CS162
Operating Systems and Systems Programming
Lecture 4

Abstractions 2: Process Management, Files and I/O
A quick programmer's viewpoint

October 10th, 2024

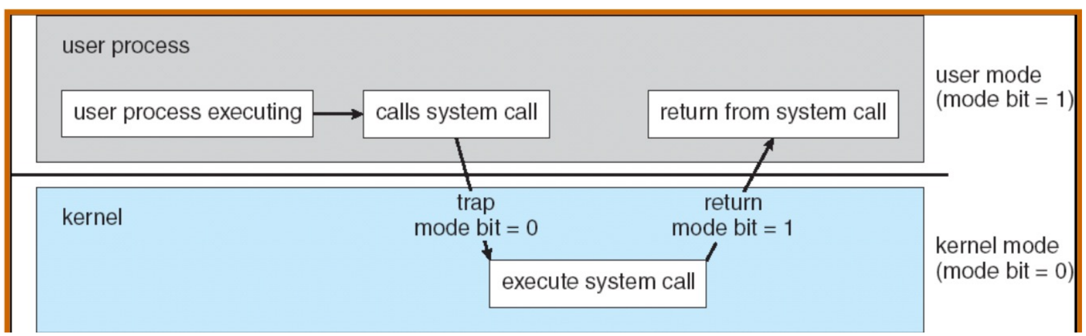
Prof. Ion Stoica

http://cs162.eecs.Berkeley.edu

Slides courtesy of David Culler, Natacha Crooks, Anthony D. Joseph, John Kubiatowicz, AJ Shankar, Alex Aiken, Eric Brewer, Ras Bodik, Doug Tygar, and David Wagner.

Recall: Dual Mode Operation

- Hardware provides at least two modes (at least 1 mode bit):
  1. Kernel Mode (or "supervisor" mode)
  2. User Mode
- Certain operations are prohibited when running in user mode
  - Changing the page table pointer, disabling interrupts, interacting directly w/ hardware, writing to kernel memory
- Carefully controlled transitions between user mode and kernel mode
  - System calls, interrupts, exceptions



Implementing Safe Kernel Mode Transfers

- Important aspects:
  - Controlled transfer into kernel (e.g., syscall table)
  - Separate kernel stack!

- Carefully constructed kernel code packs up the user process state and sets it aside
  - Details depend on the machine architecture
  - More on this next time

- Should be impossible for buggy or malicious user program to cause the kernel to corrupt itself!

3 types of Kernel Mode Transfer

- Syscall
  - Process requests a system service, e.g., exit
  - Like a function call, but "outside" the process
  - Does not have the address of the system function to call
  - Like a Remote Procedure Call (RPC) – for later
  - Marshall the syscall id and args in registers and exec syscall
- Interrupt
  - External asynchronous event triggers context switch
  - E.g., Timer, I/O device
  - Independent of user process
- Trap or Exception
  - Internal synchronous event in process triggers context switch
  - e.g., Protection violation (segmentation fault), Divide by zero, …

