Introduction to File Systems

David E. Culler

CS162 – Operating Systems and Systems Programming

Lecture 5

Sept 12, 2019

Reading: A&D 5.8, 11.1-2
HW 1 due 9/18
Proj 1 Design Doc 9/17

Objective of this lecture

- Resolve tension in understanding Threads

- Show how Operating System functionality distributes across layers in the system.
- Introduce I/O & storage services – i.e., file systems

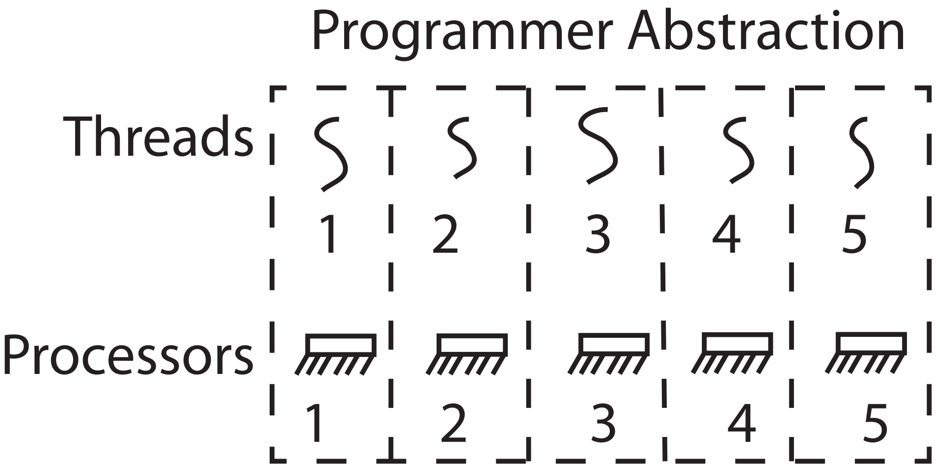
Review: Threads

- Independently schedulable entity
- Sequential thread of execution that runs concurrently with other threads
  - It can block waiting for something while others progress
  - It can work in parallel with others (ala cs61c)
- Has local state (its stack) and shared (static data and heap)
- In the absence of synchronization operations, arbitrary interleaving of threads may occur

Recall: Thread State

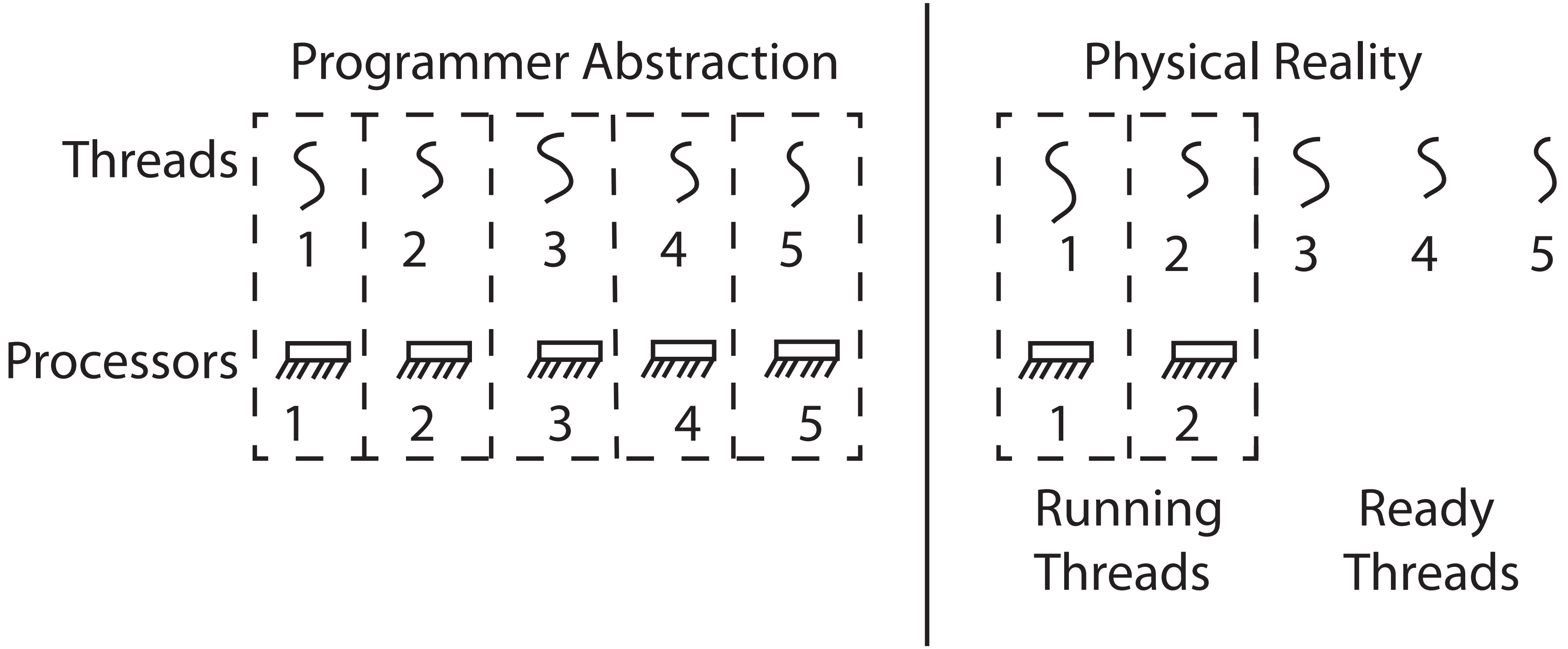
- State shared by all threads in process/addr space
  - Content of memory (global variables, heap)
  - I/O state (file system, network connections, etc)
- Execution Stack (logically private to thread)
  - Parameters, temporary variables
  - Return PCs while called procedures are executing
- State for each thread
  - CPU registers (including, program counter)
  - Ptr to Execution stack
  - Kept in Thread Control Block, when thread not running
- Scheduler works on TCBs

Recall: Thread Abstraction



- Illusion: Infinite number of processors

Recall: Thread Abstraction



- Illusion: Infinite number of processors
- Reality: Threads execute with variable "speed"
  - Programs must be designed to work with any schedule

Recall: Synchronization

- **Mutual Exclusion:** Ensuring only one thread does a particular thing at a time (one thread *excludes* the others)
- **Critical Section:** Code exactly one thread can execute at once
  - Result of mutual exclusion
- **Lock:** An object only one thread can hold at a time
  - **Provides** mutual exclusion
- Offers two **atomic** operations:
  - `Lock.Acquire()` – wait until lock is free; then grab
  - `Lock.Release()` – Unlock, wake up waiters
- Need other tools for "cooperation"
  - e.g., Java monitors, semaphores, condition variables)

