CS162
Operating Systems and Systems Programming
Lecture 2

Introduction to Processes

January 24th, 2019

Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu

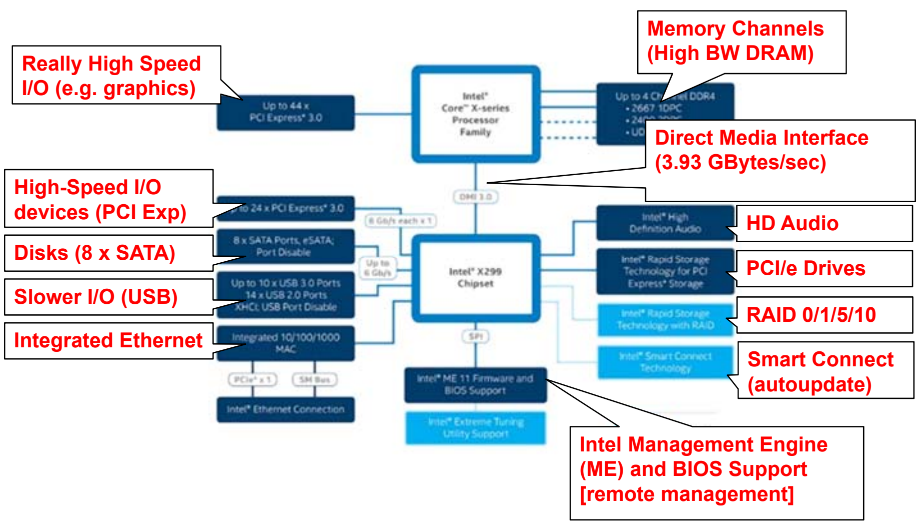
Review: What is an Operating System?

- Referee
  - Manage sharing of resources, Protection, Isolation
    - » Resource allocation, isolation, communication

- Illusionist
  - Provide clean, easy to use abstractions of physical resources
    - » Infinite memory, dedicated machine
    - » Higher level objects: files, users, messages
    - » Masking limitations, virtualization

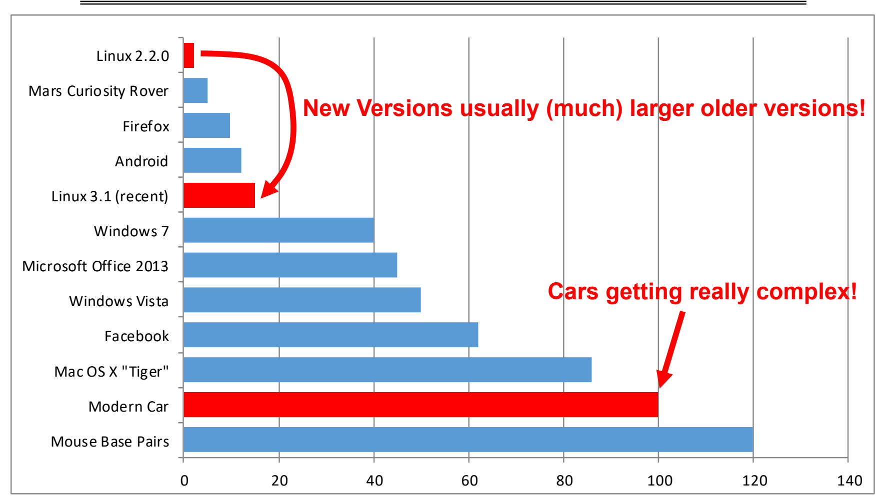
- Glue
  - Common services
    - » Storage, Window system, Networking
    - » Sharing, Authorization
    - » Look and feel

Recall: HW Functionality ⇒ great complexity!



Intel Skylake-X I/O Configuration

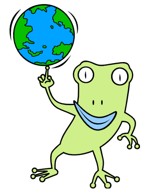
Recall: Increasing Software Complexity



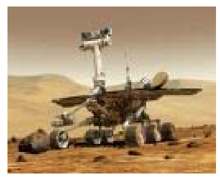
Millions of Lines of Code

(source https://informationisbeautiful.net/visualizations/million-lines-of-code/)

Example: Some Mars Rover ("Pathfinder") Requirements



- Pathfinder hardware limitations/complexity:
  - 20Mhz processor, 128MB of DRAM, VxWorks OS
  - cameras, scientific instruments, batteries, solar panels, and locomotion equipment
  - Many independent processes work together
- Can't hit reset button very easily!
  - Must reboot itself if necessary
  - Must always be able to receive commands from Earth
- Individual Programs must not interfere
  - Suppose the MUT (Martian Universal Translator Module) buggy
  - Better not crash antenna positioning software!
- Further, all software may crash occasionally
  - Automatic restart with diagnostics sent to Earth
  - Periodic checkpoint of results saved?



- Certain functions time critical:
  - Need to stop before hitting something
  - Must track orbit of Earth for communication
- A lot of similarity with the Internet of Things?
  - *Complexity,* QoS, Inaccessbility, Power limitations … ?

