CS162
Operating Systems and Systems Programming
Lecture 5

Abstractions 3: Files and I/O, Sockets and IPC

January 30st, 2024

Prof. John Kubiatowicz

http://cs162.eecs.Berkeley.edu

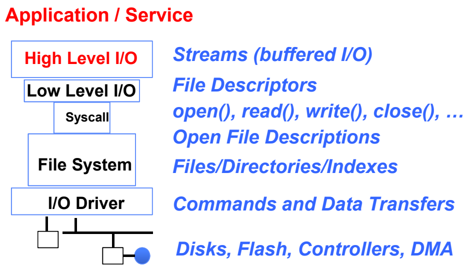
Recall: Process Creating New Processes

- `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…
  - For now–your mental model of `fork()` should be *complete* duplication of Parent

Recall: 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

I/O and Storage Layers



Application / Service

High Level I/O — *Streams (buffered I/O)*

Low Level I/O — *File Descriptors*

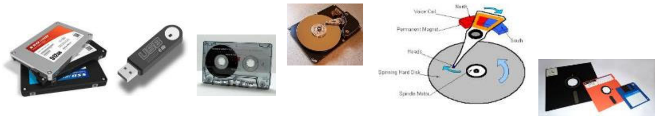
Syscall — *open(), read(), write(), close(), …*

Open File Descriptions

File System — *Files/Directories/Indexes*

I/O Driver — *Commands and Data Transfers*

Disks, Flash, Controllers, DMA



C High-Level File API – Streams

- Operates on "streams" – unformatted sequences of bytes (wither 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, ... );
```

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, sizeof(char), BUFFER_SIZE, input);
  while (length > 0) {
    fwrite(buffer, sizeof(char), length, output);
    length = fread(buffer, sizeof(char), BUFFER_SIZE, input);
  }
  fclose(input);
  fclose(output);
}
```

Aside: Check your Errors!

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

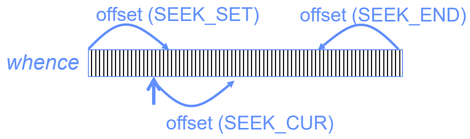
```
FILE* input = fopen(“input.txt”, “r”);
if (input == NULL) {
  // Prints our string and error msg.
  perror(“Failed to open input file”)
}
```

- 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!

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

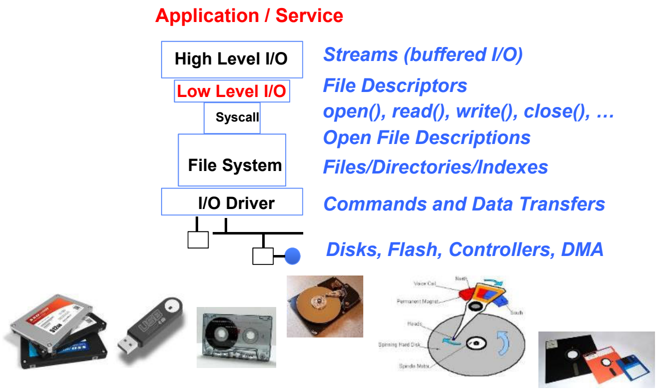


- Overall preserves high-level abstraction of a uniform stream of objects

Administrivia

- Kubiatowicz Office Hours (673 Soda Hall):
  - – 3pm-4pm, Tuesday/Thursday
- Friday was drop deadline. If you forgot to drop, we can't help you!
  - – You need to speak with advisor services in your department about how to drop
- Be careful on Ed: Don't give away solutions when you post questions or answers
  - – Remember that everyone is supposed to do their own work!
- Recommendation: Read assigned readings *before* lecture
- Group sign up should have happened already
  - – If you don't have 4 members in your group, we will try to find you other partners
  - – Want everyone in your group to have the same TA
  - – Go to your assigned section on Friday, starting this week!
- Midterm 1 conflicts
  - – Watch for announcements on EdStem (remember: MT1 is 2/15)

I/O and Storage Layers



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?

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)
```

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)
```

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

Low-Level vs High-Level file API

- Low-level direct use of syscall interface: `open(), read(), write(), close()`
- Opening of file returns file descriptor:
  `int myfile = open(…);`
- File descriptor only meaningful to kernel
  - Index into process (PDB) which holds pointers to kernel-level structure ("file description") describing file.
- Every `read()` or `write()` causes syscall no matter how small (could read a single byte)
- Consider loop to get 4 bytes at a time using `read()`:
  - Each iteration enters kernel for 4 bytes.

