

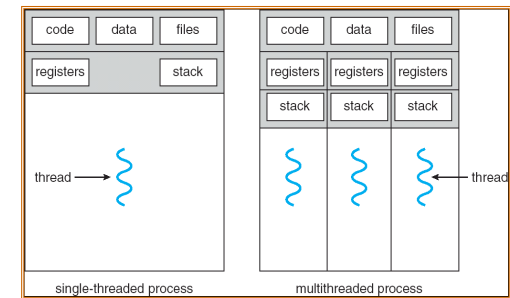
CS162
Operating Systems and
Systems Programming
Lecture 4

Process Management, Fork, and
Introduction to I/O (Everything is a File!)

January 26th, 2023
Prof. John Kubiatowicz
<http://cs162.eecs.Berkeley.edu>

Recall: Processes

- How to manage process state?
 - How to create a process?
 - How to exit from a process?
- Remember: Everything outside of the kernel is running in a process!
 - Including the shell! (Homework 2)
- **Processes are created and managed... by processes!**



1/26/2023

Kubiatowicz CS162 @UCB Spring 2023

Lec 4.2

Bootstrapping

- If processes are created by other processes, how does the first process start?
- First process is started by the kernel
 - Often configured as an argument to the kernel *before* the kernel boots
 - Often called the “init” process
- After this, all processes on the system are created by other processes

Process Management API

- `exit` – terminate a process
- `fork` – copy the current process
- `exec` – change the *program* being run by the current process
- `wait` – wait for a process to finish
- `kill` – send a *signal* (interrupt-like notification) to another process
- `sigaction` – set handlers for signals

1/26/2023

Kubiatowicz CS162 @UCB Spring 2023

Lec 4.3

1/26/2023

Kubiatowicz CS162 @UCB Spring 2023

Lec 4.4

Process Management API

- `exit` – terminate a process
- `fork` – copy the current process
- `exec` – change the *program* being run by the current process
- `wait` – wait for a process to finish
- `kill` – send a *signal* (interrupt-like notification) to another process
- `sigaction` – set handlers for signals

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.5

pid.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
int main(int argc, char *argv[])
{
    /* get current processes PID */
    pid_t pid = getpid();
    printf("My pid: %d\n", pid);

    exit(0);
}
```

Q: What if we let main return without ever calling exit?

- The OS Library calls `exit()` for us!
- The entrypoint of the executable is in the OS library
- OS library calls `main`
- If `main` returns, OS library calls `exit`
- You'll see this in Project 0: `init.c`

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.6

Process Management API

- `exit` – terminate a process
- `fork` – copy the current process
- `exec` – change the *program* being run by the current process
- `wait` – wait for a process to finish
- `kill` – send a *signal* (interrupt-like notification) to another process
- `sigaction` – set handlers for signals

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.7

Creating Processes

- `pid_t fork()` – copy the current process
 - New process has different `pid`
 - New process contains a single thread
- Return value from `fork()`: `pid` (like an integer)
 - When > 0 :
 - » Running in (original) **Parent** process
 - » return value is `pid` of new child
 - When $= 0$:
 - » Running in new **Child** process
 - When < 0 :
 - » Error! Must handle somehow
 - » Running in original process
- State of original process duplicated in *both* Parent and Child!
 - Address Space (Memory), File Descriptors (covered later), etc...

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.8

fork1.c

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

int main(int argc, char *argv[]) {
    pid_t cpid, mypid;
    pid_t pid = getpid();          /* get current processes PID */
    printf("Parent pid: %d\n", pid);
    cpid = fork();
    if (cpid > 0) {                /* Parent Process */
        mypid = getpid();
        printf("[%d] parent of [%d]\n", mypid, cpid);
    } else if (cpid == 0) {        /* Child Process */
        mypid = getpid();
        printf("[%d] child\n", mypid);
    } else {
        perror("Fork failed");
    }
}
```

1/26/2023

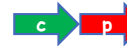
Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.9

fork1.c

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

int main(int argc, char *argv[]) {
    pid_t cpid, mypid;
    pid_t pid = getpid();          /* get current processes PID */
    printf("Parent pid: %d\n", pid);
    cpid = fork();
    if (cpid > 0) {                /* Parent Process */
        mypid = getpid();
        printf("[%d] parent of [%d]\n", mypid, cpid);
    } else if (cpid == 0) {        /* Child Process */
        mypid = getpid();
        printf("[%d] child\n", mypid);
    } else {
        perror("Fork failed");
    }
}
```



1/26/2023

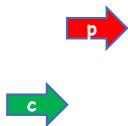
Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.10

fork1.c

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

int main(int argc, char *argv[]) {
    pid_t cpid, mypid;
    pid_t pid = getpid();          /* get current processes PID */
    printf("Parent pid: %d\n", pid);
    cpid = fork();
    if (cpid > 0) {                /* Parent Process */
        mypid = getpid();
        printf("[%d] parent of [%d]\n", mypid, cpid);
    } else if (cpid == 0) {        /* Child Process */
        mypid = getpid();
        printf("[%d] child\n", mypid);
    } else {
        perror("Fork failed");
    }
}
```



