Recall: Four fundamental OS concepts

- Thread
  - Single unique execution context
  - Program Counter, Registers, Execution Flags, Stack
- Address Space with Translation
  - Programs execute in an address space that is distinct from the memory space of the physical machine
- Process
  - An instance of an executing program is a process consisting of an address space and one or more threads of control
- Dual Mode operation/Protection
  - Only the “system” has the ability to access certain resources
  - The OS and the hardware are protected from user programs and user programs are isolated from one another by controlling the translation from program virtual addresses to machine physical addresses

Recall: Process

- Process: execution environment with Restricted Rights
  - Address Space with One or More Threads
  - Owns memory (address space)
  - Owns file descriptors, file system context, ...
  - Encapsulate one or more threads sharing process resources
- Why processes?
  - Protected from each other!
  - OS Protected from them
  - Navigate fundamental tradeoff between protection and efficiency
  - Processes provides memory protection
  - Threads more efficient than processes (later)
- Application instance consists of one or more processes

Single and Multithreaded Processes

- Threads encapsulate concurrency: “Active” component
- Address spaces encapsulate protection: “Passive” part
  - Keeps buggy program from trashing the system
- Why have multiple threads per address space?
Recall: give the illusion of multiple processors?

- Assume a single processor. How do we provide the illusion of multiple processors?
  - Multiplex in time!
- Each virtual “CPU” needs a structure to hold:
  - Program Counter (PC), Stack Pointer (SP)
  - Registers (Integer, Floating point, others...?)
- How switch from one virtual CPU to the next?
  - Save PC, SP, and registers in current state block
  - Load PC, SP, and registers from new state block
- What triggers switch?
  - Timer, voluntary yield, I/O, other things

Simultaneous MultiThreading/Hyperthreading

- Hardware technique
  - Superscalar processors can execute multiple instructions that are independent.
  - Hyperthreading duplicates register state to make a second “thread,” allowing more instructions to run.
- Can schedule each thread as if were separate CPU
  - But, sub-linear speedup!
- Original technique called “Simultaneous Multithreading”
  - SPARC, Pentium 4/Xeon (“Hyperthreading”), Power 5

Recall: User/Kernal(Priviledged) Mode

User Mode

Kernel Mode

Limited HW access Full HW access

Recall: A simple address translation (B&B)

- Can the program touch OS?
- Can it touch other programs?
Recall: Address Mapping

![Address Mapping Diagram]

Translation Map 1  Translation Map 2

Physical Address Space

Putting it together: web server

![Web Server Diagram]

Running Many Programs

- We have the basic mechanism to
  - switch between user processes and the kernel,
  - the kernel can switch among user processes,
  - Protect OS from user processes and processes
    from each other

- Questions ???
  - How do we represent user processes in the OS?
  - How do we decide which user process to run?
  - How do we pack up the process and set it aside?
  - How do we get a stack and heap for the kernel?
  - Aren't we wasting a lot of memory?
  - ...

Process Control Block

- Kernel represents each process as a process control block (PCB)
  - Status (running, ready, blocked, ...)
  - Register state (when not ready)
  - Process ID (PID), User, Executable, Priority, ...
  - Execution time, ...
  - Memory space, translation, ...

- Kernel Scheduler maintains a data structure containing the PCBs

- Scheduling algorithm selects the next one to run
Scheduler

```c
if (readyProcesses(PCBs)) {
    nextPCB = selectProcess(PCBs);
    run(nextPCB);
} else {
    run_idle_process();
}
```

- Scheduling: Mechanism for deciding which processes/threads receive the CPU
- Lots of different scheduling policies provide ...
  - Fairness or
  - Realtime guarantees or
  - Latency optimization or ..

Implementing Safe Kernel Mode Transfers

- Important aspects:
  - Separate kernel stack
  - Controlled transfer into kernel (e.g. syscall table)
- Carefully constructed kernel code packs up the user process state and sets it aside.
  - Details depend on the machine architecture
- Should be impossible for buggy or malicious user program to cause the kernel to corrupt itself.

