香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

Operating Systems
Lecture 2

Four fundamental concepts: thread, address space, process, dual mode

Prof. Yunming XIAO

School of Data Science

Acknowledgments: Ion Stoica and Matei Zaharia, Berkeley CS 162

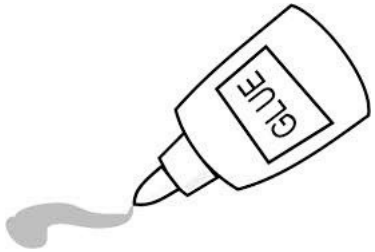
Recall: what is an Operating System?

Referee

Manage protection, isolation, and sharing of resources

Illusionist

Provide clean, easy-to-use abstractions of physical resources



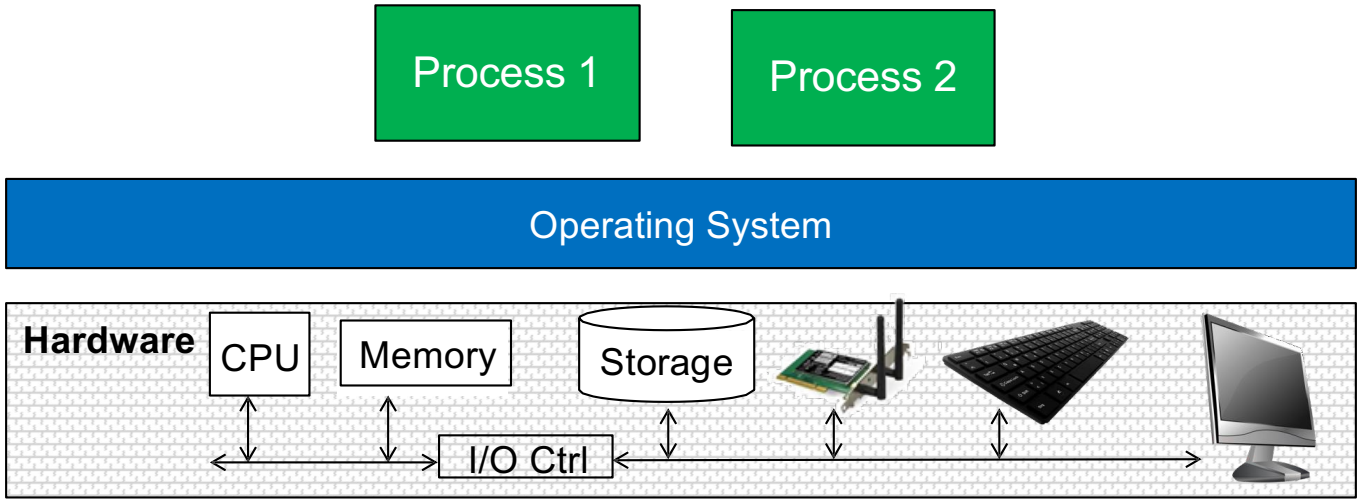
Glue

Provides a set of common services

Virtualization: Operating System

- Operating System virtualizes* hardware to processes (applications) providing high-level abstractions (e.g., process thread, sockets, etc)

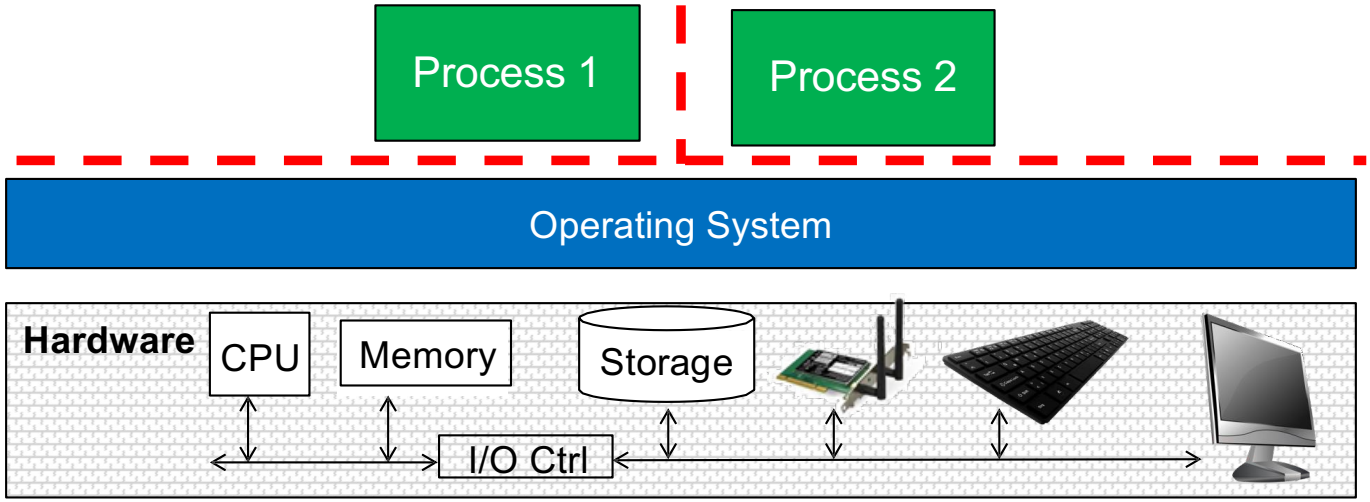
Process: an active instance of a program



***virtualization: provides the illusion that each process uses its own machine; also provides illusion if infinite memory and processor (though there will be hit in performance)**

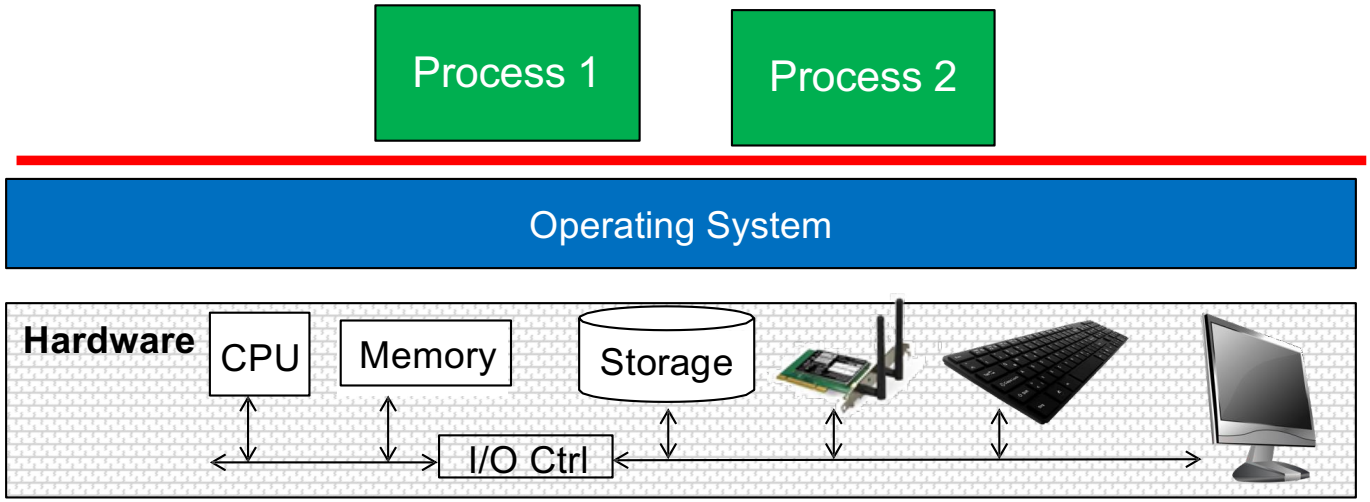
Virtualization: Operating System

- OS as referee:
  - Manage resource sharing, protection, isolation



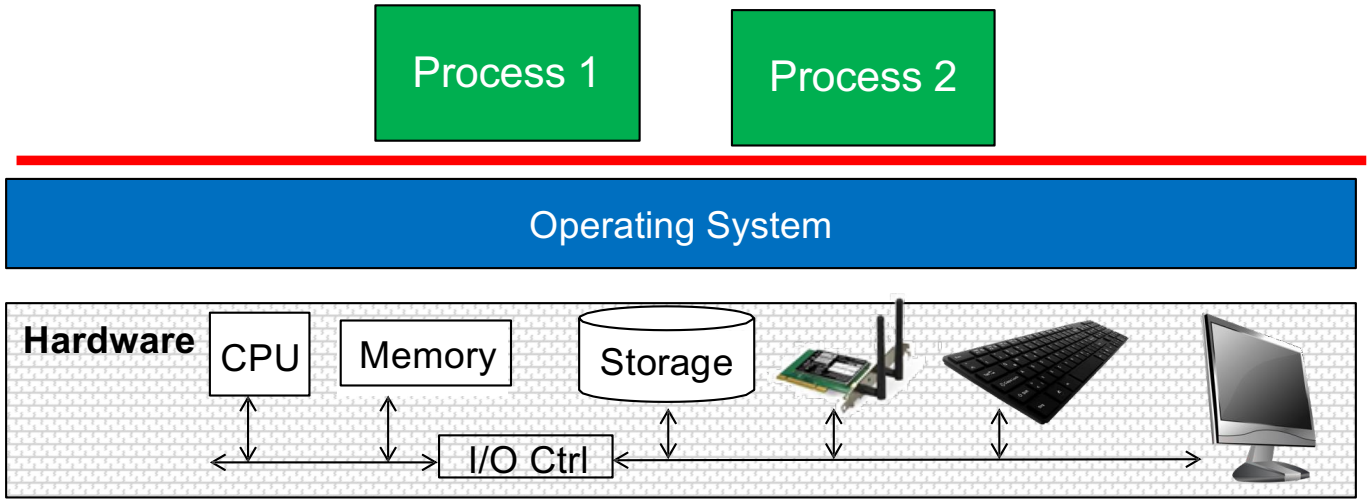
Virtualization: Operating System

- OS as an illusionist:
  - Hides hardware complexity;
  - Virtualizes hardware, e.g., provides clean abstractions



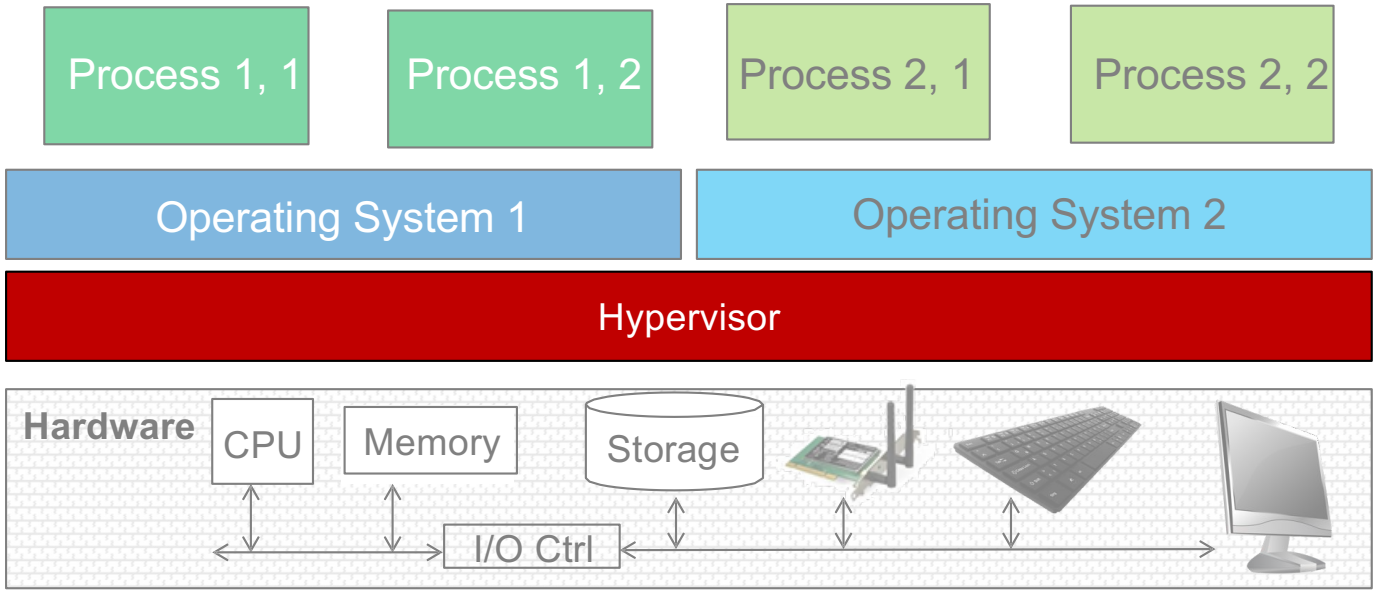
Virtualization: Operating System

- OS as glue:
  - Provides higher level services to make
  - programmer's job easier, e.g., file systems, networking, windows