1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.11

Mystery: fork_race.c

```
int i;
pid_t cpid = fork();
if (cpid > 0) {
    for (i = 0; i < 10; i++) {
        printf("Parent: %d\n", i);
        // sleep(1);
    }
} else if (cpid == 0) {
    for (i = 0; i > -10; i--) {
        printf("Child: %d\n", i);
        // sleep(1);
    }
}
```

Recall: a process consists of one or more threads executing in an address space

- Here, each process has a single thread
- These threads execute concurrently

- What does this print?
- Would adding the calls to `sleep()` matter?

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.12

Process Management API

- `exit` – terminate a process
- `fork` – copy the current process
- `exec` – change the *program* being run by the current process
- `wait` – wait for a process to finish
- `kill` – send a *signal* (interrupt-like notification) to another process
- `sigaction` – set handlers for signals

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.13

Starting new Program: variants of exec

```
...
cpid = fork();
if (cpid > 0) {
    /* Parent Process */
    tcpid = wait(&status);
} else if (cpid == 0) {
    /* Child Process */
    char *args[] = {"ls", "-l", NULL};
    execv("/bin/ls", args);

    /* execv doesn't return when it works.
       So, if we got here, it failed! */

    perror("execv");
    exit(1);
}
...
```

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.14

fork2.c – parent waits for child to finish

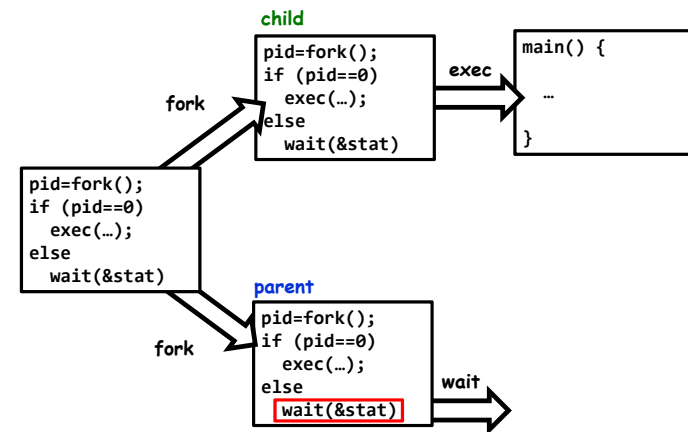
```
int status;
pid_t tcpid;
...
cpid = fork();
if (cpid > 0) {
    /* Parent Process */
    mypid = getpid();
    printf("[%d] parent of [%d]\n", mypid, cpid);
    tcpid = wait(&status);
    printf("[%d] bye %d(%d)\n", mypid, tcpid, status);
} else if (cpid == 0) {
    /* Child Process */
    mypid = getpid();
    printf("[%d] child\n", mypid);
    exit(42);
}
...
```

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.15

Process Management: The Shell pattern



1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.16

Administrivia

- Kubiawicz Office Hours (Starting next week)
 - 2pm-3pm, Monday & Wednesday
- **TOMORROW (Friday) is Drop Deadline! VERY HARD TO DROP LATER!**
- Recommendation: Read assigned readings *before* lecture
- Starting next week, we will be adhering to strict slip-day policies for non-DSP students
 - Slip days are no-questions asked (or justification needed) extensions
 - Anything beyond this requires documentation (i.e. doctor's note, etc)
 - If you run out of slip days, assignments will be discounted 10%/day
- You get 4 slip days for homework and 5 slip days for group projects
 - No project extensions on design documents, since we need to keep design reviews on track
- You should be going to sections – Important information covered in section
 - Any section will do until groups assigned
- Get finding groups of 4 people ASAP
 - Priority for same section; if cannot make this work, keep same TA
 - Remember: Your TA needs to see you in section!
- Midterm 1 will be on 2/16 from 7-9pm

1/26/2023

Kubiawicz CS162 ©UCB Spring 2023

Lec 4.17

Process Management API

- `exit` – terminate a process
- `fork` – copy the current process
- `exec` – change the *program* being run by the current process
- `wait` – wait for a process to finish
- **`kill` – send a *signal* (interrupt-like notification) to another process**
- **`sigaction` – set handlers for signals**