Little Example: Stack of Strings (SoS)

```
struct str_lst_elem {
  char *str;
  struct str_lst_elem *next;
};

struct str_lst {
  struct str_lst_elem *head;

};

void str_lst_init(struct str_lst *lst) {
  lst->head = NULL;

};
```

SoS (cont)

```
void str_lst_push(struct str_lst *lst, char *str) {
  struct str_lst_elem *new_elem = malloc(sizeof(struct str_lst_elem));
  new_elem->str = str;

  new_elem->next = lst->head;
  lst->head = new_elem;

};
```

Must be atomic if multiple threads

```
char *str_lst_pop(struct str_lst *lst) {
  char *topval;

  struct str_lst_elem *top = lst->head;
  if (!top) {
    topval = NULL;
  } else {
    topval = top->str;
    lst->head = top->next;
  }

  return topval;
};
```

Must be atomic if multiple threads

Thread Safe: Stack of Strings

```
struct str_lst_elem {
  char *str;
  struct str_lst_elem *next;
};

struct str_lst {
  struct str_lst_elem *head;
  pthread_mutex_t lock;
};

void str_lst_init(struct str_lst *lst) {
  lst->head = NULL;
  pthread_mutex_init(&lst->lock, NULL);
};
```

Thread safe: SoS (cont)

```
void str_lst_push(struct str_lst *lst, char *str) {
  struct str_lst_elem *new_elem = malloc(sizeof(struct str_lst_elem));
  new_elem->str = str;
  pthread_mutex_lock (&lst->lock);
  new_elem->next = lst->head;
  lst->head = new_elem;
  pthread_mutex_unlock (&lst->lock);
};
```

Critical Section

```
char *str_lst_pop(struct str_lst *lst) {
  char *topval;
  pthread_mutex_lock (&lst->lock);
  struct str_lst_elem *top = lst->head;
  if (!top) {
    topval = NULL;
  } else {
    topval = top->str;
    lst->head = top->next;
  }
  pthread_mutex_unlock (&lst->lock);
  return topval;
};
```

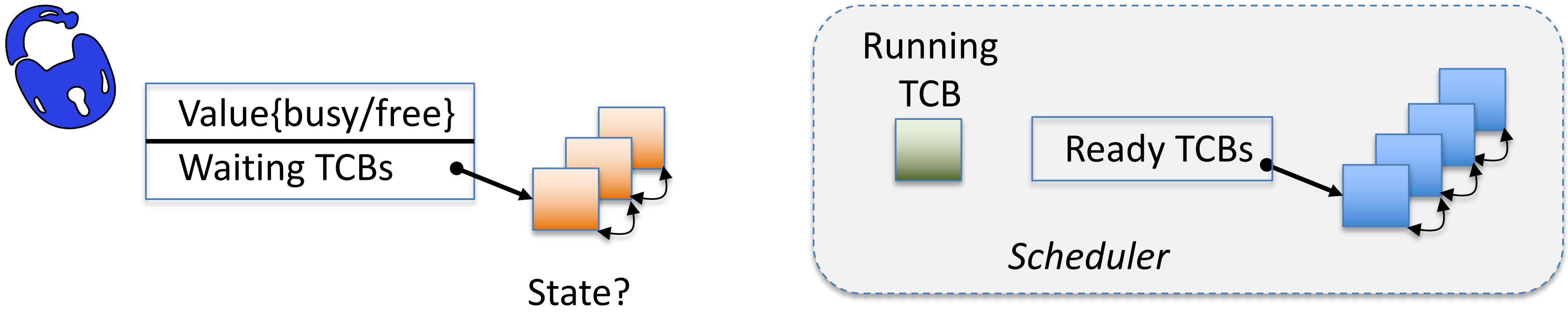
Critical Section

Lock Implementation Conundrum

Lock Implementation Conundrum

- Manipulating a data structure, like a list, requires a sequence of operations that must be atomic
- To make the list thread safe, protect it with a lock
  - Operations on the list are a critical section
  - i.e., lock; manipulate; unlock *(lots of places in the kernel)*
- Lock implementation needs to manipulate lists (of TCBs)
  - Thread that tries to acquire a busy lock is placed on the list of threads waiting on the lock (!!!) – and some other thread scheduled
  - Releasing a lock causes a thread to be removed from the lock's list and placed on the scheduler's list of ready threads
- How do we create critical sections for the lock acquire/release operations themselves ???
- We disable interrupts so no other thread can interleave with this kernel code

Basic Lock Implementation



```
Acquire(*lock) {
   disable interrupts;
   if (lock->value == BUSY) {
      put thread on lock's wait_Q
      "i.e, Go to sleep"
      allow a ready thread to run
   } else {
      lock->value = BUSY;
   }
   enable interrupts;
}
```

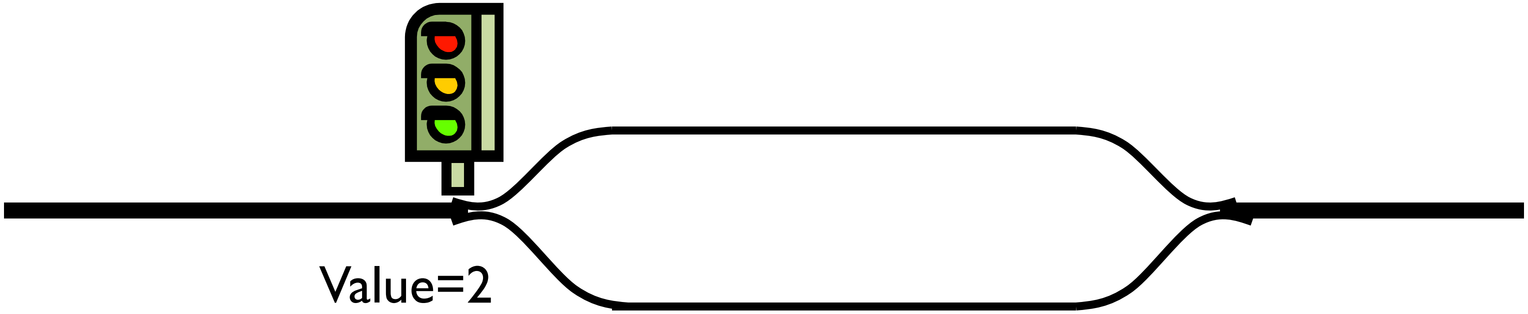
```
Release(*lock) {
   disable interrupts;
   if (any TCB on lock wait_Q) {
      "i.e., lock busy";
      take thread off wait queue
      Place on ready queue;
   } else {
      lock->value = FREE;
   }
   enable interrupts;
}
```

Is that all?