Very Brief History of OS

- Several Distinct Phases:

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*

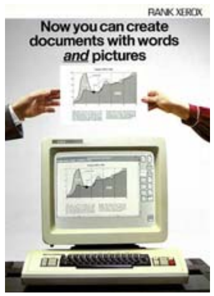
Very Brief History of OS

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

Thomas Watson was often called "the worlds greatest salesman" by the time of his death in 1956

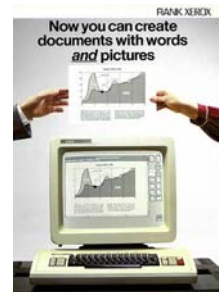
Very Brief History of OS

- Several Distinct Phases:
  – Hardware Expensive, Humans Cheap
    » Eniac, … Multics
  – Hardware Cheaper, Humans Expensive
    » PCs, Workstations, Rise of GUIs

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

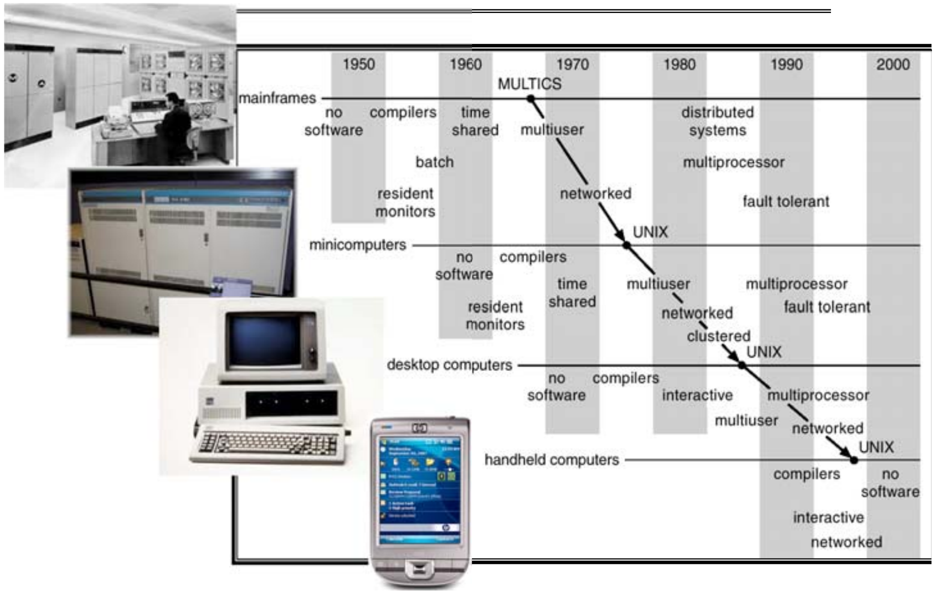
Very Brief History of OS

- 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

OS Archaeology

- Because of the cost of developing an OS from scratch, most modern OSes have a long lineage:
- Multics → AT&T Unix → BSD Unix → Ultrix, SunOS, NetBSD,…
- Mach (micro-kernel) + BSD → NextStep → XNU → Apple OS X, iPhone iOS
- MINIX → Linux → Android OS, Chrome OS, RedHat, Ubuntu, Fedora, Debian, Suse,…
- CP/M → QDOS → MS-DOS → Windows 3.1 → NT → 95 → 98 → 2000 → XP → Vista → 7 → 8 → 10 → …