1/26/2023

Kubiawicz CS162 ©UCB Spring 2023

Lec 4.18

inf_loop.c

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

void signal_callback_handler(int signum) {
    printf("Caught signal!\n");
    exit(1);
}

int main() {
    struct sigaction sa;
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);
    sa.sa_handler = signal_callback_handler;
    sigaction(SIGINT, &sa, NULL);
    while (1) {}
}
```

Q: What would happen if the process receives a SIGINT signal, but does not register a signal handler?

A: The process dies!

For each signal, there is a default handler defined by the system

1/26/2023

Kubiawicz CS162 ©UCB Spring 2023

Lec 4.19

Common POSIX Signals

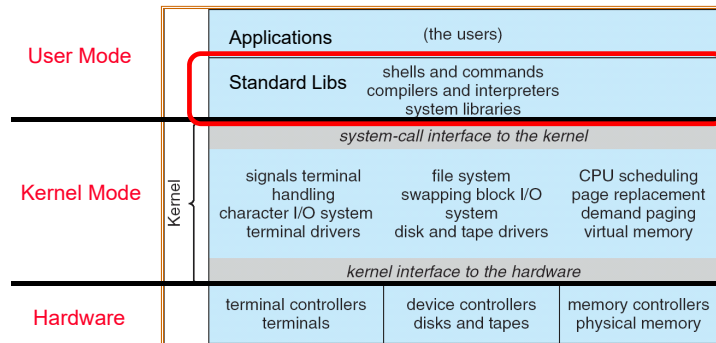
- SIGINT – control-C
- SIGTERM – default for `kill` shell command
- SIGSTP – control-Z (default action: stop process)
- SIGKILL, SIGSTOP – terminate/stop process
 - Can't be changed with `sigaction`
 - Why?

1/26/2023

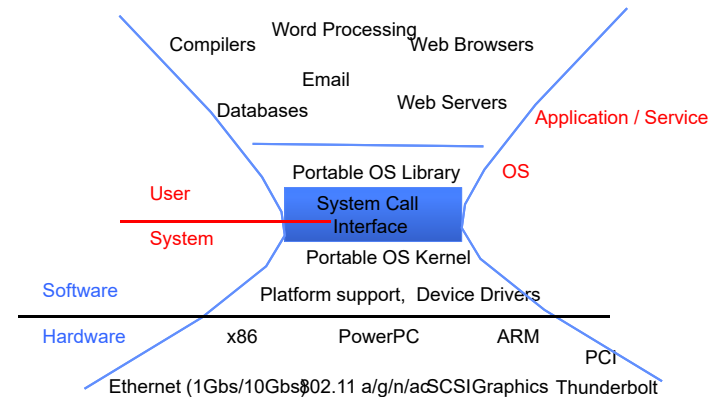
Kubiawicz CS162 ©UCB Spring 2023

Lec 4.20

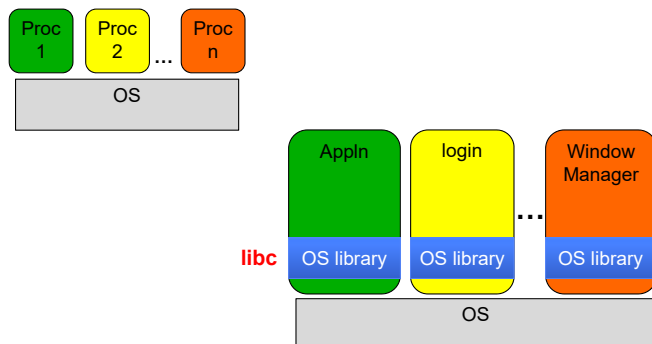
Recall: UNIX System Structure



A Kind of Narrow Waist



Recall: OS Library (libc) Issues Syscalls



- OS Library: Code linked into the user-level application that provides a clean or more functional API to the user than just the raw syscalls
 - Most of this code runs at user level, but makes syscalls (which run at kernel level)

Unix/POSIX Idea: Everything is a "File"

- Identical interface for:
 - Files on disk
 - Devices (terminals, printers, etc.)
 - Regular files on disk
 - Networking (sockets)
 - Local interprocess communication (pipes, sockets)
- Based on the system calls `open()`, `read()`, `write()`, and `close()`
- Additional: `ioctl()` for custom configuration that doesn't quite fit
- Note that the "Everything is a File" idea was a radical idea when proposed
 - Dennis Ritchie and Ken Thompson described this idea in their seminal paper on UNIX called "The UNIX Time-Sharing System" from 1974
 - I posted this on the resources page if you are curious

Aside: POSIX interfaces

- **POSIX: Portable Operating System Interface** (for uniX?)
 - Interface for application programmers (mostly)
 - Defines the term “Unix,” derived from AT&T Unix
 - Created to bring order to many Unix-derived OSes, so applications are portable
 - » Partially available on non-Unix OSes, like Windows
 - Requires standard system call interface

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.25

The File System Abstraction

- **File**
 - Named collection of data in a file system
 - POSIX File data: sequence of bytes
 - » Could be text, binary, serialized objects, ...
 - File Metadata: information about the file
 - » Size, Modification Time, Owner, Security info, 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/fa14/index.html
 - Links and Volumes (later)

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.26

Connecting Processes, File Systems, and Users

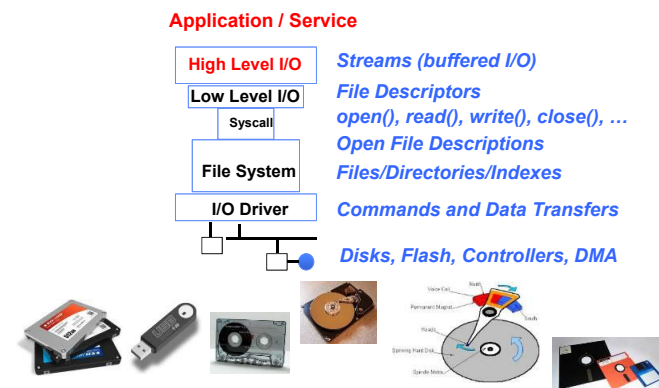
- **Every process has a *current working directory (CWD)***
 - Can be set with system call:
`int chdir(const char *path); //change CWD`
- **Absolute paths ignore CWD**
 - /home/oski/cs162
- **Relative paths are relative to CWD**
 - index.html, ./index.html
 - » Refers to index.html in current working directory
 - ../index.html
 - » Refers to index.html in parent of current working directory
 - ~/index.html, ~cs162/index.html
 - » Refers to index.html in the home directory

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.27

I/O and Storage Layers



1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.28

C High-Level File API – Streams

- Operates on “streams” – unformatted sequences of bytes (with text or binary 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

- Open stream represented by **pointer** to a **FILE** data structure
 - Error reported by returning a NULL pointer

C API Standard Streams – `stdio.h`

- Three predefined streams are opened implicitly when the program is executed.
 - `FILE *stdin` – normal source of input, can be redirected
 - `FILE *stdout` – normal source of output, can too
 - `FILE *stderr` – diagnostics and errors
- STDIN / STDOUT enable composition in Unix
- All can be redirected
 - `cat hello.txt | grep "World!"`
 - `cat's stdout goes to grep's stdin`

C High-Level File API