- There are further subtleties about when interrupts are re-enabled
  - We’ll tackle this later, as we get closer to Project 2
- The low level mechanics of thread switch are so simple and subtle that it may still seem like magic
- Still a question of how much of “threads” and ”synchronization” could be moved out of the kernel to user level (for performance).
  - later

Recall: Semaphores

- No negative values
- Only operations allowed are P and V
  - can’t read or write value, except to set it initially
- Operations must be atomic
  - Two P’s together can’t decrement value below zero
  - Similarly, thread going to sleep in P won’t miss wakeup from V – even if they both happen at same time
- Semaphore from railway analogy
  - Here is a semaphore initialized to 2 for resource control:



Recall: Important Semaphore Patterns

- **Mutual Exclusion:** (Like lock)
  - Called a "binary semaphore"

```
initial value of semaphore = 1;
semaphore.down();
// Critical section goes here
semaphore.up();
```

- **Signaling** other threads, e.g. **ThreadJoin**

```
Initial value of semaphore = 0

ThreadJoin {
    semaphore.down();
}
```

```
ThreadFinish {
    semaphore.up();
}
```

Intuition for Semaphores

- What do you need to wait for?
  - Example: Critical section to be finished
  - Example: Queue to be non-empty, or no longer full
  - Example: Some thread to be done with something
- What can you count that will be 0 when you need to wait?
  - Example: # of threads currently in critical section
  - Example: # of items currently in queue
  - Example: # of free slots in array
  - Example: status of 1 for still active
- Can use semaphore operations to maintain count

So what's in our PCB now?

- Process ID, name, etc
- Thread object(s) – TCBs
  - Place to save registers when not running
  - Thread status
  - Links to form lists
- Thread Stack
- Lock object for *any lock used by its kernel thread*
- User level lock info ???

Processes and Threads

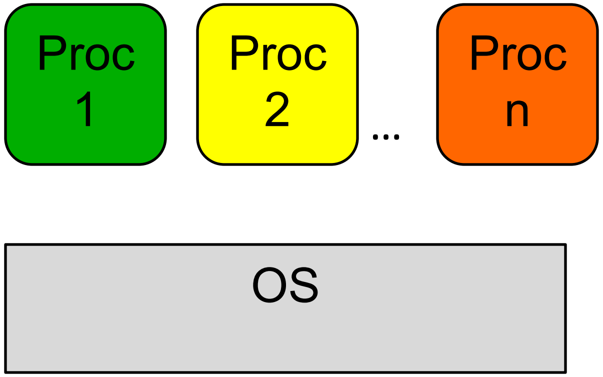
- ???

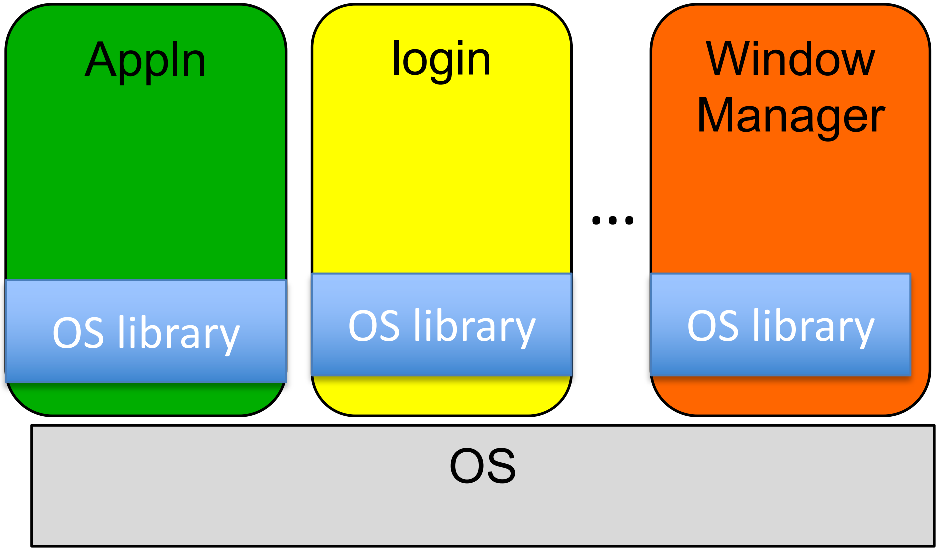
Processes issue syscalls …

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

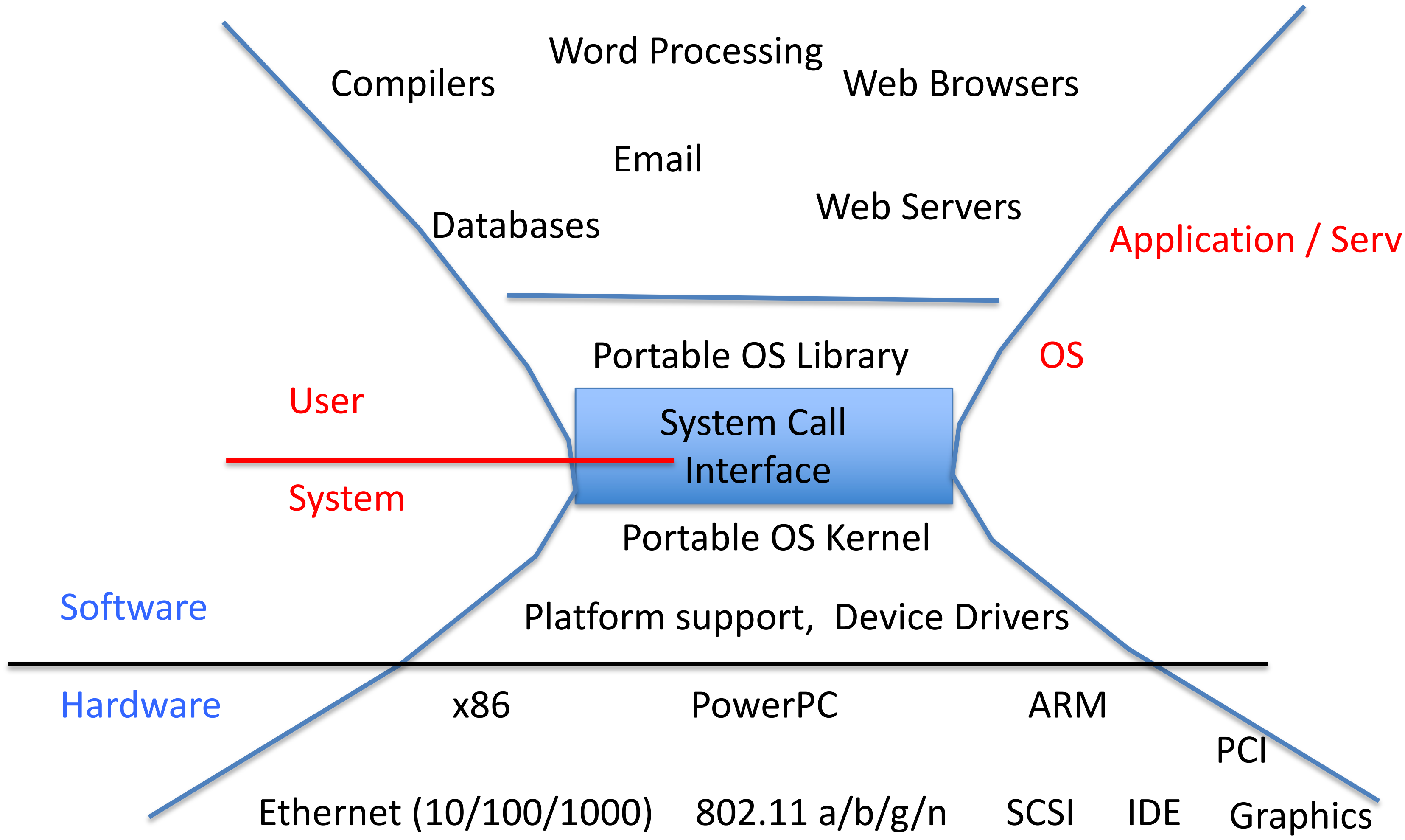
- That’s right.
- It was buried in the programming language runtime library (e.g., libc.a)
- … Layering

