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

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

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?
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();
}
```

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: A simple address translation (B&B)

- Can the program touch OS?
- Can it touch other programs?

Simple B&B: User code running

- How does kernel switch between processes?
- First question: How to return to system?

Simple B&B: Interrupt

- How to save registers and set up system stack?

Simple B&B: Switch User Process
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 prevents any memory sharing by its very nature

Alternative: Address Mapping

- Virtual Address Space 1
  - Prog 1
    - Code 1
    - Stack 1
    - Heap 1
    - OS code
    - OS data
    - OS heap & Stacks
  - Translation Map 1

- Virtual Address Space 2
  - Prog 2
    - Code 2
    - Stack 2
    - Heap 2
    - OS code
    - OS data
    - OS heap & Stacks
  - Translation Map 2

Processor Registers

- CS: EIP
- SS: ESP
- DS: EAX
- ES: ECX
- ESI
- EDI

Start address, length and access rights associated with each segment
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! Try to keep same section; if cannot make this work, keep same TA

Administrivia (Con’t)

- Conflict between Midterm 2 and EE16A
  - We are thinking of moving Midterm 2 from Wed 11/18
  - Possibilities: Mon 11/23 (my favorite) or Mon 11/16
- Midterm 1 conflicts
  - I know about one problem with Midterm 1 scheduling, and it can be dealt with. Have I missed any others?
- Finals conflicts: We will not be moving the exam or providing makeup finals...
  - I don’t know of any current conflicts
  - If you have a significant conflict that you think should cause us to change our policy, let me know now (note that CS186 is not conflicting any more).

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 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, ...

Recall: User/Kernal(Priviledged) Mode

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

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:

Exception Stack

During

User-level Process

code:
handler() {
  pusha
  ...
}

stack:

Exception Stack

Kernel

code:

Before

User-level Process

Registers

Kernel

code:

Before

User-level Process

Registers

Kernel

code:

Before

User-level Process

Registers

Kernel

code:

Before

User-level Process

Registers

Kernel

code:
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, 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 of the state of original process duplicated in both Parent and Child!
  - Memory, File Descriptors (next topic), etc...
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

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

exec
   main () {
      ...
   }

wait
```

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) {}
}
```

HW1
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

User Mode

- Applications (the users)
- Standard Libs
  - shells and commands
  - compilers and interpreters
  - system libraries

Kernel Mode

- system-call interface to the kernel
- signals terminal handling
- character I/O system
- terminal drivers
- file system
- swapping block I/O system
- disk and tape drivers
- CPU scheduling
- page replacement
- demand paging
- virtual memory

Hardware

- terminal controllers
- terminals
- device controllers
- disks and tapes
- memory controllers
- physical memory

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
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

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