- High-level buffered access: `fopen(), fread(), fwrite(), fclose()`
- Opening of file returns ptr to FILE:
  `FILE *myfile = fopen(…);`
- FILE structure is user space contains:
  - a chunk of memory for a buffer
  - the file descriptor for the file (`fopen()` will call `open()` automatically)
- Every `fread()` or `fwrite()` filters through buffer and may not call `read()` or `write()` on every call.
- Consider loop to get 4 bytes at a time using `fread()`:
  - First call to `fread()` calls `read()` for block of bytes (say 1024). Puts in buffer and returns first 4 to user.
  - Subsequent `fread()` grab bytes from buffer

Low-Level vs. High-Level File API

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

  Return data to caller
};
```

High-Level Operation:

```
ssize_t fread(…) {
  Check buffer for contents
  Return data to caller if available
    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
  Update buffer with excess data
  Return data to caller
};
```

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");
```

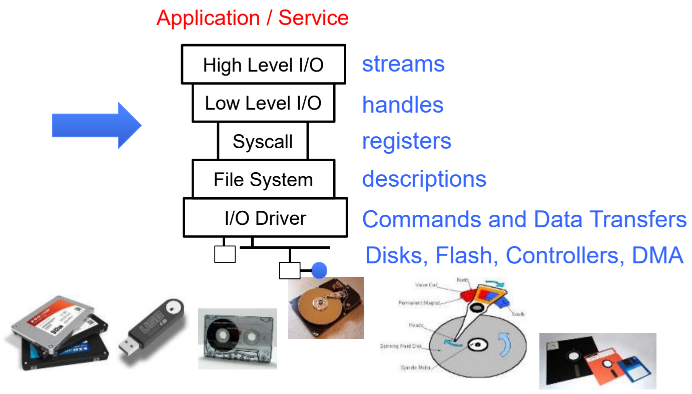
Prints out everything at once

- 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 ??


Application / Service

High Level I/O — streams

Low Level I/O — handles

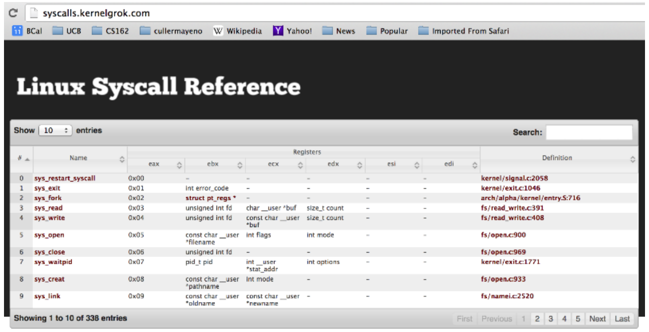
Syscall — registers

File System — descriptions

I/O Driver — Commands and Data Transfers

Disks, Flash, Controllers, DMA

Recall: SYSCALL


syscalls.kernelgrok.com
BCal UCB CS162 cullermayeno Wikipedia Yahoo! News Popular Imported From Safari

Linux Syscall Reference

Show 10 entries Search:

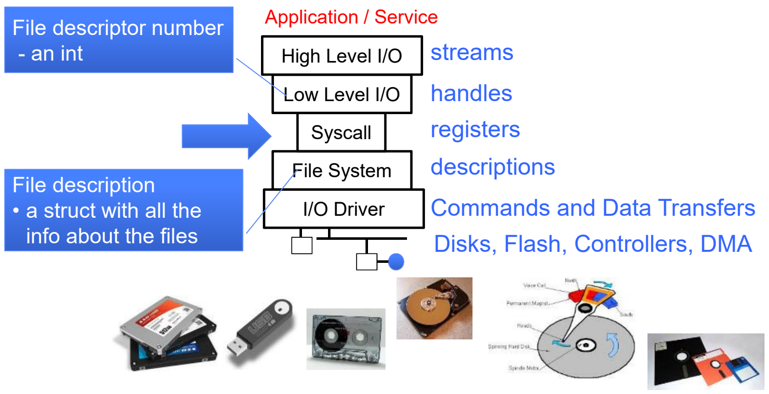
#	Name	Registers						Definition
		eax	ebx	ecx	edx	esi	edi	
0	sys_restart_syscall	0x00	-	-	-	-	-	kernel/signal.c:2058
1	sys_exit	0x01	int error_code	-	-	-	-	kernel/exit.c:1046
2	sys_fork	0x02	struct pt_regs *	-	-	-	-	arch/alpha/kernel/entry.S:716
3	sys_read	0x03	unsigned int fd	char __user *buf	size_t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

Showing 1 to 10 of 338 entries

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

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

What’s below the surface ??


File descriptor number
- an int

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

Application / Service

High Level I/O — streams

Low Level I/O — handles

Syscall — registers

File System — descriptions

I/O Driver — Commands and Data Transfers

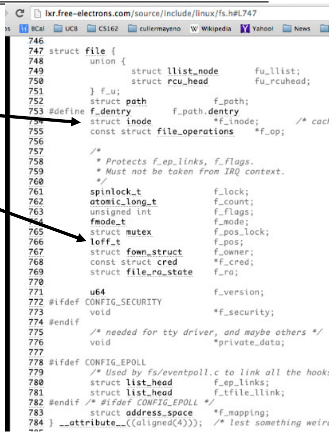
Disks, Flash, Controllers, DMA

What’s in an Open File Description?

