CSE150
Operating Systems
Lecture 3

Processes, Threads and Address Spaces (contd.)

Threads (Review)

- Thread: Single unique execution context
  - Program Counter, Registers, Stack

Value	Description
0x00000001	Thread's object memory is not de-allocated
0x00000002	Thread's stack is not freed
0x00000004	Thread is terminated
0x00000008	Thread cannot be resumed
0x00000010	Interrupt service thread
0x00000020	Thread has used FPU/MMX/XMM
0x00000040	Timer service thread
0x00000080	Shutdown service thread

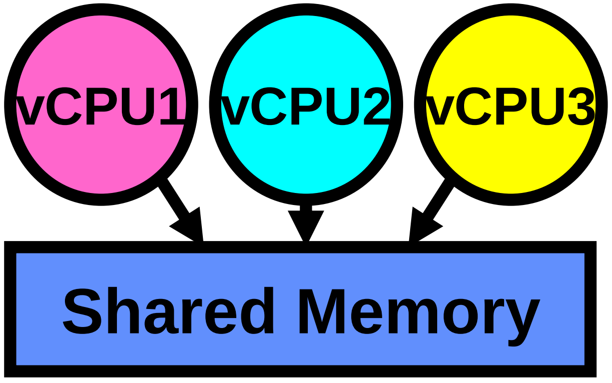
Threads (Review)

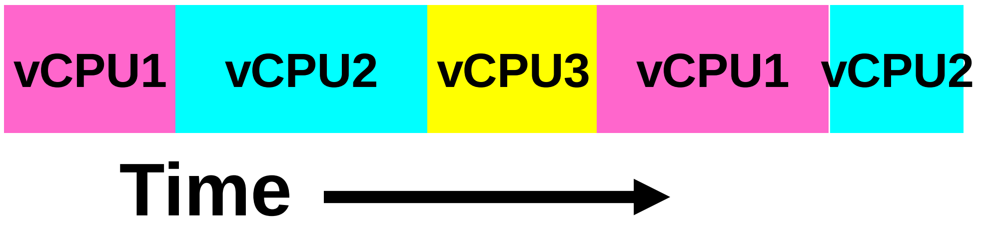
- Thread: Single unique execution context
  - Program Counter, Registers, Stack
- A thread is executing on a processor when it is resident in the processor registers.
- Program Counter (PC) register holds the address of executing instruction in the thread
- Certain registers hold the *context* of thread
  - 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
- Registers hold the root state of the thread.
  - The rest is “in memory”

The Basic Problem of Concurrency (Review)

- The basic problem of concurrency involves resources:
  - Hardware: single CPU, single DRAM, single I/O devices
  - Multiprogramming API: users think they have exclusive access to shared resources
- OS Has to coordinate all activity
  - Multiple users, I/O interrupts, …
  - How can it keep all these things straight?
- Basic Idea: Use Virtual Machine abstraction
  - Simple machine abstraction for processes
  - Then, worry about multiplexing these abstract machines

Illusion of multiple processors? (Review)

