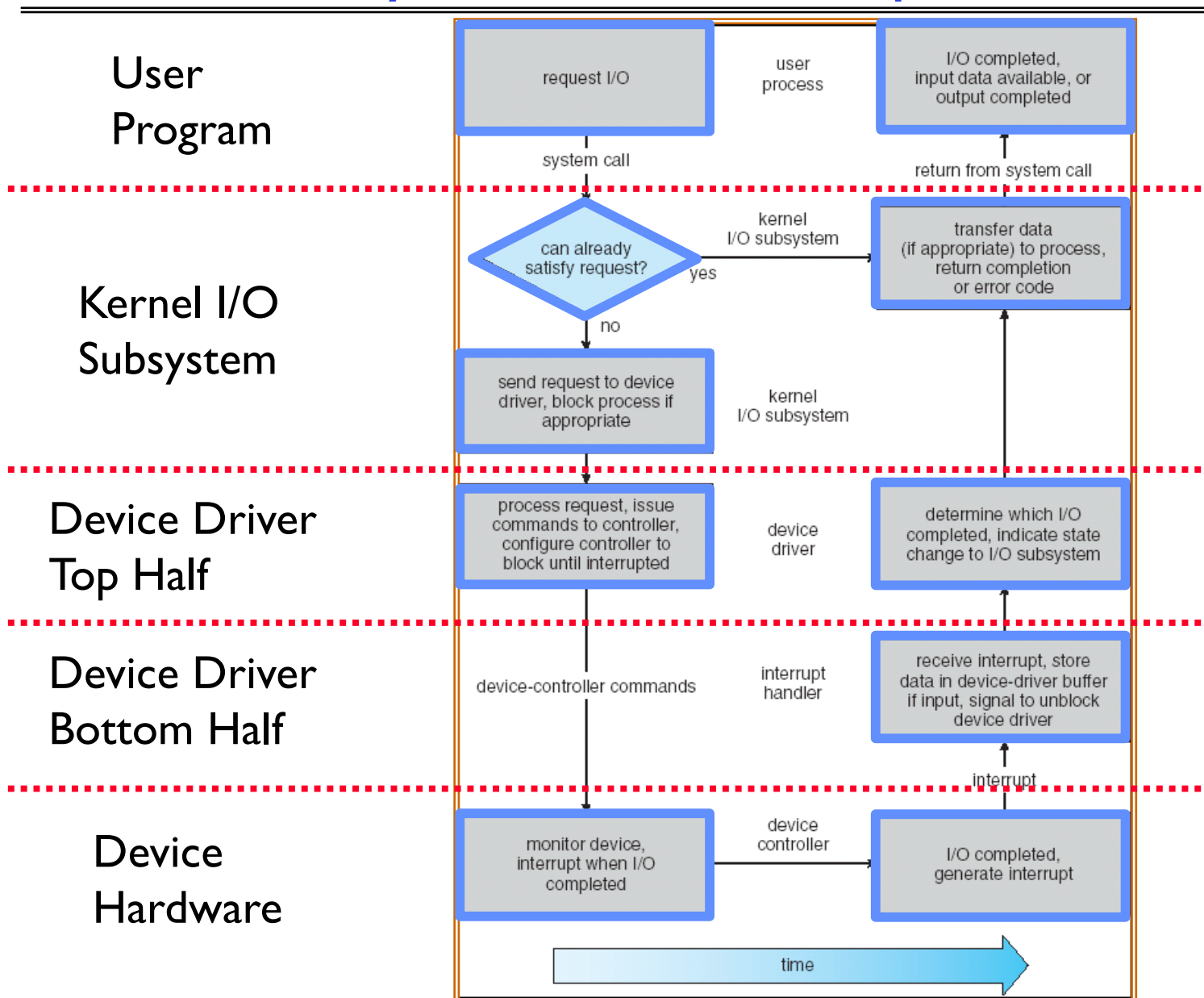
Lower Level Driver

- 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 *);
    [...]
};
```

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	request I/O — user process — I/O completed, input data available, or output completed	

User Program

Kernel I/O Subsystem

Device Driver Top Half

Device Driver Bottom Half

Device Hardware

system call — can already satisfy request? — kernel I/O subsystem — yes — transfer data (if appropriate) to process, return completion or error code — return from system call

no — send request to device driver, block process if appropriate — kernel I/O subsystem

process request, issue commands to controller, configure controller to block until interrupted — device driver — determine which I/O completed, indicate state change to I/O subsystem

device-controller commands — interrupt handler — receive interrupt, store data in device-driver buffer if input, signal to unblock device driver

monitor device, interrupt when I/O completed — device controller — I/O completed, generate interrupt — interrupt

time

Communication between processes

- Can we view files as communication channels?
