Acknowledgments: Lecture slides are from the Operating Systems course taught by John Kubiatowicz at Berkeley, with few minor updates/changes. When slides are obtained from other sources, a reference will be noted on the bottom of that slide, in which case a full list of references is provided on the last slide.
Recall: UNIX System Structure

User Mode

- Applications (the users)
  - Standard Libs
    - shells and commands
    - compilers and interpreters
    - system libraries

Kernel Mode

- Kernel
  - system-call interface to the kernel
    - 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

Hardware

- terminal controllers
  - terminals
- device controllers
  - disks and tapes
- memory controllers
  - physical memory
Recall: A Kind of Narrow Waist

Compilers
Word Processing
Web Browsers
Email
Web Servers
Databases
Email
Word Processing
Portable OS Library
Portable OS Kernel
System Call Interface
Platform support, Device Drivers
Application / Service

User
System
Software
Hardware
x86
PowerPC
ARM
PCI
Ethernet (10/100/1000)
802.11 a/b/g/n
SCSI IDE
Graphics

x86

Application / Service

Hardware
Software

System Call Interface

Portable OS Library

Word Processing

Compilers

Email

Web Browsers

Web Servers

Databases
Recall: web server

Client

Request

Reply
(retrieved by web server)

Web Server
Recall: web server

1. network socket read
2. copy arriving packet (DMA)
3. kernel copy
4. parse request
5. file read
6. disk request
7. disk data (DMA)
8. kernel copy
9. format reply
10. network socket write
11. kernel copy from user buffer to network buffer
12. format outgoing packet and DMA

Server
Kernel
Hardware
Network interface
Disk interface

Request
Reply
POSIX I/O: Everything is a “File”

• Identical interface for:
  – Devices (terminals, printers, etc.)
  – Regular files on disk
  – Networking (sockets)
  – Local interprocess communication (pipes, sockets)

• Based on `open()`, `read()`, `write()`, and `close()`

• Allows simple composition of programs
  » `find | grep | wc` ...
POSIX I/O Design Patterns

• Open before use
  – Access control check, setup happens here

• Byte-oriented
  – Least common denominator
  – OS responsible for hiding the fact that real devices may not work this way (e.g. hard drive stores data in blocks)

• Explicit close
POSIX I/O: Kernel Buffering

- **Reads are buffered**
  - Part of making everything byte-oriented
  - Process is blocked while waiting for device
  - Let other processes run while gathering result

- **Writes are buffered**
  - Complete in background (more later on)
  - Return to user when data is “handed off” to kernel
Putting it together: web server

1. network socket read
2. copy arriving packet (DMA)
3. kernel copy
4. parse request
5. file read
6. disk request
7. disk data (DMA)
8. kernel copy
9. format reply
10. network socket write
11. kernel copy from user buffer to network buffer
12. format outgoing packet and DMA

Kernel buffer reads

Request

Reply

Network interface

Disk interface
Putting it together: web server

1. Network socket read
2. Copy arriving packet (DMA)
3. Kernel copy
4. Parse request
5. File read
6. Disk request
7. Disk data (DMA)
8. Kernel copy
9. Format reply
10. Network socket write
11. Kernel copy from user buffer to network buffer
12. Format outgoing packet and DMA

Kernel buffer write

Request
Reply

Network interface
Disk interface
I/O & Storage Layers

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
The File System Abstraction

• High-level idea
  – Files live in hierarchical namespace of filenames

• File
  – Named collection of data in a file system
  – POSIX File data: sequence of bytes
    » 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
  – Hierarchical (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

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

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

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

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 — stdio.h

• 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

• All can be redirected (for instance, using “pipe” symbol: ‘|’):
  — cat hello.txt | grep “World!”
     » Cat’s stdout goes to grep’s stdin!
# C high level File API – stream ops

```c
#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, ...);
```
#include <stdio.h>

int main(void) {
    FILE* input = fopen("input.txt", "r");
    FILE* output = fopen("output.txt", "w");
    int c;

    c = fgetc(input);
    while (c != EOF) {
        fputc(output, c);
        c = fgetc(input);
    }

    fclose(input);
    fclose(output);
}`
# What if we wanted block by block I/O?

```c
#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, ...);
```
#include <stdio.h>
#define BUFFER_SIZE 1024
int main(void) {
    FILE* input = fopen("input.txt", "r");
    FILE* output = fopen("output.txt", "w");
    char buffer[BUFFER_SIZE];
    size_t length;
    length = fread(buffer, BUFFER_SIZE, sizeof(char), input);
    while (length > 0) {
        fwrite(buffer, length, sizeof(char), output);
        length = fread(buffer, BUFFER_SIZE, sizeof(char), input);
    }
    fclose(input);
    fclose(output);
}
Aside: Systems Programming