OS run-time library





Recall: A Kind of Narrow Waist



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()`**

POSIX I/O Design Patterns

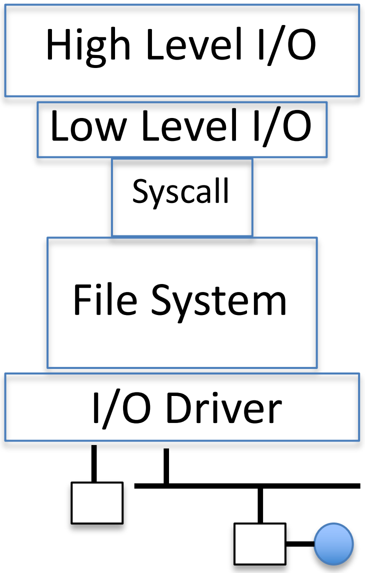
- Open before use
  - Access control check, setup happens here
- Byte-oriented
  - Least common denominator
  - OS responsible for hiding the fact that real devices may not work this way (e.g. hard drive stores data in blocks)
- Explicit close

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

I/O & Storage Layers

Application / Service



High Level I/O

Low Level I/O

Syscall

File System

I/O Driver

streams

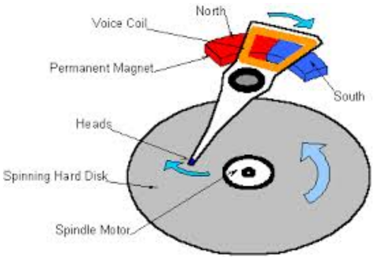
handles

registers

descriptors

Commands and Data Transfers

Disks, Flash, Controllers, DMA



The file system abstraction

- 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
    - Basis for access control
- Directory
  - "Folder" containing files & Directories
  - Hierachical (graphical) naming
    - Path through the directory graph
    - Uniquely identifies a file or directory
      - /home/ff/cs162/public_html/fa14/index.html
  - Links and Volumes (later)

C high level File API – streams (review)

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

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

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

Don’t forget to flush

Connecting Processes, Filesystem, and Users

- Process has a ‘current working directory’
- Absolute Paths
  - /home/oski/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 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 – stream ops

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

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

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

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

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

C Streams: char by char I/O

```
#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?

```
#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, ...);
```

stdio Block-by-Block I/O

```
#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);
}
```

stdio Block-by-Block I/O

```
#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

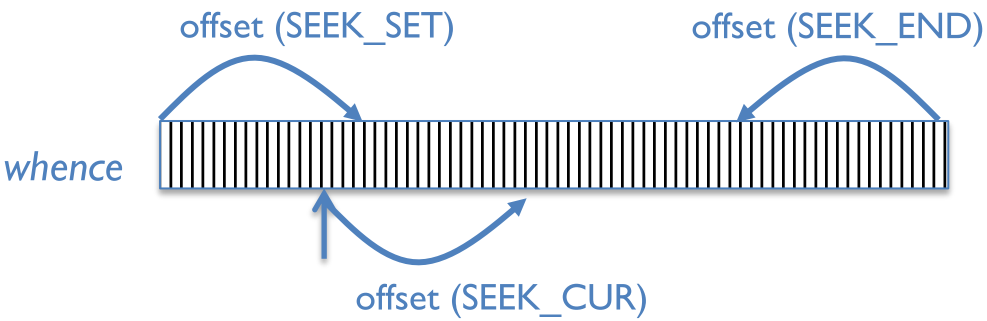
- Systems programmers are paranoid
- We should really be writing things like:

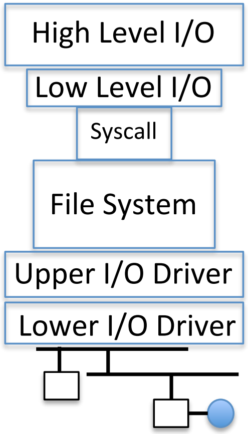
```
FILE* input = fopen(“input.txt”, “r”);
if (input == NULL) {
  // Prints our string and error msg.
  perror(“Failed to open input file”)
}
```

- Be **thorough about checking return values**
  - Want failures to be systematically caught and dealt with