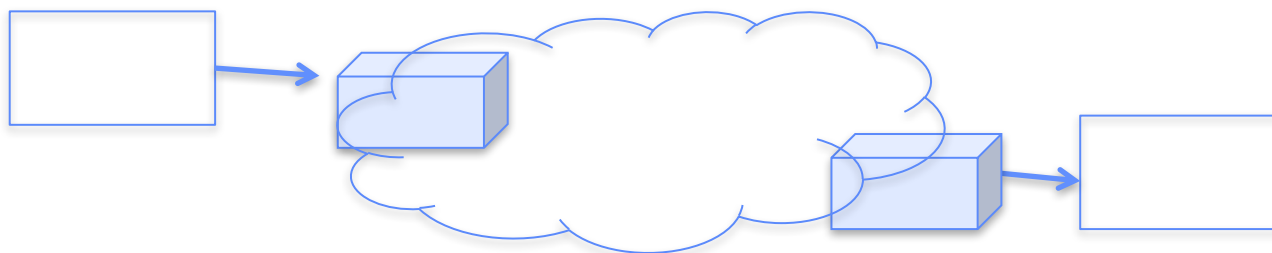
```
write(wfd, wbuf, wlen);
```
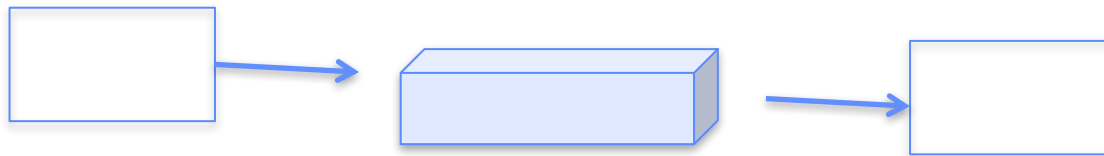


```
n = read(rfd,rbuf,rmax);
```

- 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);`

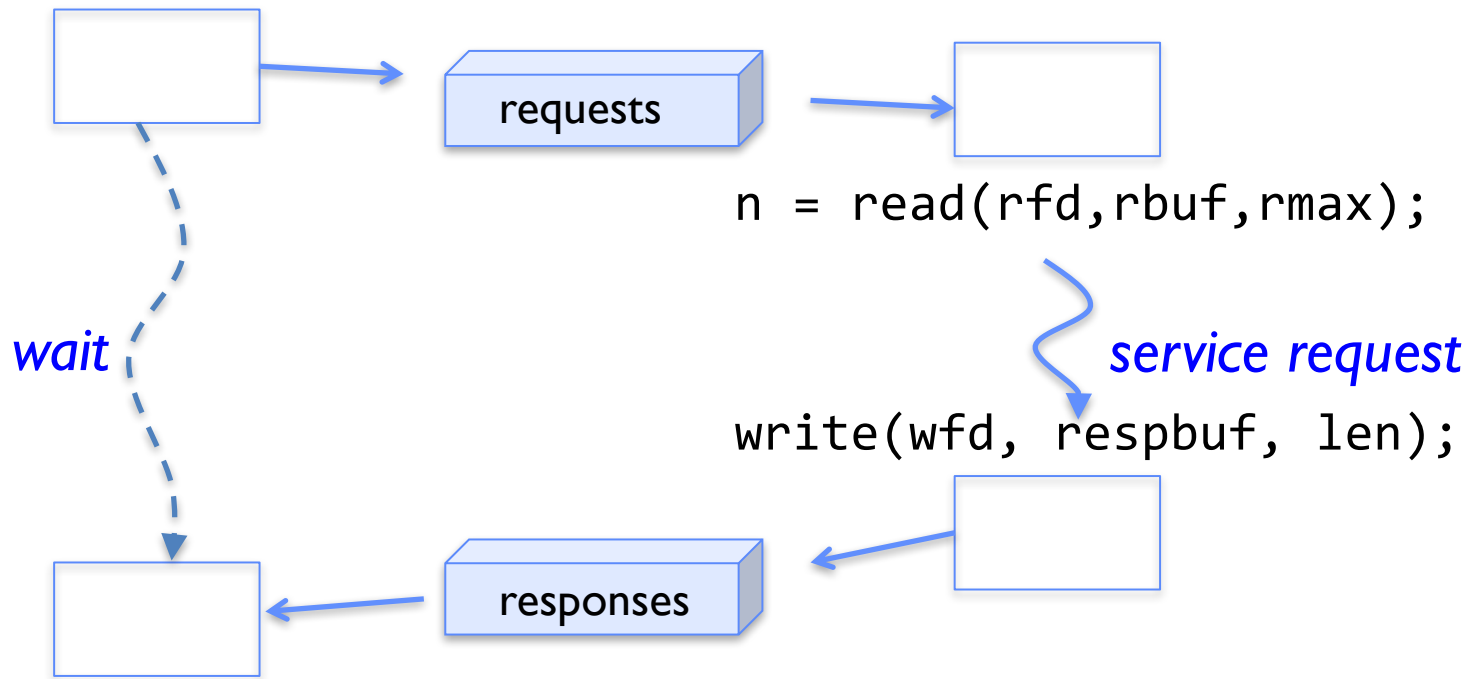`n = read(rfd,rbuf,rmax);`

- 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

