Recall: OS Library API for Threads: *pthreads*

Here: the “p” is for “POSIX” which is a part of a standardized API

```c
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, 
                  void *(*start_routine)(void*), void *arg);

- thread is created executing start_routine with arg as its sole argument.
- return is implicit call to pthread_exit

void pthread_exit(void *value_ptr);

- terminates the thread and makes value_ptr available to any successful join

int pthread_join(pthread_t thread, void **value_ptr);

- suspends execution of the calling thread until the target thread terminates.
- On return with a non-NULL value_ptr the value passed to pthread_exit() by
  the terminating thread is made available in the location referenced
  by value_ptr.
```

Recall: pThreads Example

- How many threads are in this program?
- What function does each thread run?
- One possible result:
  - Yes: Loop calls Join in thread order
  - Does the main thread join with the threads in the same order that they were created?
  - Yes: Depends on scheduling order!
  - Does the threads exit in the same order they were created?
  - No: Depends on scheduling order!
  - Would the result change if run again?
  - Yes: Depends on scheduling order!
  - Is this code safe/correct???
  - No – threads share a variable that is used without locking and there is a race
    condition!

Recall: Locks

- Locks provide two *atomic* operations:
  - Lock Acquire() – wait until lock is free; then mark it as busy
    » After this returns, we say the calling thread *holds* the lock
  - Lock Release() – mark lock as free
    » Should only be called by a thread that currently holds the lock
    » After this returns, the calling thread no longer holds the lock
- For now, don’t worry about how to implement locks!
  - We’ll cover that in substantial depth later on in the class
OS Library Locks: * pthreads

```c
int pthread_mutex_init(pthread_mutex_t *mutex,
        const pthread_mutexattr_t *attr)

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

You’ll get a chance to use these in Homework 1

---

Our Example: Fixing the Race Condition for increment (++)

```c
int common = 162;
pthread_mutex_t common_lock = PTHREAD_MUTEX_INITIALIZER;
void *threadfun(void *threadid)
{
    long tid = (long)threadid;
    pthread_mutex_lock(&common_lock);
    int my_common = common++;
    pthread_mutex_unlock(&common_lock);
    printf("Thread %lx stack: %lx common: %lx (%d)\n", tid,
           (unsigned long) &tid,
           (unsigned long) &common, my_common);
    thread_exit(NULL);
}
```

---

Recall: Adding locking to a Red/Black tree

**Thread A**
- Insert(3)
  - Lock.acquire()
  - Insert 3 into the data structure
  - Lock.release()

**Thread B**
- Insert(4)
  - Lock.acquire()
  - Insert 4 into the data structure
  - Lock.release()
- Get(6)
  - Lock.acquire()
  - Check for membership
  - Lock.release()

---

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
  – eg. 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
  – Atomicly set to kernel mode at same time as jump to systemcall 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
- CPU can disable all interrupts with internal flag
- Non-Contactable Interrupt line (NMI) can't be disabled

Network

**Interrupt Vector**

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

**Interrupt Vector**

- Address and properties of each interrupt handler

```
intrpHandler_i () {
...
}
```

**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
- Registers
- Kernel

```
code:
foo () {
  while(1)
    x = x+1
    y = y+2
}

stack:
```
During Interrupt/System Call

- User-level Process
- Registers
- Kernel

```
foo () {
  i = i + 1;
  y = 2 * y;
}
```

- stack:
- Exception Stack

```
SS ESP
IFLAGS
CS
EIP
```

Managing Processes

- How to manage process state?
  - How to create a process?
  - How to exit from a process?

- Processes are created and managed... by processes!

Administrivia

- Kubiatowicz Office Hours
  - 3pm-4pm, Tuesday/Thursday

- TOMORROW (Friday) is Drop Deadline! VERY HARD TO DROP LATER!

- Recommendation: Read assigned readings before lecture

- You should be going to sections – Important information covered in sections
  - 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)

- Starting next week, we will be adhering to strict slip-day policies for non-DSP students
  - Slip days are no-questions asked (or justification needed) extensions
  - Anything beyond this requires documentation (i.e. doctor’s note, etc)
  - If you run out of slip days, assignments will be discounted
    » 10% first day, 20% second day, 40% third day, 80% fourth day

- You get 4 slip days for homework and 5 slip days for group projects
  - No project extensions on design documents, since we need to keep design reviews on track
  - Conserve your slip days!

- Midterm 1 will be on 2/15 from 8-10pm
  - No class on day of midterm (extra office hours!)
  - Closed book
  - One page of handwritten notes – both sides
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

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[]) {
    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()`:
  - 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...

