Introduction to the Process

David E. Culler

CS162 – Operating Systems and Systems Programming

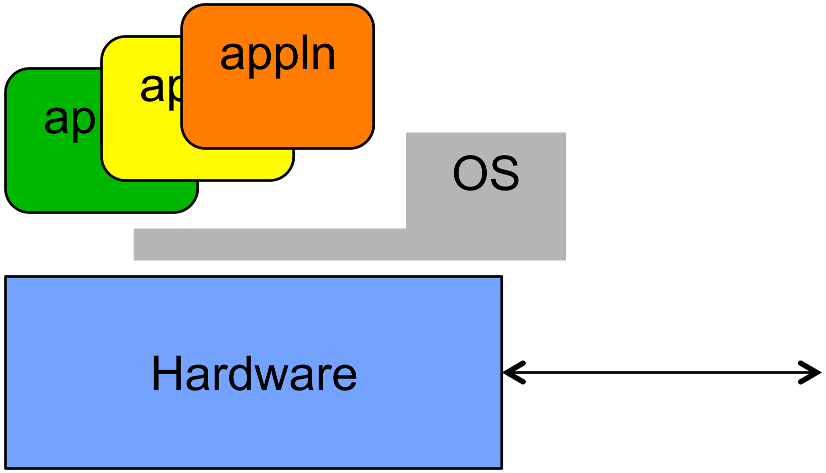
Lecture 2

Sept 3, 2014

Reading: A&D CH2.1-7
HW: 0 out, due 9/8

Recall: What is an operating system?

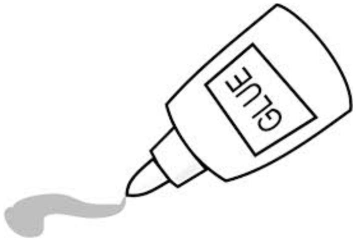
- Special layer of software that provides application software access to hardware resources
  - Convenient abstraction of complex hardware devices
  - Protected access to shared resources
  - Security and authentication
  - Communication amongst logical entities



What is an Operating System?

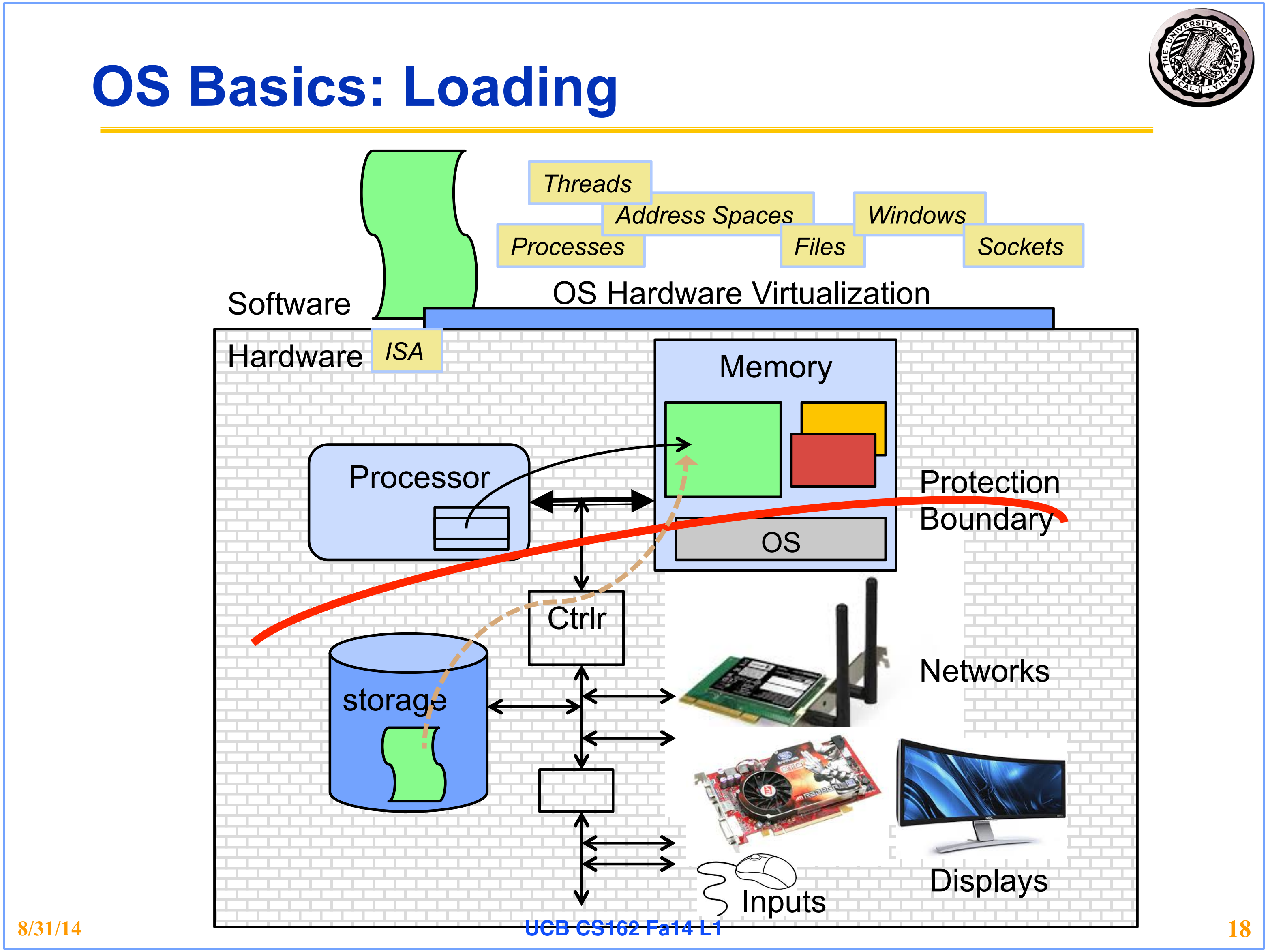
- **Referee**
  - Manage sharing of resources, Protection, Isolation
    - » Resource allocation, isolation, communication