```
// 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, ... );
```

C Streams: Char-by-Char I/O

```
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);
}
```


C High-Level File API

```
// 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, ... );
```

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.33

C Streams: Block-by-Block I/O

```
#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);
}
```

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.34

Aside: Check your Errors!

- Systems programmers should always be paranoid!
 - Otherwise you get intermittently buggy code
- We should really be writing things like:

```
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
- I may be a bit loose with error checking for examples in class (to keep short)
 - **Do as I say, not as I show in class!**

1/26/2023

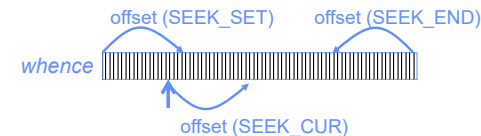
Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.35

C High-Level File API: Positioning The Pointer

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

- For `fseek()`, the offset is interpreted based on the `whence` argument (constants in `stdio.h`):
 - `SEEK_SET`: Then offset interpreted from beginning (position 0)
 - `SEEK_END`: Then offset interpreted backwards from end of file
 - `SEEK_CUR`: Then offset interpreted from current position



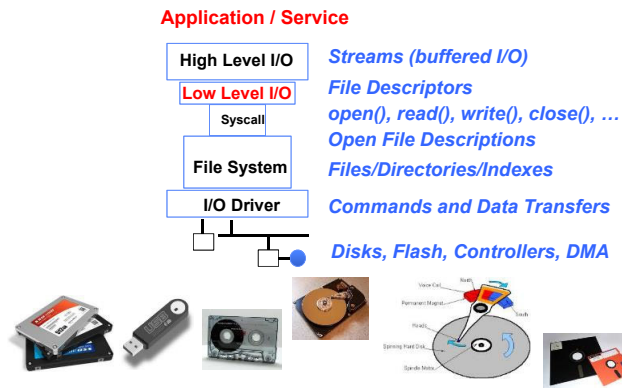
- Overall preserves high-level abstraction of a uniform stream of objects

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.36

I/O and Storage Layers



1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.37

Low-Level File I/O: The RAW system-call interface

```
#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

- Integer return from `open()` is a *file descriptor*
 - *Error indicated by return < 0: the global `errno` variable set with error (see man pages)*
- Operations on *file descriptors*:
 - Open system call created an *open file description* entry in system-wide table of open files
 - *Open file description* object in the kernel represents an instance of an open file
 - *Why give user an integer instead of a pointer to the file description in kernel?*

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.38

C Low-Level (pre-opened) Standard Descriptors