Virtualization: Virtual Machine

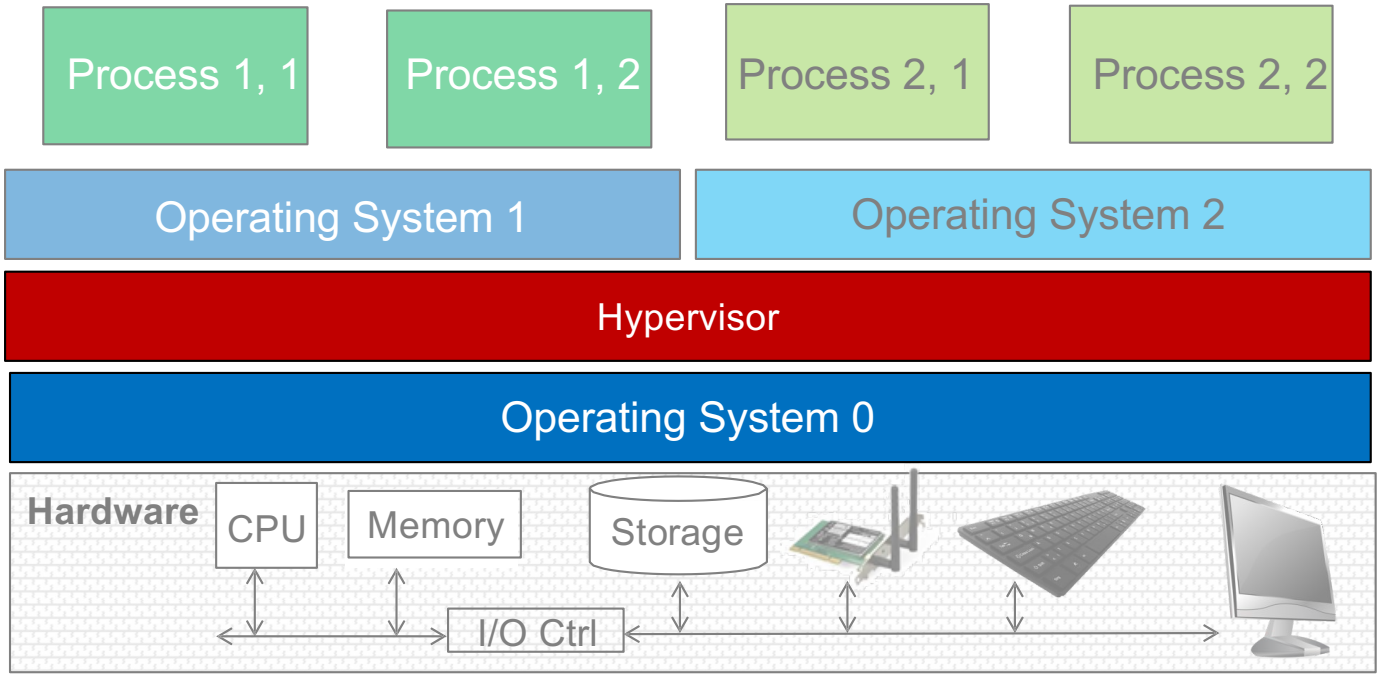
- Hypervisor creates virtual machines. A virtual machine virtualizes* hardware to Operating System providing same hardware abstraction



***virtualization: provides the illusion that each operating system is using the same physical machine; It provides fault isolation; if an OS crashes won't affect other OSes**

Virtualization: Virtual Machine

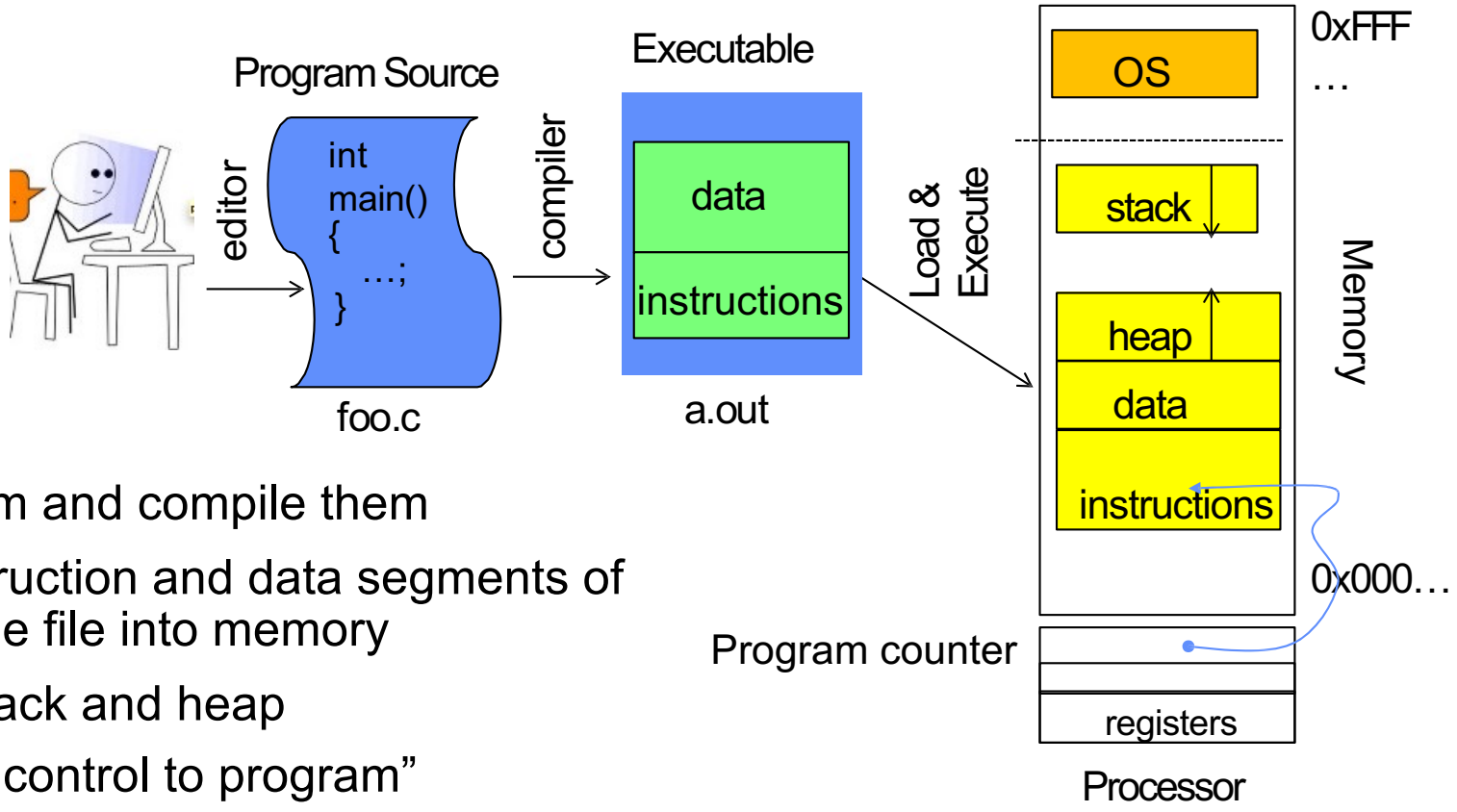
- Hypervisor creates virtual machines. A virtual machine virtualizes* hardware to Operating System providing same hardware abstraction



Today: four fundamental OS concepts

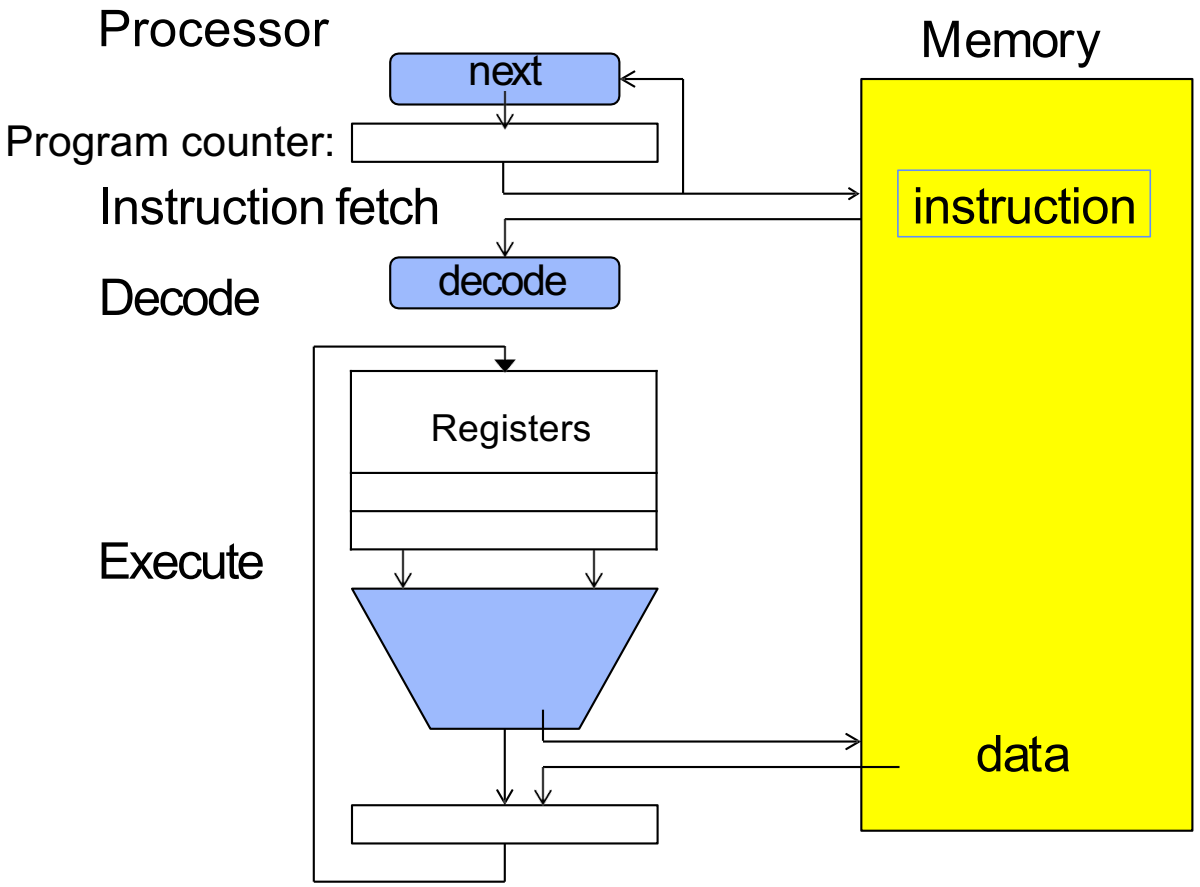
- **Thread: execution context**
  - Fully describes program state
  - Program Counter, Registers, Stack, Execution Flags,
- **Address space (with or w/o translation)**
  - Set of memory addresses accessible to program (for read or write)
  - May be distinct from memory space of the physical machine (in which case programs operate in a virtual address space)
- **Process: an instance of a running program**
  - Address Space + one or more Threads
- **Dual mode operation / Protection**
  - Only the “system” has the ability to access certain resources
  - Combined with translation, isolates programs from each other and the OS from programs