- **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



OS Basics: Loading

Threads

Address Spaces

Windows

Processes

Files

Sockets

Software

OS Hardware Virtualization

Hardware

ISA

Memory

OS

Processor

Protection Boundary

Ctrlr

storage

Networks

Displays

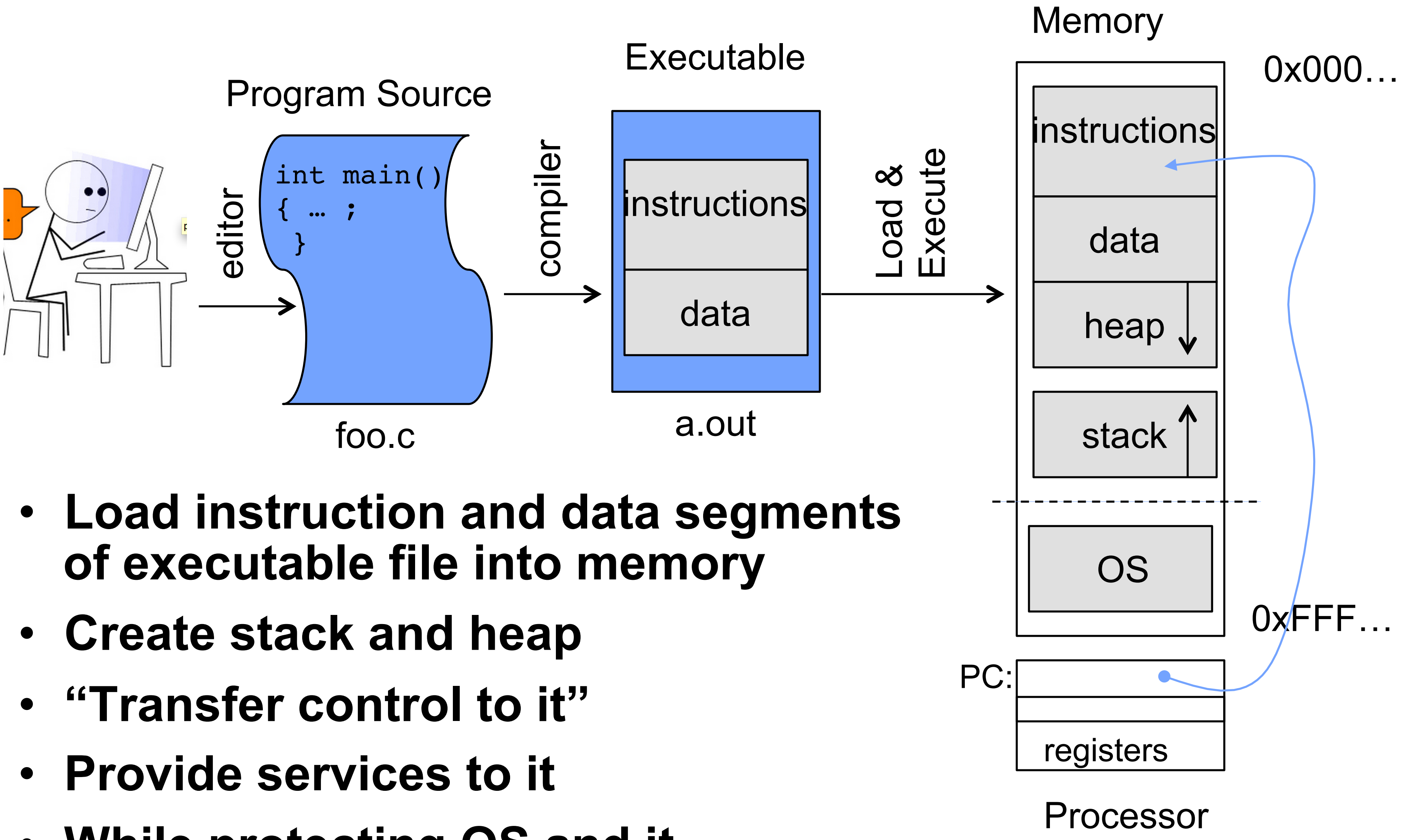
Inputs

8/31/14 UCB CS162 Fa14 L1 18

Today: four fundamental OS concepts

- Privileged/User Mode
  - The hardware can operate in two modes, with only the “system” mode having the ability to access certain resources.
- Address Space
  - 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*
- Protection
  - 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