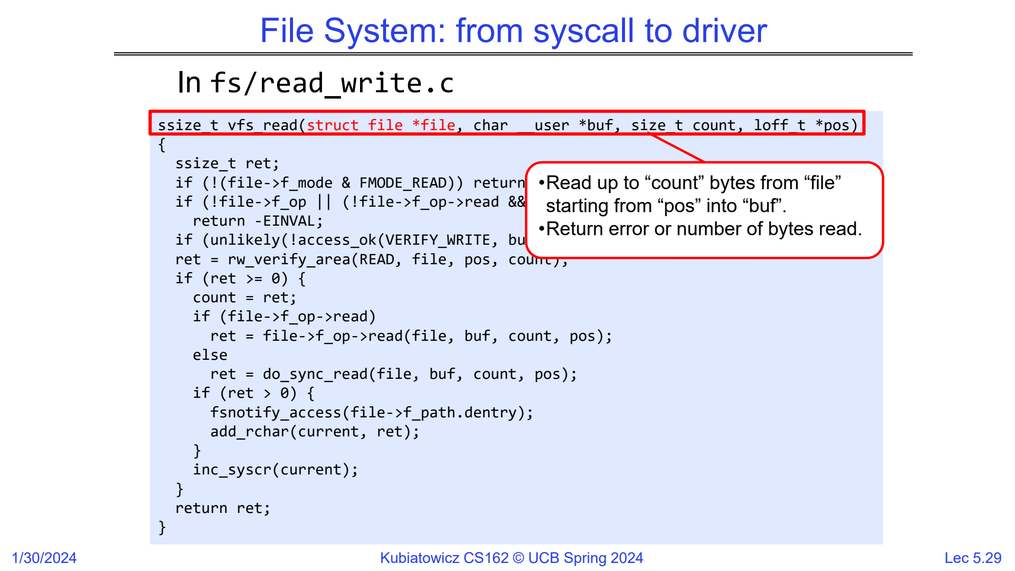
Inside Kernel!

For our purposes, the two most important things are:

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


lxr.free-electrons.com/source/include/linux/fs.h#L747
BCal UCB CS162 cullermayeno Wikipedia Yahoo! News

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

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

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

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

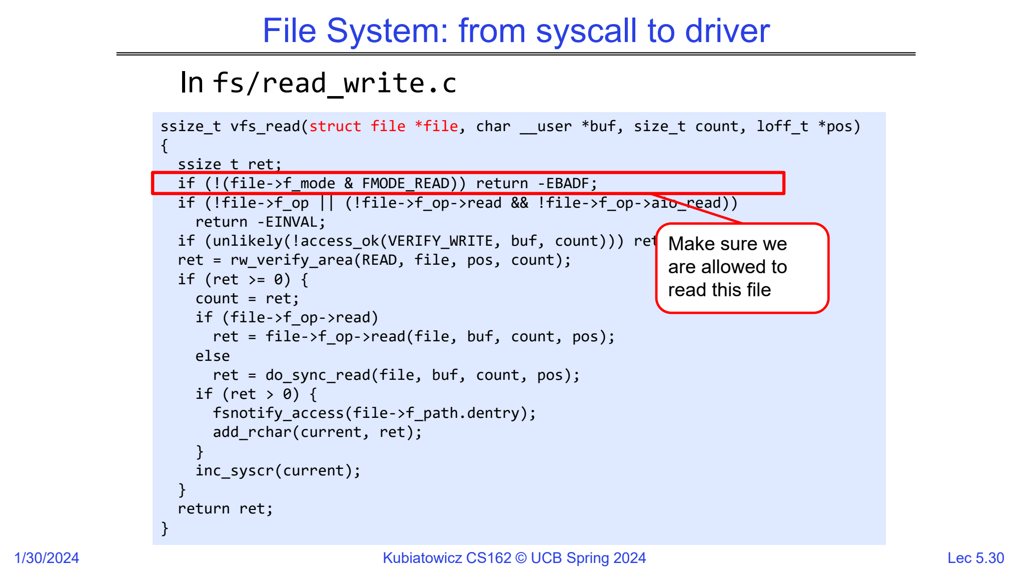


File System: from syscall to driver

In `fs/read_write.c`

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
  ssize_t ret;
  if (!(file->f_mode & FMODE_READ)) return
  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 read.