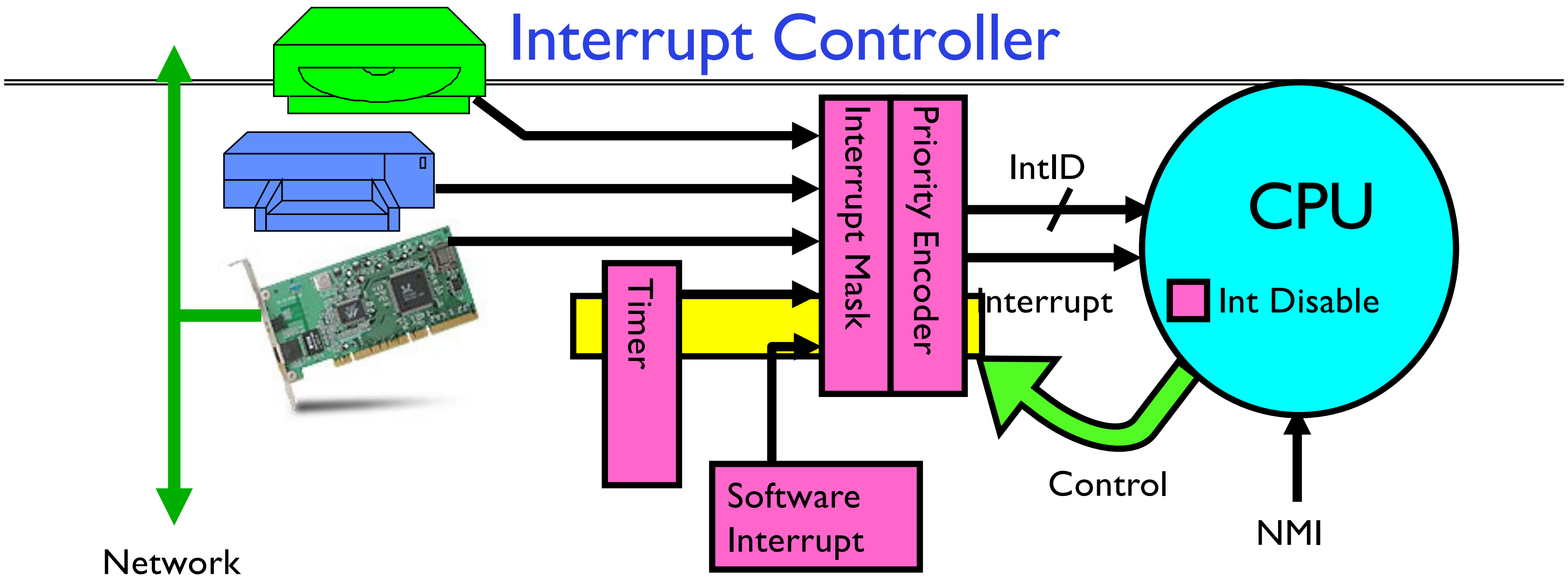
Handling System Calls safely

- Vector through well-defined syscall entry points!
  - Table mapping *system call number* to *handler*
  - Atomically set to kernel mode at same time as jump to system call code in kernel
  - Separate Kernel Stack in kernel memory during syscall execution
- System call handler must never trust user and must validate everything!
- On entry: Copy arguments
  - From user memory/registers/stack into kernel memory
  - Protect kernel from malicious code evading checks
- On entry: Validate arguments
  - Protect kernel from errors in user code
  - Protect kernel from invalid values and addresses
- On exit: Copy results back
  - Into user memory

How do we take interrupts safely?

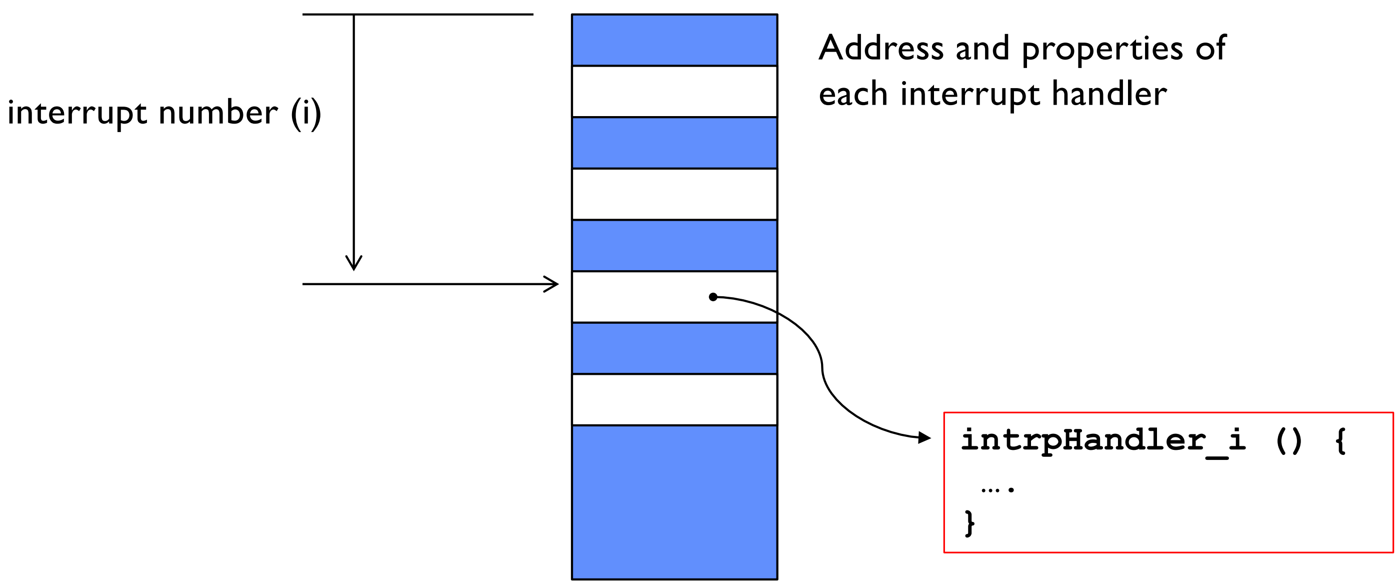
- Interrupt processing not visible to the user process:
  - Occurs between instructions, restarted transparently
  - No change to process state
  - What can be observed even with perfect interrupt processing?
- Interrupt vector
  - Limited number of entry points into kernel
- Kernel interrupt stack
  - Handler works regardless of state of user code
- Interrupt masking
  - Handler is non-blocking
- Atomic transfer of control
  - "Single instruction"-like to change:
    - » Program counter
    - » Stack pointer
    - » Memory protection
    - » Kernel/user mode
- Exceptions handled similarly, except *synchronously* (attached to particular instruction)

Interrupt Controller



- Interrupts invoked with interrupt lines from devices
- Interrupt controller chooses interrupt request to honor
  - Interrupt identity specified with ID line
  - Mask enables/disables interrupts
  - Priority encoder picks highest enabled interrupt
  - Software Interrupt Set/Cleared by Software
- CPU can disable all interrupts with internal flag
- Non-Maskable Interrupt line (NMI) can't be disabled