Client (issues requests)   **Server (performs operations)**

`write(rqfd, rqbuf, buflen);`

requests

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

service request
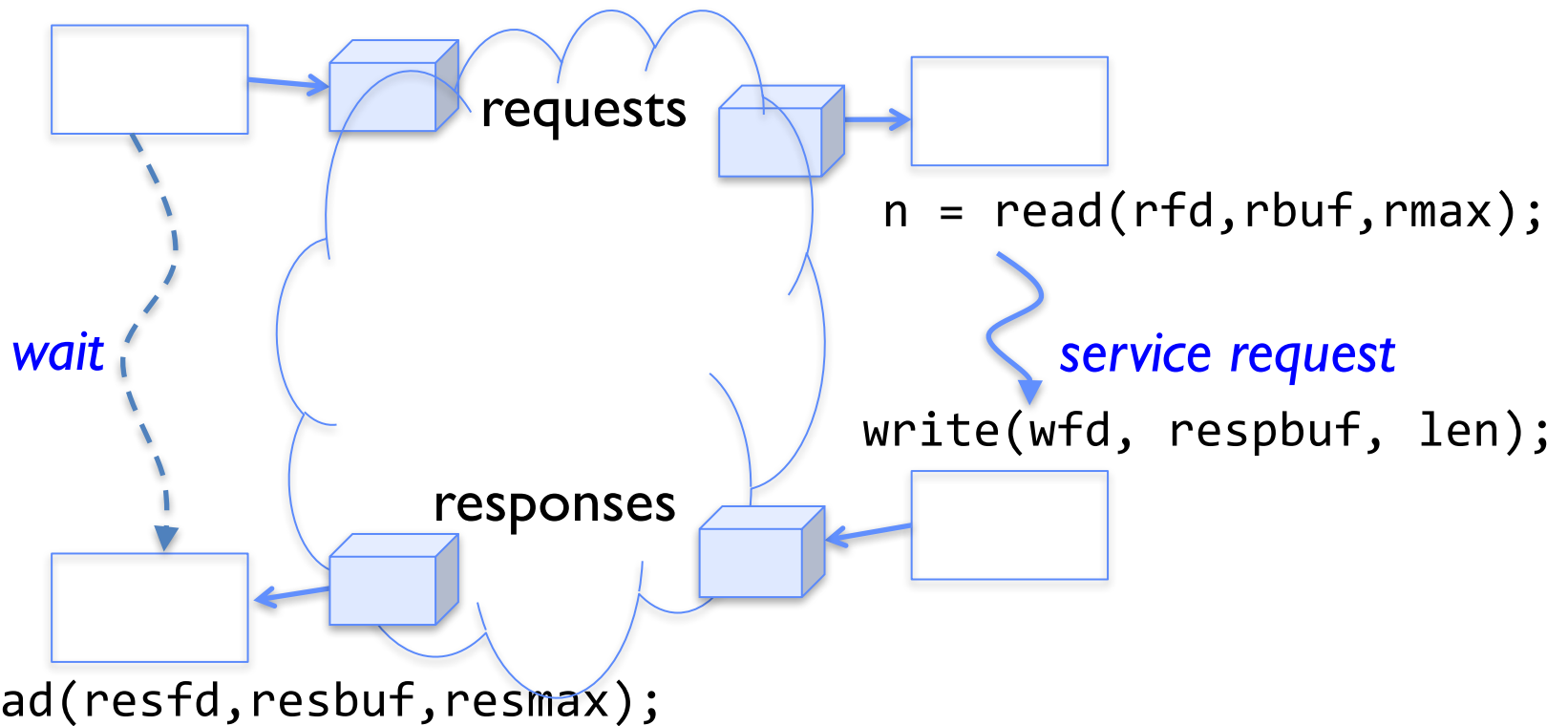
`write(wfd, respbuf, len);`

wait

responses

`n = read(resfd,resbuf,resmax);`

Request Response Protocol