OS bottom line: run programs

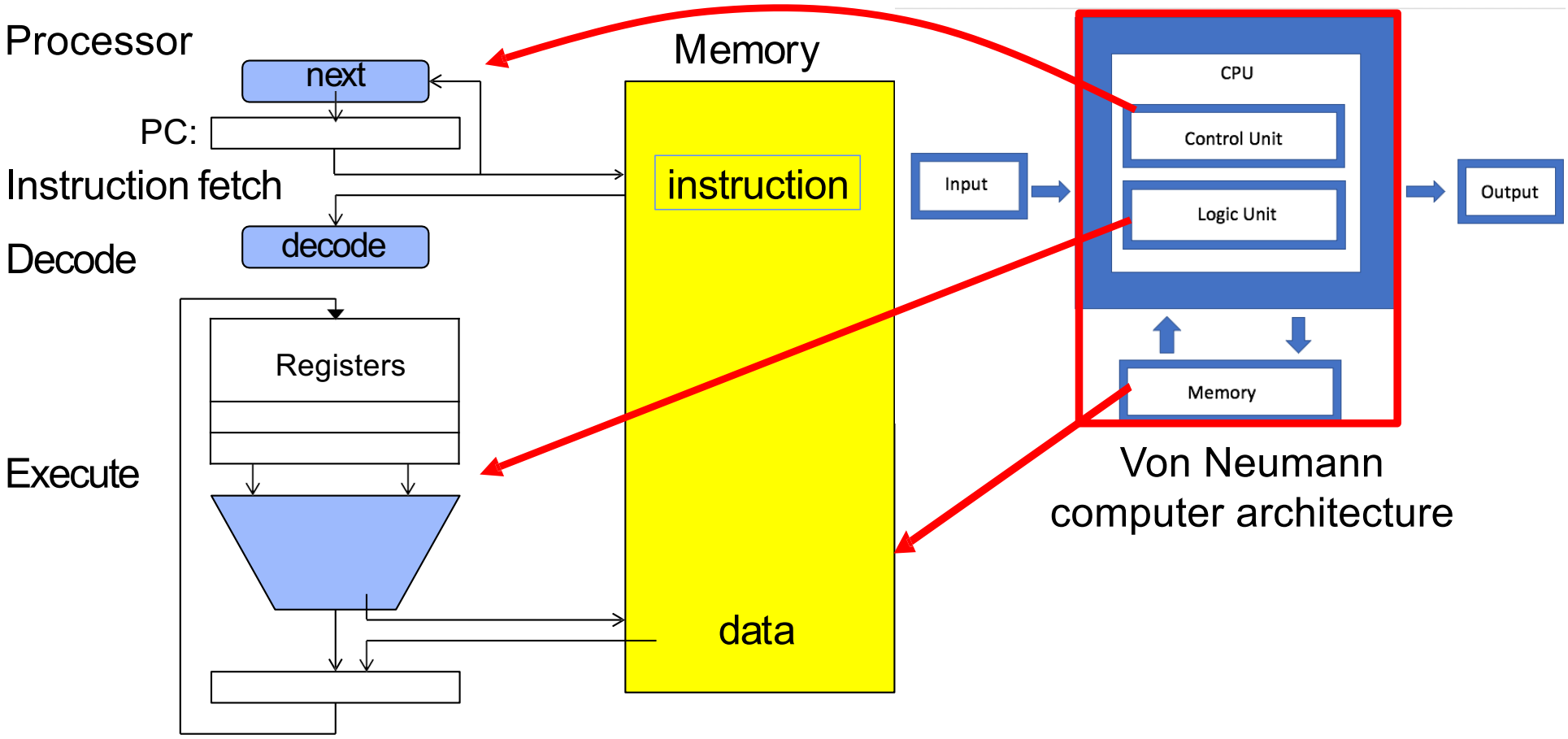


- Write them and compile them
- Load instruction and data segments of executable file into memory
- Create stack and heap
- “Transfer control to program”
- Provide services to program while protecting OS and program

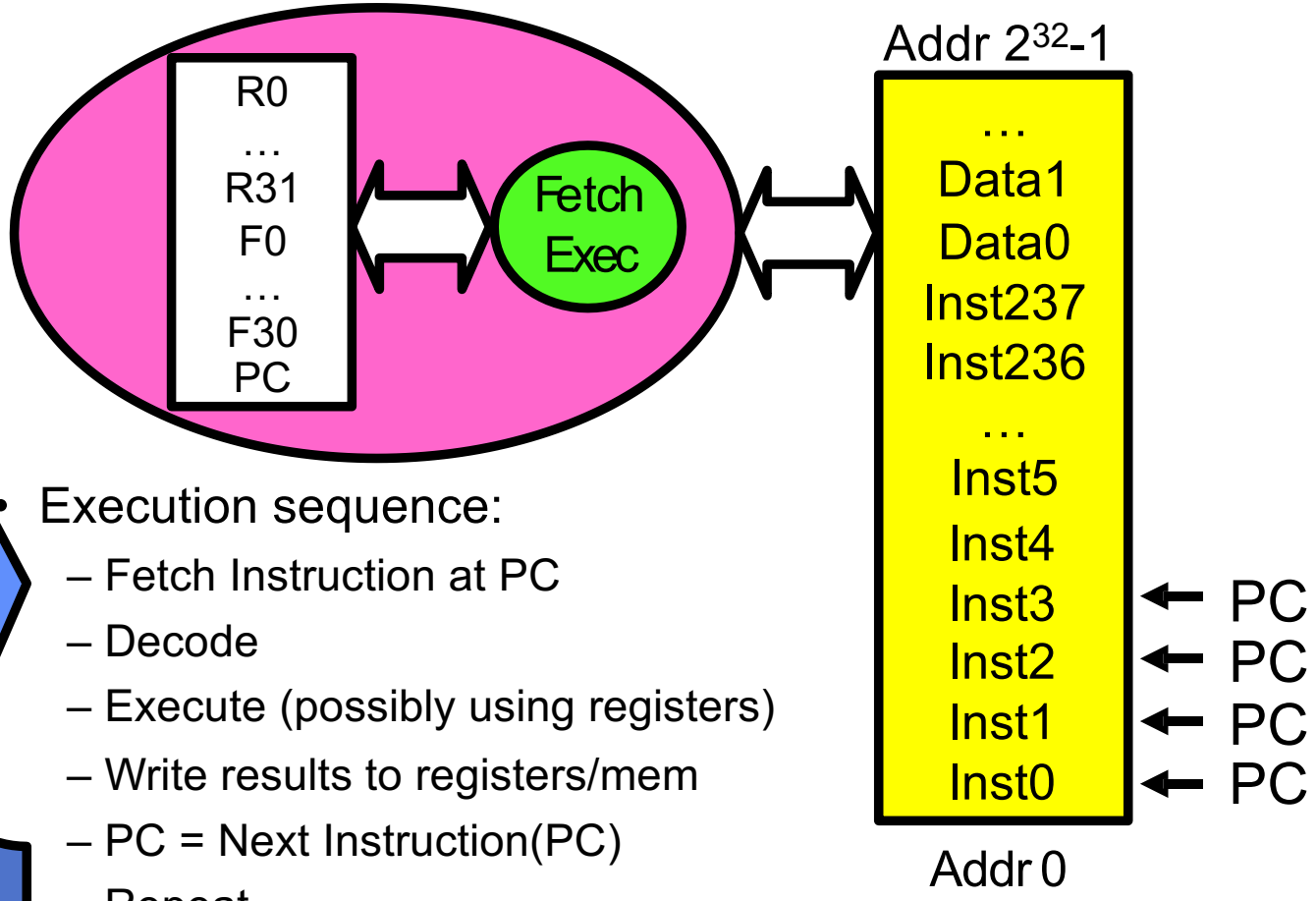
Instruction Fetch/Decode/Execute



Instruction Fetch/Decode/Execute



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

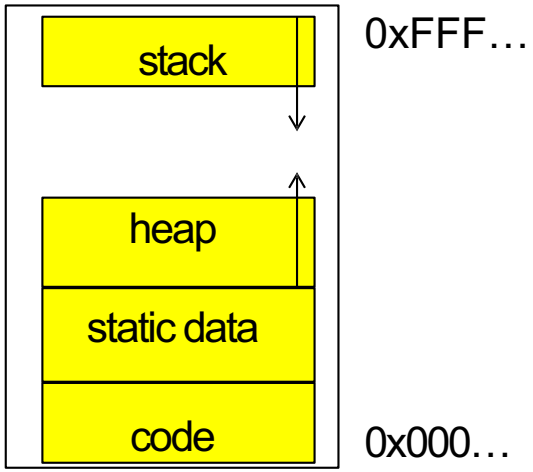
- Thread: single unique execution context
  - Program Counter, Registers, Execution Flags, Stack, Memory State
- A thread is *executing* on a processor (core) when it is *resident* in the processor registers
- Resident means: registers hold the root state (context) of the thread:
  - Including program counter (PC) register & currently executing instruction
    - PC points at next instruction in memory
    - Instructions stored in memory
  - Including intermediate values for ongoing computations
    - Can include actual values (like integers) or pointers to values in memory
  - Stack pointer holds the address of the top of stack (which is in memory)
  - The rest is “in memory”

First OS concept: thread of control

- A thread is suspended (not executing) when its state *is not* loaded (resident) into the processor
  - Processor state pointing at some other thread
  - Program counter register *is not* pointing at next instruction from this thread
  - Often: a copy of the last value for each register stored in memory

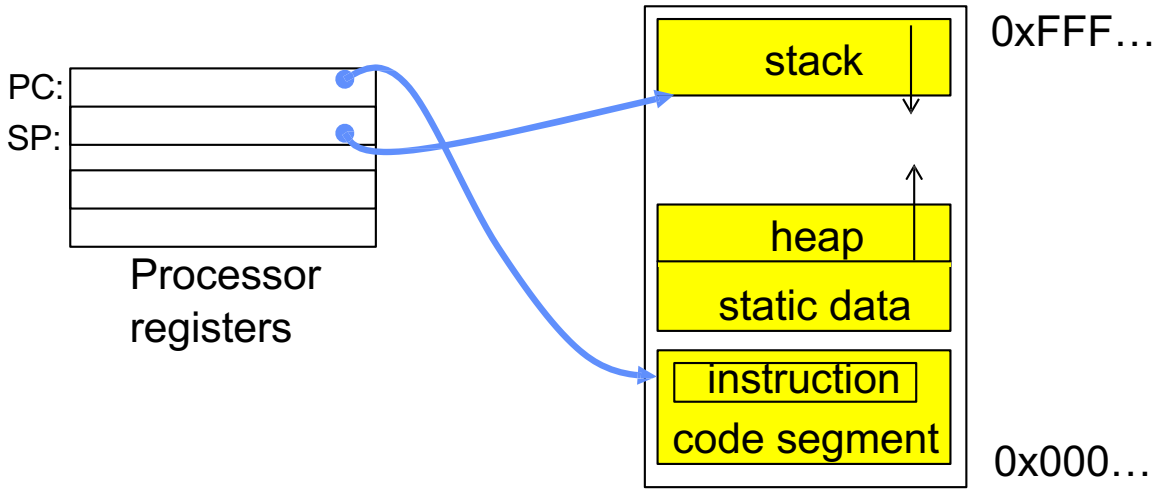
Second OS concept: address space

- Address space: the set of accessible addresses + state associated with them:
  - For 32-bit processor: 232 = 4 billion (10^9) addresses
  - For 64-bit processor: 264 = 18 quintillion (10^{18}) addresses



- What happens when you read or write to an address?
  - Perhaps acts like regular memory
  - Perhaps ignores writes
  - Perhaps causes I/O operation (Memory-mapped I/O)
  - Perhaps causes exception (fault)
  - Communicates with another program
  - ….

