Recall: Four fundamental OS concepts

- **Thread**
  - Single unique execution context
  - Program Counter, Registers, Execution Flags, Stack

- **Address Space w/ 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
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!
  - Multiple “virtual CPUs”
- 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 (Privileged) Mode

User Mode
- syscall
- interrupt
- exception
- exit
- rtn
- rfi
- exec

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?
Alternative: Address Mapping

Program 1
Virtual Address Space 1

Translation Map 1

Program 2
Virtual Address Space 2

Translation Map 2

Physical Address Space
Putting it together: web server

1. network socket read
2. copy arriving packet (DMA)
3. kernel copy
4. parse request
5. file read
6. disk request
7. disk data (DMA)
8. kernel copy
9. format reply
10. network socket write
11. kernel copy from user buffer into network buffer
12. format outgoing packet and DMA

Server

Kernel

RTU

Hardware

Network Interface

Disk Interface
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

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

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

User-level Process

code:

```c
foo() {
    while(...) {
        x = x+1;
        y = y-2;
    }
}
```

stack:

Registers

- SS: ESP
- CS: EIP
- EFLAGS
- other registers: EAX, EBX, ...

Kernel

code:

```c
handler() {
    pusha
    ...
}
```

Exception Stack
During

User-level
Process

code:

foo () {
  while(...) {
    x = x+1;
    y = y-2;
  }
}

stack:

Registers

Kernel
code:

handler() {
  pusha
  ...
}

Exception Stack

| SS    |
| ESP   |
| EFLAGS|
| CS    |
| EIP   |
| error |

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)
- 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, eg., 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

- Office Hours:
  - 1630 to 1700 Monday, or email me for an alternate time
- Homework 0 immediately ⇒ Due on Wednesday!
  - Get familiar with all the tools
  - importance of git
- TA session time slot
  - Monday 12:30 to 13:15
- Late registration is this week
  - If you are not serious about taking the course, please drop the course now
- Group sign up form out next week (after “Tarmim”)
  - think of selecting group members ASAP
  - 4 people in a group!
• 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 operation on processes themselves?
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

#define BUFSIZE 1024

int main(int arg, char *argv[])
{
    int c;

    pid_t pid = getpid(); /* get current process PID */

    printf("My pid: %d\n", pid);
    c = fgetc(stdin);
    exit(0);
}
Can a process create a process?

• Yes
  - Unique identity of process is the “process ID” (or pid).
• Fork() system call creates a copy of current process with a new pid
• 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

#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);
}
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
int status;
...
cpid = fork();
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);
}
...
UNIX Process Management

fork

pid = fork();
if (pid == 0)
    exec(...);
else
    wait(pid);

exec

main () {
    ... 
}

wait

pid = fork();
if (pid == 0)
    exec(...);
else
    wait(pid);
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
  cc -c sourcefile1.c
  cc -c sourcefile2.c
  ln -o program sourcefile1.o sourcefile2.o
  ./program
#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) {}
}

Got top?
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?
Break
## Recall: UNIX System Structure

<table>
<thead>
<tr>
<th>User Mode</th>
<th>Kernel Mode</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applications</td>
<td>Standard Libs</td>
<td>Hardware</td>
</tr>
<tr>
<td>(the users)</td>
<td>shells and commands, compilers and interpreters, system libraries</td>
<td>terminal controllers, terminals</td>
</tr>
<tr>
<td></td>
<td></td>
<td>device controllers, disks and tapes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>memory controllers, physical memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>kernel interface to the hardware</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>signals terminal handling, character I/O system, terminal drivers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>file system, swapping block I/O system, disk and tape drivers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CPU scheduling, page replacement, demand paging, virtual memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>system-call interface to the kernel</strong></td>
</tr>
</tbody>
</table>
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

Proc 1  Proc 2  ...  Proc n

OS

AppIn

login

Window Manager

OS library

OS library

OS library

OS
A Kind of Narrow Waist

Compilers

Word Processing

Web Browsers

Email

Web Servers

Databases

Portable OS Library

OS

System Call

Interface

Portable OS Kernel

Platform support, Device Drivers

User

System

Software

Hardware

x86

PowerPC

ARM

PCI

Ethernet (10/100/1000)

802.11 a/b/g/n

SCSI

IDE

Graphics

Application / Service
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

Low Level I/O

Syscall

File System

I/O Driver

streams

handles

registers

descriptors

Commands and Data Transfers

Disks, Flash, Controllers, DMA
The file system abstraction

• File
  - Named collection of data in a file system
  - 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 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>
</tr>
<tr>
<td>w</td>
<td>wb</td>
<td>Open for writing; created if does not exist</td>
</tr>
<tr>
<td>a</td>
<td>ab</td>
<td>Open for appending; created if does not exist</td>
</tr>
<tr>
<td>r+</td>
<td>rb+</td>
<td>Open existing file for reading &amp; writing.</td>
</tr>
<tr>
<td>w+</td>
<td>wb+</td>
<td>Open for reading &amp; writing; truncated to zero if exists, create otherwise</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>
</tr>
</tbody>
</table>

Don't forget to flush
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
    » find | grep | wc ...
#include <stdio.h>

// character oriented
int fputc( int c, FILE *fp ); // rtn c or EOF on err
int fputs( const char *s, FILE *fp ); // rtn >0 or EOF

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

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

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

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