Recall: What is an operating system?

- Special layer of software that provides application software access to hardware resources
  - Convenient abstraction of complex hardware devices
  - Protected access to shared resources
  - Security and authentication
  - Communication amongst logical entities

Very Brief History of OS

- Several Distinct Phases:
  - Hardware Expensive, Humans Cheap
    - Eniac, ... Multics

“I think there is a world market for maybe five computers.” – Thomas Watson, chairman of IBM, 1943
Very Brief History of OS

• Several Distinct Phases:
  – Hardware Expensive, Humans Cheap
    » Eniac, … Multics
  – Hardware Cheaper, Humans Expensive
    » PCs, Workstations, Rise of GUIs
  – Hardware Really Cheap, Humans Really Expensive
    » Ubiquitous devices, Widespread networking

Thomas Watson was often called “the world's greatest salesman” by the time of his death in 1956

Very Brief History of OS

• Several Distinct Phases:
  – Hardware Expensive, Humans Cheap
    » Eniac, … Multics
  – Hardware Cheaper, Humans Expensive
    » PCs, Workstations, Rise of GUIs
  – Hardware Really Cheap, Humans Really Expensive
    » Ubiquitous devices, Widespread networking

Rapid change in hardware leads to changing OS
  – Batch ⇒ Multiprogramming ⇒ Timesharing ⇒ Graphical UI ⇒ Ubiquitous Devices
  – Gradual migration of features into smaller machines

• Today
  – Small OS: 100K lines / Large: 10M lines (5M browser!)
  – 100-1000 people-years
Today: Four Fundamental OS Concepts

- **Thread**
  - Single unique execution context: fully describes program state
  - 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

OS Bottom Line: Run Programs

- Load instruction and data segments of executable file into memory
- Create stack and heap
- “Transfer control to program”
- Provide services to program
- While protecting OS and program
Recall (61B): Instruction Fetch/Decode/Execute

The instruction cycle

- Processor
  - PC: Instruction fetch
  - Registers
  - ALU
  - Execute
    - Data
      - Registers
      - ALU
      - Memory

Recall (61C): What happens during program execution?

Execution sequence:
- Fetch Instruction at PC
- Decode
- Execute (possibly using registers)
- Write results to registers/mem
- PC = Next Instruction(PC)
- Repeat

First OS Concept: Thread of Control

- Certain registers hold the context of thread
  - Stack pointer holds the address of the top of stack
    - Other conventions: Frame pointer, Heap pointer, Data
  - May be defined by the instruction set architecture or by compiler conventions
- Thread: Single unique execution context
  - Program Counter, Registers, Execution Flags, Stack
- A thread is executing on a processor when it is resident in the processor registers.
- PC register holds the address of executing instruction in the thread
- Registers hold the root state of the thread.
  - The rest is “in memory”

Second OS Concept: Program’s Address Space

- Address space ⇒ the set of accessible addresses + state associated with them:
  - For a 32-bit processor there are \(2^{32} = 4\) billion addresses
- What happens when you read or write to an address?
  - Perhaps nothing
  - Perhaps acts like regular memory
  - Perhaps ignores writes
  - Perhaps causes I/O operation
    - (Memory-mapped I/O)
  - Perhaps causes exception (fault)
### Address Space: In a Picture

**Processor registers**

- **PC:**
  - 0x000...
  - 0xFFF...

- **SP:**
  - 0xFFF...

- **Stack:**
  - 0x000...

- **Heap:**
  - 0xFFF...

- **Static Data:**
  - 0x000...

- **Instruction:**
  - 0xFFF...

- **Code Segment:**
  - 0xFFF...

- **What's in the code segment? Static data segment?**
- **What's in the Stack Segment?**
  - How is it allocated? How big is it?
- **What's in the Heap Segment?**
  - How is it allocated? How big?