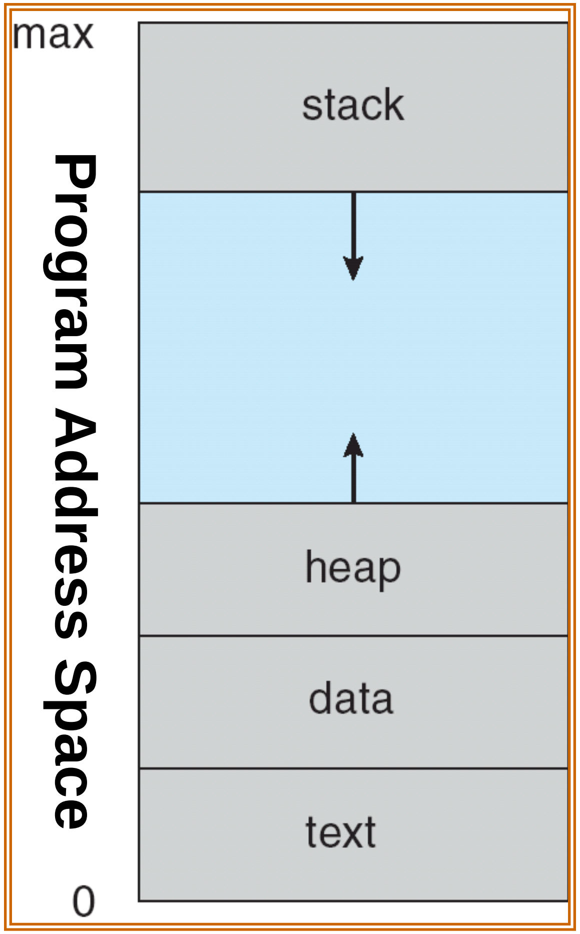




- Assume a single processor. How do we provide the illusion of multiple processors?
  – Multiplex in time!
- Each virtual "CPU" needs a structure to hold:
  – Program Counter (PC), Stack Pointer (SP)
  – Registers (Integer, Floating point, others…?)
- How switch from one CPU to the next?
  – Save PC, 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

Program’s Address Space (Review)

- Address space $\Rightarrow$ the set of accessible addresses + state associated with them:
  - For a 32-bit processor there are 2^{32} = 4 billion addresses.



Today: Fundamental OS Concepts (contd.)

- 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

Third OS Concept: Process

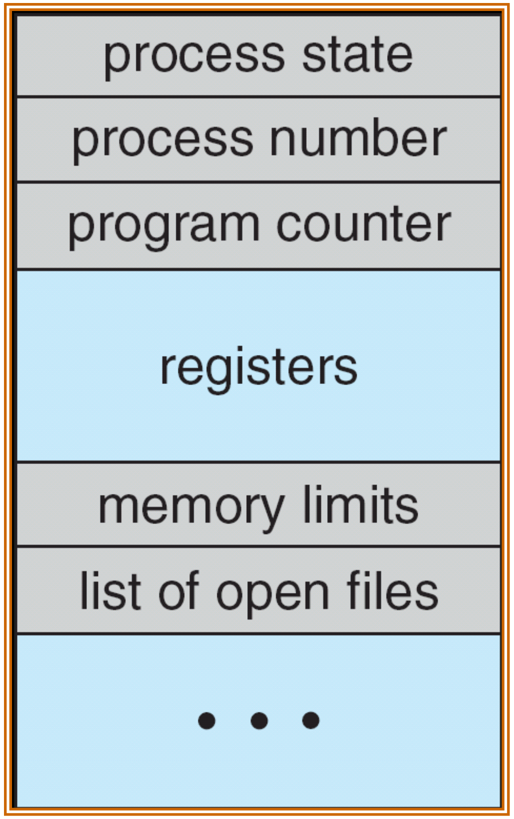
- **Process**: execution environment with Restricted Rights
  - – Address Space with One or More Threads
  - – Owns memory (address space)
  - – Owns file descriptors, file system context, …
  - – Encapsulate one or more threads sharing process resources
- Why processes?
  - – Protected from each other!
  - – OS Protected from them
  - – Threads more efficient than processes (later)
- Fundamental tradeoff between protection and efficiency
  - Communication easier *within* a process
  - Communication harder *between* processes

Traditional UNIX Process

- Process: *Operating system abstraction to represent what is needed to run a single program*
  - Often called a “HeavyWeight Process”
  - Formally: a single, sequential stream of execution in its *own* address space
- Two parts:
  - Sequential Program Execution Stream
    - » Code executed as a *single, sequential* stream of execution
    - » Includes State of CPU registers
  - Protected Resources:
    - » Main Memory State (contents of Address Space)
    - » I/O state (i.e. file descriptors)