```
#include <unistd.h>
STDIN_FILENO - macro has value 0
STDOUT_FILENO - macro has value 1
STDERR_FILENO - macro has value 2

// Get file descriptor inside FILE *
int fileno (FILE *stream)

// Make FILE * from descriptor
FILE * fdopen (int filedes, const char *opentype)
```

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.39

Low-Level File API

- Read data from open file using file descriptor:

```
ssize_t read (int filedes, void *buffer, size_t maxsize)
```

 - Reads up to `maxsize` bytes – **might actually read less!**
 - returns bytes read, 0 => EOF, -1 => error
- Write data to open file using file descriptor

```
ssize_t write (int filedes, const void *buffer, size_t size)
```

 - returns number of bytes written
- Reposition file offset within kernel (this is independent of any position held by high-level FILE descriptor for this file!)

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

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.40

Example: lowio.c

```
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);
}
```

- How many bytes does this program read?

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 inside kernel**
 - Part of making everything byte-oriented
 - Process is **blocked** while waiting for device
 - Let other processes run while gathering result
- **Writes are buffered inside kernel**
 - Complete in background (more later on)
 - Return to user when data is “handed off” to kernel
- This buffering is part of global buffer management and caching for block devices (such as disks)
 - Items typically cached in quanta of disk block sizes
 - We will have many interesting things to say about this buffering when we dive into the kernel

Low-Level I/O: Other Operations

- Operations specific to terminals, devices, networking, ...
 - e.g., `ioctl`
- Duplicating descriptors
 - `int dup2(int old, int new);`
 - `int dup(int old);`
- Pipes – channel
 - `int pipe(int pipefd[2]);`
 - Writes to `pipefd[1]` can be read from `pipefd[0]`
- File Locking
- Memory-Mapping Files
- Asynchronous I/O

High-Level vs. Low-Level File API

High-Level Operation:

```
size_t fread(...) {
    Do some work like a normal fn...
```

asm code ... syscall # into %eax
put args into registers %ebx, ...
special trap instruction

Kernel:
get args from regs
dispatch to system func
Do the work to read from the file
Store return value in %eax

```
get return values from regs
Do some more work like a normal fn...
};
```

Low-Level Operation:

```
ssize_t read(...) {
```

asm code ... syscall # into %eax
put args into registers %ebx, ...
special trap instruction

Kernel:
get args from regs
dispatch to system func
Do the work to read from the file
Store return value in %eax

```
get return values from regs
};
```

High-Level vs. Low-Level File API

- Streams are buffered in user memory:


```
printf("Beginning of line ");
sleep(10); // sleep for 10 seconds
printf("and end of line\n");
```

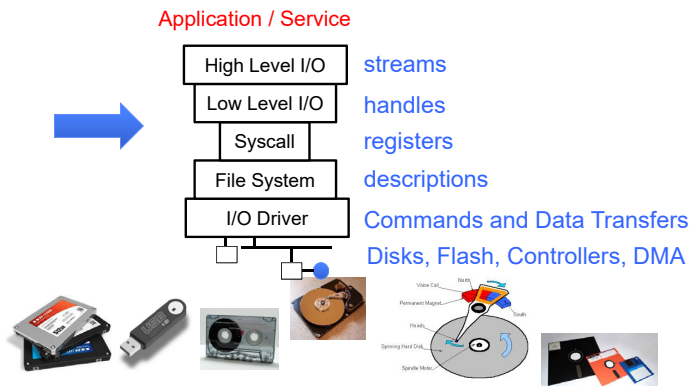
Prints out everything at once

- Operations on file descriptors are visible immediately