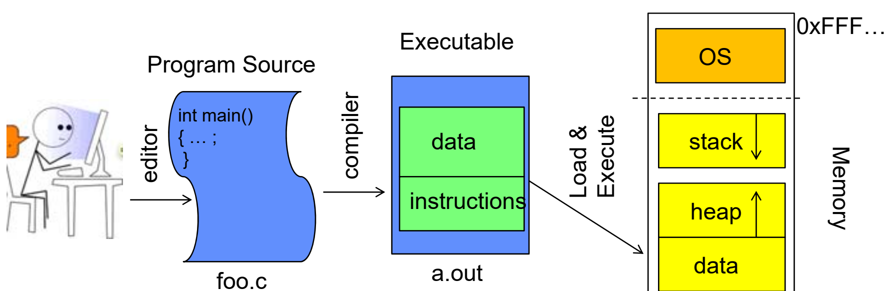
Migration of OS Concepts and Features



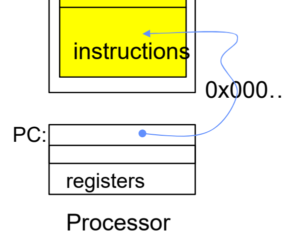
Today: Four Fundamental OS Concepts

- **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

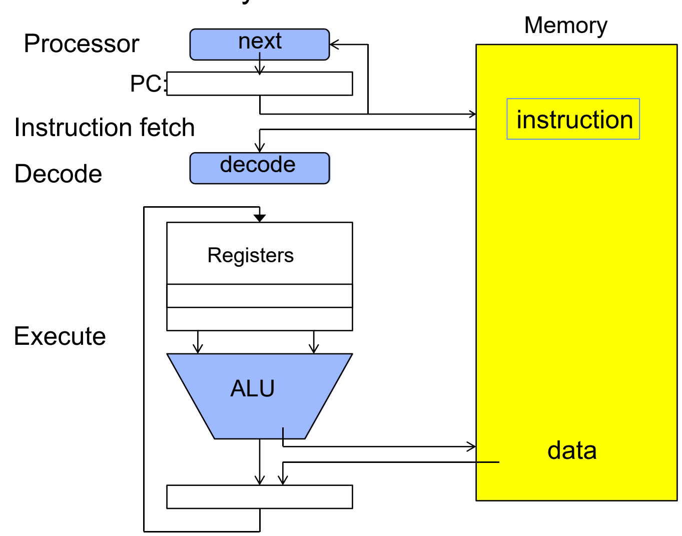


- Load instruction and data segments of executable file into memory
- Create stack and heap
- "Transfer control to program"
- Provide services to program
- While protecting OS and program

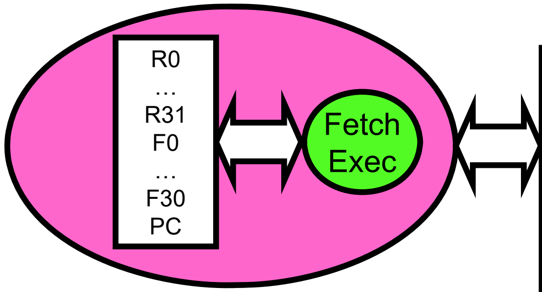


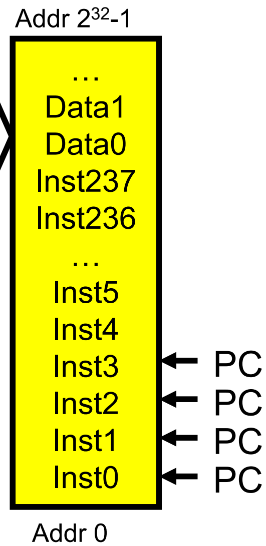
Recall (61C): Instruction Fetch/Decode/Execute

The instruction cycle



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





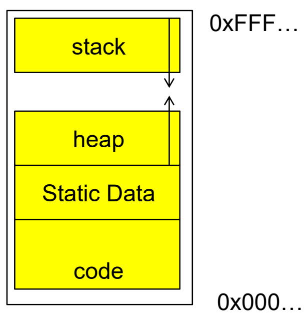
- Execution sequence:
  – Fetch Instruction at PC
  – Decode
  – Execute (possibly using registers)
  – Write results to registers/mem
  – PC = Next Instruction(PC)
  – Repeat

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"

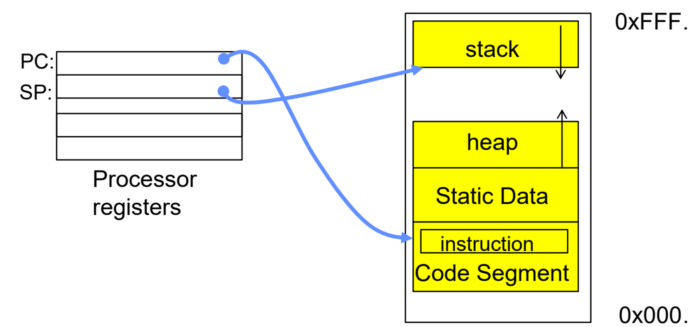
Second OS Concept: Program's Address Space

- 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



- 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?

Administrivia: Getting started

- Start homework 0 immediately ⇒ Due next Tuesday (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
- Should go to section tomorrow!
- Friday (2/1) is drop day!
  - Very hard to drop afterwards…
  - Please drop sooner if you are going to anyway ⇒ Let someone else in!
- Group sign up form out next week (due after drop deadline)
  - Start finding groups ASAP
  - 4 people in a group!
  - Try to attend either same section or 2 sections by same TA

Administrivia (Con't)

- Midterm conflicts:
  - There are a couple of people with midterm conflicts – we are still figuring out what to do (if anything)
- Kubiatowicz Office Hours:
  - 1pm-2pm, Monday/Thursday
  - May change as need arises (still have a bit of fluidity here)
- 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/ (CalNet sign in)
  - Webcast is *NOT* a replacement for coming to class!