```
#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);
        perror("Fork failed");
    } else {                 /* 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

```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 process 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

```c
int i;
int 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);
    }
}
```

- **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

```c
int 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

```c
int status;
pid_t tcpid;

if (cpid > 0) {
    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) {
    mypid = getpid();
    printf("[%d] child\n", mypid);
    exit(42);
}
```

Process Management: The Shell pattern

```
child
pid=fork();
if (pid==0)
    exec(...);
else
    wait(&stat)

parent
pid=fork();
if (pid==0)
    exec(...);
else
    wait(&stat)
```

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

```
#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) {}
}
```

inf_loop.c

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
- SIGSTP – control-Z (default action: stop process)
- SIGKILL, SIGSTOP – terminate/stop process
  - Can't be changed with sigaction
  - Why?

Recall: UNIX System Structure

User Mode

Kernel Mode

Hardware

Recall: OS Library (libc) Issues Syscalls

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

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

Aside: POSIX interfaces

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

The File System Abstraction

- File
  - Named collection of data in a file system
  - POSIX File data: sequence of bytes
    - Could be text, binary, serialized objects, ...
  - File Metadata: information about the file
    - Size, Modification Time, Owner, Security info, Access control
- Directory
  - “Folder” containing files & directories
  - Hierarchical (graphical) naming
    - Path through the directory graph
    - Uniquely identifies a file or directory
      - `/home/ff/cs162/public_html/fa14/index.html`
  - Links and Volumes (later)

Connecting Processes, File Systems, and Users

- Every process has a *current working directory* (CWD)
  - Can be set with system call:
    ```c
    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
  - `~/cs162/index.html`
    - Refers to index.html in the home directory
I/O and Storage Layers

Application / Service

High Level I/O
- Streams (buffered I/O)
- File Descriptors
  - open(), read(), write(), close(), ...
- Open File Descriptions
  - Files/Directories/Indexes

Low Level I/O
- Syscall

File System
- Commands and Data Transfers
- Disks, Flash, Controllers, DMA

C High-Level File API – Streams

- Operates on “streams” – unformatted sequences of bytes (whether text or binary data), with a position:

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

<table>
<thead>
<tr>
<th>Mode</th>
<th>Text</th>
<th>Binary</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>rb</td>
<td>Open existing file for reading</td>
<td></td>
</tr>
<tr>
<td>w</td>
<td>wb</td>
<td>Open for writing; created if does not exist</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>ab</td>
<td>Open for appending; created if does not exist</td>
<td></td>
</tr>
<tr>
<td>r+</td>
<td>rb+</td>
<td>Open existing file for reading &amp; writing.</td>
<td></td>
</tr>
<tr>
<td>w+</td>
<td>wb+</td>
<td>Open for reading &amp; writing; truncated to zero if exists, create otherwise</td>
<td></td>
</tr>
<tr>
<td>a+</td>
<td>ab+</td>
<td>Open for reading &amp; writing. Created if does not exist. Read from beginning, write as append</td>
<td></td>
</tr>
</tbody>
</table>

- 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

```c
// 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);
```

```c
// 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);
```

```c
// 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

```c
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 Streams: Block-by-Block I/O

```c
#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);
}
```

### C High-Level File API

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

### Aside: Check your Errors!

- Systems programmers should always be paranoid!
  - Otherwise you get intermittently buggy code
- We should really be writing things like:
  ```c
  FILE* input = fopen("input.txt", "r");
  if (input == NULL) {
      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

```c
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

<table>
<thead>
<tr>
<th>Application / Service</th>
<th>Streams (buffered I/O)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>High Level I/O</strong></td>
<td><strong>Low Level I/O</strong></td>
</tr>
<tr>
<td>Syscall</td>
<td>File System</td>
</tr>
<tr>
<td>File System</td>
<td>I/O Driver</td>
</tr>
<tr>
<td>Open File Descriptions</td>
<td>Commands and Data Transfers</td>
</tr>
<tr>
<td>Disks, Flash, Controllers, DMA</td>
<td></td>
</tr>
</tbody>
</table>

---

C Low-Level (pre-opened) Standard Descriptors

```c
#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)
```

- 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?

---

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

```c
#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, ...)

---

```c
// 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:
  
  ```c
  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

  ```c
  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!)

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

Example: **lowio.c**

```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**

- Streams are buffered in user memory:
  - printf("Beginning of line \n");
  - 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(18);
  - 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