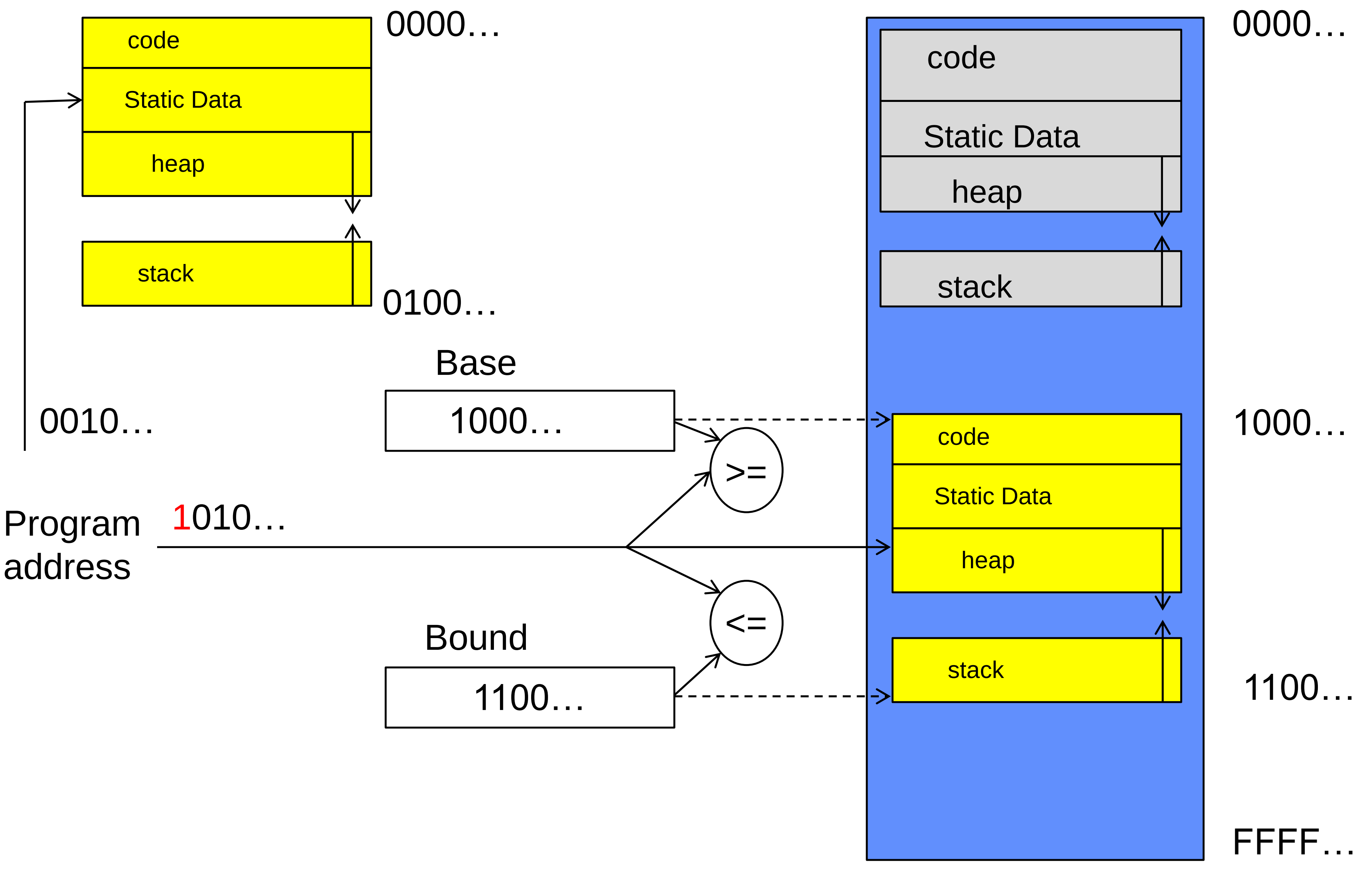
How do we multiplex processes?

- The current state of process held in a process control block (PCB):
  - This is a "snapshot" of the execution and protection environment
  - Only one PCB active at a time
- Give out CPU time to different processes (Scheduling):
  - Only one process "running" at a time
  - Give more time to important processes
- Give pieces of resources to different processes (Protection):
  - Controlled access to non-CPU resources
  - Sample mechanisms:
    - » Memory Mapping: Give each process their own address space
    - » Kernel/User duality: Arbitrary multiplexing of I/O through system calls