- Systems programmers are paranoid
- We should really be writing things like:
  ```c
  FILE* input = fopen("input.txt", "r");
  if (input == NULL) {
    // Prints our string and error msg.
    perror("Failed to open input file")
  }
  ```
- Be thorough about checking return values
  - Want failures to be systematically caught and dealt with
C Stream API: Positioning

```c
int fseek(FILE *stream, long int offset, int whence);
```

```c
long int ftell(FILE *stream)
```

```c
void rewind(FILE *stream)
```

- Preserves high level abstraction of a uniform stream of objects
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
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

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!
A little example: lowio.c

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

int main() {
    char buf[1000];
    int fd = open("lowio.c", O_RDONLY, S_IRUSR | S_IWUSR);
    ssize_t rd = read(fd, buf, sizeof(buf));
    int err = close(fd);
    ssize_t wr = write(STDOUT_FILENO, buf, rd);
}
```
And lots more!

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

```c
int dup2 (int old, int new)
int dup (int old)
```
```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);
}
```
#include <fcntl.h>
#include <unistd.h>

#define BUFFER_SIZE 1024

int main(void) {
    int input_fd = open("input.txt", O_RDONLY);
    int output_fd = open("output.txt", O_WRONLY);
    char buffer[BUFFER_SIZE];
    ssize_t length;
    length = read(input_fd, buffer, BUFFER_SIZE);
    while (length > 0) {
        write(output_fd, buffer, length);
        length = read(input_fd, buffer, BUFFER_SIZE);
    }
    close(input_fd);
    close(output_fd);
}
Streams vs. File Descriptors

- Streams are buffered in user memory:
  ```c
  printf("Beginning of line ");
  sleep(10); // sleep for 10 seconds
  printf("and end of line\n");
  ```
  ⇒ Prints out everything at once

- Operations on file descriptors are visible immediately
  ```c
  write(STDOUT_FILENO, "Beginning of line ", 18);
  sleep(10);
  write("and end of line \n", 16);
  ```
  ⇒ Outputs "Beginning of line" 10 seconds earlier
Summary: Key Unix I/O Design Concepts

• Uniformity – everything is a file
  – 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 yielding processor to other task

• Kernel buffered writes
  – Completion of out-going transfer decoupled from the application, allowing it to continue

• Explicit close
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

- 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??

- 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

File descriptor number
- an int

File Descriptors
- a struct with all the info about the files
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`
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;
}
```

- Read up to “count” bytes from “file” starting from “pos” into “buf”.
- Return error or number of bytes read.
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;

Make sure we are allowed to read this file
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
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 whether we can write to `buf` (e.g., `buf` is in the user space range)
- `unlikely()`: hint to branch prediction this condition is unlikely
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 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()
 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;

In fs/read_write.c
Notify the parent of this file that the file was read (see http://www.fieldses.org/~bfields/kernel/vfs.txt)
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 bytes read by “current” task (for scheduling purposes)
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
- User Program requests I/O
- User process completes
- User Program returns from system call

Kernel I/O Subsystem
- Kernel I/O subsystem checks if request can be satisfied
  - Yes: send request to device driver, block process if appropriate
  - No: continue

Device Driver Top Half
- Device driver processes request, issues commands to controller, configures controller to block
  - Device driver completes
  - Device controller change to I/O subsystem

Device Driver Bottom Half
- Device controller receives interrupt, stores data in device-driver buffer
  - Device controller issues interrupt

Device Hardware
- Device hardware monitors device, generates interrupt
  - I/O completed

Time

System Call
Communication between processes

- Can we view files as communication channels?

\[ \text{write}(\text{wfd, wbuf, wlen}); \]

\[ \text{n = read}(\text{rfd, rbuf, rmax}); \]