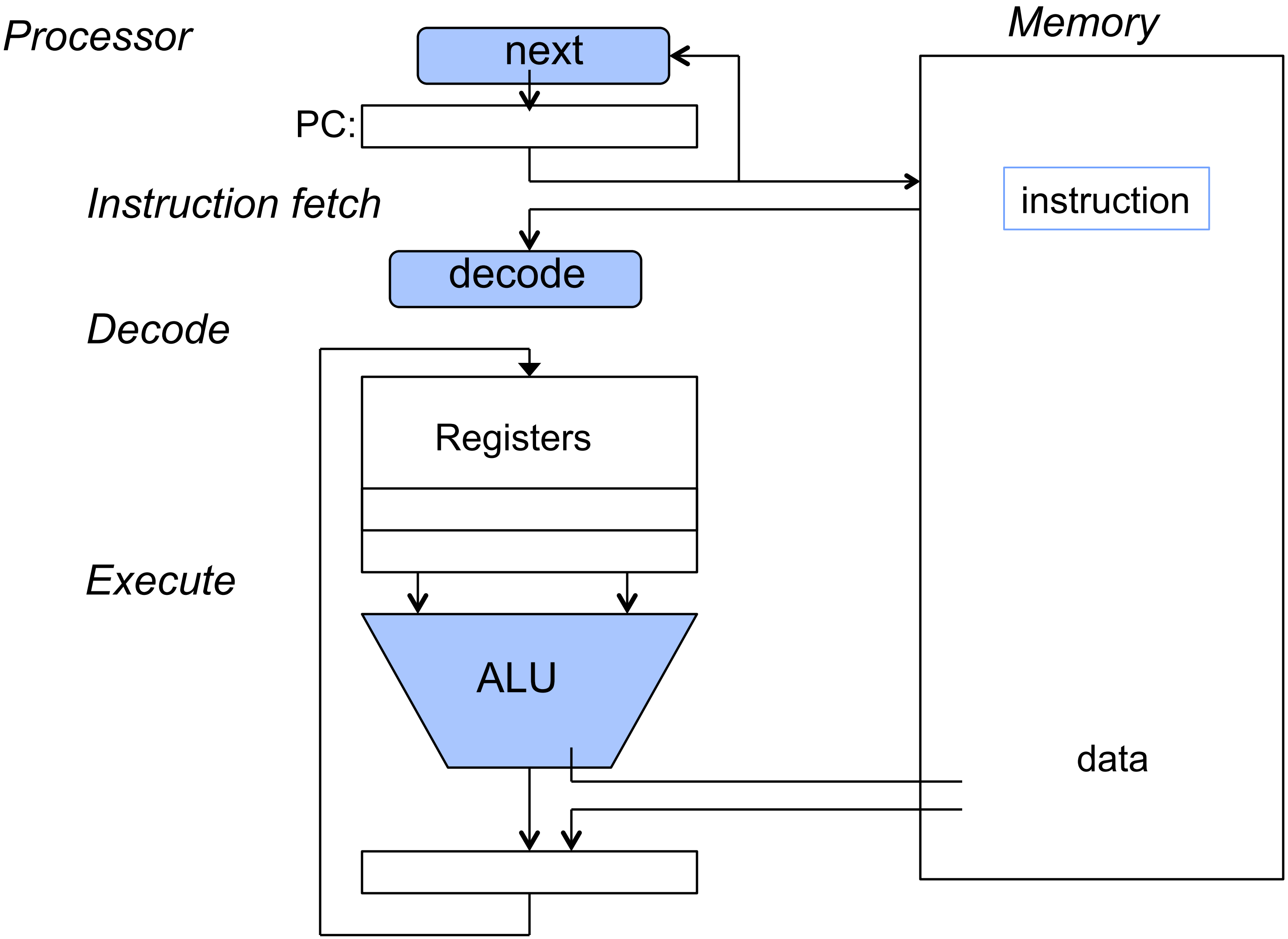
- **Load instruction and data segments of executable file into memory**
- **Create stack and heap**
- **“Transfer control to it”**
- **Provide services to it**
- **While protecting OS and it**

Why no stack or heap in executable?

- What about data segment?

Today we need one key 61B concept

The instruction cycle



Key OS Concept: the Process

Process **= Execution of a Program with Restricted Rights**

“User Programs”

“Applications”

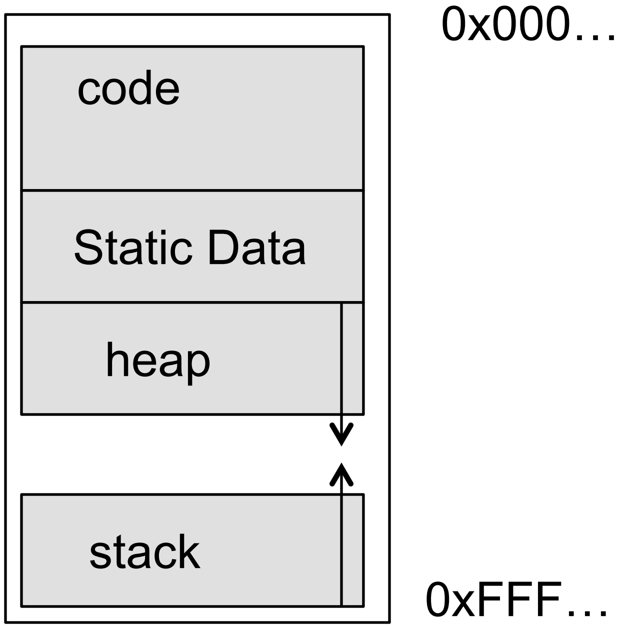
Operating System Boundary

Process **= Address Space with one or more threads of control**

“Kernel”

“Operating System”

