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
3 types of 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, …

All 3 are an **UNPROGRAMMED CONTROL TRANSFER**

Where does it go?

Recall: User/Kernel (Privileged) Mode

**User Mode**
- Limited HW access
- Full HW access

**Kernel Mode**

How do we get the system target address of the “unprogrammed control transfer?”
Interrupt Vector

- Where else do you see this dispatch pattern?

Simple B&B: User => Kernel

- How to return to system?

Simple B&B: Interrupt

- How to save registers and set up system stack?

Simple B&B: Interrupt

- How to save registers and set up system stack?
What’s wrong with this simplistic address translation mechanism?

- **Fragmentation:**
  - Kernel has to somehow fit whole processes into contiguous block of memory
  - After a while, memory becomes fragmented!
- **Sharing:**
  - Very hard to share any data between Processes or between Process and Kernel
  - Simple segmentation
Virtual Address Translation

- Simpler, more useful schemes too!
- Give every process the illusion of its own BIG FLAT ADDRESS SPACE
  - Break it into pages
  - More on this later

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 decide which user process to run?
  - How do we represent user processes in the OS?
  - 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 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();
}
```

Putting it together: web server

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 ?????

---

Page 6
**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**

- **THIS** Friday (1/27) 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
    » 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 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);
}
```

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) {
    /* 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);
}
```

UNIX Process Management

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) {
```

Got top!
**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?

**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