- 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

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

```c
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)

write(rqfd, rqbuf, buflen);

wait

requests

n = read(rfd, rbuf, rmax);

service request

write(wfd, respbuf, len);

responses

n = read(resfd, resbuf, resmax);

Server (performs operations)
Request Response Protocol

Client (issues requests)

write(rqfd, rqbuf, buflen);

n = read(rfd, rbuf, rmax);

wait

Server (performs operations)

requests

service request

responses

write(wfd, respbuf, len);

n = read(resfd, resbuf, resmax);
Client-Server Models

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

• Client “sometimes on”
  – Initiates a request to the server when interested
  – E.g., Web browser on your laptop or cell phone
  – Doesn’t communicate directly with other clients
  – Needs to know the server’s address

• Server is “always on”
  – Services requests from many client hosts
  – E.g., Web server for the www.cnn.com Web site
  – Doesn’t initiate contact with the clients
  – Needs a fixed, well-known address
Sockets

- **Socket**: an abstraction of a network I/O queue
  - Mechanism for inter-process communication
  - Embodies one side of a communication channel
    » Same interface regardless of location of other end
    » Could be local machine (called “UNIX socket”) or remote machine (called “network socket”)
  - First introduced in 4.2 BSD UNIX: big innovation at time
    » Now most operating systems provide some notion of socket

- Data transfer like files
  - Read / Write against a descriptor

- Over ANY kind of network
  - Local to a machine
  - Over the internet (TCP/IP, UDP/IP)
  - OSI, Appletalk, SNA, IPX, SIP, NS, …
Silly Echo Server – running example

Client (issues requests)

Server (performs operations)

gets(fd, sndbuf, ...);

write(fd, buf, len);

requests

wait

responses

n = read(fd, buf,);

print

write(fd, buf,);

n = read(fd, rcvbuf,);

print

print
void client(int sockfd) {
    int n;
    char sndbuf[MAXIN]; char rcvbuf[MAXOUT];
    getreq(sndbuf, MAXIN); /* prompt */
    while (strlen(sndbuf) > 0) {
        write(sockfd, sndbuf, strlen(sndbuf)); /* send */
        memset(rcvbuf,0,MAXOUT);               /* clear */
        n = read(sockfd, rcvbuf, MAXOUT-1);    /* receive */
        write(STDOUT_FILENO, rcvbuf, n);       /* echo */
        getreq(sndbuf, MAXIN);                /* prompt */
    }
}

void server(int consockfd) {
    char reqbuf[MAXREQ];
    int n;
    while (1) {
        memset(reqbuf,0, MAXREQ);
        n = read(consockfd,reqbuf,MAXREQ-1); /* Recv */
        if (n <= 0) return;
        n = write(STDOUT_FILENO, reqbuf, strlen(reqbuf)); /* echo */
        n = write(consockfd, reqbuf, strlen(reqbuf)); /* echo */
    }
}
What assumptions are we making?

- Reliable
  - Write to a file => Read it back. Nothing is lost.
  - Write to a (TCP) socket => Read from the other side, same.
  - Like pipes

- In order (sequential stream)
  - Write X then write Y => read gets X then read gets Y

- When ready?
  - File read gets whatever is there at the time. Assumes writing already took place.
  - Like pipes!
Socket creation and connection

- File systems provide a collection of permanent objects in structured name space
  - Processes open, read/write/close them
  - Files exist independent of the processes
- Sockets provide a means for processes to communicate (transfer data) to other processes.
- Creation and connection is more complex
- Form 2-way pipes between processes
  - Possibly worlds away
- How do we name them?
- How do these completely independent programs know that the other wants to “talk” to them?
Namespaces for communication over IP

- Hostname
  - www.eecs.berkeley.edu
- IP address
  - 128.32.244.172 (ipv6?)
- Port Number
  - 0-1023 are “well known” or “system” ports
    » Superuser privileges to bind to one
  - 1024 – 49151 are “registered” ports (registry)
    » Assigned by IANA for specific services
  - 49152–65535 (2^{15}+2^{14} to 2^{16}−1) are “dynamic” or “private”
    » Automatically allocated as “ephemeral Ports”
Socket Setup over TCP/IP

- Special kind of socket: server socket
  - Has file descriptor
  - Can’t read or write
- Two operations:
  1. `listen()`: Start allowing clients to connect
  2. `accept()`: Create a new socket for a particular client connection
Socket Setup over TCP/IP

- **Server Socket**: Listens for new connections
  - Produces new sockets for each unique connection
  - 3-way handshake to establish new connection!

- **Things to remember**:
  - Connection involves 5 values:
    - [Client Addr, Client Port, Server Addr, Server Port, Protocol]
  - Often, Client Port “randomly” assigned
    » Done by OS during client socket setup
  - Server Port often “well known”
    » 80 (web), 443 (secure web), 25 (sendmail), etc
    » Well-known ports from 0—1023
Web Server using Sockets (in concept)

**Client**

- Create Client Socket
- Connect it to server (host:port)
- Connection Socket ↔ Connection Socket
- write request
- read response
- Close Client Socket

**Server**

- Create Server Socket
- Bind it to an Address (host:port)
- Listen for Connection
- Accept syscall()
- Connection Socket ↔ Connection Socket
- read request
- write response
- Close Connection Socket
- Close Server Socket
char *host_name, port_name;

// Create a socket
struct addrinfo *server = lookup_host(host_name, port_name);
int sock_fd = socket(server->ai_family, server->ai_socktype,
                       server->ai_protocol);

// Connect to specified host and port
connect(sock_fd, server->ai_addr, server->ai_addrlen);

// Carry out Client-Server protocol
run_client(sock_fd);

/* Clean up on termination */
close(sock_fd);
struct addrinfo *lookup_host(char *host_name, char *port) {
    struct addrinfo *server;
    struct addrinfo hints;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;

    int rv = getaddrinfo(host_name, port, &hints, &server);
    if (rv != 0) {
        printf("getaddrinfo failed: %s\n", gai_strerror(rv));
        return NULL;
    }
    return server;
}
/ Create socket to listen for client connections
char *port_name;
struct addrinfo *server = setup_address(port_name);
int server_socket = socket(server->ai_family,
    server->ai_socktype, server->ai_protocol);

// Bind socket to specific port
bind(server_socket, server->ai_addr, server->ai_addrlen);

// Start listening for new client connections
listen(server_socket, MAX_QUEUE);

while (1) {
    // Accept a new client connection, obtaining a new socket
    int conn_socket = accept(server_socket, NULL, NULL);
    serve_client(conn_socket);
    close(conn_socket);
}

close(server_socket);
Server Address - itself