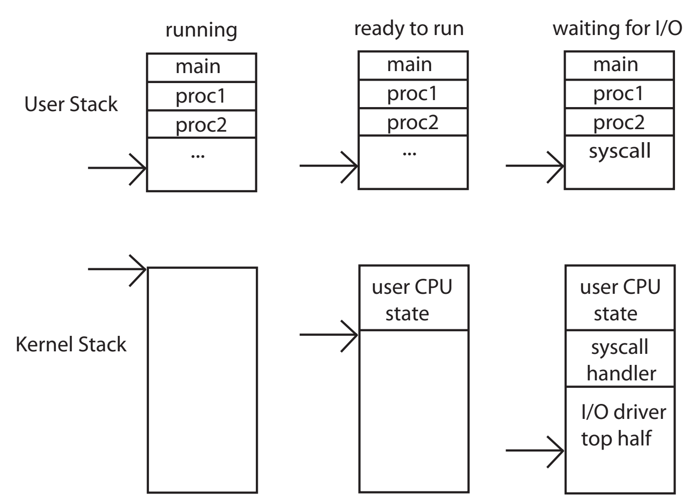
Interrupt Vector



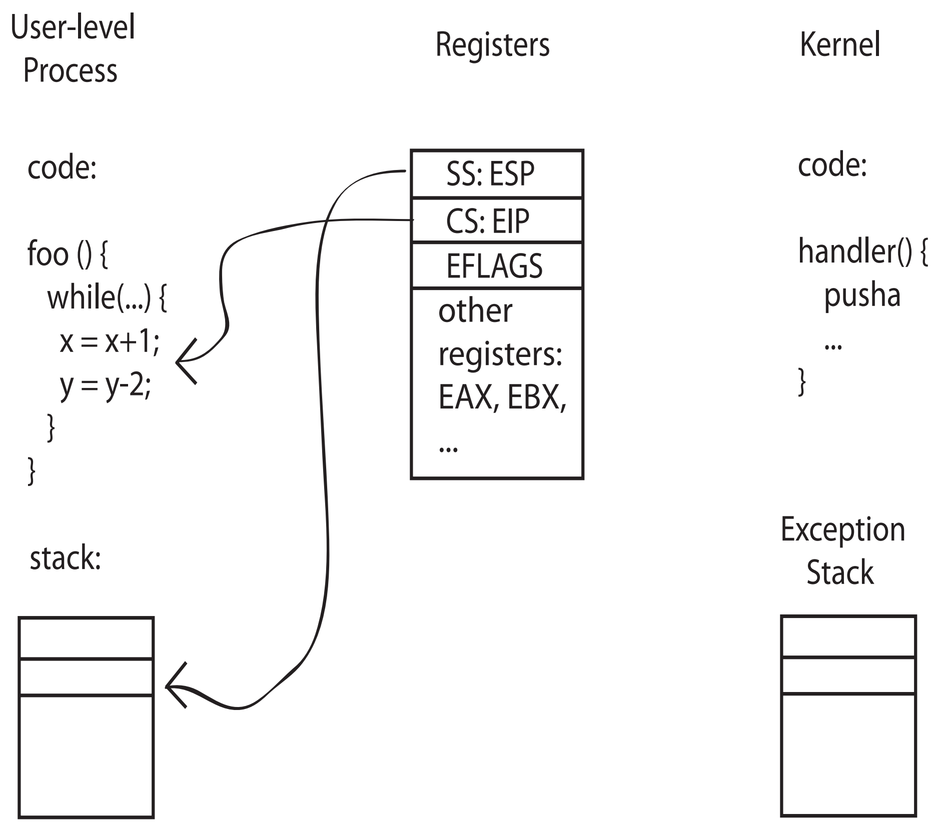
- Where else do you see this dispatch pattern?
  - System Call
  - Exceptions