Address Space

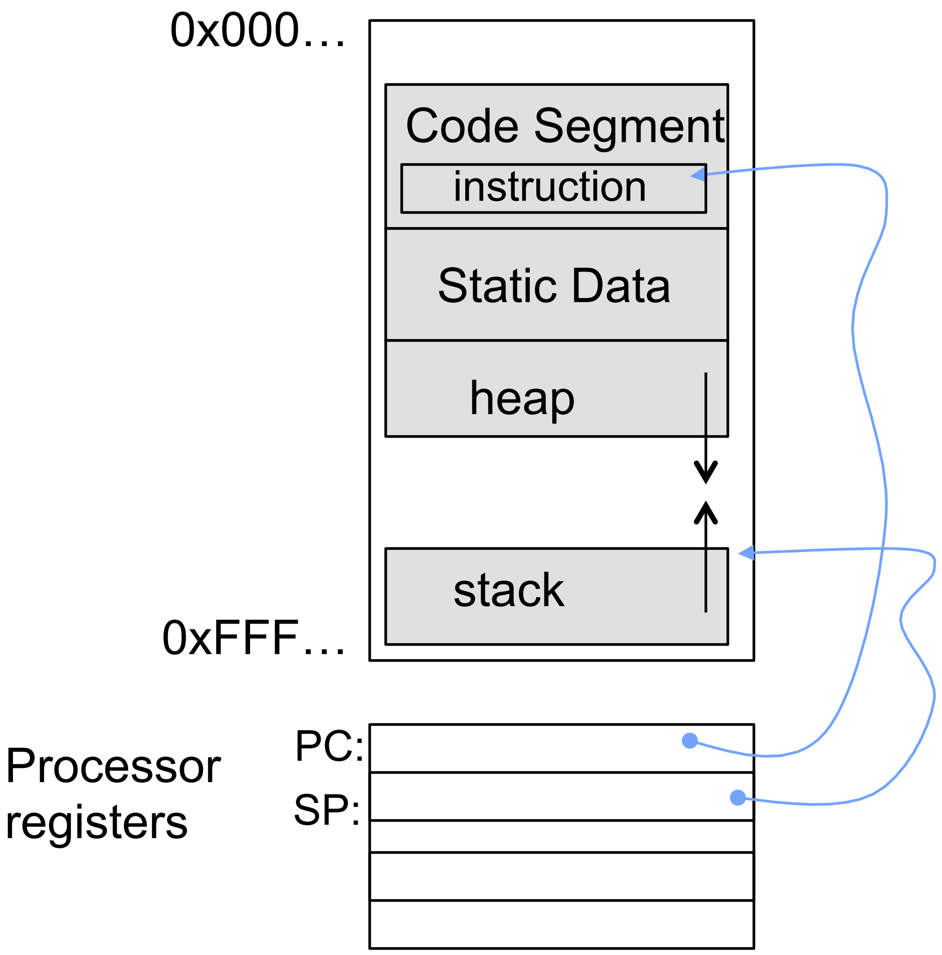


- **What's in the code segment? Data?**
- **What's in the stack segment?**
  - **How is it allocated? How big is it?**
- **What's in the heap segment?**
  - **How is it allocated? How big?**

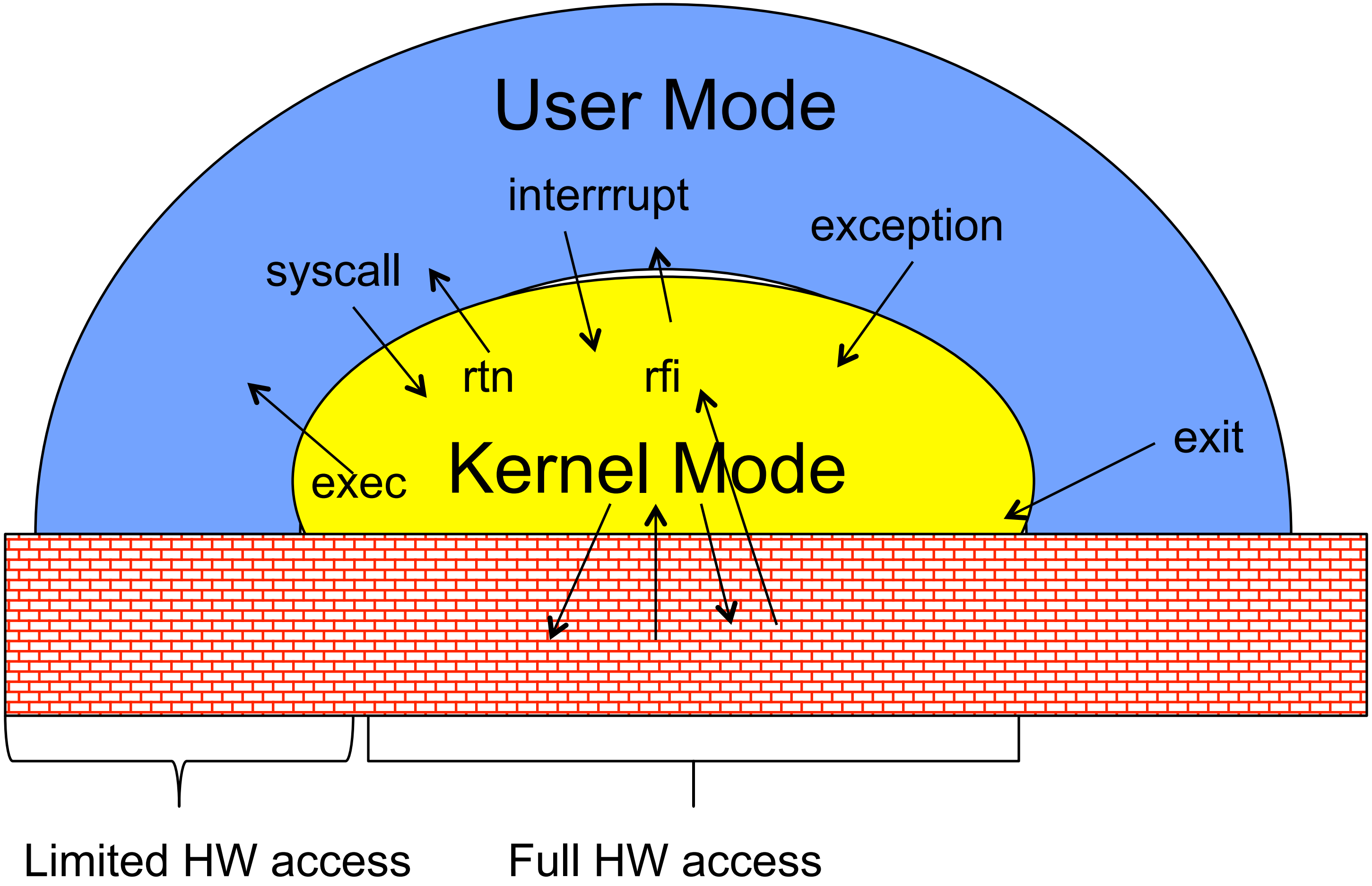
Thread of Control

- 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.
- 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"