```
write(STDOUT_FILENO, "Beginning of line ", 18);
sleep(10);
write("and end of line \n", 16);
```

Outputs "Beginning of line" 10 seconds earlier than "and end of line"

What's below the surface ??

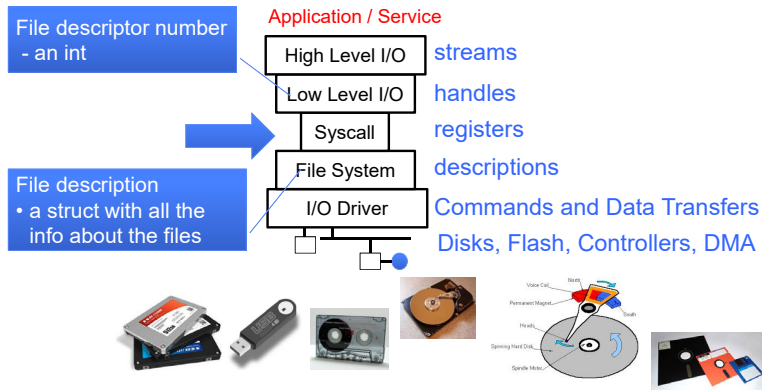


Recall: SYSCALL

#	Name	Registers	Definition
0	sys_restart_syscall	0x00	kernel/signal.c:2058
1	sys_exit	0x01	kernel/exit.c:1046
2	sys_fork	0x02	arch/alpha/kernel/entry.S:716
3	sys_read	0x03	fs/read_write.c:393
4	sys_write	0x04	fs/read_write.c:408
5	sys_open	0x05	fs/open.c:900
6	sys_close	0x06	fs/open.c:969
7	sys_waitpid	0x07	kernel/exit.c:1771
8	sys_creat	0x08	fs/open.c:933
9	sys_link	0x09	fs/namei.c:2320

- 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 ??



1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.49

What's in an Open File Description?

Inside Kernel!

For our purposes, the two most important things are:

- Where to find the file data on disk
- The current position within the file

```

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

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.50

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 -EIO;
    if (!file->f_op || (!file->f_op->read && !file->f_op->read_iter))
        return -EINVAL;
    if (unlikely(!access_ok(VERIFY_WRITE, buf, count)))
        return -EIO;
    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.

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.51

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 -EIO;
    if (!file->f_op || (!file->f_op->read && !file->f_op->read_iter))
        return -EINVAL;
    if (unlikely(!access_ok(VERIFY_WRITE, buf, count)))
        return -EIO;
    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

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.52

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;
```

Check if file has read methods

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.53

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;
```

•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

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.54

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;
```

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

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.55

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;
```

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

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.56

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;
}
```

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

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.57

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;
}
```

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

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.58

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;
}
```

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

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.59

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 *, filowner_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 *);
    [...];
};
```

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.60

Device Drivers

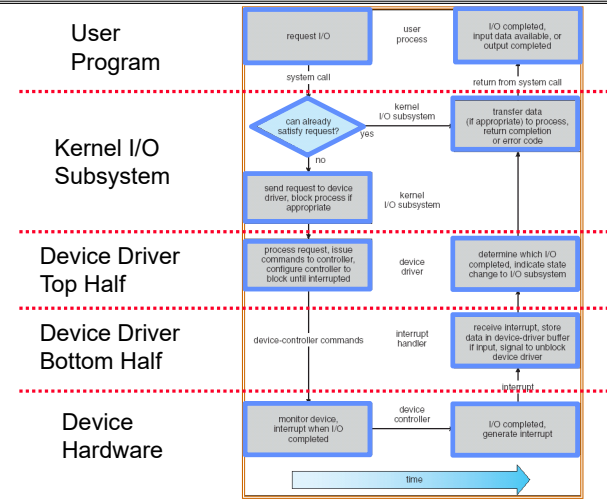
- **Device Driver:** Device-specific code in the kernel that interacts directly with the device hardware
 - Supports a standard, internal interface
 - Same kernel I/O system can interact easily with different device drivers
 - Special device-specific configuration supported with the `ioctl()` system 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

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.61

Life Cycle of An I/O Request



1/26/2023

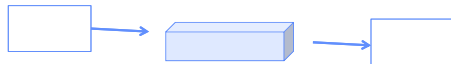
Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.62

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!

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.63

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?