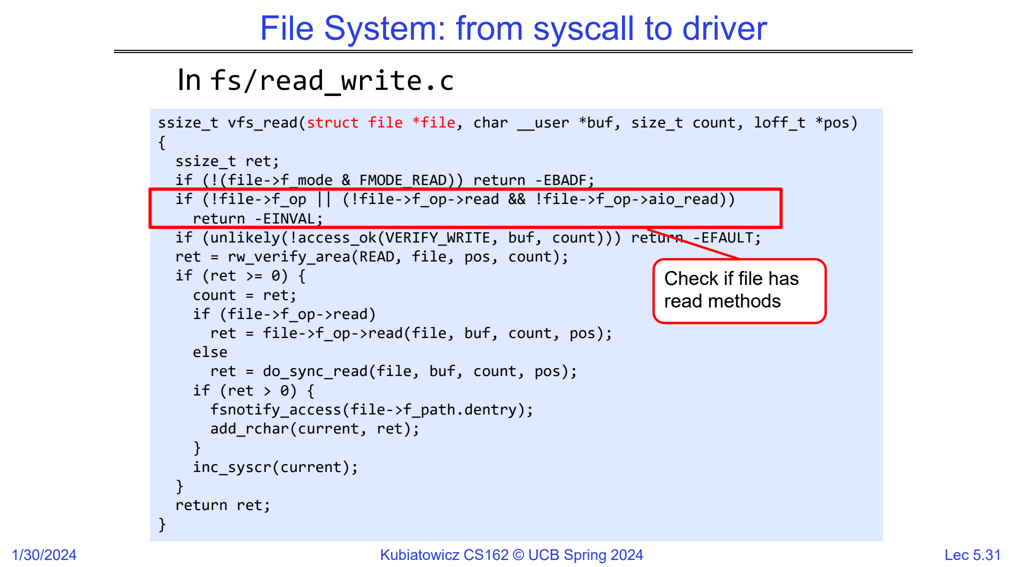


File System: from syscall to driver

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))) 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

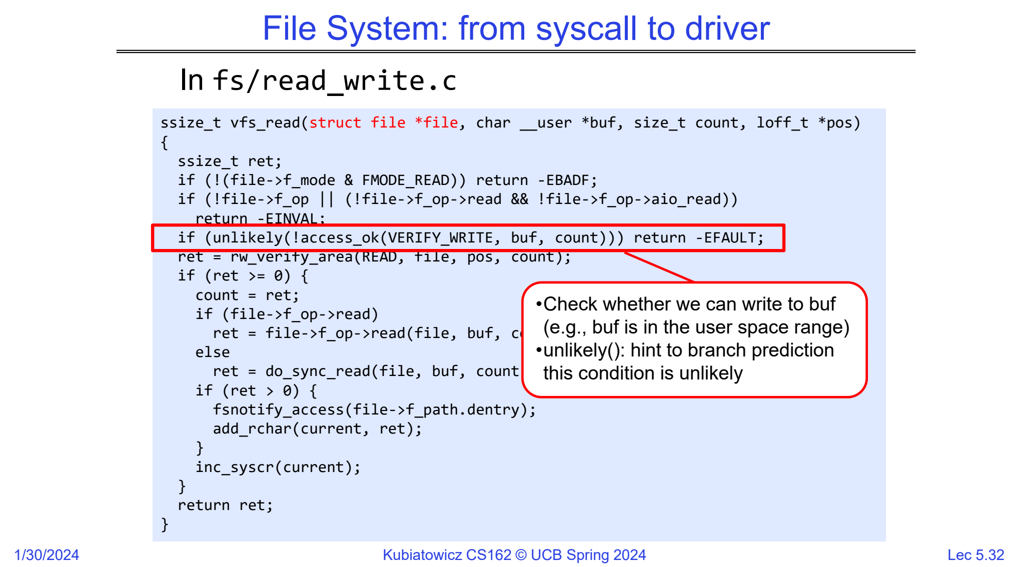


File System: from syscall to driver

In `fs/read_write.c`

```
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`

```
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, c
    else
      ret = do_sync_read(file, buf, count
    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`

```
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`

```
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`

```
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->re
    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)

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, 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 read by “current” task (for scheduling purposes)

File System: from syscall to driver

In `fs/read_write.c`

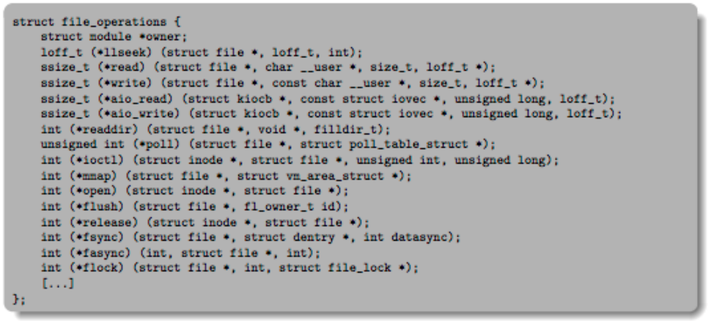
```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
  ssize_t ret;
  if (!(file->f_mode & FMODE_READ)) return -EBADF;
  if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
    return -EINVAL;
  if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -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)

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

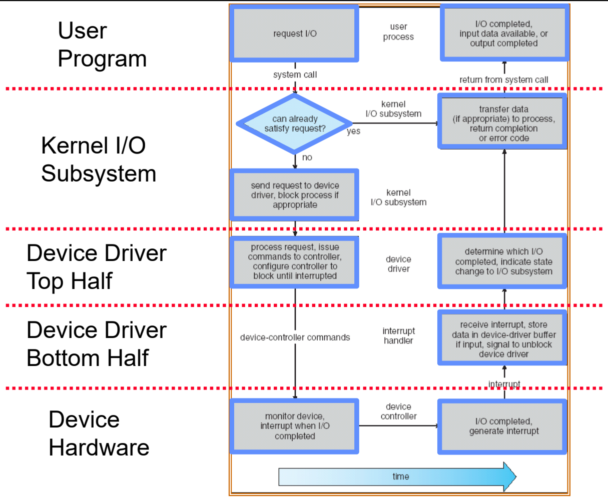
Lower Level Driver



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

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

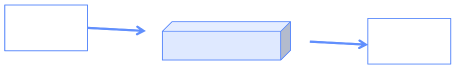
Life Cycle of An I/O Request


User Program
Kernel I/O Subsystem
Device Driver Top Half
Device Driver Bottom Half
Device Hardware
request I/O
user process
I/O completed, input data available, or output completed
system call
return from system call
can already satisfy request?
yes
no
kernel I/O subsystem
transfer data (if appropriate) to process, return completion or error code
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
interrupt
monitor device, interrupt when I/O completed
device controller
I/O completed, generate interrupt
time

Communication between processes

- Can we view files as communication channels?