C Stream API: Positioning

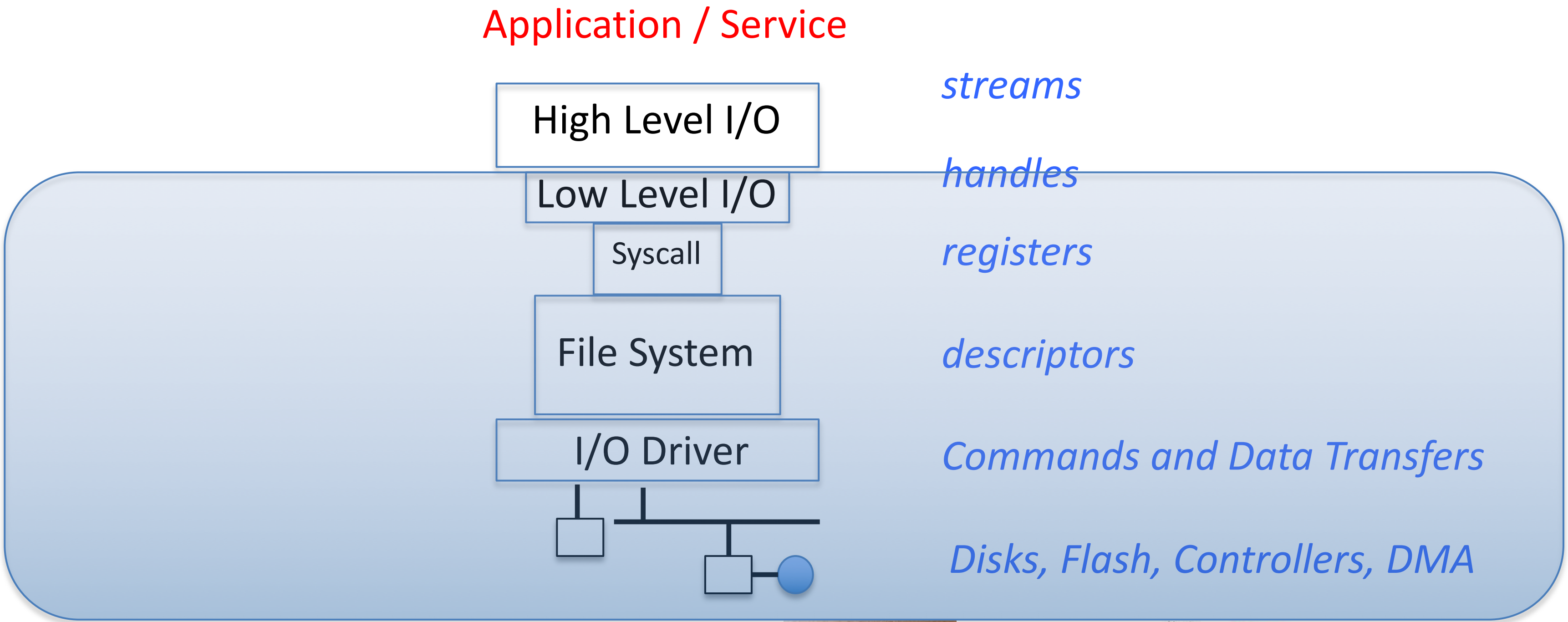
```
int fseek(FILE *stream, long int offset, int whence);

long int ftell (FILE *stream)

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

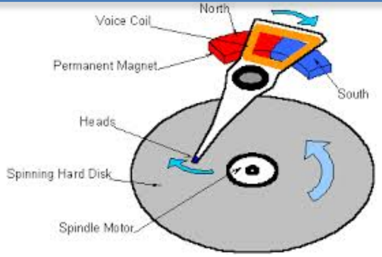
handles

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

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

C Low Level: standard descriptors

```
#include <unistd.h>

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

int fileno (FILE *stream)

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

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

C Low Level Operations

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

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

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

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

A little example: lowio.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

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

Another: lowio-std.c

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

#define BUFSIZE 1024

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

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

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

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

  exit(0);
}
```

Low-Level I/O: Example

```
#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);
}
```

Low-Level I/O: Other Operations

- Operations specific to terminals, devices, networking, …
- Duplicating descriptors
  - **`int dup2(int old, int new);`**
  - **`int dup(int old);`**
- `Pipes – bi-directional channel`
  - **`int pipe(int fileds[2]);`**
  - `Writes to fileds[1] read from fileds[0]`
- File Locking
- Memory-Mapping Files
- Asynchronous I/O

Little pipe example

```
#include <unistd.h>

#define BUFSIZE 1024
enum PipeSel {rd_pipe = 0, wt_pipe = 1};

int main(int argc, char *argv[])
{
  char *msg = "Message in a pipe.\n";
  char buf[BUFSIZE];
  int pipe_fd[2];
  if (pipe(pipe_fd)) {
      fprintf (stderr, "Pipe failed.\n"); return EXIT_FAILURE;
    }
  ssize_t writelen = write(pipe_fd[wt_pipe], msg, strlen(msg)+1);
  printf("Sent: %s [%ld, %ld]\n", msg, strlen(msg)+1, writelen);

  ssize_t readlen  = read(pipe_fd[rd_pipe], buf, BUFSIZE);
  printf("Rcvd: %s [%ld]\n", msg, readlen);
  close(pipe_fd[wt_pipe]);
  close(pipe_fd[rd_pipe]);
}
```

Inter-Process Communication (IPC)

- One process reads a file the other writes, or …