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

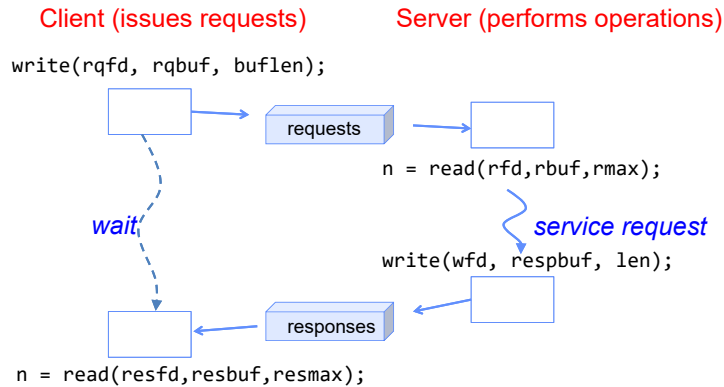
Lec 4.64

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.64

Request Response Protocol

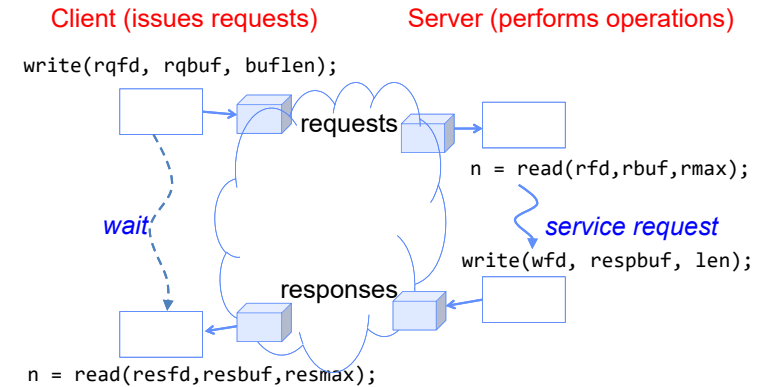


1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.65

Request Response Protocol: Across Network



1/26/2023

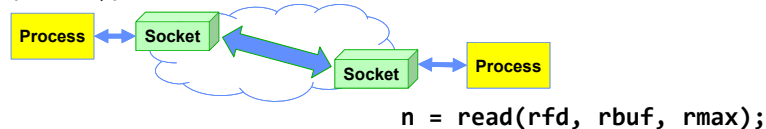
Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.66

The Socket Abstraction: Endpoint for Communication

- **Key Idea:** Communication across the world looks like File I/O

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



- Sockets: Endpoint for Communication
 - Queues to temporarily hold results
- Connection: Two Sockets Connected Over the network ⇒ IPC over network!
 - How to **open()**?
 - What is the namespace?
 - How are they connected in time?

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.67

Sockets: More Details

- **Socket:** An abstraction for one endpoint of a network connection
 - Another mechanism for **inter-process communication**
 - Most operating systems (Linux, Mac OS X, Windows) provide this, even if they don't copy rest of UNIX I/O
 - Standardized by POSIX
- First introduced in 4.2 BSD (Berkeley Standard Distribution) Unix
 - This release had some huge benefits (and excitement from potential users)
 - Runners waiting at release time to get release on tape and take to businesses
- Same abstraction for any kind of network
 - Local (within same machine)
 - The Internet (TCP/IP, UDP/IP)
 - Things "no one" uses anymore (OSI, Appletalk, IPX, ...)

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.68

Sockets: More Details

- Looks just like a file with a **file descriptor**
 - Corresponds to a network connection (*two* queues)
 - **write** adds to output queue (queue of data destined for other side)
 - **read** removes from it input queue (queue of data destined for this side)
 - Some operations do not work, e.g. **lseek**
- How can we use sockets to support real applications?
 - A bidirectional byte stream isn't useful on its own...
 - May need messaging facility to partition stream into chunks
 - May need RPC facility to translate one environment to another and provide the abstraction of a function call over the network

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.69

Simple Example: Echo Server

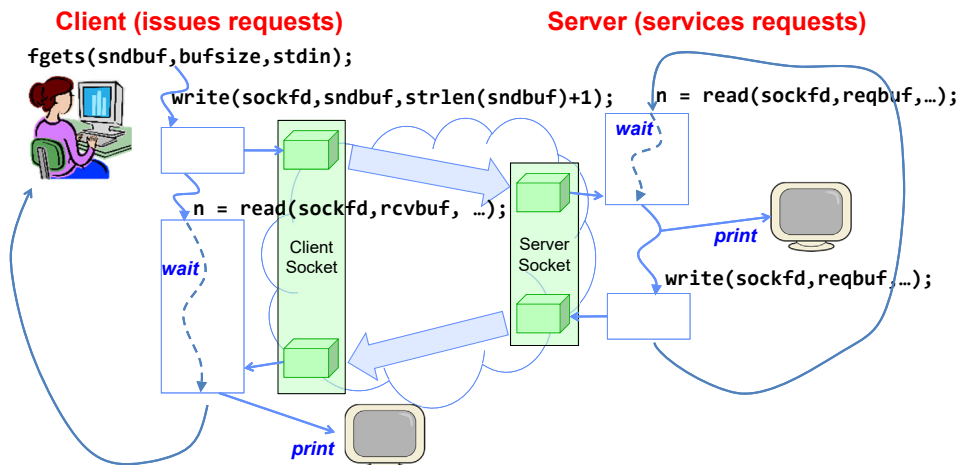


1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.70

Simple Example: Echo Server



1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.71

Echo client-server example

```
void client(int sockfd) {
    int n;
    char sndbuf[MAXIN]; char rcvbuf[MAXOUT];
    while (1) {
        fgets(sndbuf, MAXIN, stdin);          /* prompt */
        write(sockfd, sndbuf, strlen(sndbuf)+1); /* send (including null terminator) */
        memset(rcvbuf, 0, MAXOUT);          /* clear */
        n=read(sockfd, rcvbuf, MAXOUT);      /* receive */
        write(STDOUT_FILENO, rcvbuf, n);    /* echo */
    }
}
```

```
void server(int consockfd) {
    char reqbuf[MAXREQ];
    int n;
    while (1) {
        memset(reqbuf, 0, MAXREQ);
        len = read(consockfd, reqbuf, MAXREQ); /* Recv */
        if (n <= 0) return;
        write(STDOUT_FILENO, reqbuf, n);
        write(consockfd, reqbuf, n); /* echo */
    }
}
```