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

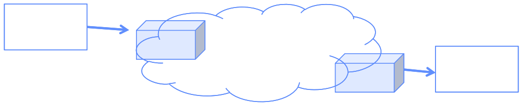


```
n = read(rfd,rbuf,rmax);
```

- 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);
```

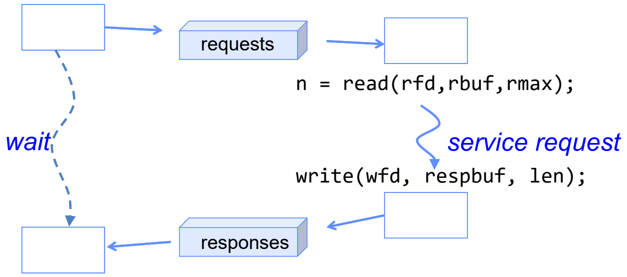


```
n = read(rfd,rbuf,rmax);
```

- 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

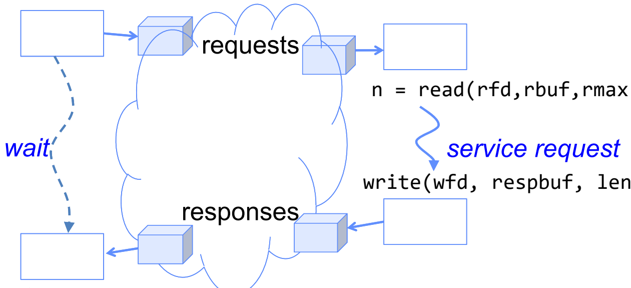
Client (issues requests) Server (performs operations)



```
write(rqfd, rqbuf, buflen);
n = read(rfd,rbuf,rmax);
write(wfd, respbuf, len);
n = read(resfd,resbuf,resmax);
```

Request Response Protocol: Across Network

Client (issues requests) Server (performs operations)

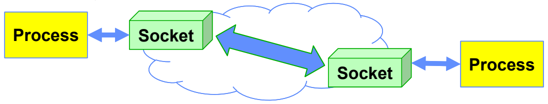