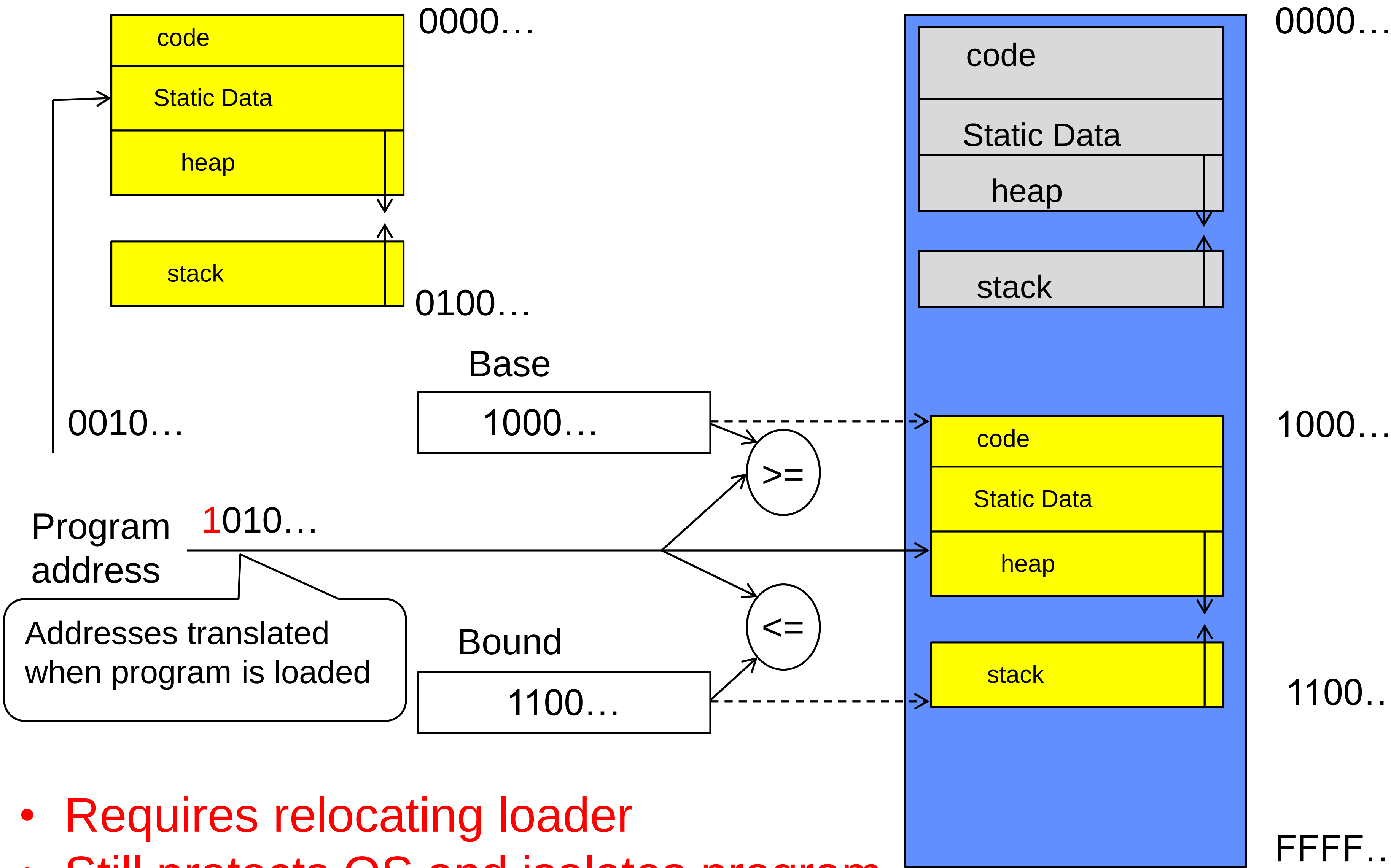


Process Control Block

Simple Protection: Base and Bound (B&B)