```
pid_t pid = fork();
if (pid < 0) {
  fprintf (stderr, "Fork failed.\n");
  return EXIT_FAILURE;
}
if (pid != 0) {
  ssize_t writelen = write(pipe_fd[wt_pipe], msg, msglen);
  printf("Parent: %s [%ld, %ld]\n", msg, msglen, writelen);
  close(pipe_fd[wt_pipe]);
} else {
  ssize_t readlen  = read(pipe_fd[rd_pipe], buf, BUFSIZE);
  printf("Child Rcvd: %s [%ld]\n", msg, readlen);
  close(pipe_fd[rd_pipe]);
}
```

Streams vs. File Descriptors

- 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

Why Buffer in Userspace? Overhead!

- Avoid system call overhead
  - Time to copy registers, transition to kernel mode, jump to system call handler, etc.

- Minimum syscall time: ~100s of nanoseconds
  - Read/write a file byte by byte?
  - Max throughput of **~10MB/second**
  - With **`fgetc`**? Keeps up with your SSD

Why Buffer in Userspace? Functionality.

- System call operations less capable
  - Simplifies operating system

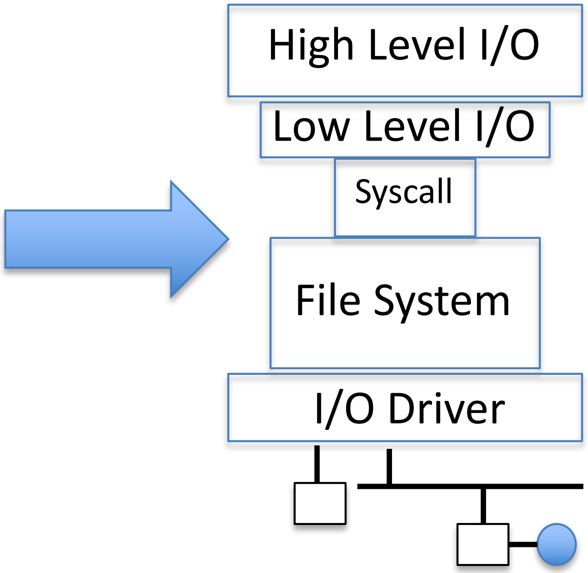
- Example: No "read until new line" operation
  - Solution: Make a big read syscall, find first new line in userspace
  - Could simulate by one syscall per character, but we already know this is a bad idea

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

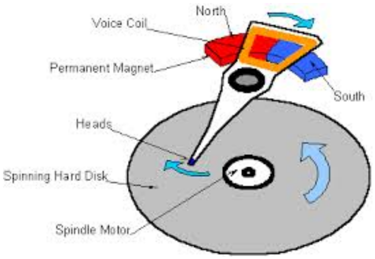
handles

registers

descriptors

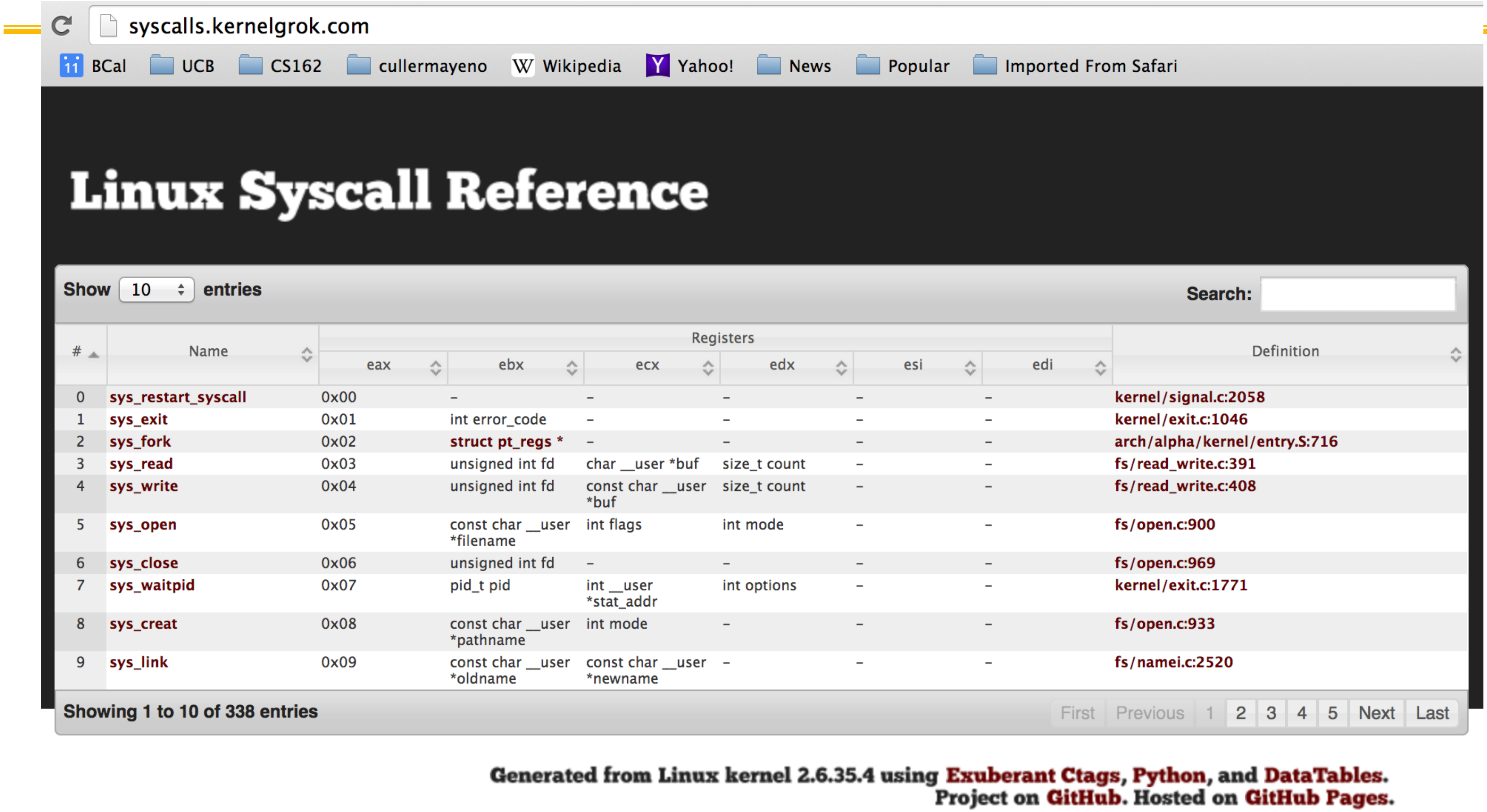
Commands and Data Transfers

Disks, Flash, Controllers, DMA



Break

Recall: SYSCALL



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

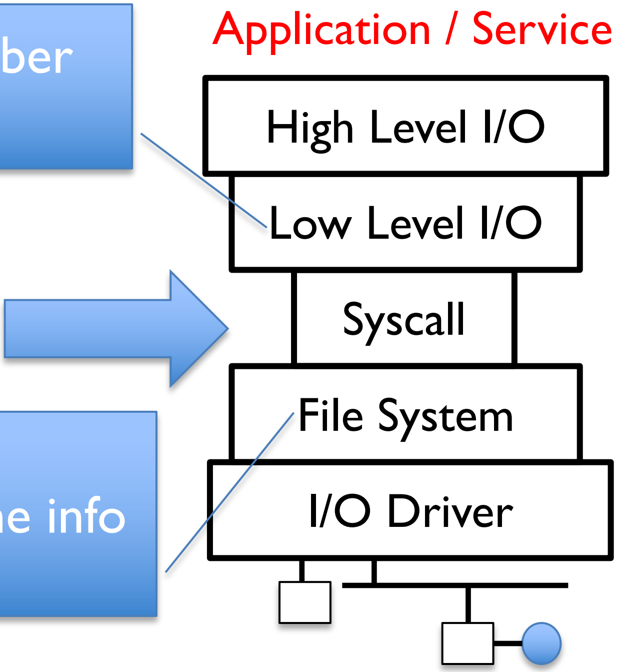
Showing 1 to 10 of 338 entries

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



Application / Service

High Level I/O

Low Level I/O

Syscall

File System

I/O Driver

streams

handles

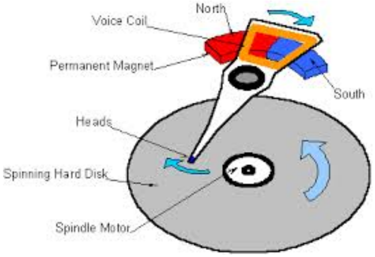
registers

descriptors

Commands and Data Transfers

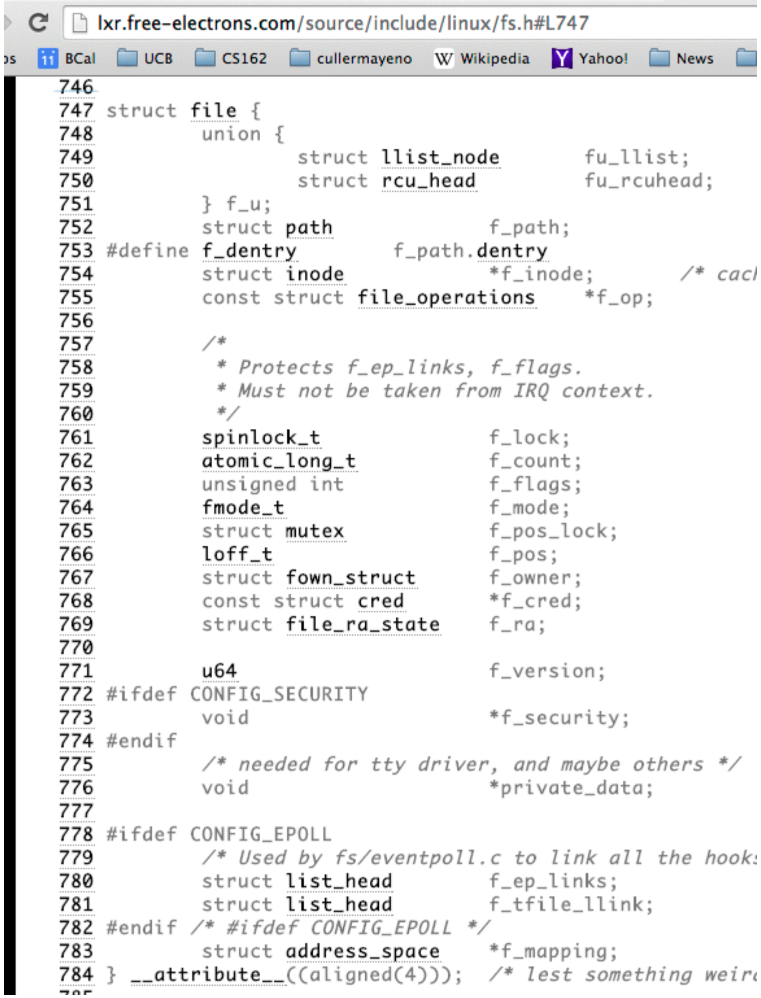
Disks, Flash, Controllers, DMA

File 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