```
write(rqfd, rqbuf, buflen);
n = read(rfd,rbuf,rmax);
write(wfd, respbuf, len);
n = read(resfd,resbuf,resmax);
```

The Socket Abstraction: Endpoint for Communication

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

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



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

- 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?

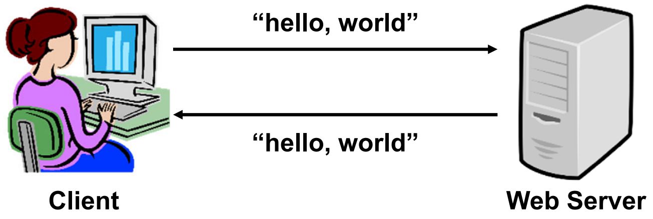
Sockets: More Details

- **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, …)

Sockets: More Details

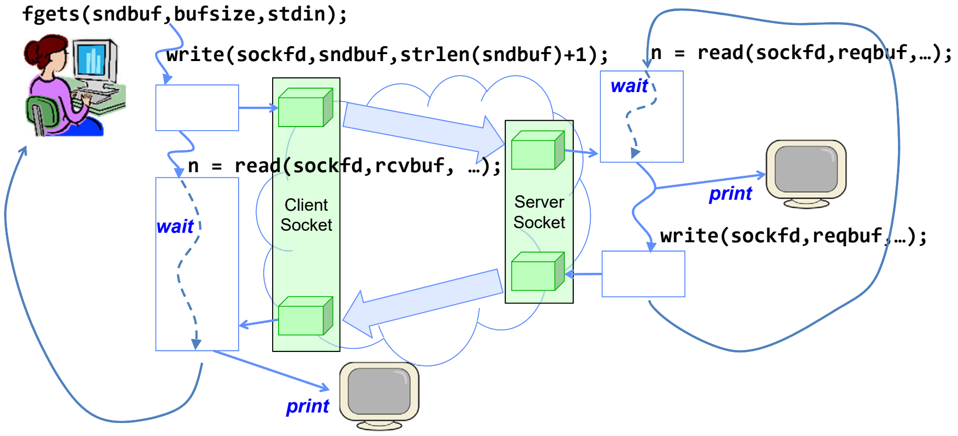
- Looks just like a file with a **file descriptor**
  - – Corresponds to a network connection (*two* queues)
  - – `write` adds to output queue (queue of data destined for other side)
  - – `read` removes from it input queue (queue of data destined for this side)
  - – Some operations do not work, e.g. `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

Simple Example: Echo Server

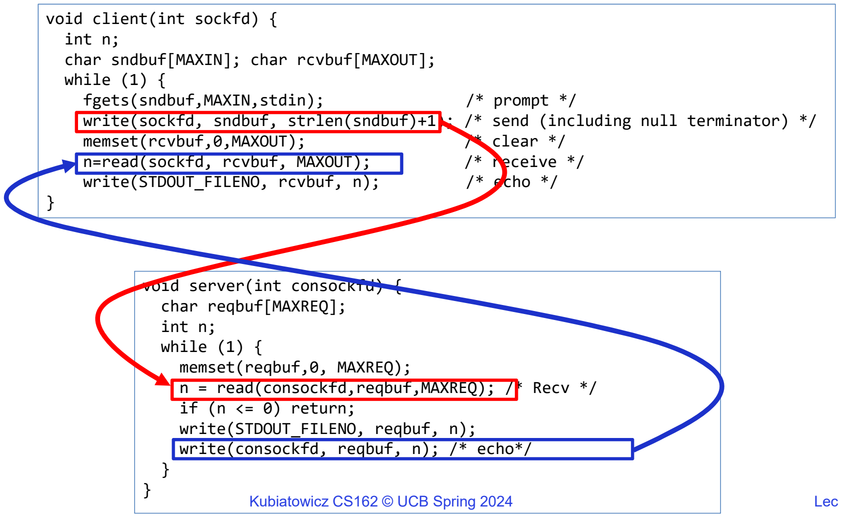


Simple Example: Echo Server

Client (issues requests) **Server (services requests)**


fgets(sndbuf,bufsize,stdin);
write(sockfd,sndbuf,strlen(sndbuf)+1);
n = read(sockfd,reqbuf,…);
wait
n = read(sockfd,rcvbuf, …);
Client Socket
Server Socket
wait
print
write(sockfd,reqbuf,…);
print

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);
    n = read(consockfd,reqbuf,MAXREQ); /* Recv */
    if (n <= 0) return;
    write(STDOUT_FILENO, reqbuf, n);
    write(consockfd, reqbuf, n); /* 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.
- 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

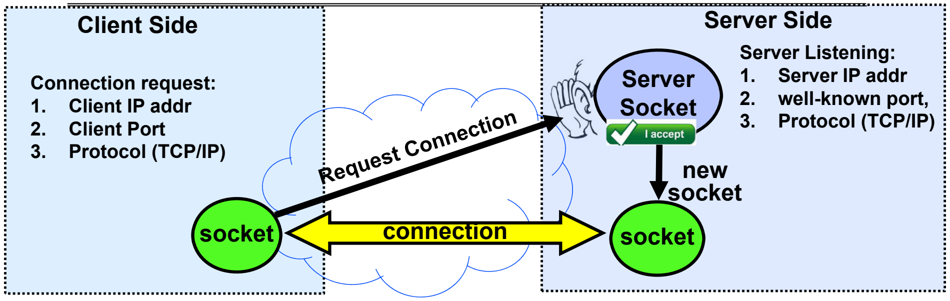
Socket Creation

- 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?

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"

Connection Setup over TCP/IP



Client Side

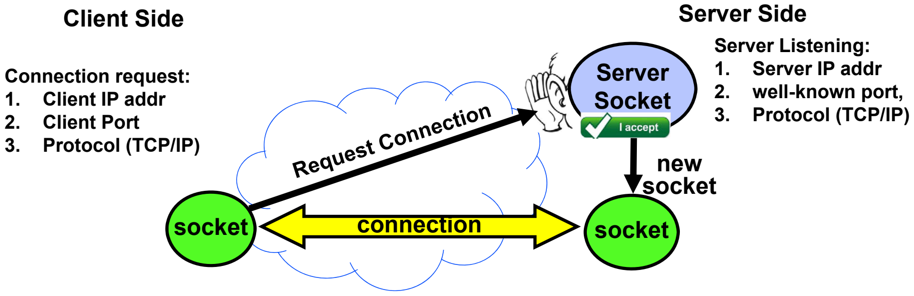
Connection request:
1. **Client IP addr**
2. **Client Port**
3. **Protocol (TCP/IP)**

Server Side

Server Listening:
1. **Server IP addr**
2. **well-known port,**
3. **Protocol (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 Setup over TCP/IP



Client Side

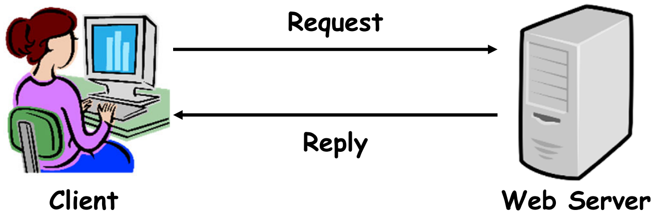
Connection request:
1. **Client IP addr**
2. **Client Port**
3. **Protocol (TCP/IP)**

Server Side

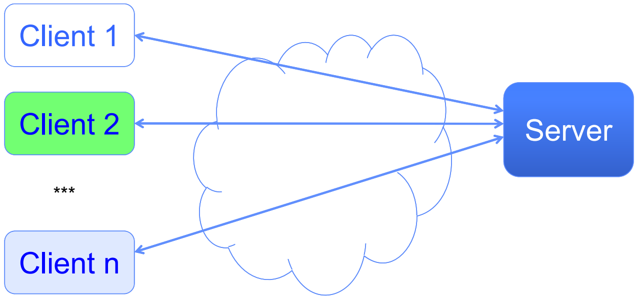
Server Listening:
1. **Server IP addr**
2. **well-known port,**
3. **Protocol (TCP/IP)**

- 5-Tuple identifies each connection:
  1. Source IP Address
  2. Destination IP Address
  3. Source Port Number
  4. Destination Port Number
  5. Protocol (always TCP here)
- 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

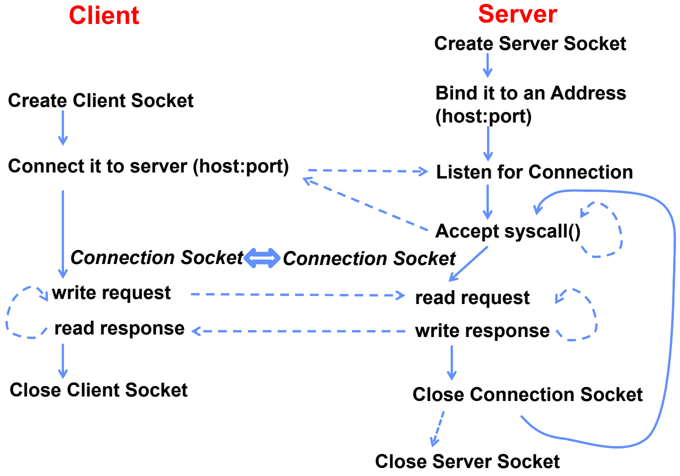


Client-Server Models



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

Simple Web Server



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

Close Server Socket

Client Code