Need for Separate Kernel Stacks

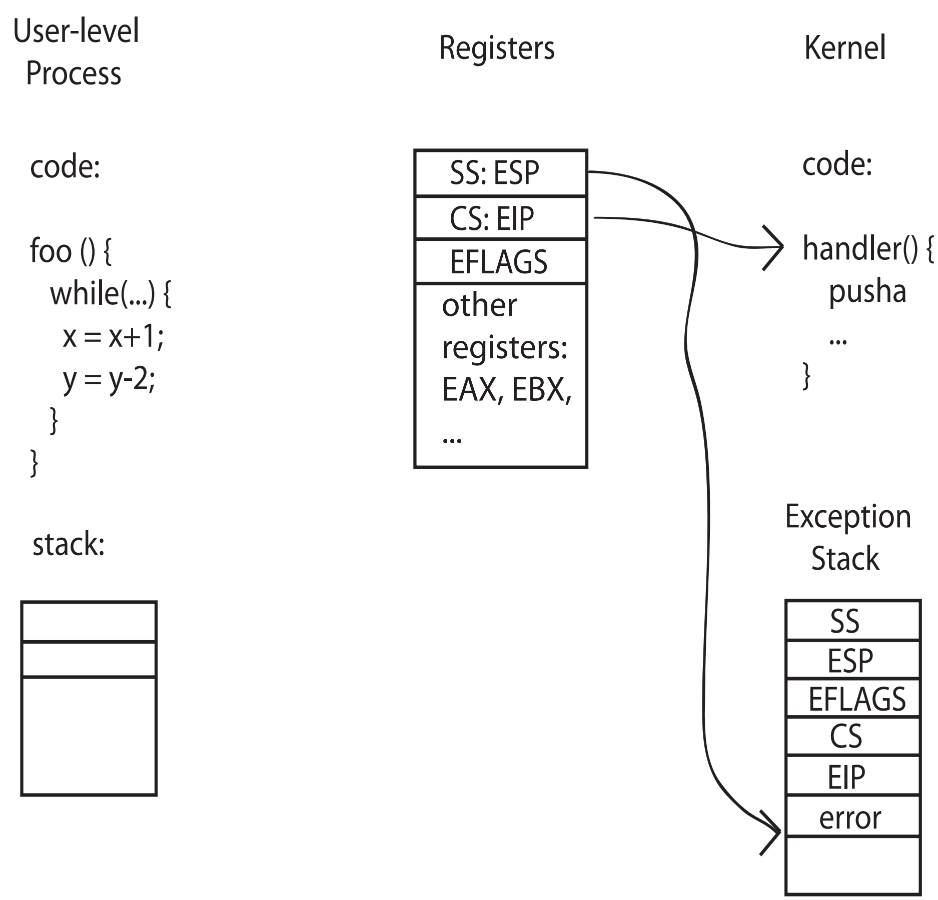
- Kernel needs space to work
- Cannot put anything on the user stack (Why?)
- Two-stack model
  – OS thread has interrupt stack (located in kernel memory) plus User stack (located in user memory)
  – Syscall handler copies user args to kernel space before invoking specific function (e.g., open)
  – Interrupts ?



Before

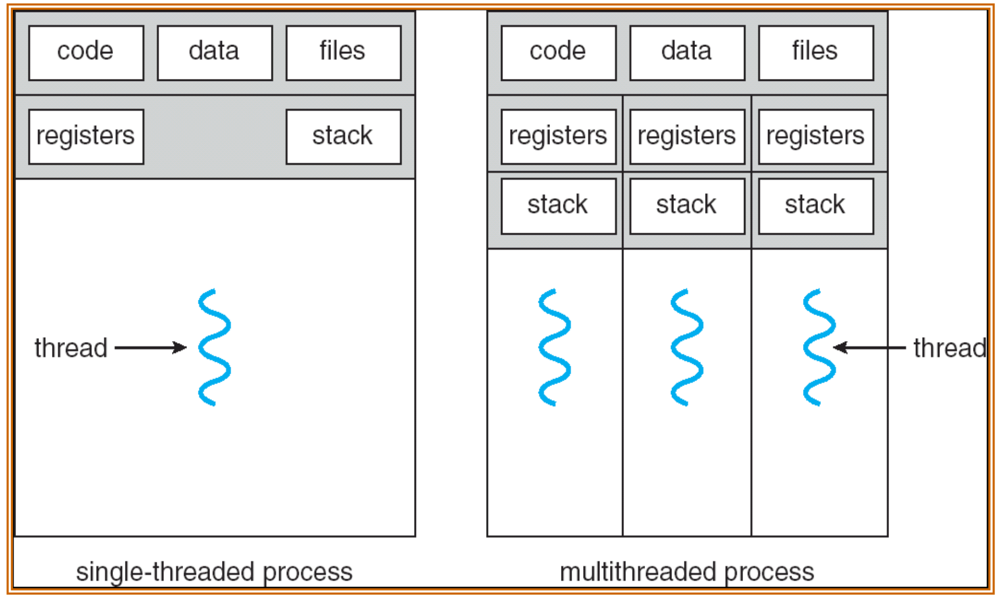

User-level
Process
code:
foo () {
while(...) {
x = x+1;
y = y-2;
}
}
stack:
Registers
SS: ESP
CS: EIP
EFLAGS
other
registers:
EAX, EBX,
...
Kernel
code:
handler() {
pusha
...
}
Exception
Stack

During Interrupt/System Call



Managing Processes

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



Bootstrapping

- If processes are created by other processes, how does the first process start?

- First process is started by the kernel
  - Often configured as an argument to the kernel *before* the kernel boots
  - Often called the "init" process

