CS162
Operating Systems and Systems Programming
Lecture 5

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

February 3rd, 2026

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

Basic UNIX Philosophy

Everything is a "File"

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

The File System Abstraction

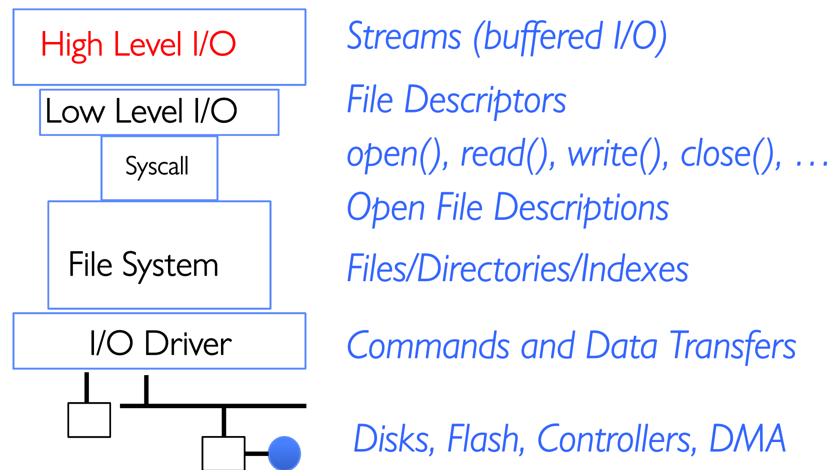
- File
  - Named collection of data in a file system
  - POSIX File data: sequence of bytes
    - » Could be text, binary, serialized objects, …
  - File Metadata: information about the file
    - » Size, Modification Time, Owner, Security info, Access control
- Directory
  - "Folder" containing files & directories
  - 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)

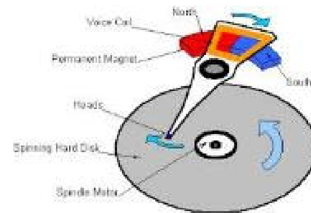
Connecting Processes, File Systems, and Users

- 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

I/O and Storage Layers

Application / Service





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(c, output);
    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:

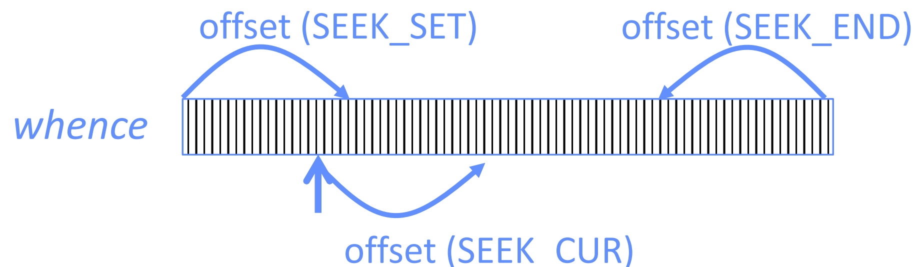
```
FILE* input = fopen(“input.txt”, “r”);
if (input == NULL) {
  // Prints our string and error msg.
  perror(“Failed to open input file”)
}
```

- 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



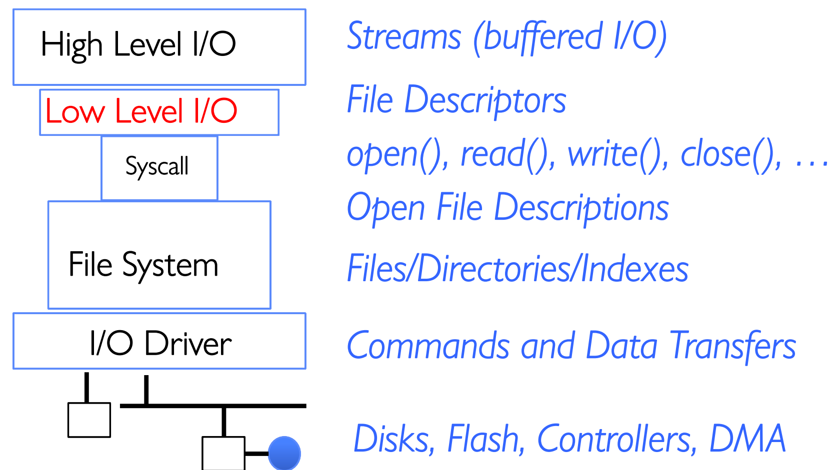
- Overall preserves high-level abstraction of a uniform stream of objects

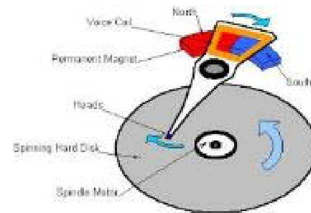
Administrivia

- Kubiatowicz Office Hours (673 Soda Hall):
  - 1pm-2pm, 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 2/24)
  - I *think* that the cs152 conflict has been resolved with a cs152 conflict time.