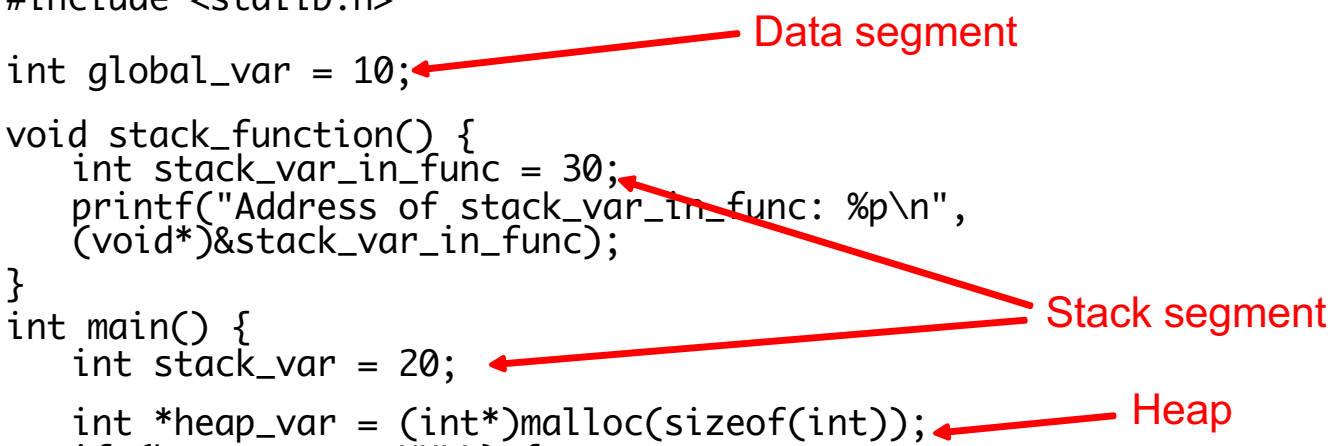
Address space in a picture



- 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 is it?

Simple example

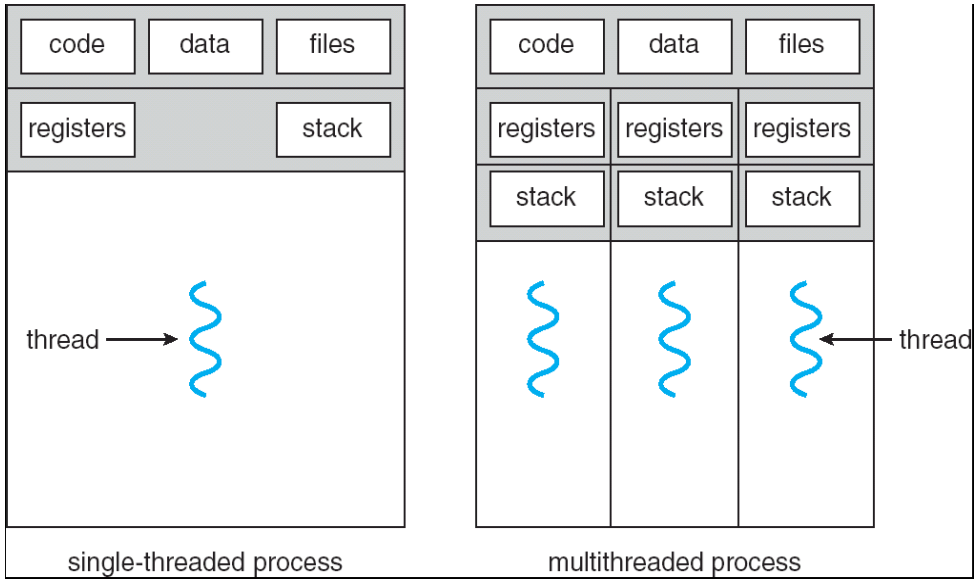
```
#include <stdio.h>
#include <stdlib.h>

int global_var = 10;

void stack_function() {
    int stack_var_in_func = 30;
    printf("Address of stack_var_in_func: %p\n",
    (void*)&stack_var_in_func);
}
int main() {
    int stack_var = 20;

    int *heap_var = (int*)malloc(sizeof(int));
    if (heap_var == NULL) {
        fprintf(stderr, "Failed to allocate memory on the
        heap\n");
        return 1;
    }
    *heap_var = 40; // Storing a value in the heap memory.

    // The programmer is responsible for freeing heap
    memory.  free(heap_var);

    return 0;
}
```



Third OS concept: process

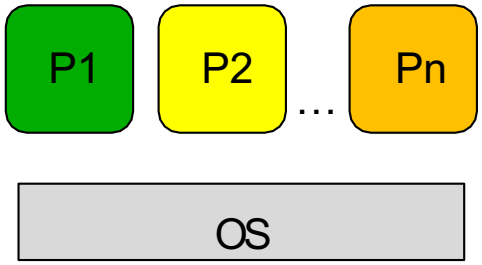
- Definition: execution environment for a program with Restricted Rights
  - Address space with one or more threads
  - Owns memory (address space)
  - Owns file descriptors, file system context, …
  - Encapsulate one or more threads sharing process resources
- Application program executes as a process
  - Complex applications can fork/exec child processes [later!]
- Why processes?
  - Protected from each other!
  - OS Protected from them
  - Processes provides memory protection
- Fundamental tradeoff between protection and efficiency
  - Communication easier within a process
  - Communication harder between processes

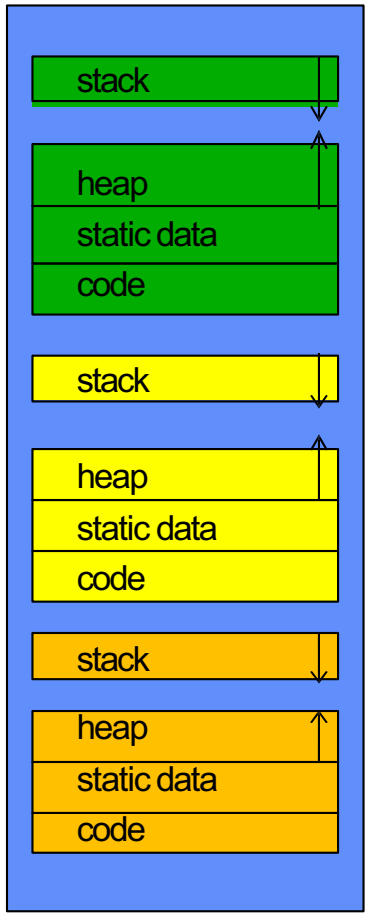
Single and multithreaded processes



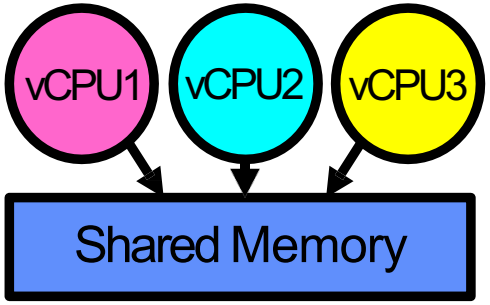
- Threads encapsulate concurrency:
  - "Active" component
- Address spaces encapsulate protection:
  - "Passive" component
  - Keeps buggy programs from crashing the system
- Why have multiple threads per address space?
  - Parallelism: take advantage of actual hardware parallelism (e.g., multicore)
  - Concurrency: ease of handling I/O and other simultaneous events

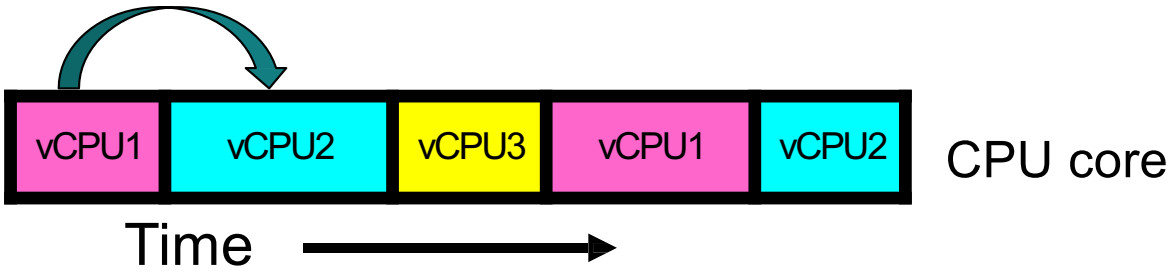
Multiprogramming – multiple processes





Illusion of multiple processors with a single core





- 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
- 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

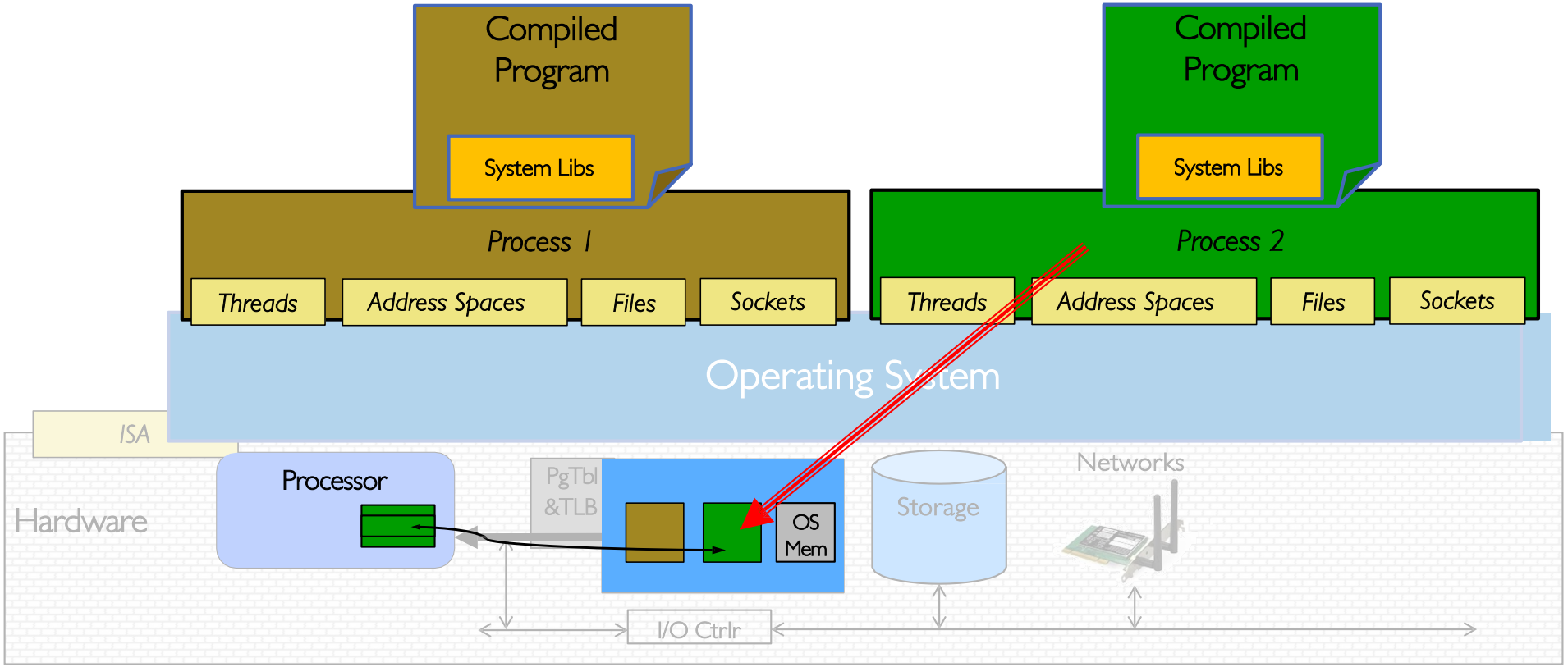
Properties of this simple multiprogramming technique