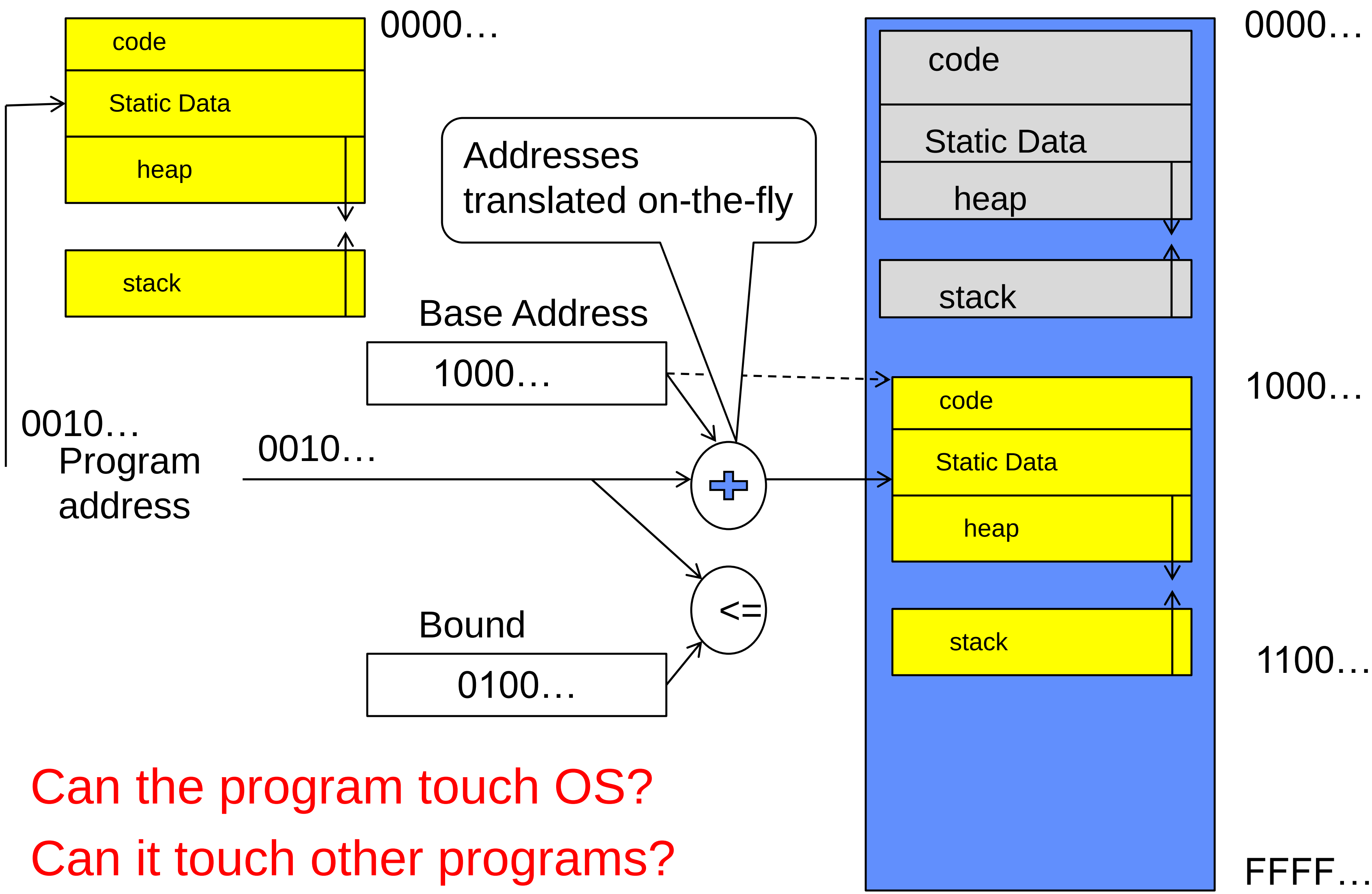


Simple Protection: Base and Bound (B&B)