```
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);
```

Client-Side: Getting the Server Address

```
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;         /* Includes AF_INET and AF_INET6 */
  hints.ai_socktype = SOCK_STREAM;     /* Essentially TCP/IP */

  int rv = getaddrinfo(host_name, port_name, &hints, &server);
  if (rv != 0) {
    printf("getaddrinfo failed: %s\n", gai_strerror(rv));
    return NULL;
  }
  return server;
}
```

Server Code (v1)

```
// 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 (wildcard IP), Passive

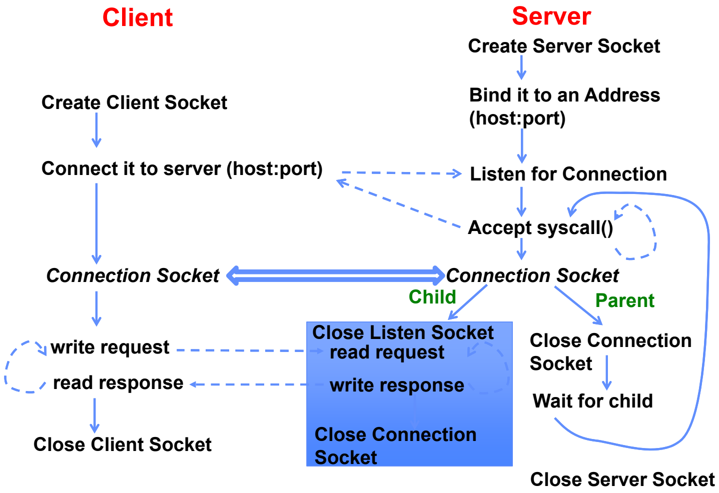
```
struct addrinfo *setup_address(char *port) {
  struct addrinfo *server;
  struct addrinfo hints;
  memset(&hints, 0, sizeof(hints));
  hints.ai_family = AF_UNSPEC;          /* Includes AF_INET and AF_INET6 */
  hints.ai_socktype = SOCK_STREAM;      /* Essentially TCP/IP */
  hints.ai_flags = AI_PASSIVE;          /* Set up for server socket */

  int rv = getaddrinfo(NULL, port, &hints, &server);  /* No address! (any local IP) */
  if (rv != 0) {
    printf("getaddrinfo failed: %s\n", gai_strerror(rv));
    return NULL;
  }
  return server;
}
```

- **Accepts any connections on the specified port**

How Could the Server Protect Itself?