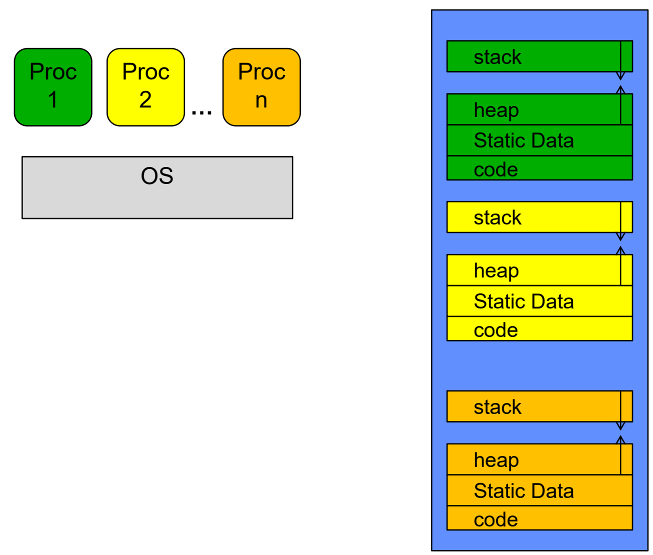
CS 162 Collaboration Policy

Explaining a concept to someone in another group
Discussing algorithms/testing strategies with other groups
Helping debug someone else's code (in another group)
Searching online for generic algorithms (e.g., hash table)

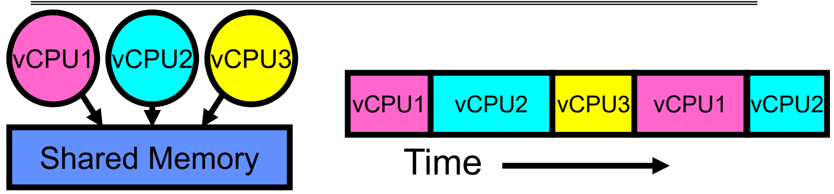
Sharing code or test cases with another group
Copying OR reading another group's code or test cases
Copying OR reading online code or test cases from from prior years

We compare all project submissions against prior year submissions and online solutions and will take actions (described on the course overview page) against offenders

Multiprogramming - Multiple Threads of Control



How can we give the illusion of multiple processors?



- Assume a single processor. How do we provide the illusion of multiple processors?
  - Multiplex in time!
- 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

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)

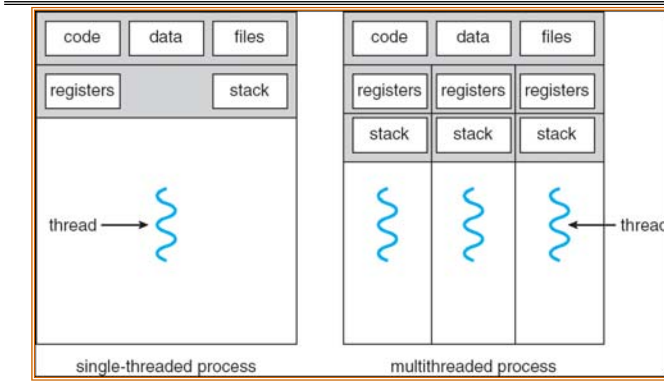
Protection

- 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)

Third OS Concept: Process

- 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

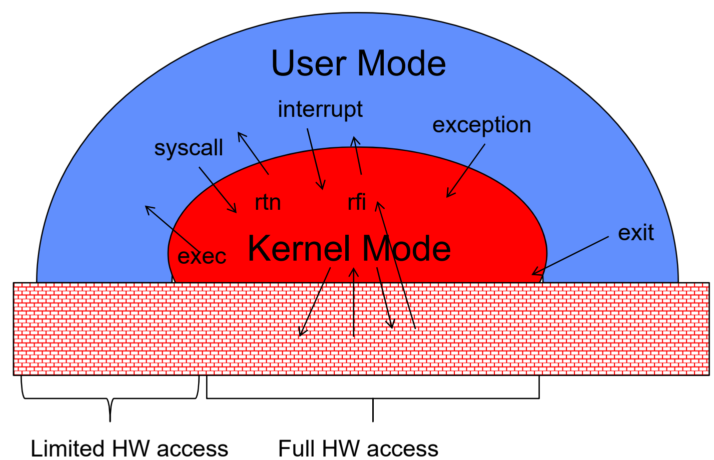


- Threads encapsulate concurrency: "Active" component
- Address spaces encapsulate protection: "Passive" part
  – Keeps buggy program from trashing the system
- Why have multiple threads per address space?