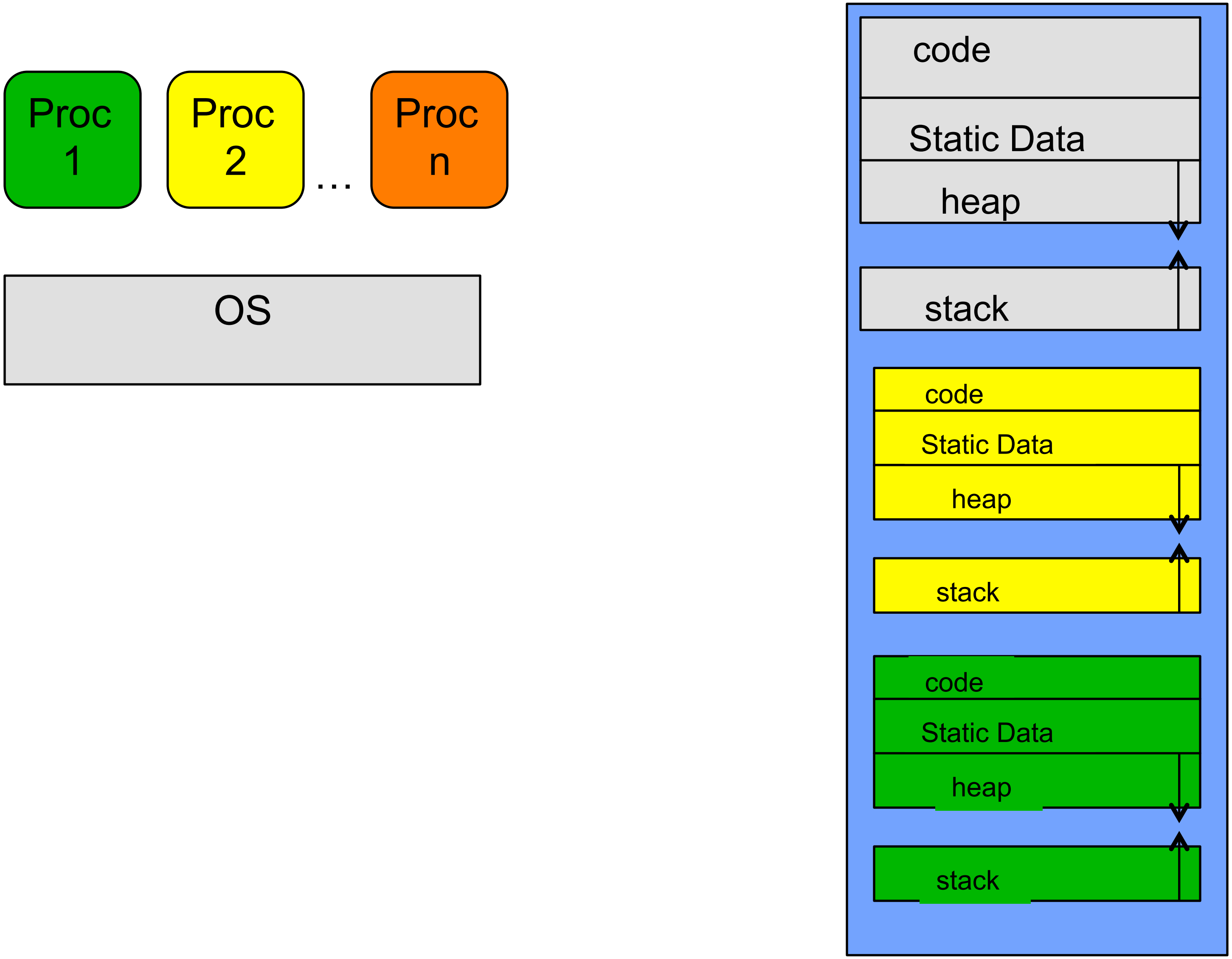
In a Picture


0x000…
Code Segment
instruction
Static Data
heap
stack
0xFFF…
Processor registers
PC:
SP:

User/Kernal(Priviledged) Mode



Multiprogramming - Multiple Processes

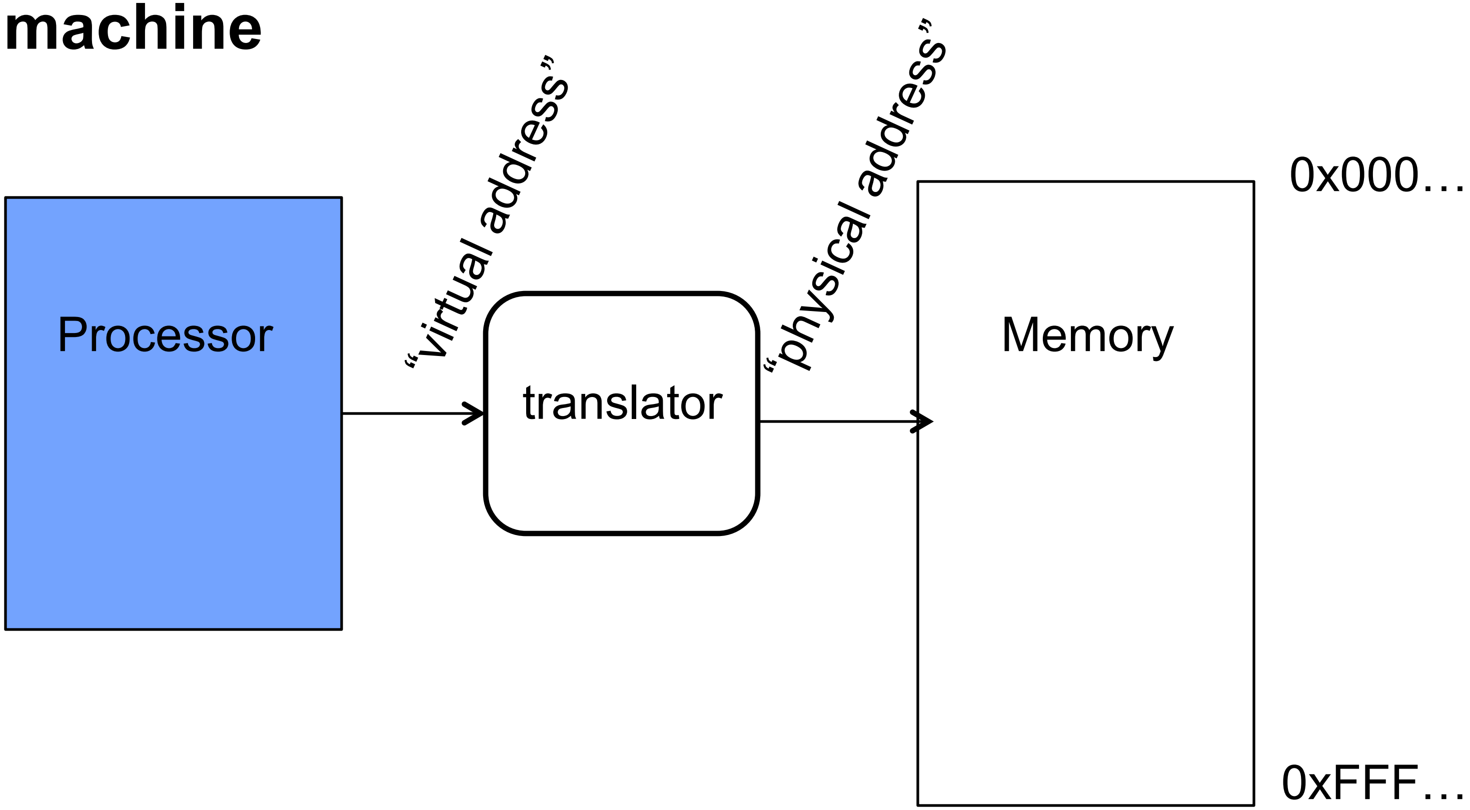


Dual Mode Operation

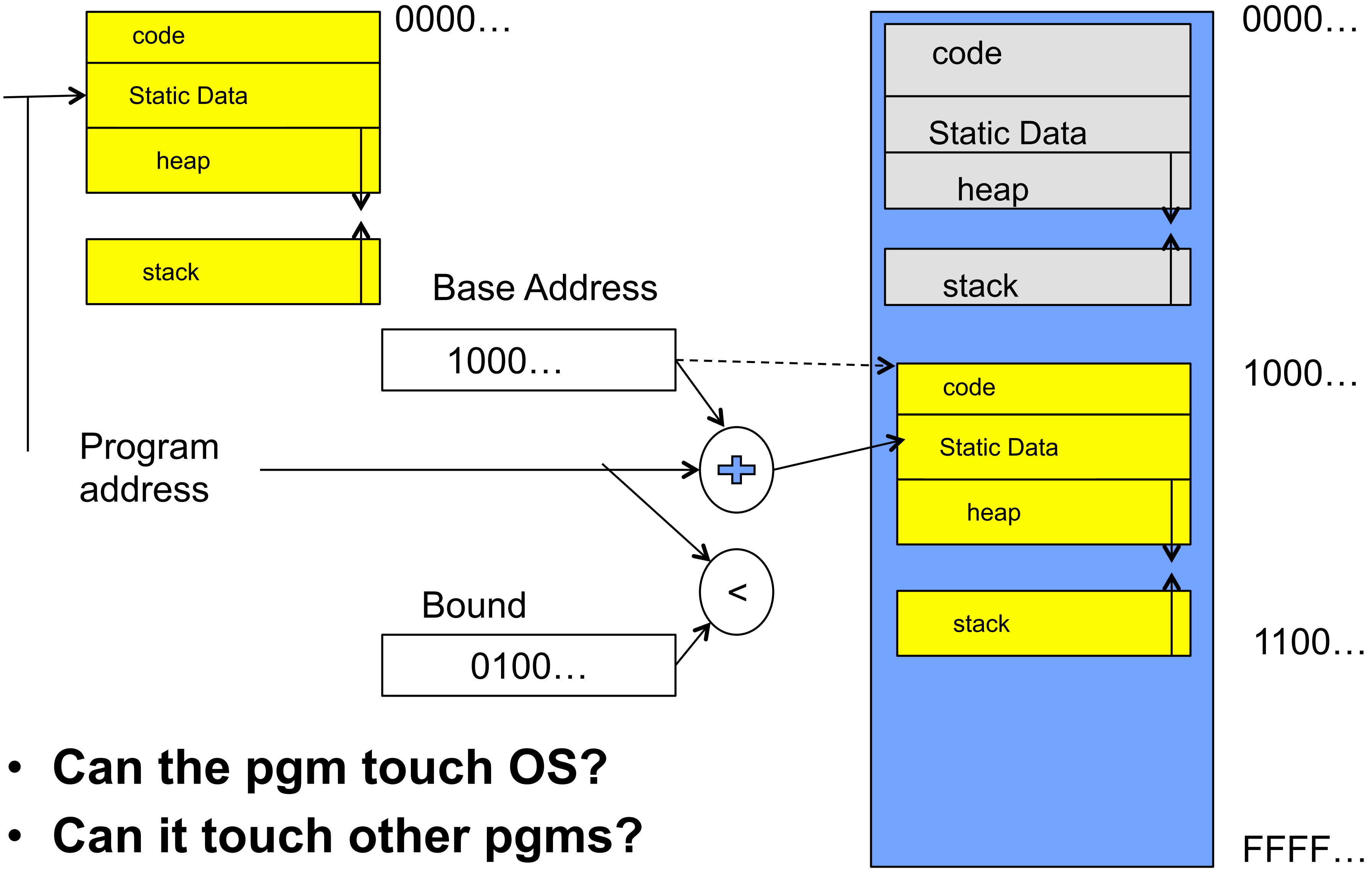
- What is needed in the hardware to support “dual mode” operation?
- a bit of state (user/system mode bit)
- Certain operations / actions only permitted in system/kernel mode
  - In user mode they fail or trap
- User->Kernel transition *sets* system mode AND saves the user PC
  - Operating system code carefully puts aside user state then performs the necessary operations
- Kernel->User transition clears system mode AND restores appropriate user PC
  - return-from-interrupt

