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

**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

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

**shared-memory diagram**

- vCPU1, vCPU2, vCPU3
- Time

- 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

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 ..

Recall: 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 arguments in registers and execute 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, …
Recall: User/Kernel (Privileged) Mode

User Mode

- interrupt
- syscall
- exit

Kernel Mode

- interrupt
- syscall
- exit

Limited HW access

Full HW access

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

User-level Process

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

- stack:
  - Kernell Stack
  - user CPU state
  - syscall
  - I/O driver top half

Kernel

- code:
  - handler() {
  - pusha
  - ...
  - }

- stack:
  - Exception Stack
  - other registers: EAX, EBX, ...
During

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

Administrivia: Getting started

- We have a new discussion section!
  - 162-112 Th 6:30-7:30P  310 Soda

- Joseph Office Hours: Mondays/Tuesdays 11-12 in 465F Soda

- THIS Friday (9/2) is early drop day! Very hard to drop afterwards...

- Work on Homework 0 immediately ⇒ Due on Monday!
  - Get familiar with all the cs162 tools
  - Submit to autograder via git

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, eg, lower priority
  - Certain Non-Maskable-Interrupts (NMI)
    » e.g., kernel segmentation fault
Interrupt Controller

- 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

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
  - When = 0:
    » Running in new Child process
  - When < 0:
    » Error! Must handle somehow
    » Running in original process
- All 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;
  cpid = fork();
  if (cpid > 0) { /* Parent Process */
    mypid = getpid();
    printf("[\%d] parent of [\%d], pid\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

- UNIX man pages: `fork(2), exec(3), wait(2), signal(3)`

fork2.c

```c
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);
}
...```

Administrivia (Con’t)

- Participation: Attend section! Get to know your TA!

- Please use private Piazza posts only for student logistics issues

- Group sign up via autograder then TA form next week (after EDD)
  - Get finding groups of 4 people ASAP
  - Priority for same section; if cannot make this work, keep same TA
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`

---

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: fork3.c

```c
int i;
cpid = fork();
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

<table>
<thead>
<tr>
<th>User Mode</th>
<th>Kernel Mode</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applications</td>
<td>(the users)</td>
<td>Applications</td>
</tr>
<tr>
<td>Standard Libs</td>
<td>shells and commands, compilers and interpreters, system libraries</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>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>
</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

<table>
<thead>
<tr>
<th>Proc 1</th>
<th>Proc 2</th>
<th>Proc n</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>App</td>
<td>login</td>
</tr>
<tr>
<td>OS library</td>
<td>OS library</td>
<td>OS library</td>
</tr>
<tr>
<td>OS library</td>
<td>OS library</td>
<td>OS library</td>
</tr>
</tbody>
</table>

A Kind of Narrow Waist

- Compilers
- Word Processing
- Web Servers
- Email
- Databases
- Portable OS Library
- System Call Interface
- Application / Service
- User
- System
- Portable OS Kernel
- Platform support, Device Drivers
- Hardware
- x86
- PowerPC
- ARM
- Ethernet (1Gbs/10Gbs)
- 802.11 a/g/n/ac
- SCSI
- Graphics
- Thunderbolt
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**
  - streams
  - handles
  - registers
  - descriptors
- **Low Level I/O**
  - Syscall
- **File System**
- **I/O Driver**

The File System Abstraction

- **High-level idea**
  - Files live in hierarchical namespace of filenames
- **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
  - Links and Volumes (later)

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