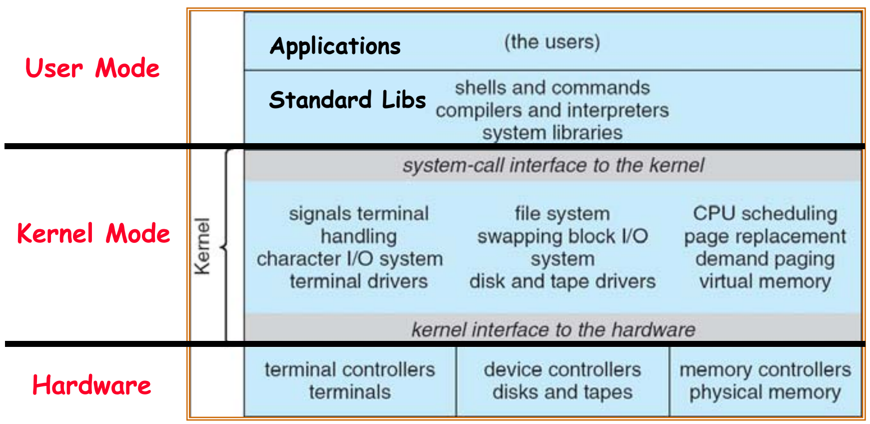
Fourth OS Concept: Dual Mode Operation

- 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

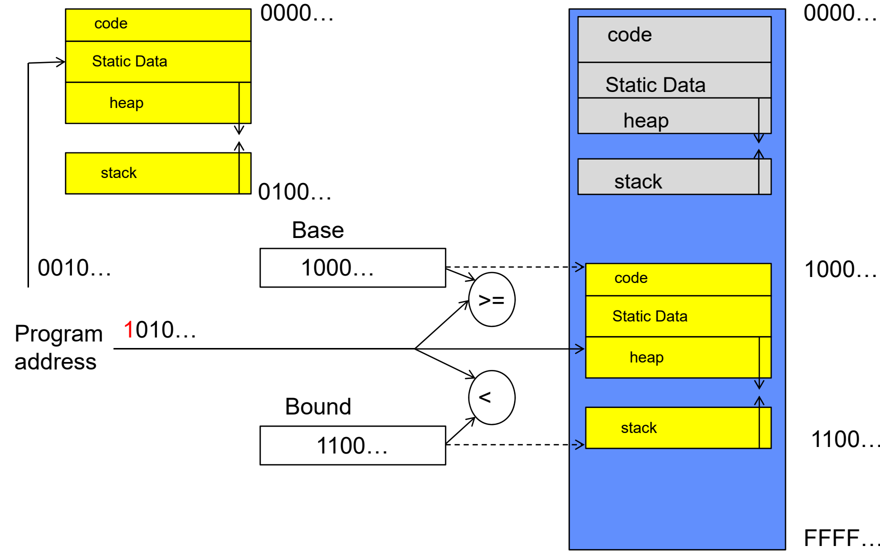
User/Kernel (Privileged) Mode



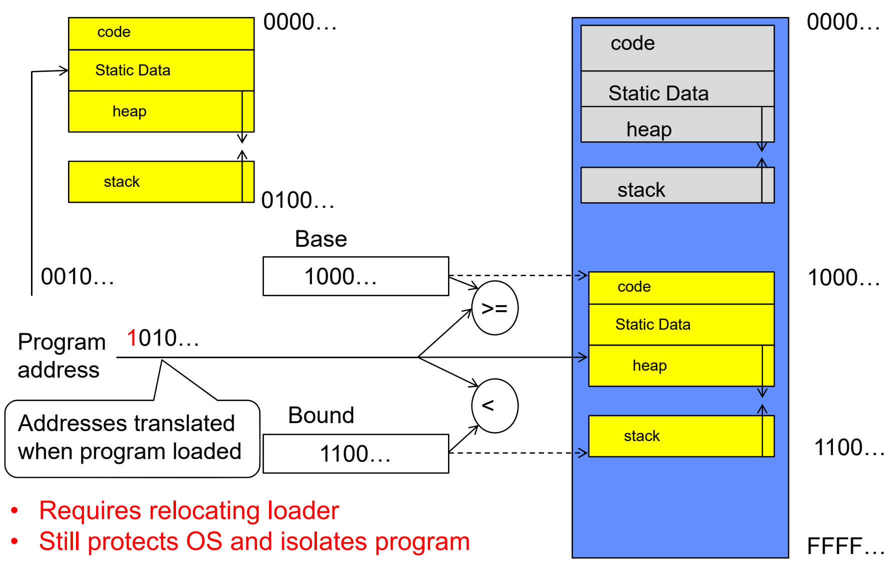
For example: UNIX System Structure



Simple Protection: Base and Bound (B&B)



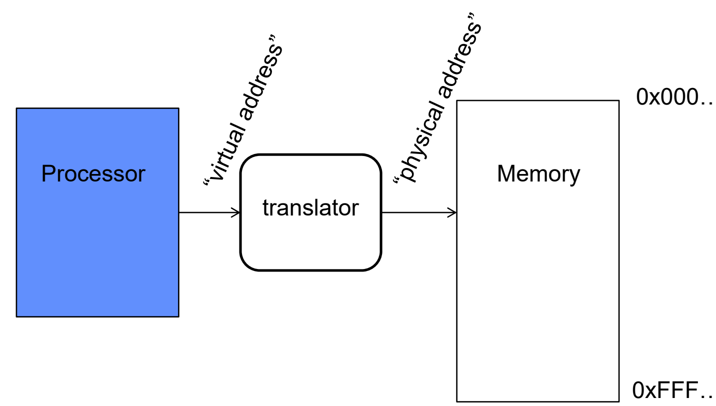
Simple Protection: Base and Bound (B&B)