- 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)
- 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)

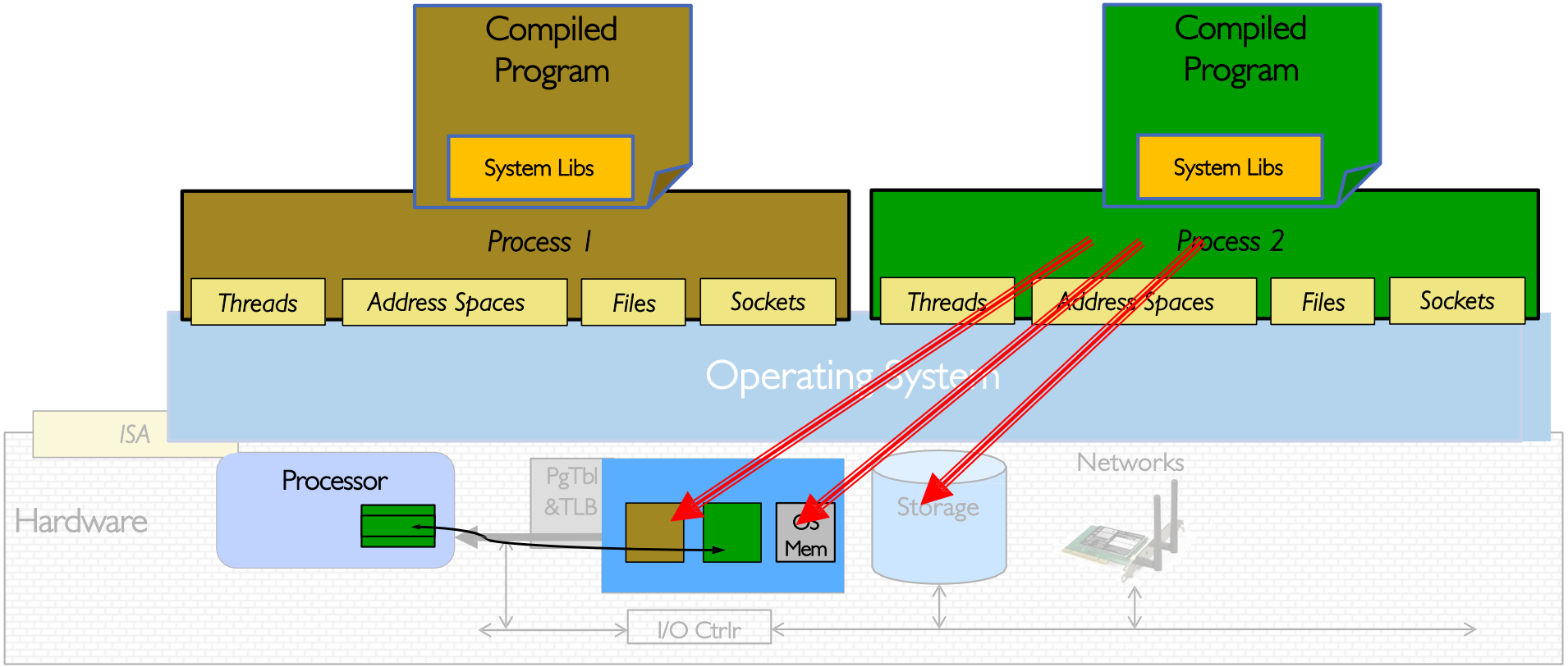
Protection and isolation

- Why do we need processes ?
  - Reliability: bugs can only overwrite memory of process they are in
  - Security and privacy: malicious or compromised process can't read or write other process' data
  - (To some degree) Fairness: enforce shares of disk, CPU

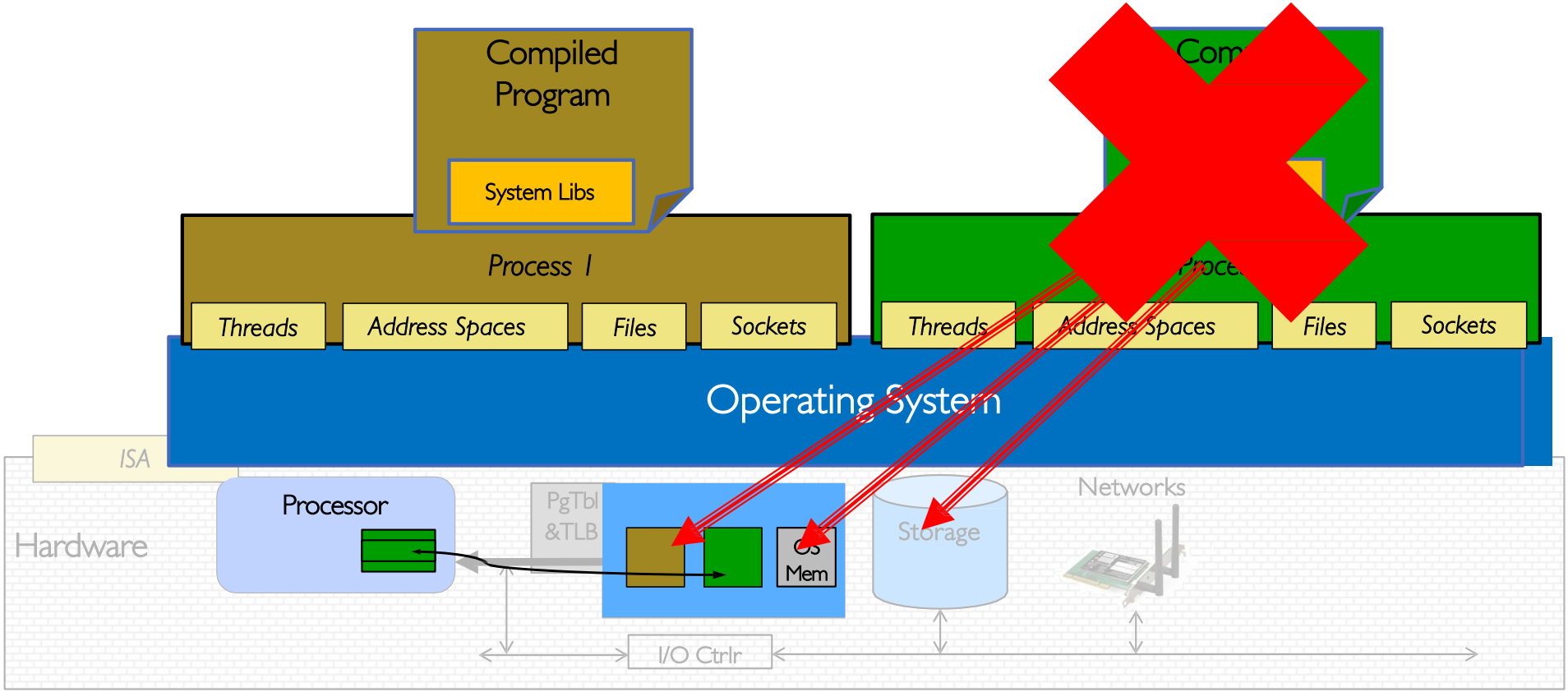
Protection


Compiled Program
System Libs
Process 1
Threads
Address Spaces
Files
Sockets
Compiled Program
System Libs
Process 2
Threads
Address Spaces
Files
Sockets
Operating System
ISA
Hardware
Processor
PgTbl &TLB
OS Mem
Storage
Networks
I/O Ctrlr

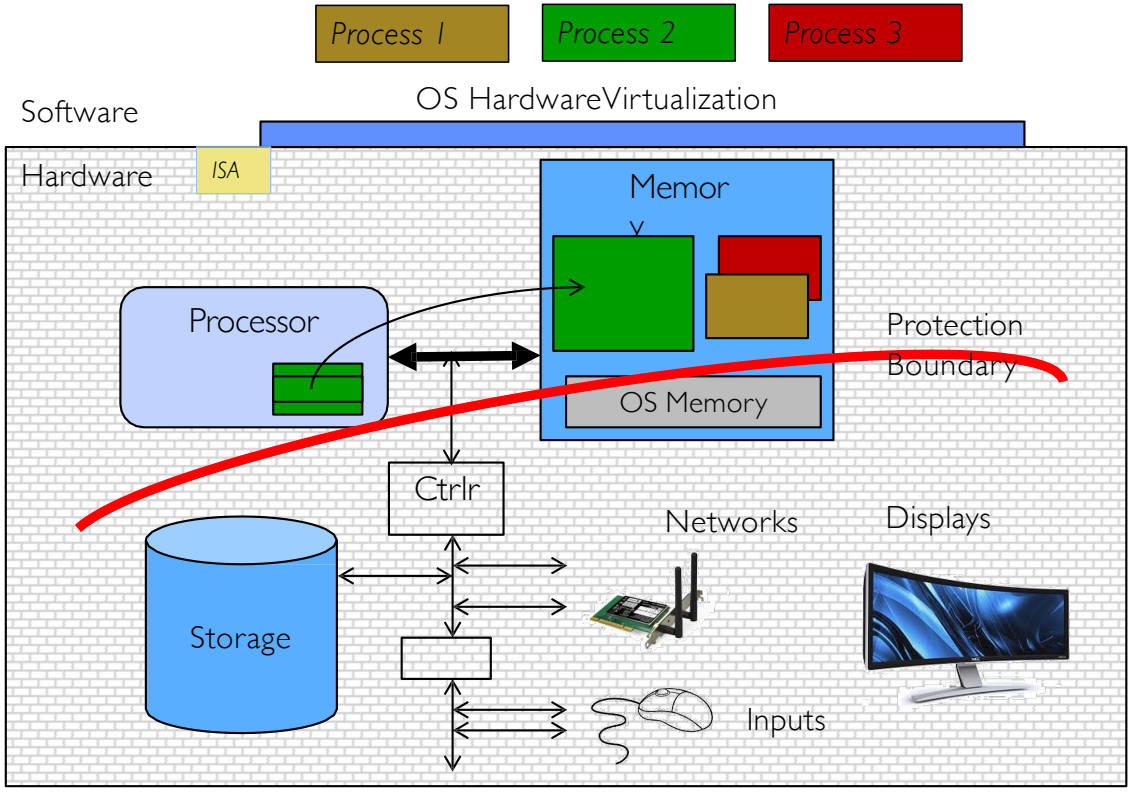
Protection


Compiled Program
System Libs
Process 1
Threads
Address Spaces
Files
Sockets
Compiled Program
System Libs
Process 2
Threads
Address Spaces
Files
Sockets
Operating System
ISA
Hardware
Processor
PgTbl &TLB
OS Mem
Storage
Networks
I/O Ctrlr

Protection


Compiled Program
System Libs
Process 1
Threads
Address Spaces
Files
Sockets
Threads
Address Spaces
Files
Sockets
Operating System
ISA
Hardware
Processor
PgTbl &TLB
OS Mem
Storage
Networks
I/O Ctrlr

Protection



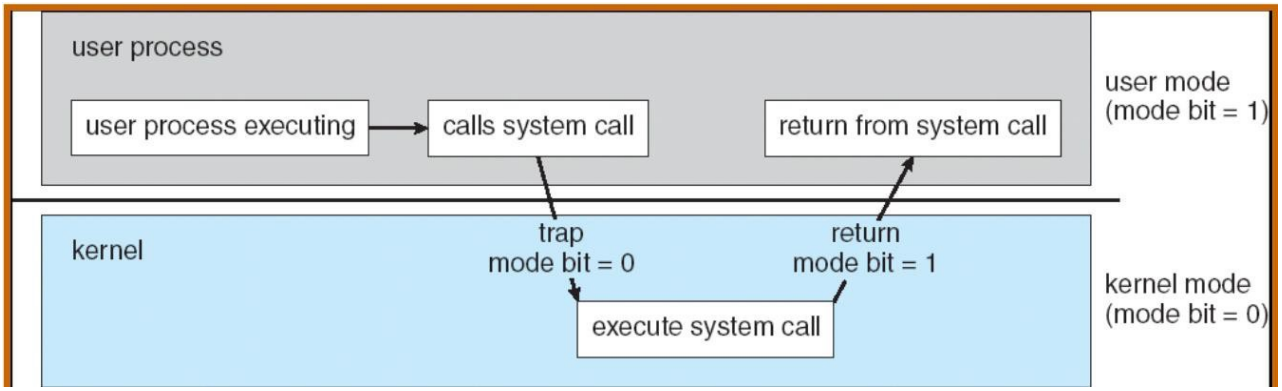
- OS isolates processes from each other
- OS isolates itself from other processes
- …even though they are actually running on the same hardware!