I/O and Storage Layers

Application / Service





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, -1 ⇒ error
  - might not write all bytes, so may need to use a loop to send all bytes

- 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");
```

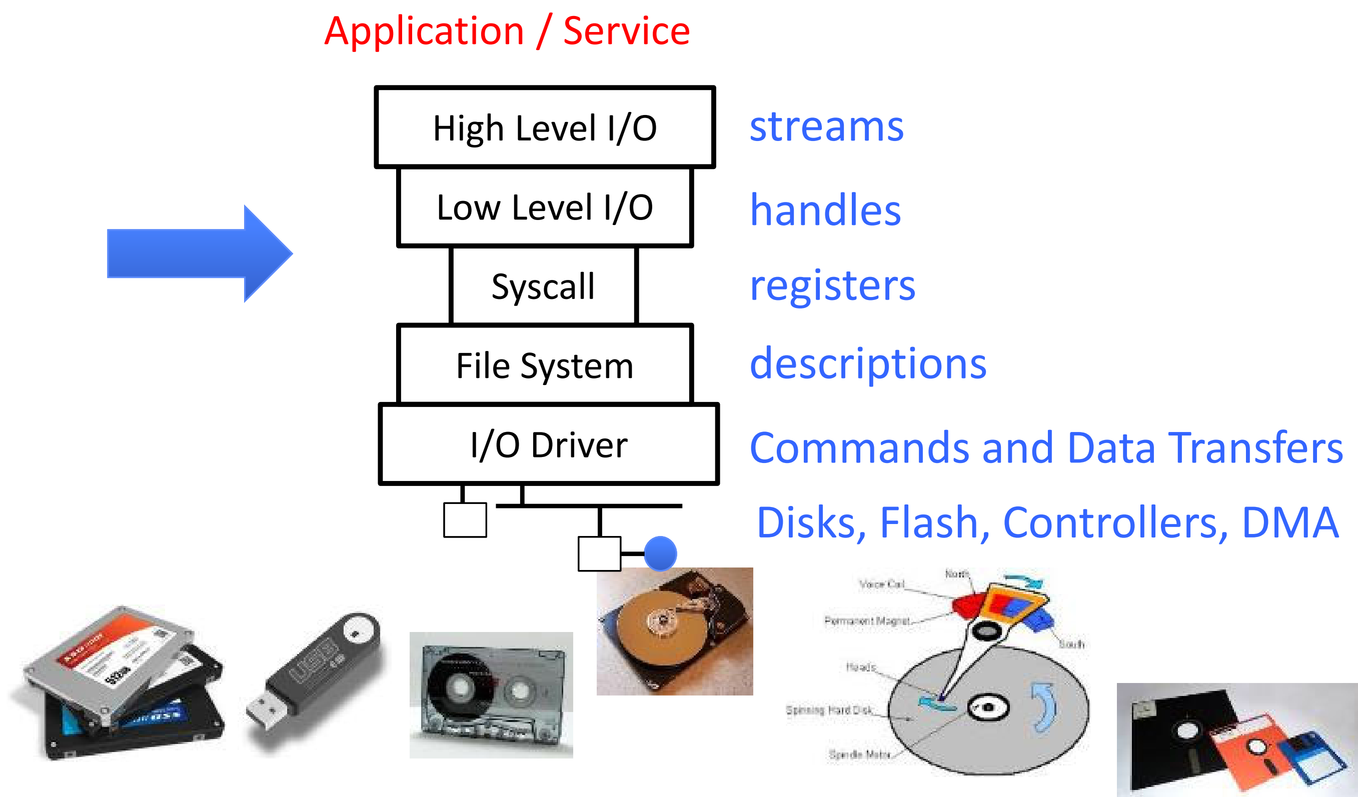
  Prints out everything at once

- 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 ??



SYSCALL

Linux Syscall Reference

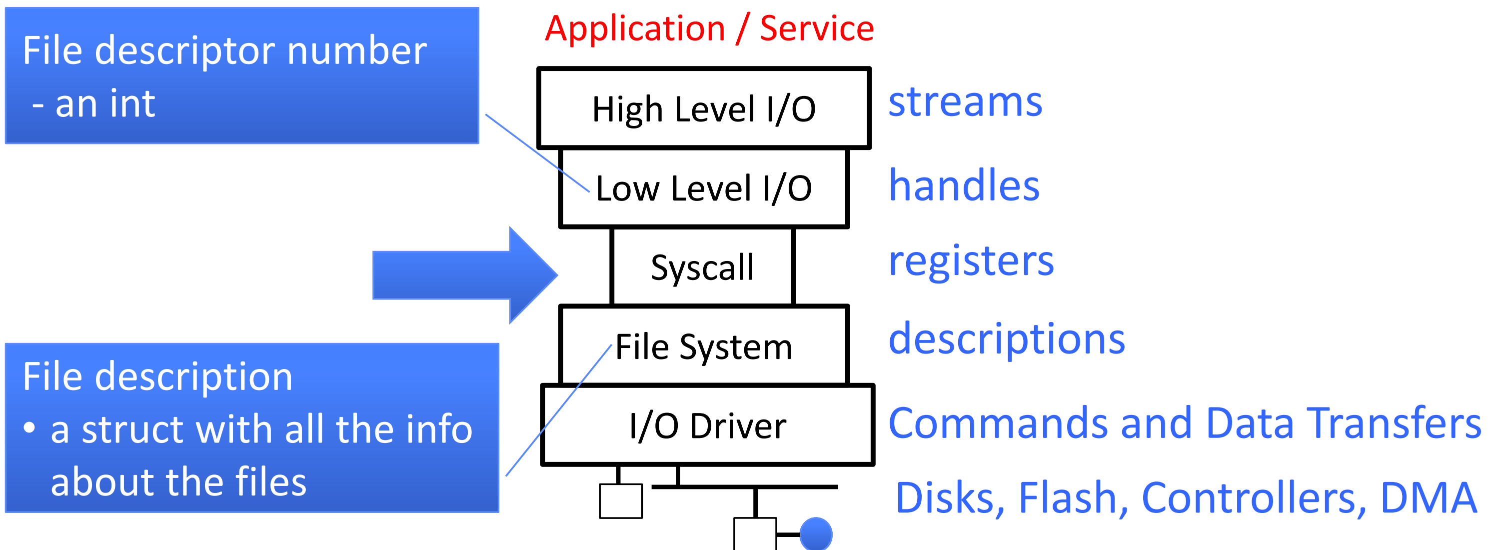
Show 10 entries | Search:

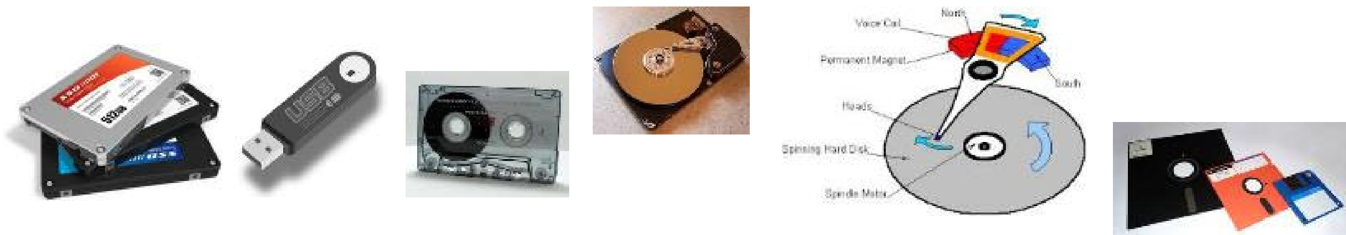
#	Name	Registers: eax	ebx	ecx	edx	esi	edi	Definition
0	sys_restart_syscall	0x00	-	-	-	-	-	kernel/signal.c:2058
1	sys_exit	0x01	int error_code	-	-	-	-	kernel/exit.c:1046
2	sys_fork	0x02	struct pt_regs *					arch/alpha/kernel/entry.S:716
3	sys_read	0x03	unsigned int fd	char __user *buf	size_t count	-	-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

Showing 1 to 10 of 338 entries — First Previous 1 2 3 4 5 Next Last

- 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 ??





Conclusion

- POSIX I/O
  - Everything is a file!
  - Based on the system calls **open()**, **read()**, **write()**, and **close()**
  - Devices (terminals, printers, etc.)
  - Regular files on disk
  - Networking (sockets)
  - Local interprocess communication (pipes, sockets)
- 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
- Next Time: 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