- After this, all processes on the system are created by other processes

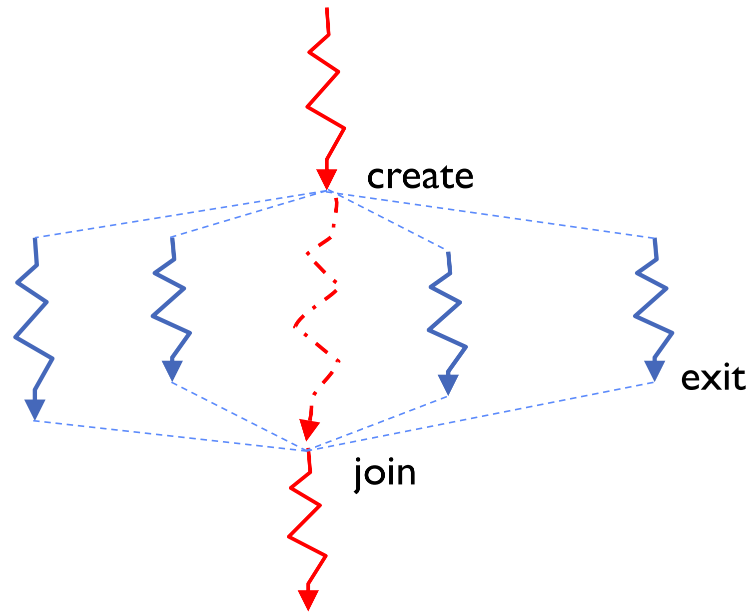
Process Management API

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

Recall threads

```
int pthread_create

void pthread_exit

pthread_join
```

Recall: Fork-Join Pattern



- Main thread *creates* (forks) collection of sub-threads passing them args to work on…
- … and then *joins* with them, collecting results.

Process Management API

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

pid.c

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

  exit(0);
}
```

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

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

Process Management API

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

Creating Processes

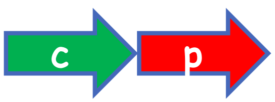
- `pid_t fork()` – copy the current process
  - New process has different pid
  - New process contains a single thread
- Return value from **`fork()`**: pid (like an integer)
  - When > 0:
    - » Running in (original) Parent process
    - » return value is pid of new child
  - When = 0:
    - » Running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

fork1.c

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

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

fork1.c

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

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



fork1.c

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

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

Mystery: fork_race.c

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

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

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

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

Process Management API

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

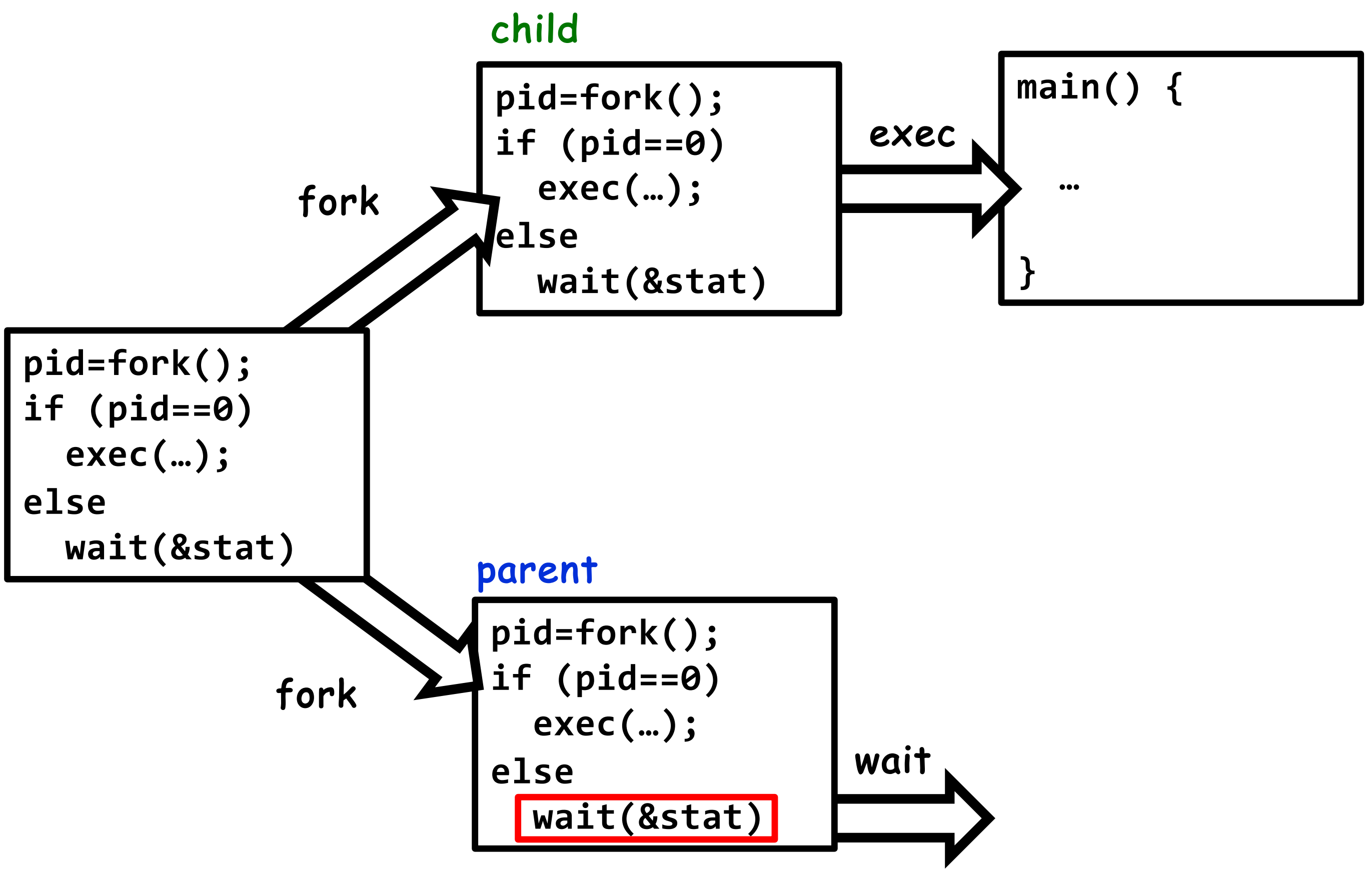
Starting new Program: variants of exec

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

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

  perror("execv");
  exit(1);
}
…
```