Client (issues requests)   Server (performs operations)

`write(rqfd, rqbuf, buflen);`
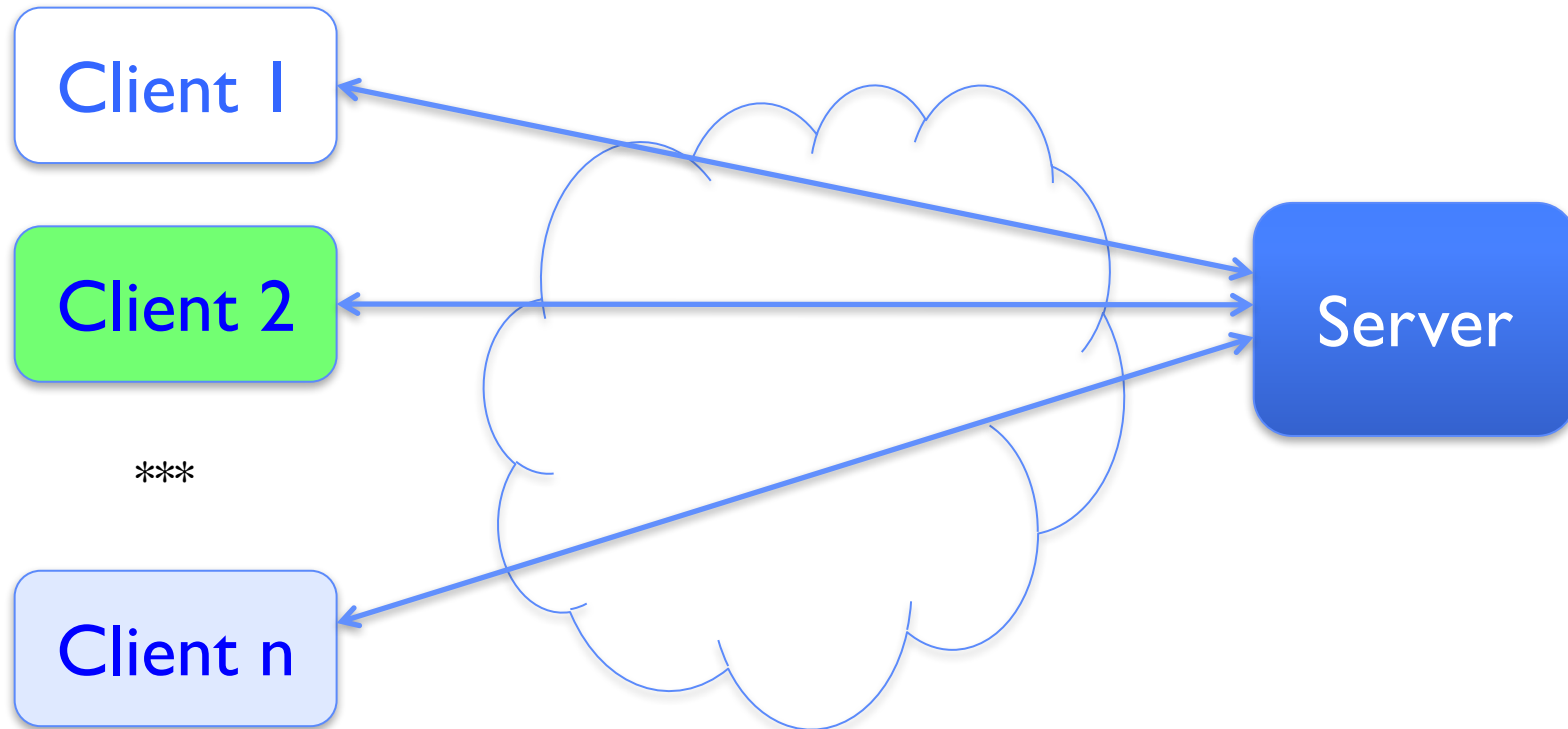
requests

wait

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

service request

`write(wfd, respbuf, len);`

responses

`n = read(resfd,resbuf,resmax);`

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)