- Handle each connection in a separate process
  - This will mean that the logic serving each request will be "sandboxed" away from the main server process
- In the following code, keep in mind:
  - `fork()` will duplicate *all* of the parent's file descriptors (i.e. pointers to sockets!)
  - We keep control over accepting new connections in the parent
  - New child connection for each remote client

Server With Protection (each connection has own process)



Client

Server

Create Server Socket

Create Client Socket

Bind it to an Address (host:port)

Connect it to server (host:port)

Listen for Connection

Accept syscall()

Connection Socket

Connection Socket

Child

Parent

write request

Close Listen Socket

read request

Close Connection Socket

read response

write response

Wait for child

Close Client Socket

Close Connection Socket

Close Server Socket

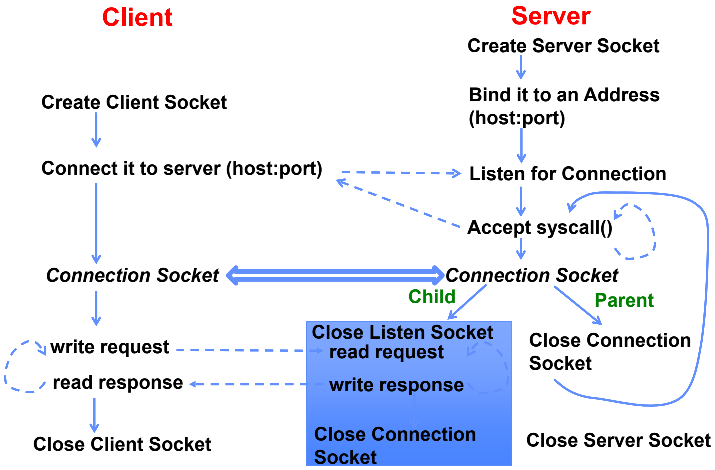
Server Code (v2)

```
// Socket setup code elided…
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(0);
  } else {
    close(conn_socket);
    wait(NULL);
  }
}
close(server_socket);
```

How to make a Concurrent Server

- So far, in the server:
  - Listen will queue requests
  - Buffering present elsewhere
  - But server *waits* for each connection to terminate before servicing the next
    - This is the standard shell pattern
- A concurrent server can handle and service a new connection before the previous client disconnects
  - Simple – just don't wait in parent!
  - Perhaps not so simple – multiple child processes better not have data races with one another through file system/etc!

Server With Protection and Concurrency



Client

Server

Create Server Socket

Bind it to an Address (host:port)

Create Client Socket

Connect it to server (host:port)

Listen for Connection

Accept syscall()

Connection Socket

Connection Socket

Child

Parent

Close Listen Socket

read request

write response

Close Connection Socket

write request

read response

Close Connection Socket

Close Client Socket

Close Server Socket

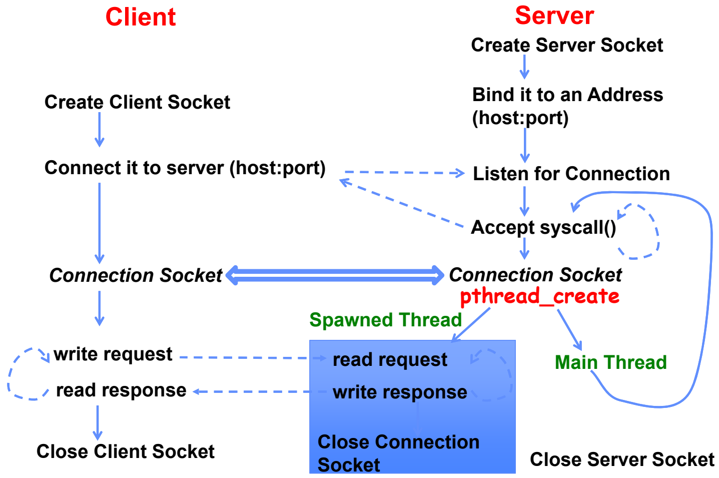
Server Code (v3)

```
// Socket setup code elided…
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(0);
  } else {
    close(conn_socket);
    //wait(NULL);
  }
}
close(server_socket);
```

Faster Concurrent Server (without Protection)

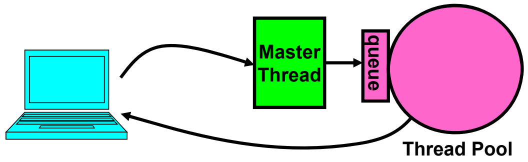
- Spawn a new *thread* to handle each connection
  - Lower overhead spawning process (less to do)
- Main *thread* initiates new client connections without waiting for previously spawned threads
- Why give up the protection of separate processes?
  - More efficient to create new threads
  - More efficient to switch between threads
- Even more potential for data races (need synchronization?)
  - Through shared memory structures
  - Through file system

Server with Concurrency, without Protection



Thread Pools: More Later!

- Problem with previous version: Unbounded Threads
  - When web-site becomes too popular – throughput sinks
- Instead, allocate a bounded "pool" of worker threads, representing the maximum level of multiprogramming



```
master() {
   allocThreads(worker,queue);
   while(TRUE) {
      con=AcceptCon();
      Enqueue(queue,con);
      wakeUp(queue);
   }
}
```

```
worker(queue) {
   while(TRUE) {
      con=Dequeue(queue);
      if (con==null)
         sleepOn(queue);
      else
         ServiceWebPage(con);
   }
}
```

Conclusion

- POSIX I/O
  - Everything is a file!
  - Based on the system calls **open()**, **read()**, **write()**, and **close()**
- Streaming IO: modeled as a stream of bytes
  - Most streaming I/O functions start with "f" (like "fread")
  - Data buffered automatically by C-library function
- Low-level I/O:
  - File descriptors are integers
  - Low-level I/O supported directly at system call level
- 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 (IPC mechanism)