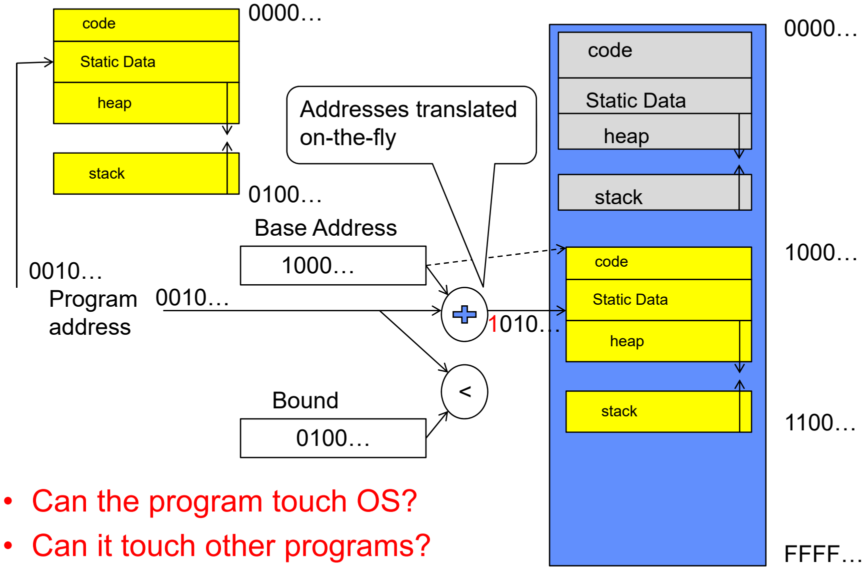
- Requires relocating loader
- Still protects OS and isolates program
- No addition on address path

Another idea: Address Space Translation

- Program operates in an address space that is distinct from the physical memory space of the machine

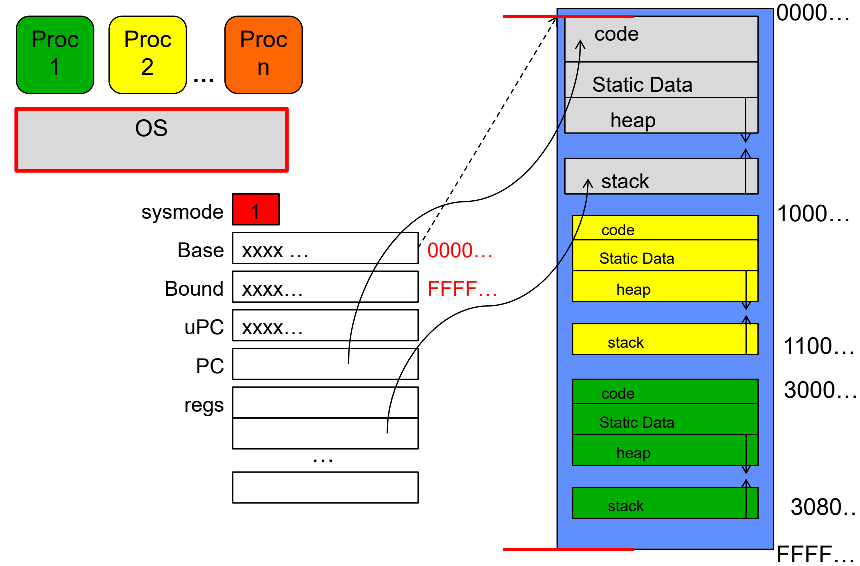


Simple address translation with Base and Bound

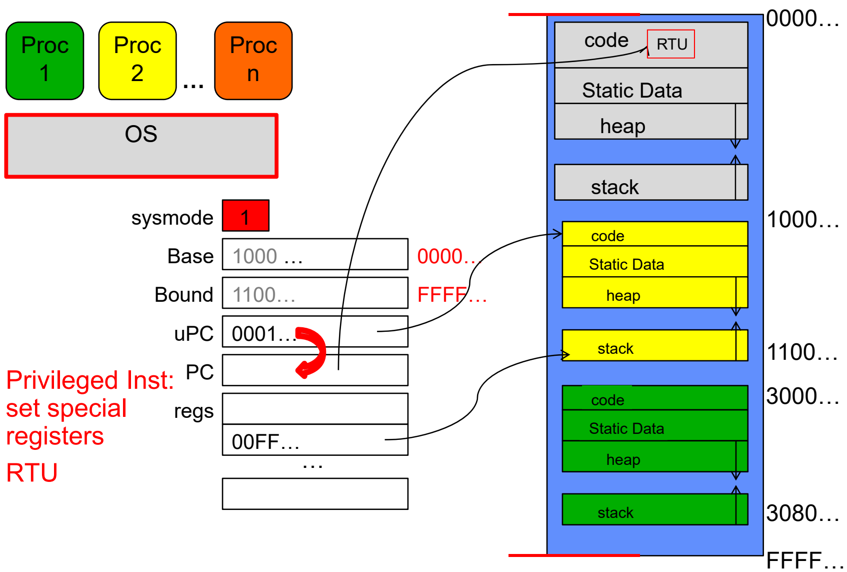


- Can the program touch OS?
- Can it touch other programs?