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
Kernel System Call Handler

- Vector through well-defined syscall entry points!
  - Table mapping system call number to handler
- Locate arguments
  - In registers or on user(!) stack
- Copy arguments
  - From user memory into kernel memory
  - Protect kernel from malicious code evading checks
- Validate arguments
  - Protect kernel from errors in user code
- Copy results back
  - Into user memory

Hardware support: Interrupt Control

- Interrupt processing not be 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 Handler invoked with interrupts 'disabled'
  - Re-enabled upon completion
  - Non-blocking (run to completion, no waits)
  - Pack up in a queue and pass off to an OS thread for hard work
    » wake up an existing OS thread
- OS kernel may enable/disable interrupts
  - On x86: CLI (disable interrupts), STI (enable)
  - Atomic section when select next process/thread to run
  - Atomic return from interrupt or syscall
- HW may have multiple levels of interrupt
  - Mask off (disable) certain interrupts, e.g., lower priority
  - Certain non-maskable-interrupts (nmi)
    » e.g., kernel segmentation fault
  - Interrupts invoked with interrupt lines from devices
  - Interrupt controller chooses interrupt request to honor
    - Mask enables/disables interrupts
    - Priority encoder picks highest enabled interrupt
    - Software Interrupt Set/Cleared by Software
    - Interrupt identity specified with ID line
- CPU can disable all interrupts with internal flag
- Non-maskable interrupt line (NMI) can’t be disabled
How do we take interrupts safely?

- **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
- **Transparent restartable execution**
  - User program does not know interrupt occurred

Administrivia: Getting started

- **Kubiatowicz Office Hours:**
  - 1pm-2pm, Monday/Wednesday
- **Homework 0 immediately ⇒ Due on Monday!**
  - Get familiar with all the cs162 tools
  - Submit to autograder via git
- Should be going to section already!
- **Participation: Get to know your TA!**
- **Friday is Drop Deadline!**
- **Group sign up form out next week (after drop deadline)**
  - Get finding groups ASAP
  - 4 people in a group!
- **Finals conflicts: Tell us now**
  - Must give us a good reason for providing an alternative
  - No alternate time if the conflict is because of an overlapping class (e.g. EE122)

Question

- Process is an instance of a program executing.
  - The fundamental OS responsibility
- Processes do their work by processing and calling file system operations
- Are there any operations on processes themselves?
- exit?

pid.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[]) {
  int c;
  pid_t pid = getpid();  /* get current processes PID */
  printf("My pid: \n");
  printf("%d\n", pid);
  c = fgetc(stdin);
  exit(0);
}
```
Can a process create a process?

- Yes
- Fork creates a copy of process
- Return value from Fork: 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
- All of the state of original process duplicated in both Parent and Child!
  - Memory, File Descriptors (next topic), etc...

fork1.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];
    size_t readlen, writelen, slen;
    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");
        exit(1);
    }
    exit(0);
}
```

fork2.c

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

UNIX Process Management

- UNIX fork - system call to create a copy of the current process, and start it running
  - No arguments!
- UNIX exec - system call to change the program being run by the current process
- UNIX wait - system call to wait for a process to finish
- UNIX signal - system call to send a notification to another process
# UNIX Process Management

A shell is a job control system:
- Allows programmer to create and manage a set of programs to do some task
- Windows, MacOS, Linux all have shells

Example: to compile a C program
```
cmake -c sourcefile1.c
cmake -c sourcefile2.c
ln -o program sourcefile1.o sourcefile2.o
./program
```

# Shell

- A shell is a job control system
- Allows programmer to create and manage a set of programs to do some task
- Windows, MacOS, Linux all have shells

Example: to compile a C program
```
cmake -c sourcefile1.c
cmake -c sourcefile2.c
ln -o program sourcefile1.o sourcefile2.o
./program
```

# Signals - infloop.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 %d - phew!\n", signum);
    exit(1);
}

int main() {
    signal(SIGINT, signal_callback_handler);