```
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;       /* cac
        const struct file_operations    *f_op;

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

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

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

File System: from syscall to driver

In fs/read_write.c

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

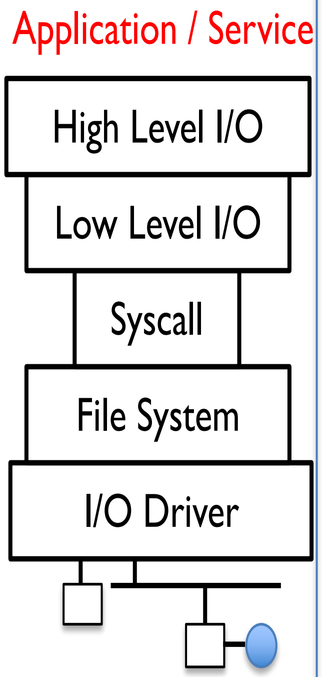
Layer by layer

User App

```
length = read(input_fd, buffer, BUFFER_SIZE);
```

User library

```
ssize_t read(int, void *, size_t){
    marshal args into registers
    issue syscall
    register result of syscall to rtn value
};
```



Exception U→K, interrupt processing

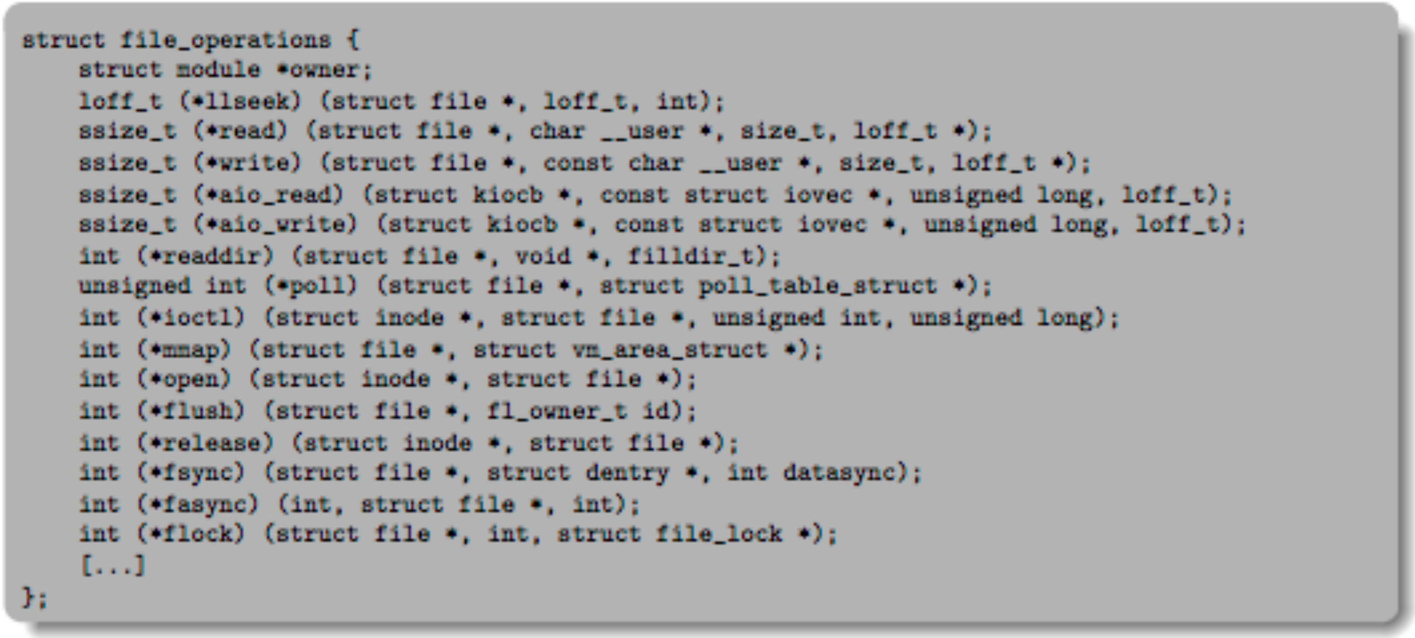
```
Void syscall_handler (struct intr_frame *f) {
    unmarshall call#, args from regs
    dispatch : handlers[call#](args)
    marshal results fo syscall ret
}
```

```
ssize_t vfs_read(struct file *file, char
__user *buf, size_t count, loff_t *pos)
{
        UserProcess/File System relationship
        call device driver to do the work
}
```