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

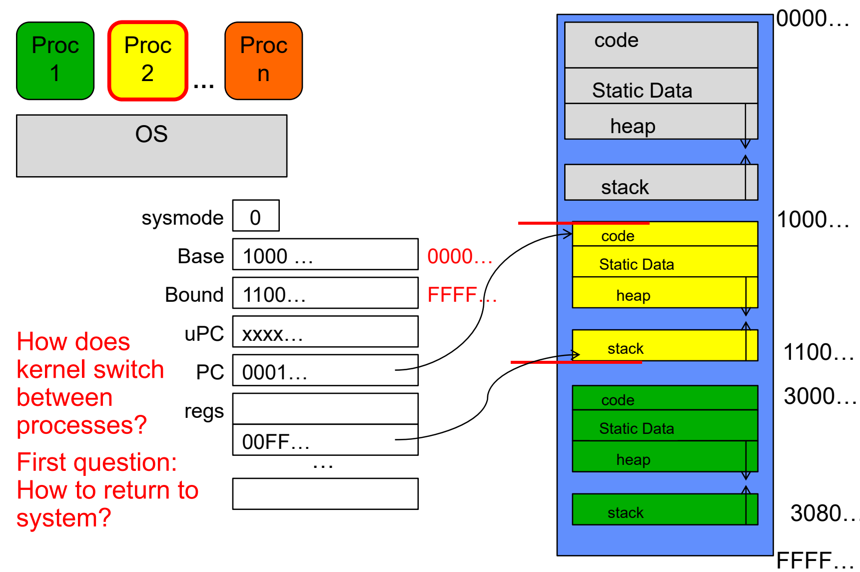


Simple B&B: OS gets ready to execute process



- Privileged Inst: set special registers
- RTU

Simple B&B: User Code Running



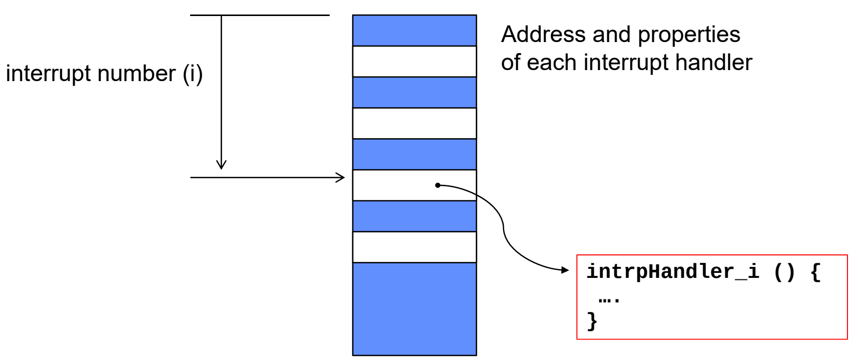
- How does kernel switch between processes?
- First question: How to return to system?

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?

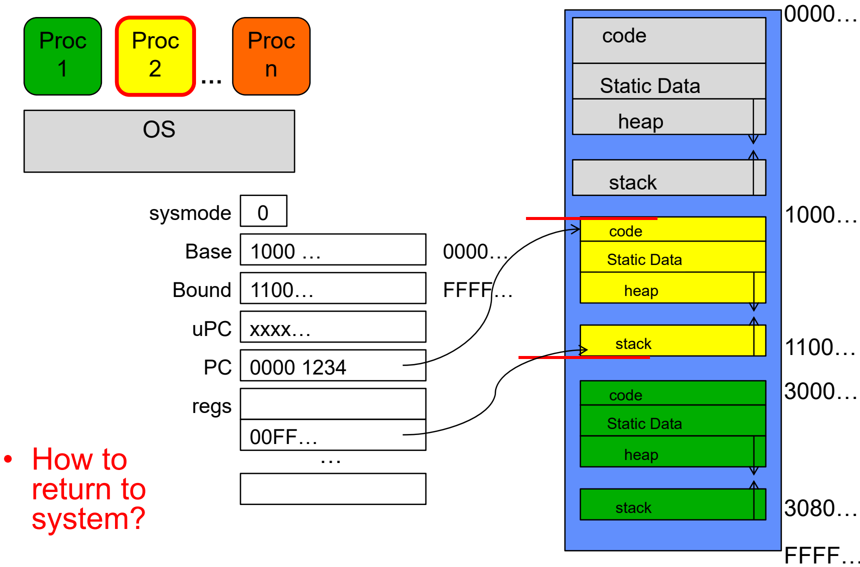
How do we get the system target address of the “unprogrammed control transfer?”