### Multiprogramming - Multiple Threads of Control

- **vCPU1**
- **vCPU2**
- **vCPU3**

- **Shared Memory**

- **Time**

### Administrivia: Getting started

- **Start homework 0 immediately ⇒ Due next Monday (1/29)!**
  - cs162-xx account, Github account, registration survey
  - Vagrant and VirtualBox – VM environment for the course
  - Consistent, managed environment on your machine
  - Get familiar with all the cs162 tools, submit to autograder via git
  - Homework slip days: You have 3 slip days

- **THIS Friday (1/26) is early drop day! Very hard to drop afterwards…**

- **Should be going to section already!**

- **Group sign up form will be out after drop deadline**
  - Work on finding groups ASAP: 4 people in a group!
  - Try to attend either same section or 2 sections by same TA
The Basic Problem of Concurrency

- The basic problem of concurrency involves resources:
  - Hardware: single CPU, single DRAM, single I/O devices
  - Multiprogramming API: processes think they have exclusive access to shared resources
- OS has to coordinate all activity
  - Multiple processes, I/O interrupts, ...
  - How can it keep all these things straight?
- Basic Idea: Use Virtual Machine abstraction
  - Simple machine abstraction for processes
  - Multiplex these abstract machines
- Dijkstra did this for the “THE system”
  - Few thousand lines vs 1 million lines in OS 360 (1K bugs)

Properties of this simple multiprogramming technique

- All virtual CPUs share same non-CPU resources
  - I/O devices the same
  - Memory the same
- Consequence of sharing:
  - Each thread can access the data of every other thread (good for sharing, bad for protection)
  - Threads can share instructions (good for sharing, bad for protection)
  - Can threads overwrite OS functions?
- This (unprotected) model is common in:
  - Embedded applications
  - Windows 3.1/Early Macintosh (switch only with yield)
  - Windows 95—ME (switch with both yield and timer)

Protection

- Operating System must protect itself from user programs
  - Reliability: compromising the operating system generally causes it to crash
  - Security: limit the scope of what processes can do
  - Privacy: limit each process to the data it is permitted to access
  - Fairness: each should be limited to its appropriate share of system resources (CPU time, memory, I/O, etc)
- It must protect User programs from one another
- Primary Mechanism: limit the translation from program address space to physical memory space
  - Can only touch what is mapped into process address space
- Additional Mechanisms:
  - Privileged instructions, in/out instructions, special registers
  - Syscall processing, subsystem implementation
    » (e.g., file access rights, etc)
- 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
  - Processes provides memory protection
  - Threads more efficient than processes (later)
- Fundamental tradeoff between protection and efficiency
  - Communication easier within a process
  - Communication harder between processes
- 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?

Fourth OS Concept: Dual Mode Operation

- Hardware provides at least two modes:
  - “Kernel” mode (or “supervisor” or “protected”)
  - “User” mode: Normal programs executed
- What is needed in the hardware to support “dual mode” operation?
  - A bit of state (user/system mode bit)
  - Certain operations / actions only permitted in system/kernel mode
    - In user mode they fail or trap
  - User → Kernel transition sets system mode AND saves the user PC
    - Operating system code carefully puts aside user state then performs the necessary operations
  - Kernel → User transition clears system mode AND restores appropriate user PC
    - return-from-interrupt

User/Kernel (Privileged) Mode

- Limited HW access
- Full HW access

Administrivia (Cont’d)

- Anthony’s OH: Wednesdays 1-3PM 465F Soda starting 1/31
- Jonathan’s OH: Mondays 1-2:45PM 525 Soda starting 1/29
- Avoid private Piazza posts – others have same question
- Three Free Online Textbooks:
  - Click on “Resources” link for a list of “Online Textbooks”
  - Can read O’Reilly books for free as long as on campus or VPN
    - One book on Git, two books on C
- Webcast: [https://CalCentral.Berkeley.edu/](https://CalCentral.Berkeley.edu/) (CalNet sign in)
  - Webcast is *NOT* a replacement for coming to class!
5 min break

Simple Protection: Base and Bound (B&B)

- Program operates in an address space that is distinct from the physical memory space of the machine
- Requires relocating loader
- Still protects OS and isolates program
- No addition on address path

Another idea: Address Space Translation
A simple address translation with Base and Bound

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

Simple B&B: OS gets ready to execute process

Simple B&B: User Code Running

Tying it together: Simple B&B: OS loads process

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

• Privileged Inst: set special registers
• RTU
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?

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: Switch User Process

- How to save registers and set up system stack?

Simple B&B: "resume"

- How to save registers and set up system stack?

Conclusion: 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