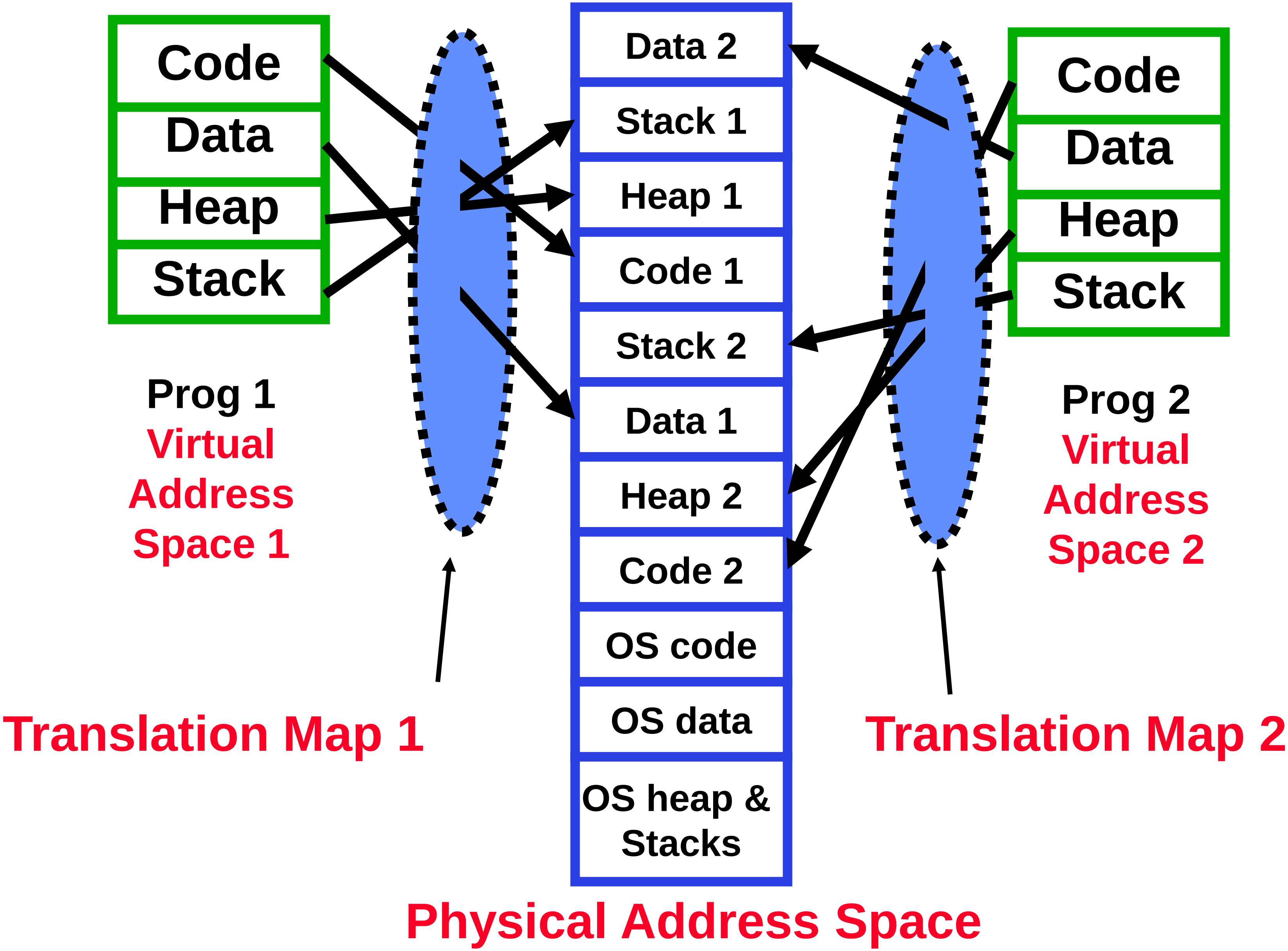
- Requires relocating loader
- Still protects OS and isolates program