fork2.c – parent waits for child to finish

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

Process Management: The Shell pattern



Process Management API

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

inf_loop.c

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

void signal_callback_handler(int signum) {
  printf(“Caught signal!\n”);
  exit(1);
}
int main() {
  struct sigaction sa;
  sa.sa_flags = 0;
  sigemptyset(&sa.sa_mask);
  sa.sa_handler = signal_callback_handler;
  sigaction(SIGINT, &sa, NULL);
  while (1) {}
}
```

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

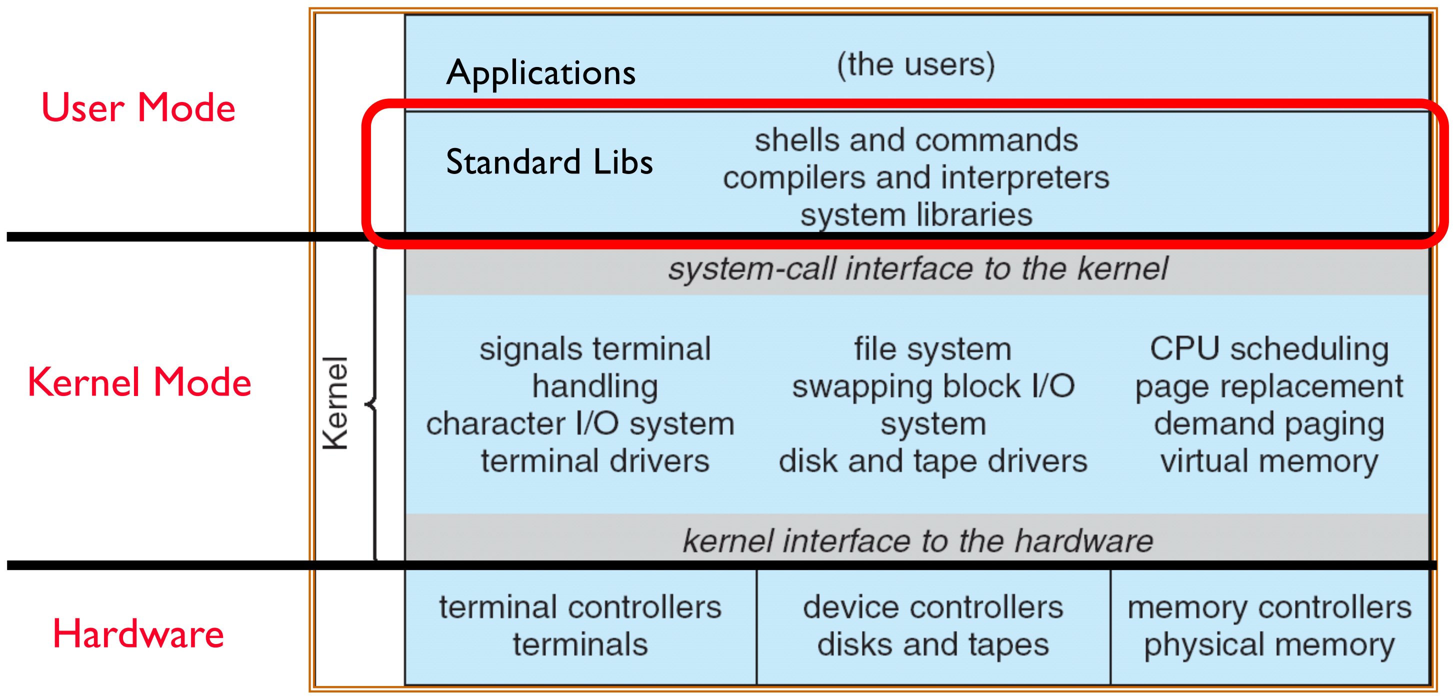
For each signal, there is a default handler defined by the system

Common POSIX Signals

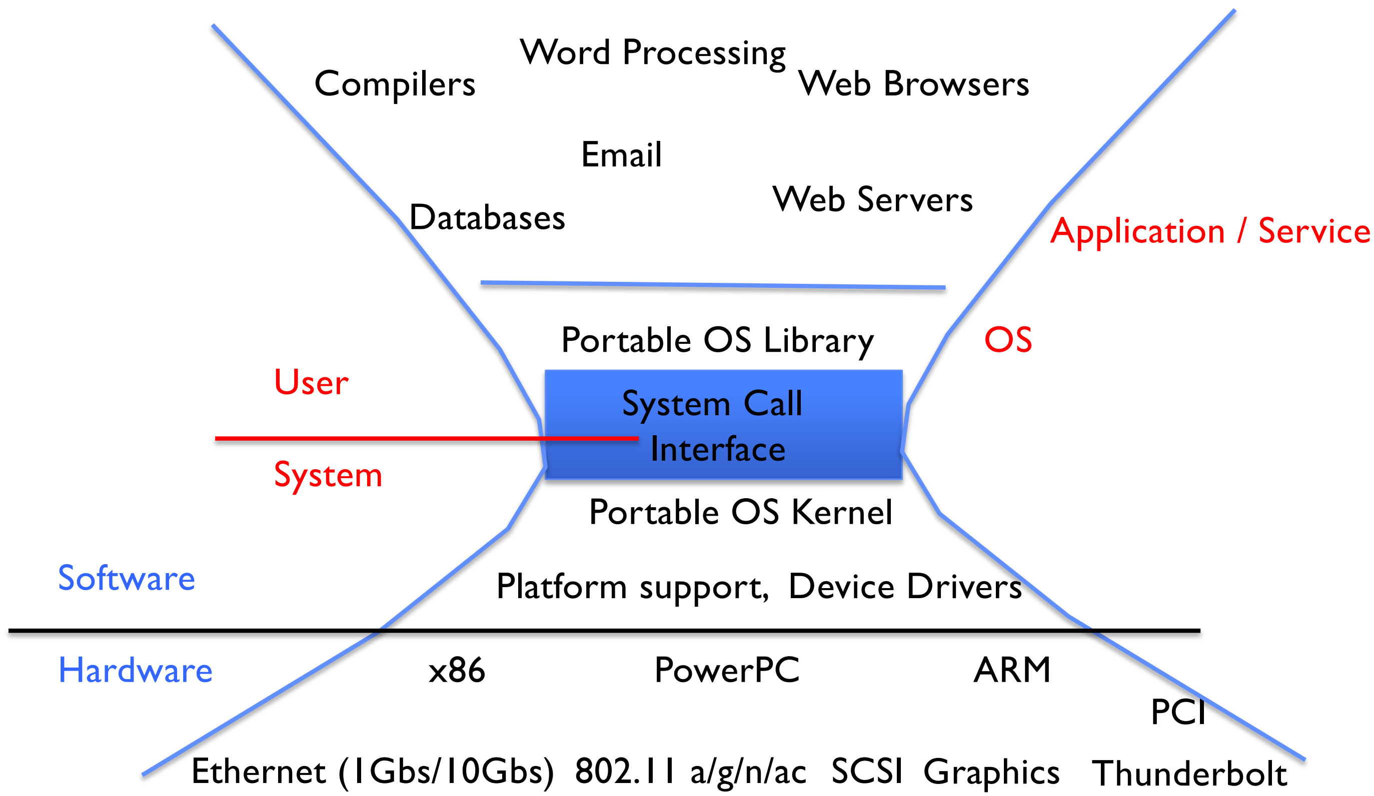
- `SIGINT` – control-C
- `SIGTERM` – default for **`kill`** shell command
- `SIGSTOP` – control-Z (default action: stop process)

- `SIGKILL`, `SIGSTOP` – terminate/stop process
  – Can't be changed with **`sigaction`**
  – Why?

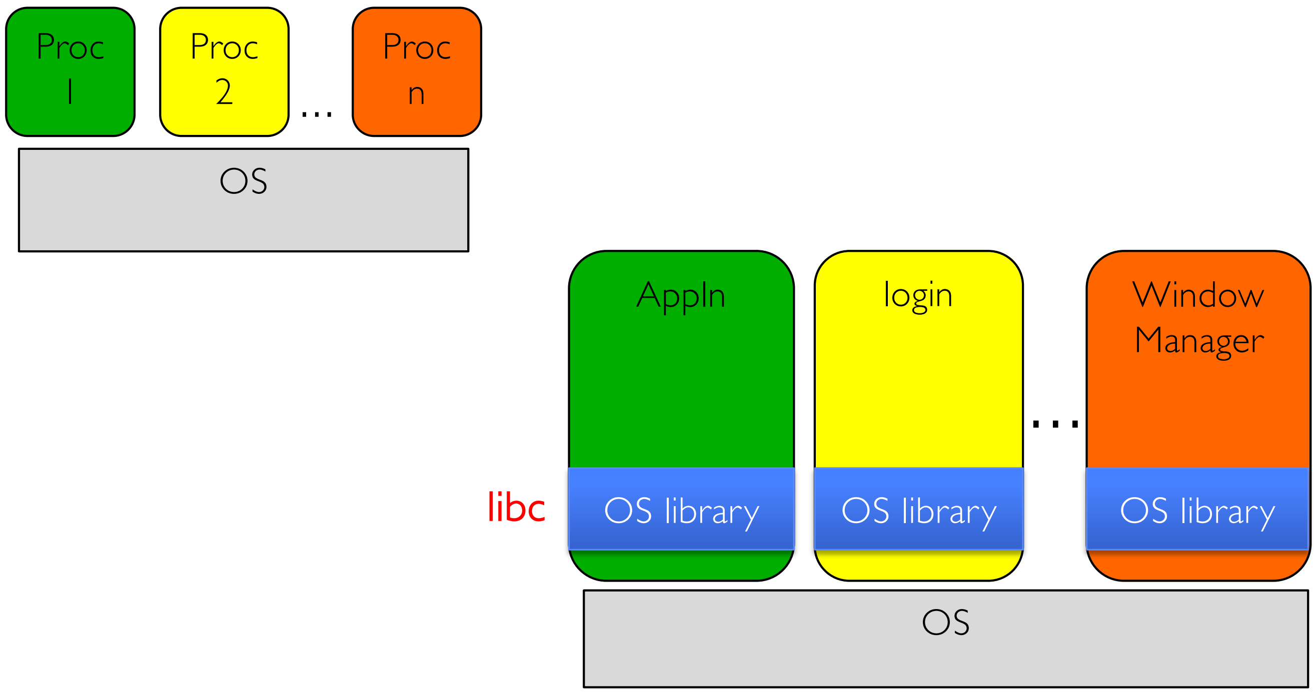
Recall: UNIX System Structure



A Kind of Narrow Waist



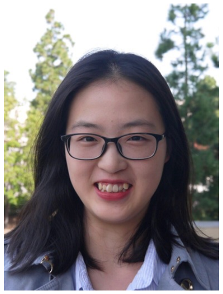
Recall: OS Library (libc) Issues Syscalls



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

Administrivia