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.72

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.
- **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
 - » Actually need to loop and read until we receive the terminator ('\0')
 - Assumes writing already took place
 - Blocks if nothing has arrived yet

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.73

Socket Creation

- File systems provide a collection of permanent objects in a structured name space:
 - Processes open, read/write/close them
 - Files exist independently of processes
 - Easy to name what file to open()
- Pipes: one-way communication between processes on same (physical) machine
 - Single queue
 - Created transiently by a call to pipe()
 - Passed from parent to children (descriptors inherited from parent process)
- Sockets: two-way communication between processes on same or different machine
 - Two queues (one in each direction)
 - Processes can be on separate machines: no common ancestor
 - How do we *name* the objects we are opening?
 - How do these completely independent programs know that the other wants to “talk” to them?

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.74

Namespaces for Communication over IP

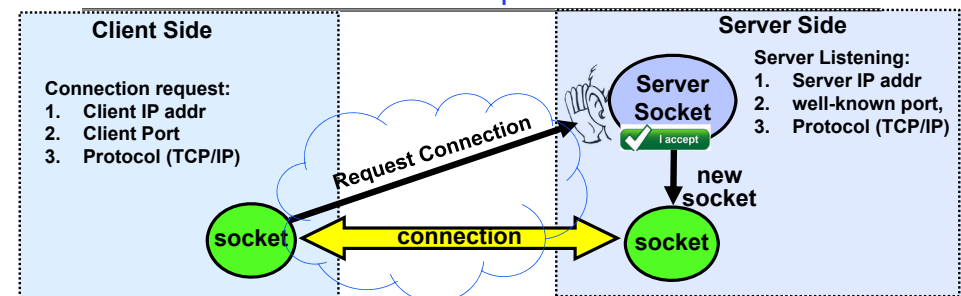
- **Hostname**
 - www.eecs.berkeley.edu
- **IP address**
 - 128.32.244.172 (IPv4, 32-bit Integer)
 - 2607:f140:0:81::f (IPv6, 128-bit Integer)
- **Port Number**
 - 0-1023 are “well known” or “system” ports
 - » Superuser privileges to bind to one
 - 1024 – 49151 are “registered” ports (**registry**)
 - » 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”

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.75

Connection Setup over TCP/IP



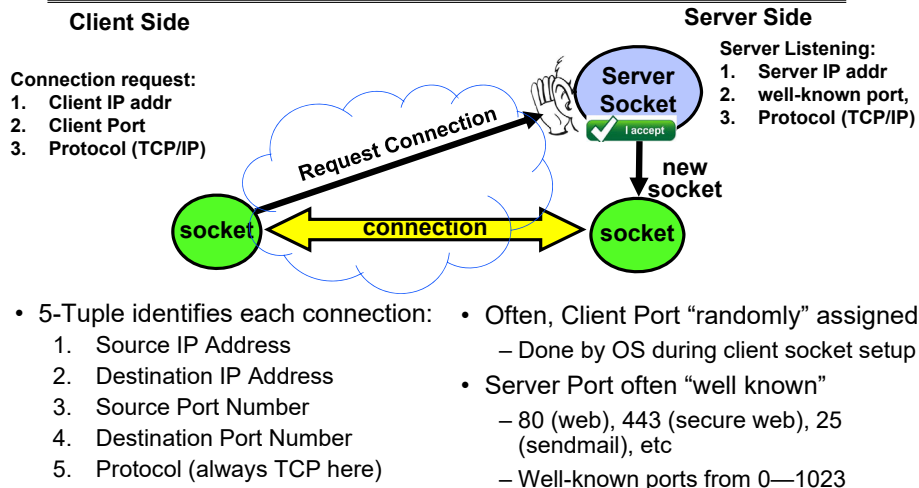
- Special kind of socket: **server socket**
 - Has file descriptor
 - Can't read or write
- Two operations:
 1. **listen()**: Start allowing clients to connect
 2. **accept()**: Create a *new socket* for a *particular* client

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.76

Connection Setup over TCP/IP



1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.77

Conclusion (I)

- System Call Interface is “narrow waist” between user programs and kernel
- Streaming I/O: 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 ...

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.78

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

1/26/2023

Kubiatowicz CS162 ©UCB Spring 2023

Lec 4.79