A simple address translation with Base and Bound

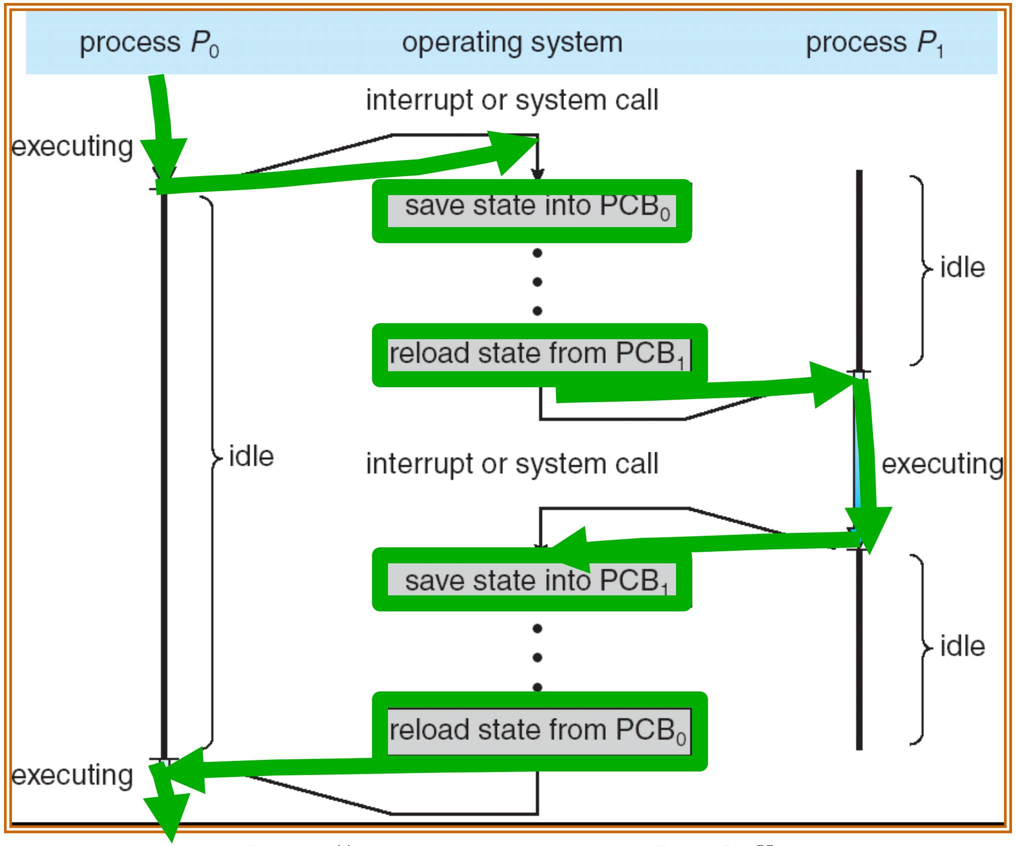


- Can the program touch OS?
- Can it touch other programs?

Providing Illusion of Separate Address Space: Load new Translation Map on Switch



CPU Switch From Process to Process

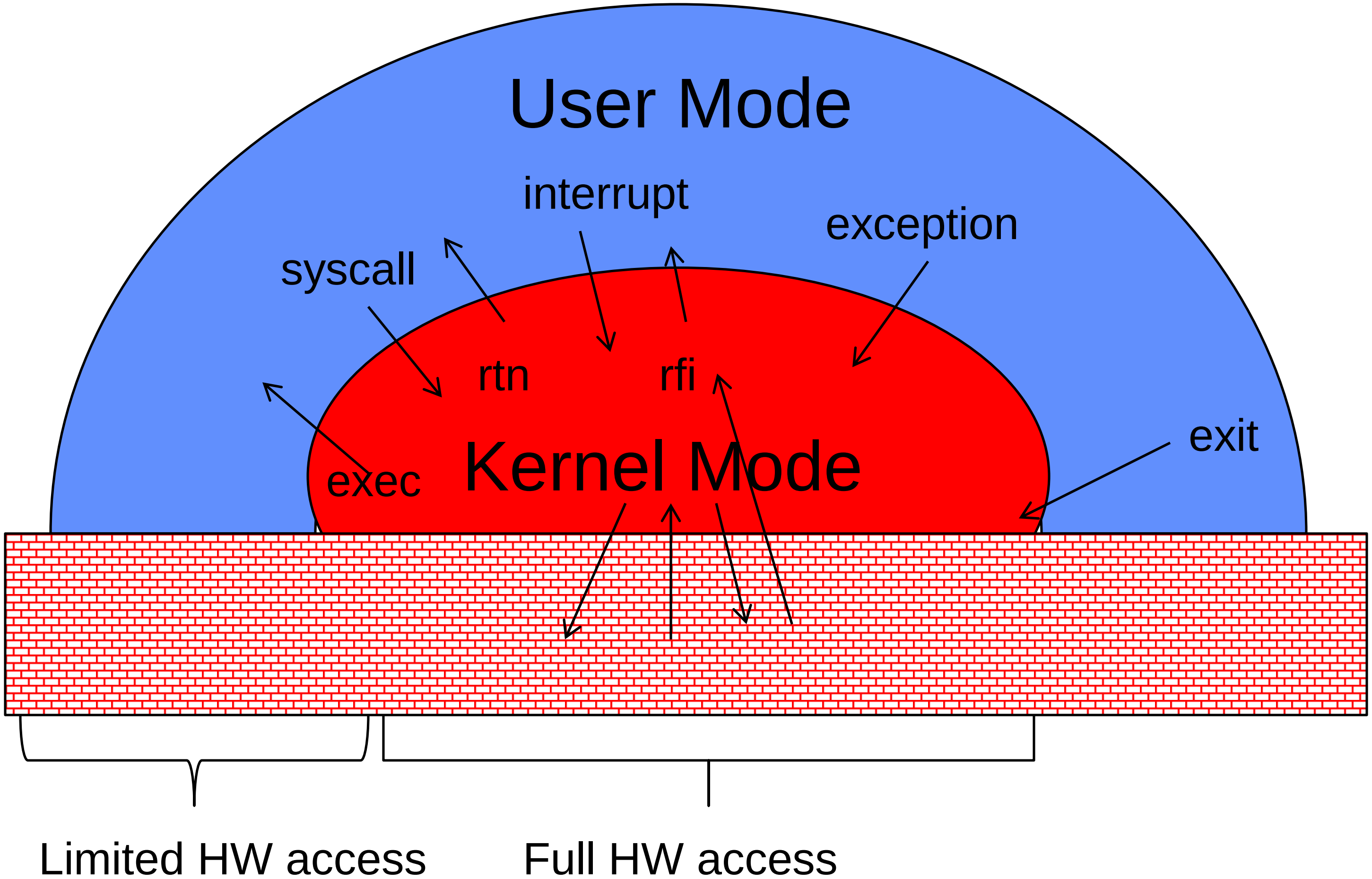


- This is also called a "context switch"
- Code executed in kernel above is overhead
  - Overhead sets minimum practical switching time