Key OS Concept: Address Space

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

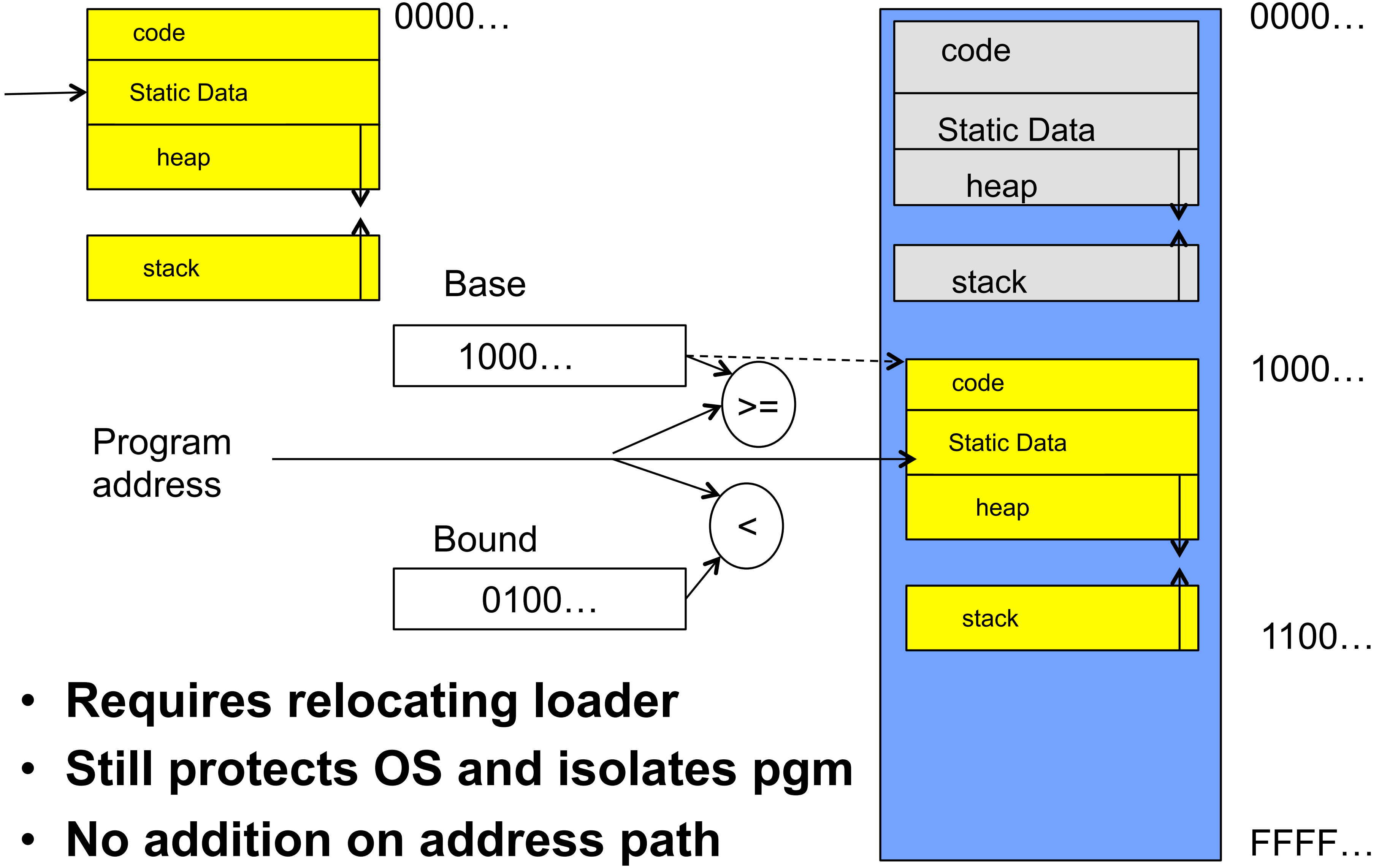


A simple address translation: B&B



- **Can the pgm touch OS?**
- **Can it touch other pgms?**

A different base and bound

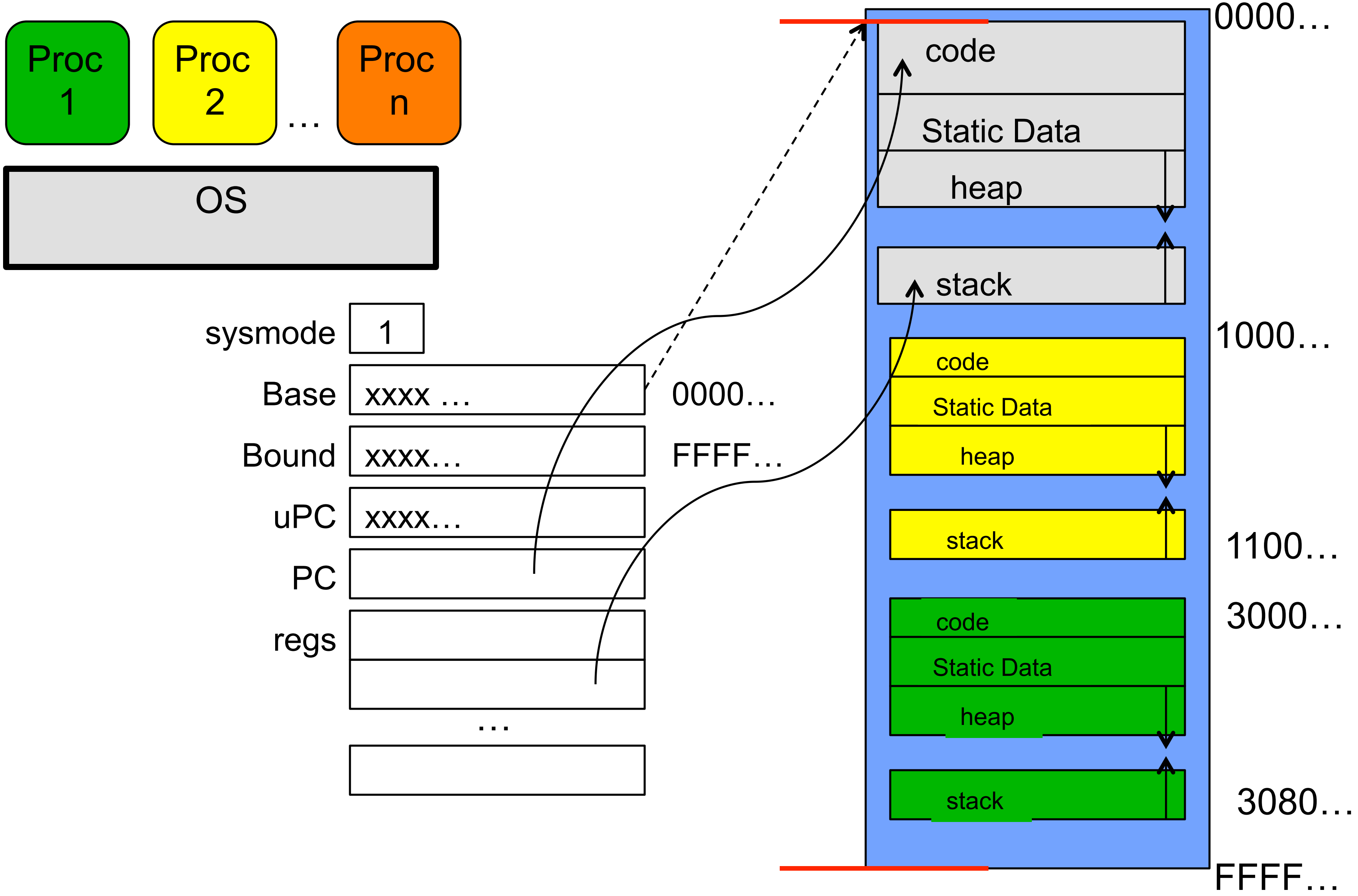


- **Requires relocating loader**
- **Still protects OS and isolates pgm**
- **No addition on address path**

Key OS Concept: 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
- 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 in