Interrupt Vector



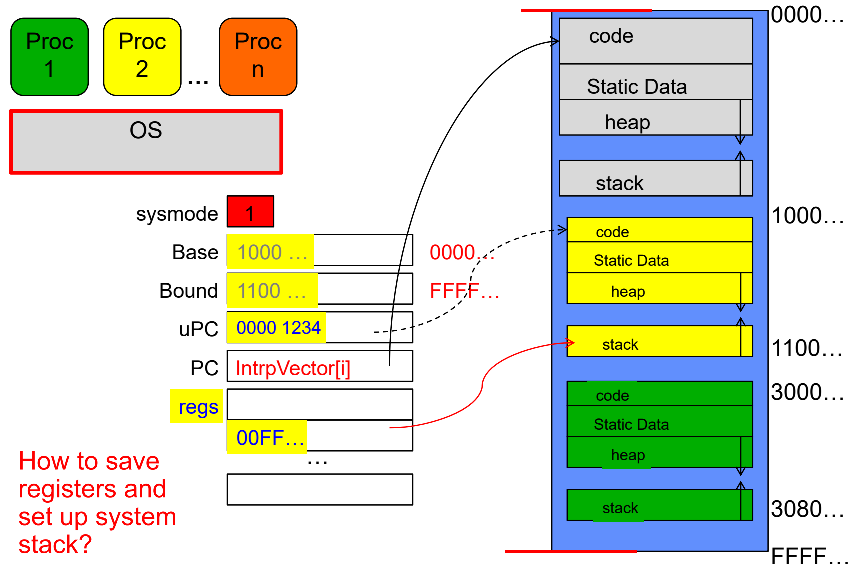
- Where else do you see this dispatch pattern?

Simple B&B: User => Kernel



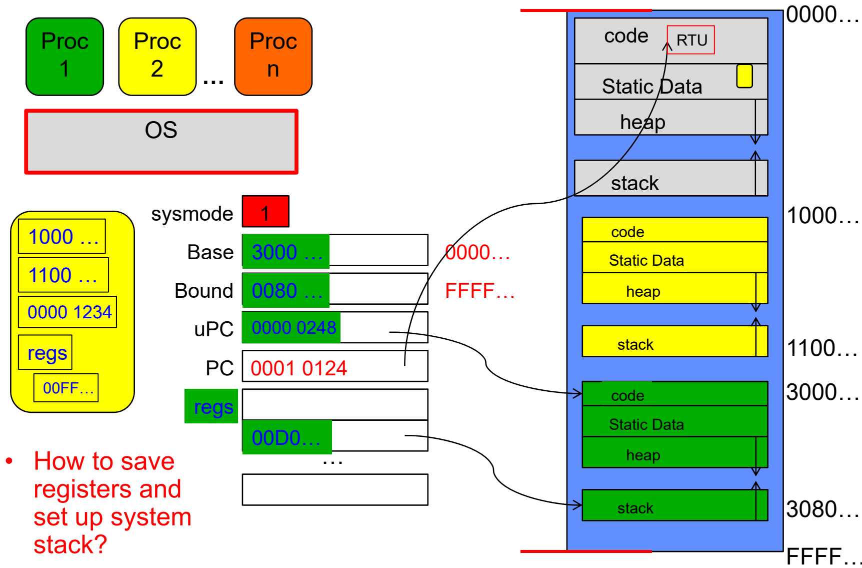
- How to return to system?

Simple B&B: Interrupt



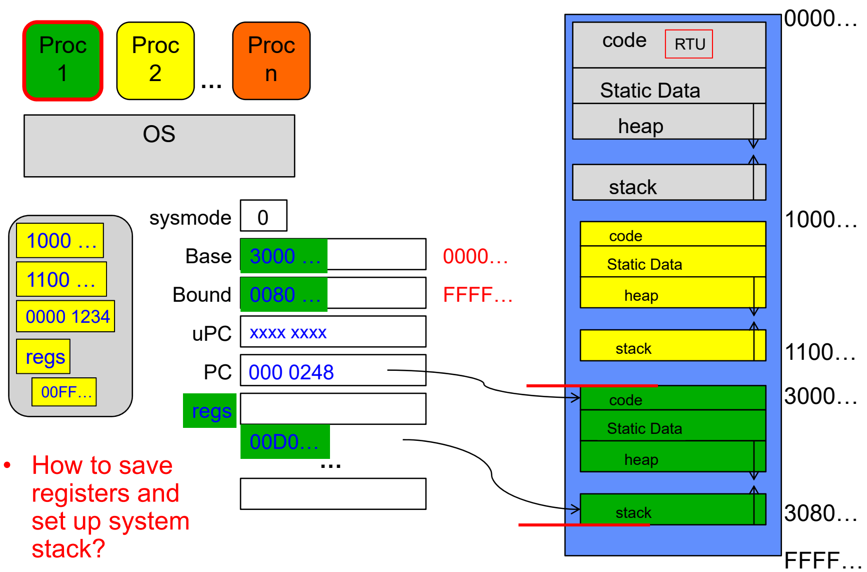
- How to save registers and set up system stack?

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?

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 are 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

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


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