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
  - 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

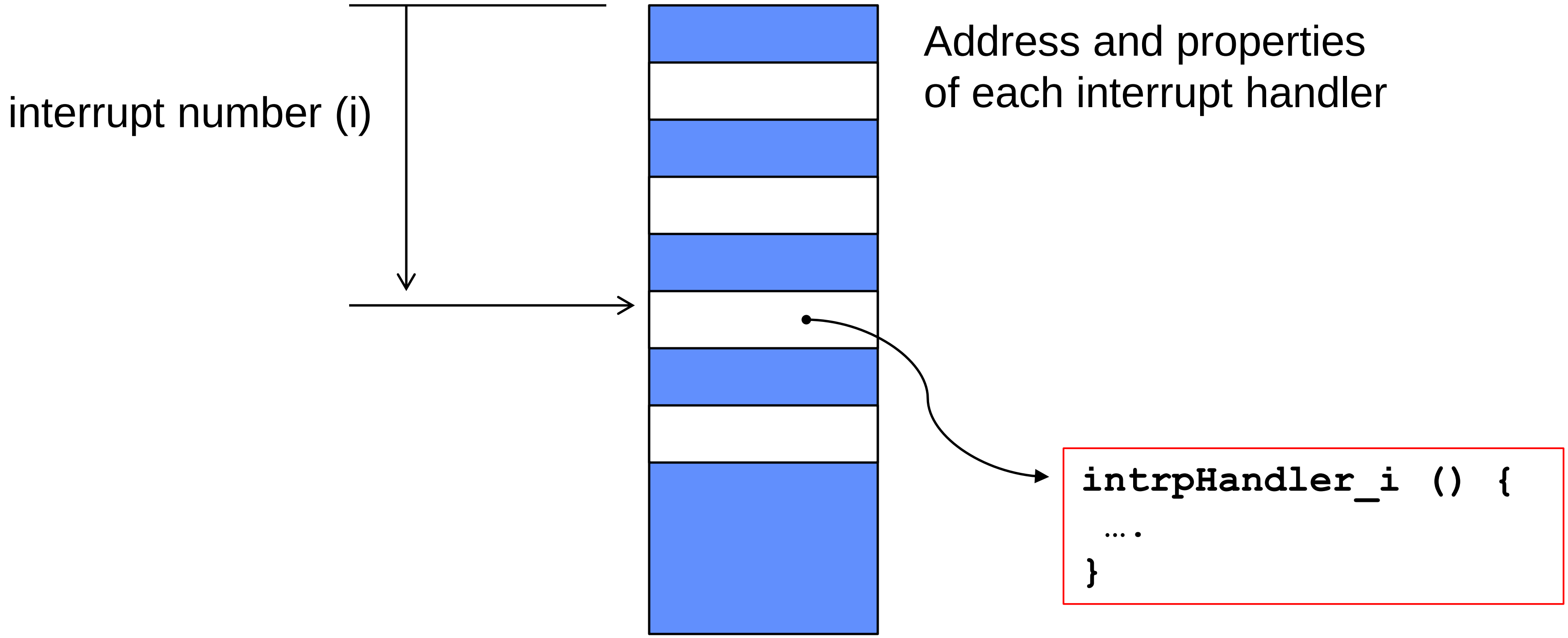


3 types of Mode Transfer

- 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
  - Put 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
  - 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



Summary

- Two more OS concepts
  - Processes (Threads + Address Space… more or less)
  - Dual Mode Operation (Hardware support for Protection)

- Base and Bounds
  - A simple protection mechanism used in Cray 1 using two registers
  - Modern systems use a translation map per process

- Book talks about processes
  - When this concerns concurrency, really talking about thread portion of a process
  - When this concerns protection, talking about address space portion of a process

- User/Kernel Mode
  - Must (carefully) control User→Kernel transitions
  - Transitions controlled via Syscalls, Interrupts, Traps/Exceptions