Protection and isolation

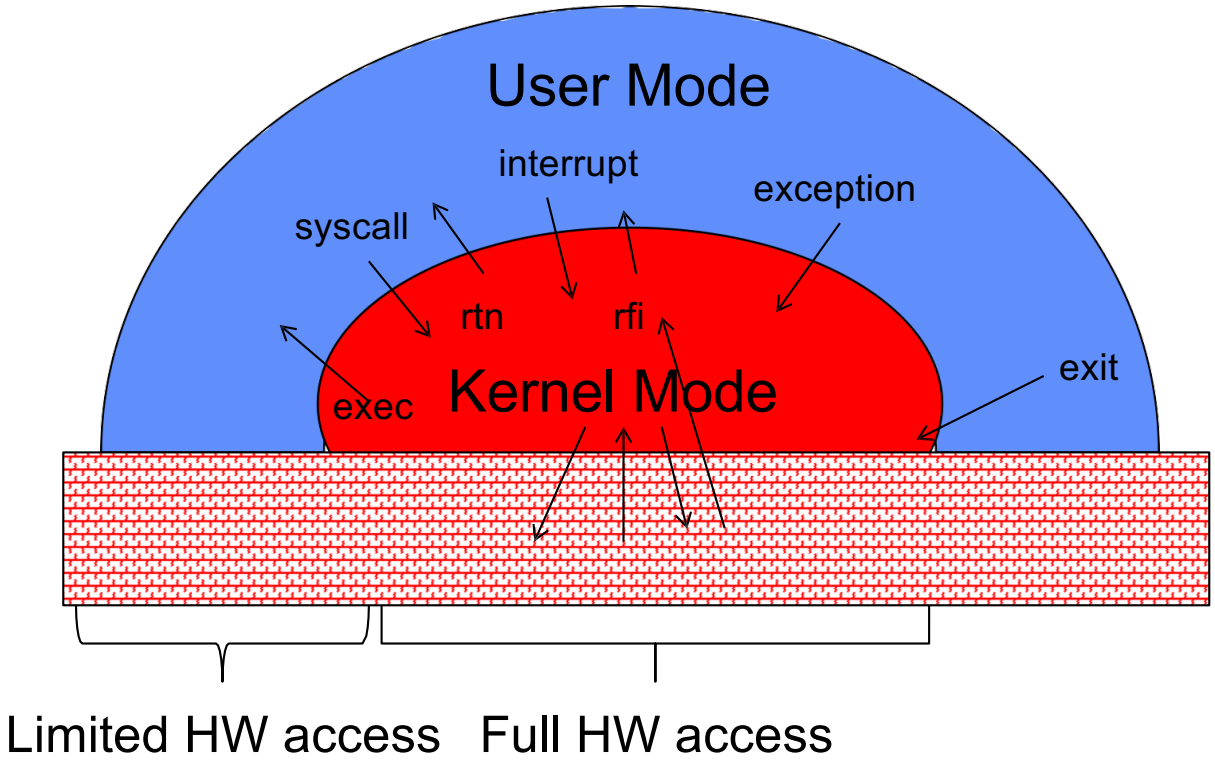
- OS must protect user programs from one another
- Operating System must protect itself from user programs
  - Reliability: compromising the operating system generally causes it to crash
  - Security: limit the scope of what processes can do
  - Privacy: limit each process to the data it is permitted to access
  - Fairness: each should be limited to its appropriate share of system resources (CPU time, memory, I/O, etc)
- 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, I/O instructions, special registers
  - syscall processing, subsystem implementation
    - (e.g., file access rights, etc.)

Fourth OS concept: dual mode operation

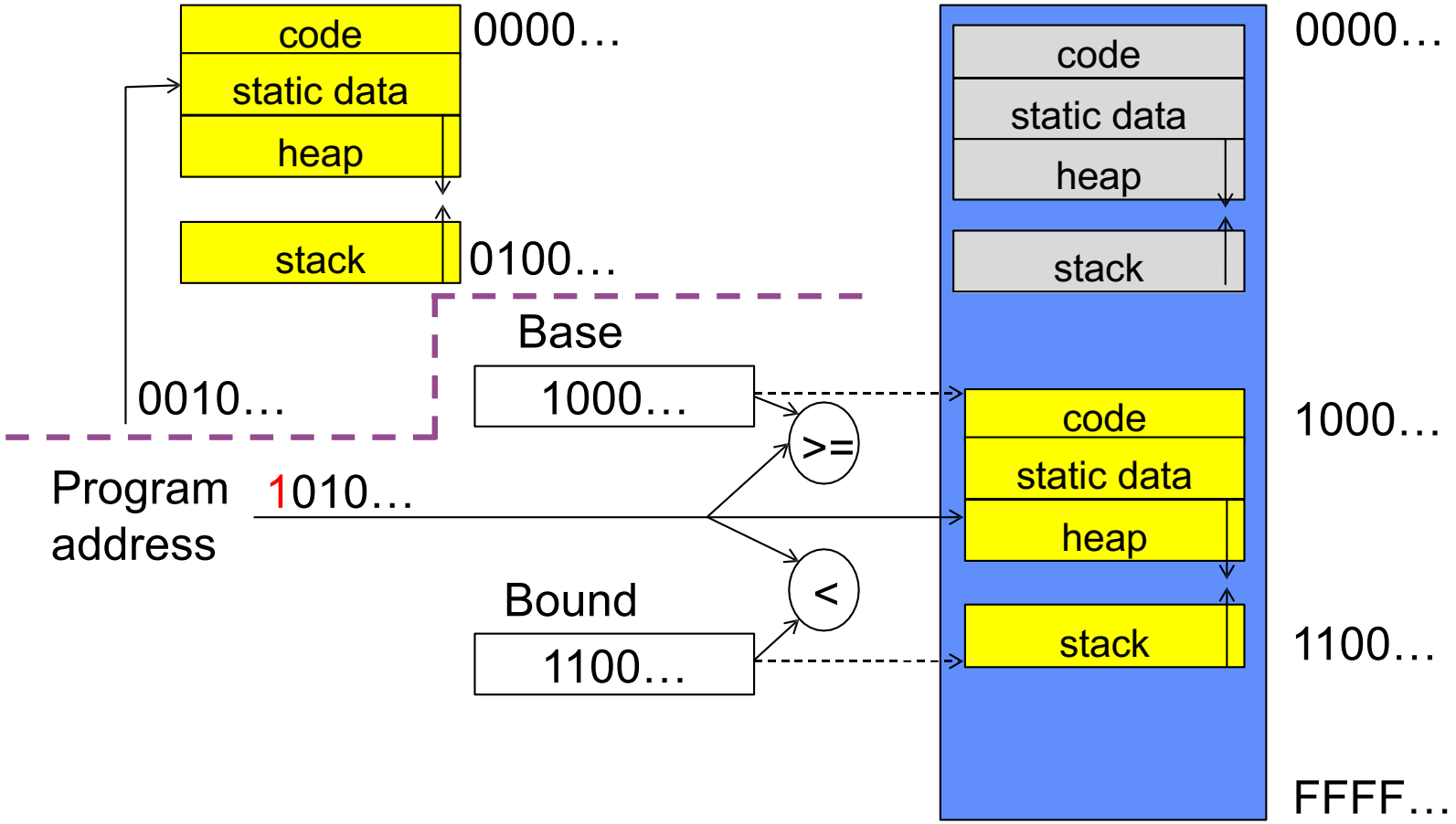
- Hardware provides at least two modes (at least 1 mode bit):
  1. Kernel Mode (or “supervisor” mode)
  2. User Mode
- Certain operations are prohibited when running in user mode
  - Changing the page table pointer, disabling interrupts, interacting directly w/ hardware, writing to kernel memory
- Carefully controlled transitions between user mode and kernel mode
  - System calls, interrupts, exceptions



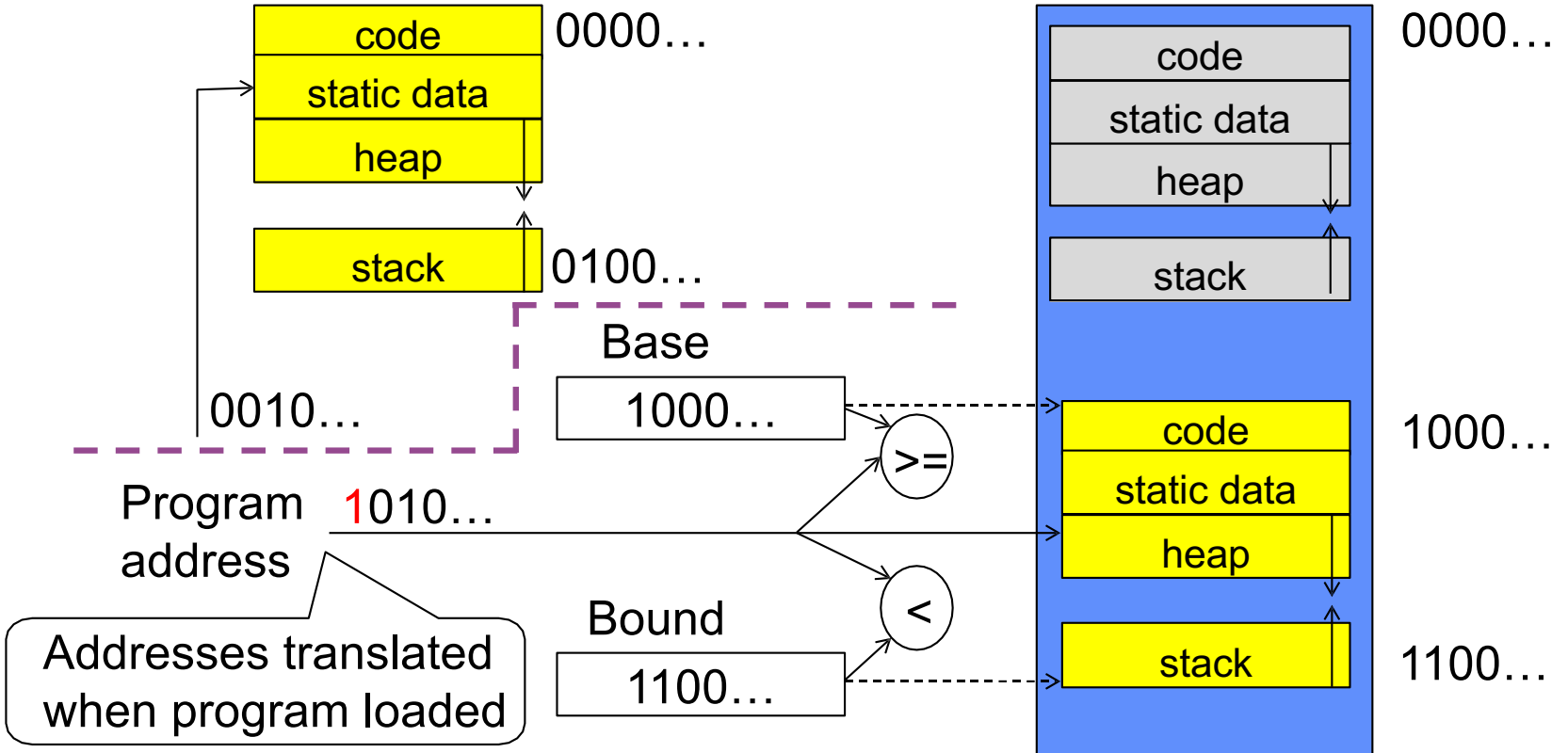
User/Kernel (privileged) mode



Simple protection: base and bound (B&B)