    while (1) {}
}
```

# Process races: fork.c

```c
if (cpid > 0) {
    mypid = getpid();
    printf("[\%d] parent of [\%d]\n", mypid, cpid);
    for (i=0; i<100; i++) {
        printf("[\%d] parent: %d\n", mypid, i);
        sleep(1);
    }
}
else if (cpid == 0) {
    mypid = getpid();
    printf("[\%d] child\n", mypid);
    for (i=0; i>-100; i--) {
        printf("[\%d] child: %d\n", mypid, i);
        sleep(1);
    }
}
```

Question: What does this program print?
Does it change if you add in one of the sleep() statements?
Recall: UNIX System Structure

![UNIX System Structure Diagram]

How does the kernel provide services?

- 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 Diagram]
**A Kind of Narrow Waist**

- Compilers
- Word Processing
- Web Browsers
- Email
- Databases
- Web Servers
- Application / Service

- User
- Portable OS Library
- OS

- System
- Portable OS Kernel
- System Call Interface
- Platform support, Device Drivers

- Software
- Hardware
  - x86
  - PowerPC
  - ARM
  - PCI
  - Ethernet (10/100/1000)
  - 802.11 a/b/g/n
  - SCSI IDE
  - Graphics

---

**Key Unix I/O Design Concepts**

- **Uniformity**
  - 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 process, yielding processor to other task

- **Kernel buffered writes**
  - Completion of out-going transfer decoupled from the application, allowing it to continue

- **Explicit close**

---

**I/O & Storage Layers**

- **Application / Service**
  - **High Level I/O**
    - streams
  - **Low Level I/O**
    - handles
  - **Syscall**
  - **File System**
  - **I/O Driver**
  - **Commands and Data Transfers**
    - Disks, Flash, Controllers, DMA

---

**The file system abstraction**

- **File**
  - Named collection of data in a file system
  - File data
    - Text, binary, linearized objects
  - File Metadata: information about the file
    - Size, Modification Time, Owner, Security info
    - Basis for 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)
### C high level File API – streams (review)

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

```c
#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>Descriptions</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, created 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>

### Connecting Processes, Filesystem, and Users

- Process has a 'current working directory'
- Absolute Paths
  - `/home/ff/cs152`
- Relative paths
  - `index.html`, `/index.html` - current WD
  - `../index.html` - parent of current WD
  - `~`, `~cs152` - home directory

### C API Standard Streams

- 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`
  - Recall: Use of pipe symbols connects `STDOUT` and `STDIN`

```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 Stream API positioning**

- Preserves high level abstraction of a uniform stream of objects
- Adds buffering for performance

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

---

**What's below the surface ??**

![Diagram illustrating high level I/O and low level I/O]

- **High Level I/O**
- **Low Level I/O**
- **Syscall**
- **File System**
- **I/O Driver**
- **Application / Service**

---

**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

---

**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 filedes) – 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!

**And lots more !**

- TTYs versus files
- Memory mapped files
- File Locking
- Asynchronous I/O
- Generic I/O Control Operations
- Duplicating descriptors

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

**What’s below the surface ??**

Application / Service

- streams
- handles
- registers

Low Level I/O

Syscall

File System

- descriptors

I/O Driver

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

**SYSCALL**

- 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
Internal OS File Descriptor

• Internal Data Structure describing everything about the file
  - Where it resides
  - Its status
  - How to access it

Device Drivers

• Device Driver: Device-specific code in the kernel that interacts directly with the device hardware
  - Supports a standard, internal interface
  - Same kernel I/O system can interact easily with different device drivers
  - Special device-specific configuration supported with the ioctl() system call

• Device Drivers typically divided into two pieces:
  - Top half: accessed in call path from system calls
    » implements a set of standard, cross-device calls like open(), close(), read(), write(), ioctl(), strategy()
    » This is the kernel's interface to the device driver
    » Top half will start I/O to device, may put thread to sleep until finished
  - Bottom half: run as interrupt routine
    » Gets input or transfers next block of output
    » May wake sleeping threads if I/O now complete

File System: from syscall to driver

In fs/read_write.c

```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;
}
```

Low Level Driver

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

- **Process**: execution environment with Restricted Rights
  - Address Space with One or More Threads
  - Owns memory (address space)
  - Owns file descriptors, file system context, ...
  - Encapsulate one or more threads sharing process resources

- **Interrupts**
  - Hardware mechanism for regaining control from user
  - Notification that events have occurred
  - User-level equivalent: Signals

- **Native control of Process**
  - Fork, Exec, Wait, Signal

- **Basic Support for I/O**
  - Standard interface: open, read, write, seek
  - Device drivers: customized interface to hardware

---

**Life Cycle of An I/O Request**

1. **User Program**
   - Request I/O
   - System call
   - Call should return error?

2. **Device Driver Top Half**
   - Process request, send commands to controller, return messages to user, control interrupts

3. **Device Driver Bottom Half**
   - Poll device, interrupt when I/O completed

4. **Device Hardware**
   - Monitor device, generate interrupt

---

**So what happens when you fgetc?**

1. **Application / Service**
   - Streams
   - Handles
   - Registers
   - Descriptors
   - Commands and Data Transfers
   - Disks, Flash, Controllers, DMA