- Ion's Office Hours
  - 11-12pm, Tuesdays
- Ion away this Thursday (NYC conference)
  - The lecture will be taught by Yi Xu, postdoc in the Sky Computing Lab



- Recommendation: Read assigned readings *before* lecture
- You should be going to sections – Important information covered in section
  - Any section will do until groups assigned
- Get finding groups of 4 people ASAP
  - Priority for same section; if cannot make this work, keep same TA
  - Remember: Your TA needs to see you in section!

Administrivia (Con't)

- You get 6 slip days for homework and 7 slip days for group projects
  - No project extensions on design documents, since we need to keep design reviews on track
  - Conserve your slip days!
- Midterm 1 will be on 10/3 from 7-9pm (tentative)
  - No class on day of midterm
  - Closed book
  - One page of *handwritten* notes – both sides

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
  - Seethe 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

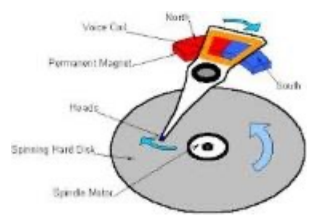
I/O and Storage Layers

Application / Service

Layer	
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

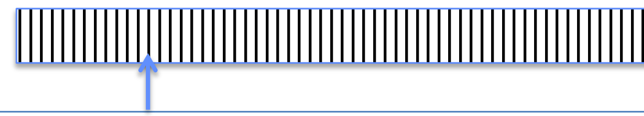







C High-Level File API – Streams

- 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:

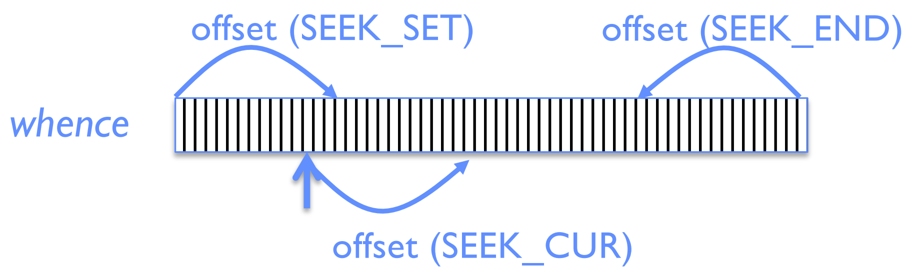
```
FILE* input = fopen(“input.txt”, “r”);
if (input == NULL) {
  // Prints our string and error msg.
  perror(“Failed to open input file”)
}
```

- 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

I/O and Storage Layers

Application / Service

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

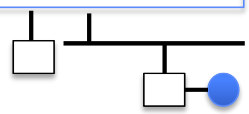
Low Level I/O — *File Descriptors*

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

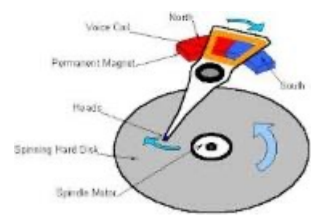
Open File Descriptions

File System — *Files/Directories/Indexes*

I/O Driver — *Commands and Data Transfers*



Disks, Flash, Controllers, DMA



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"

Conclusion

- System Call Interface is "narrow waist" between user programs and kernel
  - Must enter kernel atomically by setting PC to kernel routine at same time that CPU enters kernel mode
- Processes consist of one or more threads in an address space
  - Abstraction of the machine: execution environment for a program
  - Can use fork, exec, etc. to manage threads within a process
- We saw the role of the OS library
  - Provide API to programs
  - Interface with the OS to request services
- 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