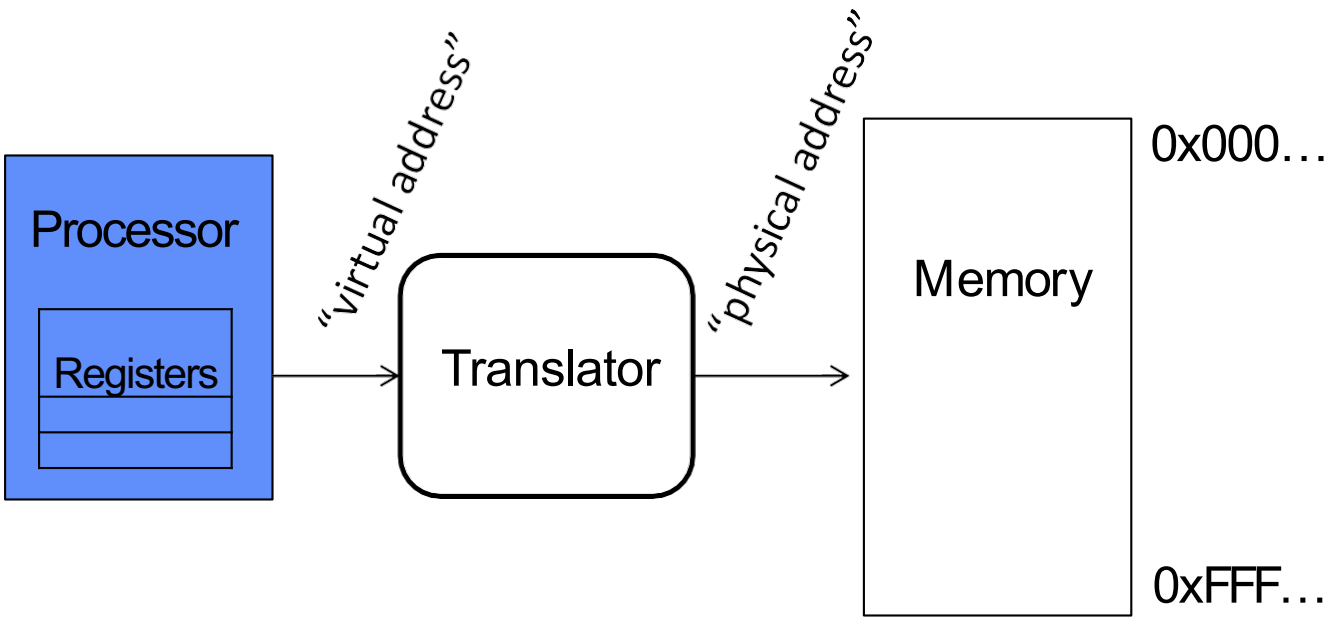
Simple protection: base and bound (B&B)



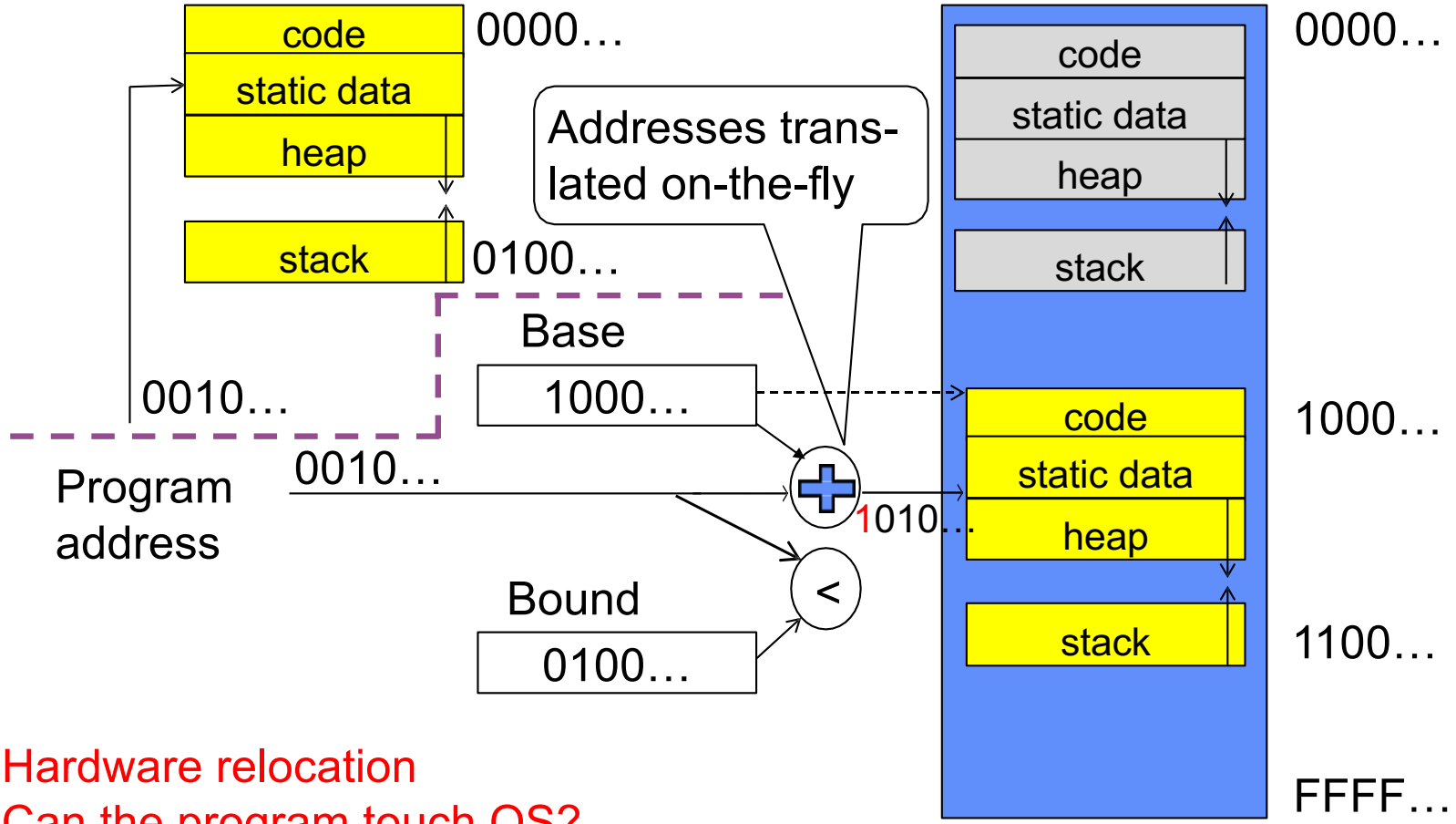
- Still protects OS and isolates program
- Requires relocating loader
- No addition on address path

Another idea: address space translation

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

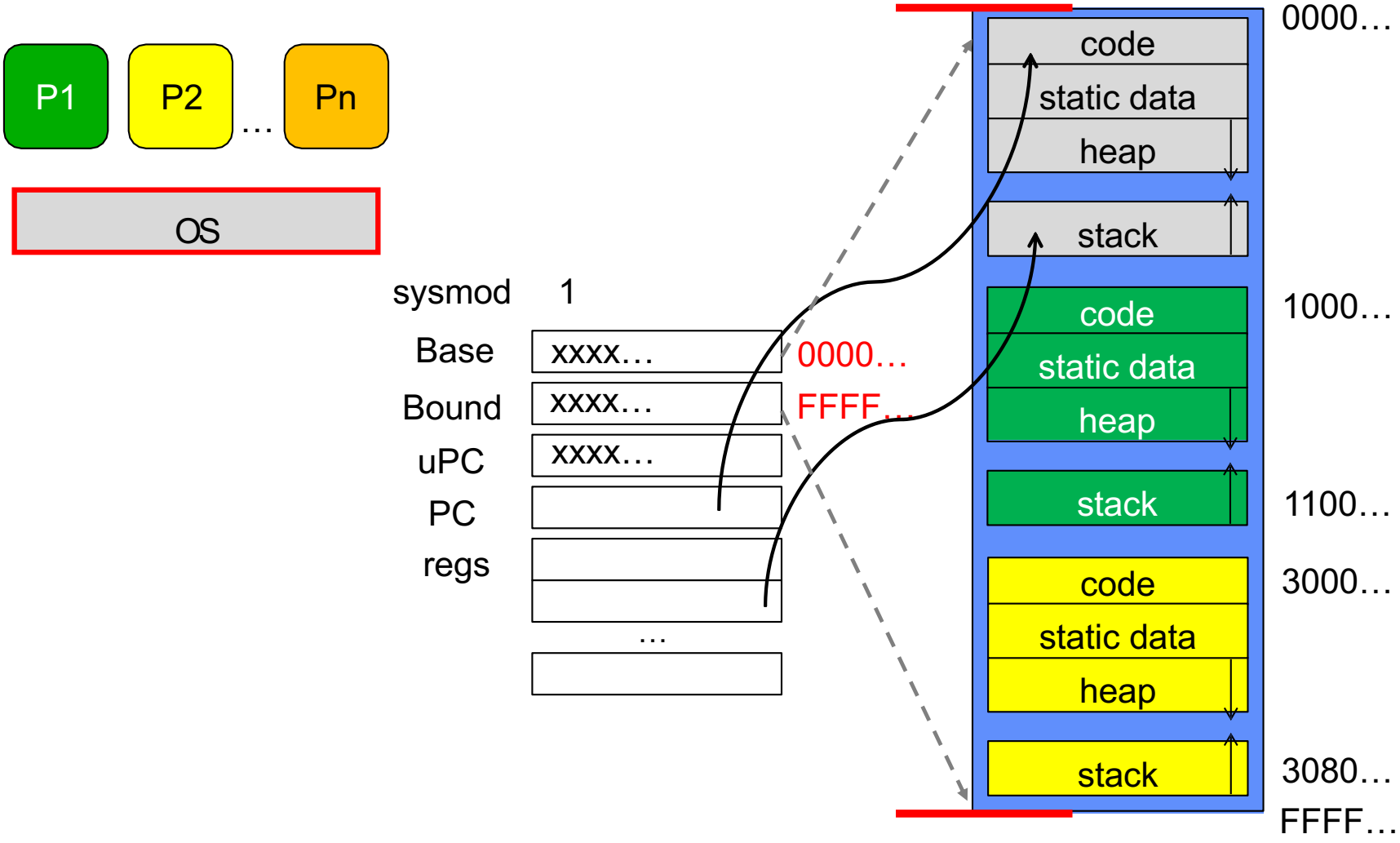


Simple protection: base and bound (B&B)