```c
struct addrinfo *setup_address(char *port) {
    struct addrinfo *server;
    struct addrinfo hints;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;
    getaddrinfo(NULL, port, &hints, &server);
    return server;
}
```

- Simple form
- Internet Protocol, TCP
- Accepting any connections on the specified port
How does the server protect itself?

• Isolate the handling of each connection
• By forking it off as another process
Sockets With Protection

**Client**
- Create Client Socket
- Connect it to server (host:port)
  - Connection Socket
    - write request
    - read response
    - Close Client Socket

**Server**
- Create Server Socket
- Bind it to an Address (host:port)
- Listen for Connection
- Accept syscall()
  - Connection Socket
    - Child
      - Close Listen Socket
      - read request
      - write response
      - Close Connection Socket
    - Parent
      - Close Connection Socket
      - Wait for child
      - Close Server Socket

Close Listen Socket
Server Protocol (v2)

// Start listening for new client connections
listen(server_socket, MAX_QUEUE);
while (1) {
    // Accept a new client connection, obtaining a new socket
    int conn_socket = accept(server_socket, NULL, NULL);

    pid_t pid = fork();
    if (pid == 0) {
        close(server_socket);
        serve_client(conn_socket);
        close(conn_socket);
        exit(EXIT_SUCCESS);
    } else {
        close(conn_socket);
        wait(NULL);
    }
}
close(server_socket);
Concurrent Server

- Listen will queue requests
- Buffering present elsewhere
- But server waits for each connection to terminate before initiating the next
Sockets With Protection and Parallelism

**Client**

- Create Client Socket
- Connect it to server (host:port)
- Connection Socket
- write request
- read response
- Close Client Socket

**Server**

- Create Server Socket
- Bind it to an Address (host:port)
- Listen for Connection
- Accept syscall()
- Connection Socket
- read request
- write response
- Close Connection Socket
- Child

- Close Listen Socket
- read request
- write response
- Close Connection Socket
- Parent

- Close Server Socket
Server Protocol (v3)

// Start listening for new client connections
listen(server_socket, MAX_QUEUE);
signal(SIGCHLD,SIG_IGN); // Prevent zombie children
while (1) {
    // Accept a new client connection, obtaining a new socket
    int conn_socket = accept(server_socket, NULL, NULL);

    pid_t pid = fork(); // New process for connection
    if (pid == 0) { // Child process
        close(server_socket); // Doesn't need server_socket
        serve_client(conn_socket); // Serve up content to client
        close(conn_socket); // Done with client!
        exit(EXIT_SUCCESS);
    } else { // Parent process
        close(conn_socket); // Don't need client socket
        // wait(NULL); // Don't wait (SIGCHLD ignored, above)
    }
}
close(server_socket);
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)

• Socket: an abstraction of a network I/O queue
  – Mechanism for inter-process communication