Device Driver

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

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

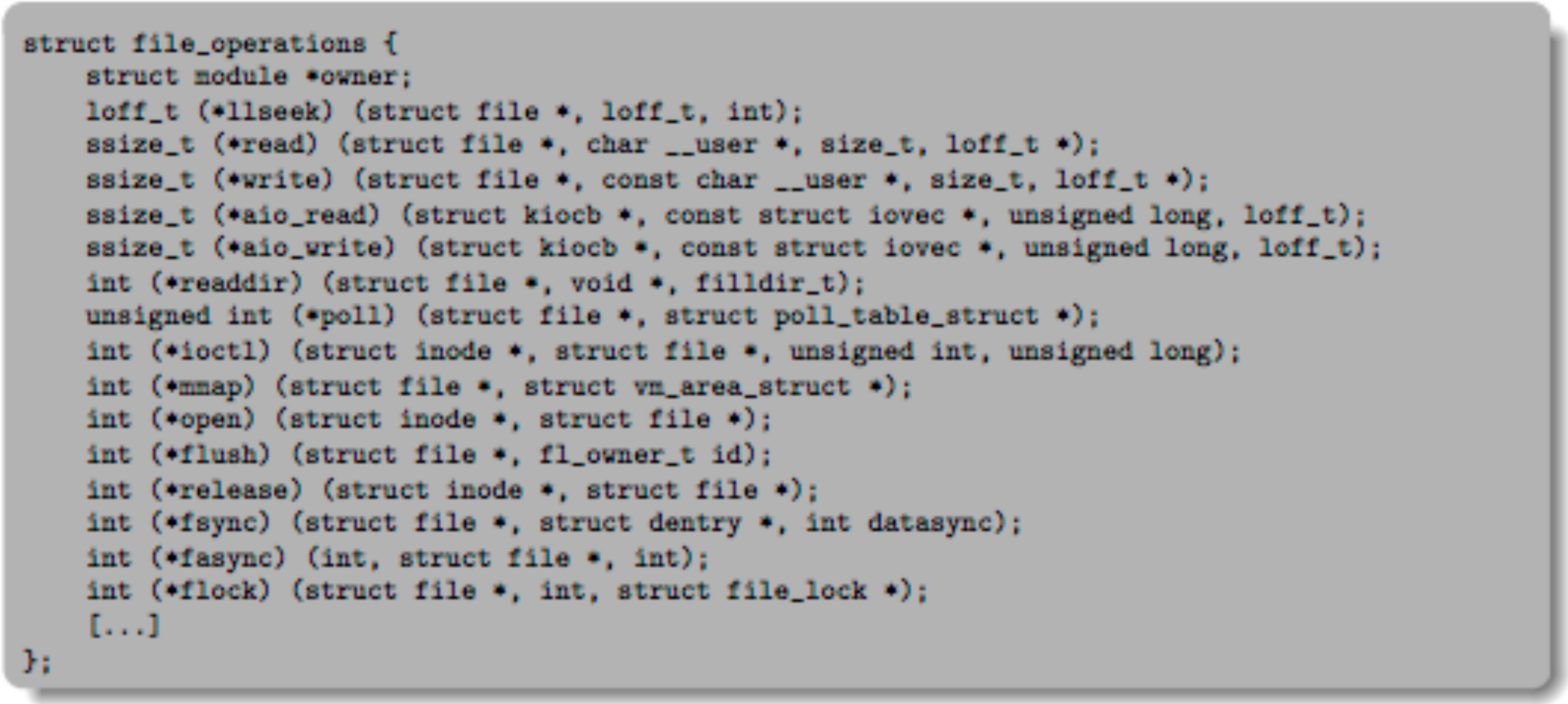
In fs/read_write.c

```
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)

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

Life Cycle of An I/O Request

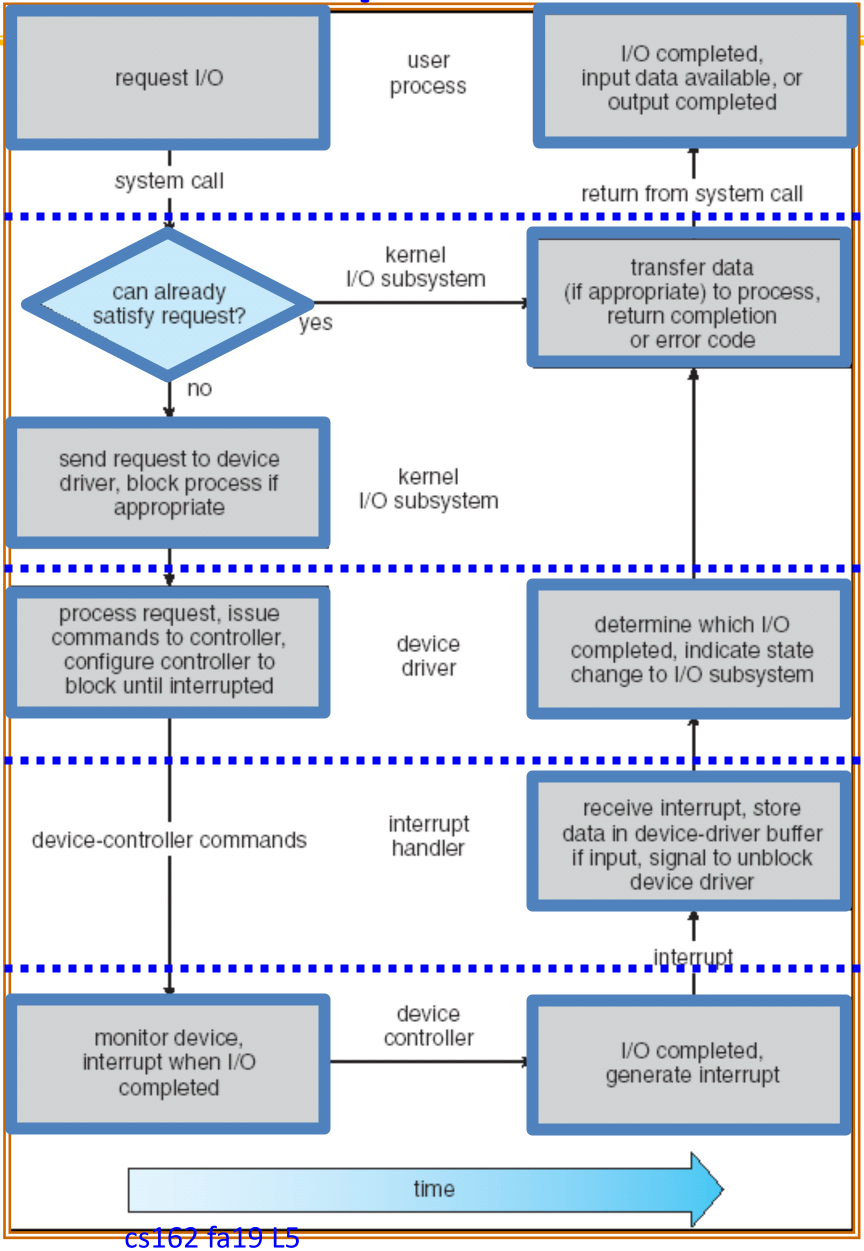
User Program

Kernel I/O Subsystem

Device Driver Top Half

Device Driver Bottom Half

Device Hardware



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

So what's in our PCB now?

- Process ID, name, etc
- Thread object(s) – TCBs
  - Place to save registers when not running
  - Thread status, Links to form lists for scheduling
- Thread Stack
- Lock object for *any lock used by its kernel thread*
  - User level lock info (if multithreaded processes)
- Current working directory
- File Descriptors/Handles for open files

BIG OS Concepts so far

- Processes
- Address Space
- Protection
- Dual Mode
- Interrupt handlers (including syscall and trap)
- Threads
- Synchronization Operations
- File System
  - Integrates processes, users, cwd, protection
- Key Layers: OS Lib, Syscall, Subsystem, Driver
  - User handler on OS descriptors
- Process control
  - fork, wait, signal, exec