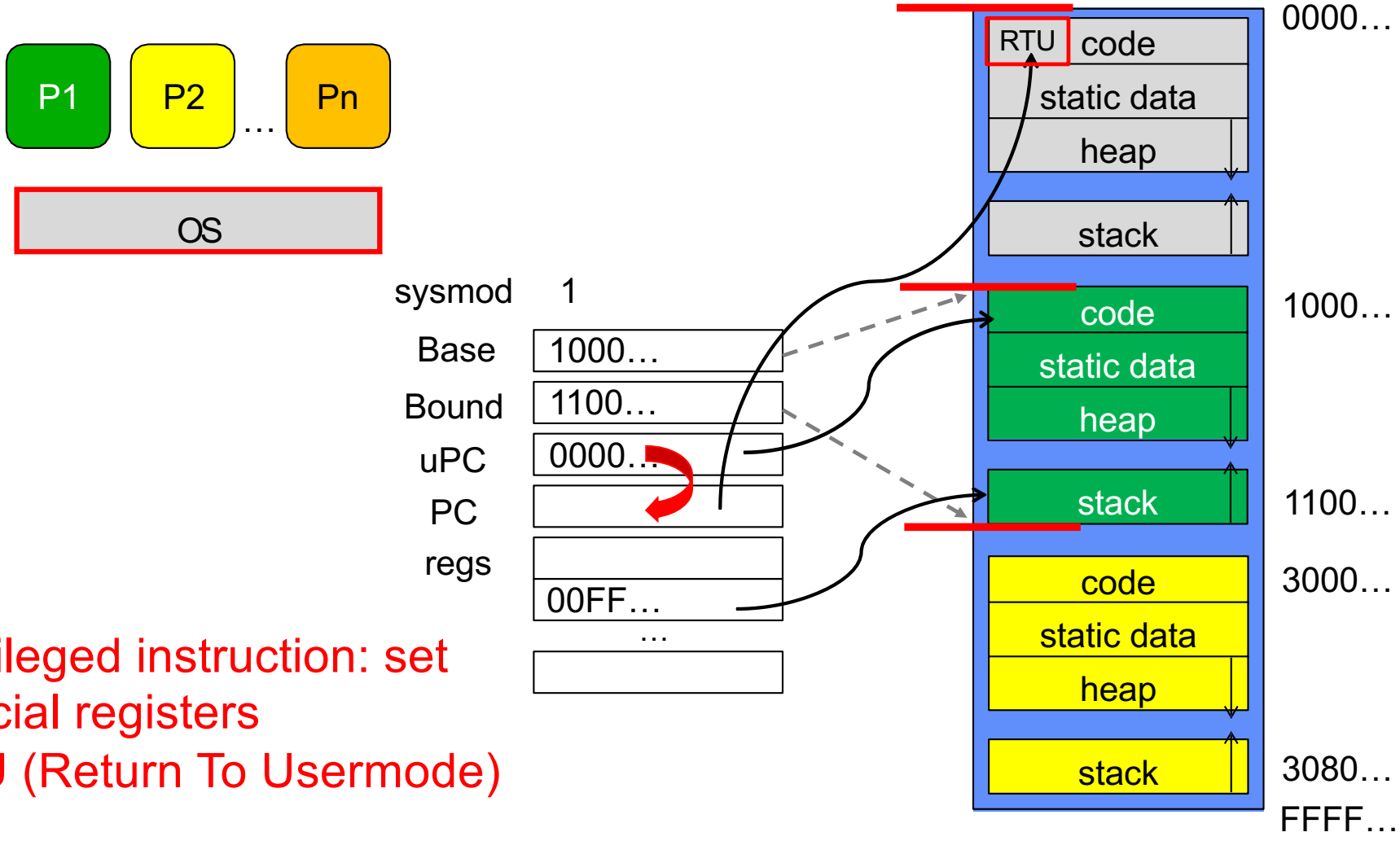


- Hardware relocation
- Can the program touch OS?
- Can it touch other programs?

Trying it together: simple B&B: OS load process

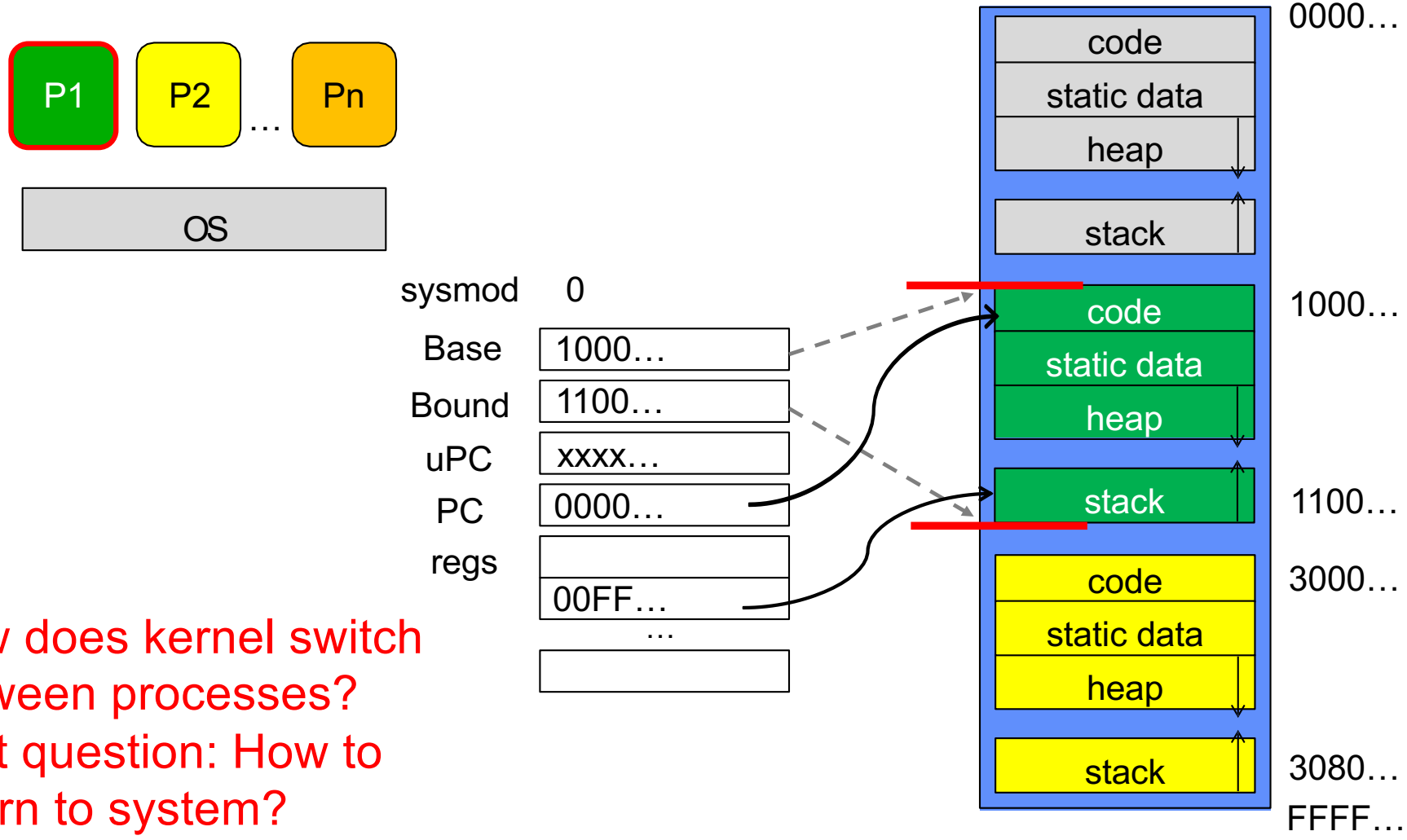


Simple B&B: OS gets ready to execute process



- Privileged instruction: set special registers
- RTU (Return To Usermode)

Simple B&B: user code running



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

3 types of user to 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
  - 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, …

3 types of user to kernel mode transfer

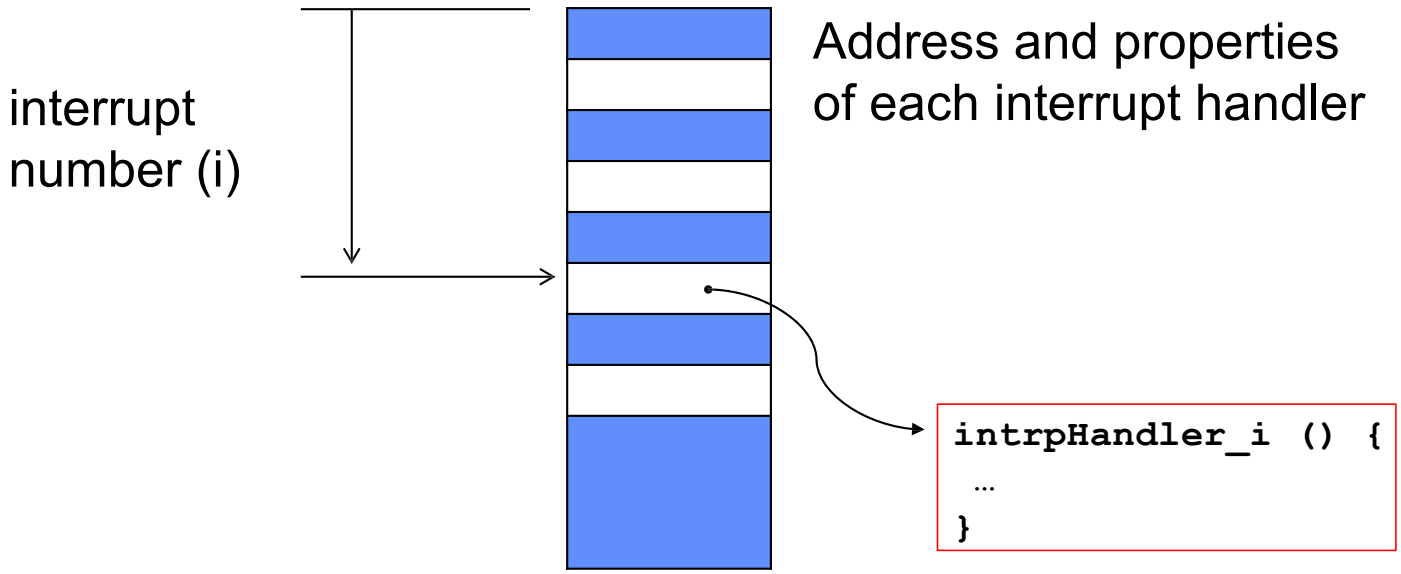
- Syscall
- Interrupt
- Trap or Exception

- All 3 are an UNPROGRAMMED* CONTROL TRANSFER
  - Where does it go?

*Switching to OS without a direct instruction from the running process

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

Interrupt vector



Conclusion: four fundamental OS concepts

- **Process: an instance of a running program**
  - Address Space + one or more Threads
- **Thread: execution context**
  - Fully describes program state
  - Program Counter, Registers, Stack, Execution Flags,
- **Address space (with or w/o translation)**
  - Set of memory addresses accessible to program (for read or write)
  - May be distinct from memory space of the physical machine (in which case programs operate in a virtual address space)
- **Dual mode operation / protection**
  - Only the “system” has the ability to access certain resources

Thanks