- Additional Mechanisms:
  - Privileged instructions, in/out instructions, special registers
  - syscall processing, subsystem implementation
    - » (e.g., file access right)

Simple B&B: OS loads process



Proc 1

Proc 2

…

Proc n

OS

sysmode 1

Base xxxx … 0000…

Bound xxxx… FFFF…

uPC xxxx…

PC

regs

…

0000…

code

Static Data

heap

stack

1000…

code

Static Data

heap

stack

1100…

3000…

code

Static Data

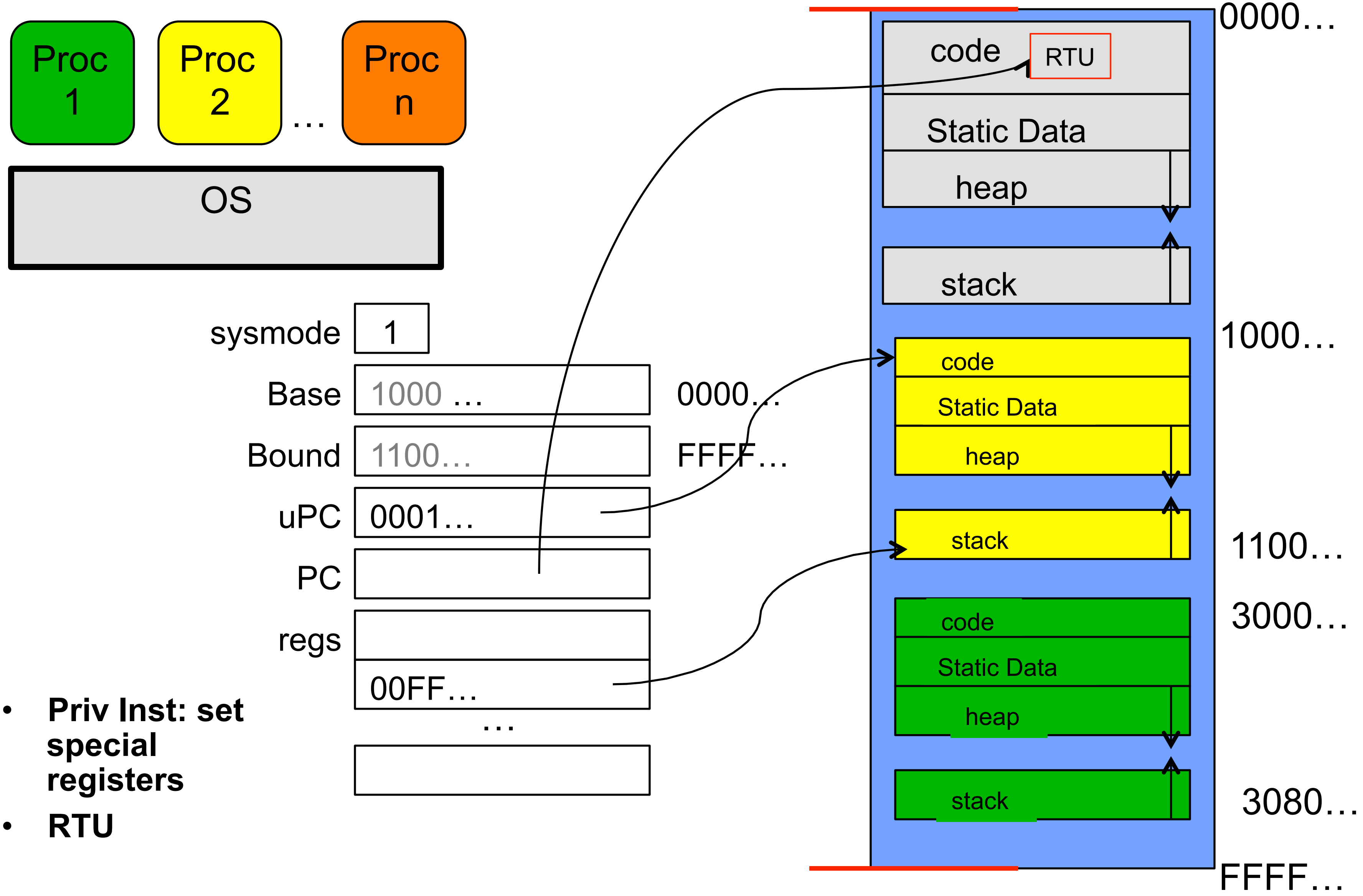
heap

stack

3080…

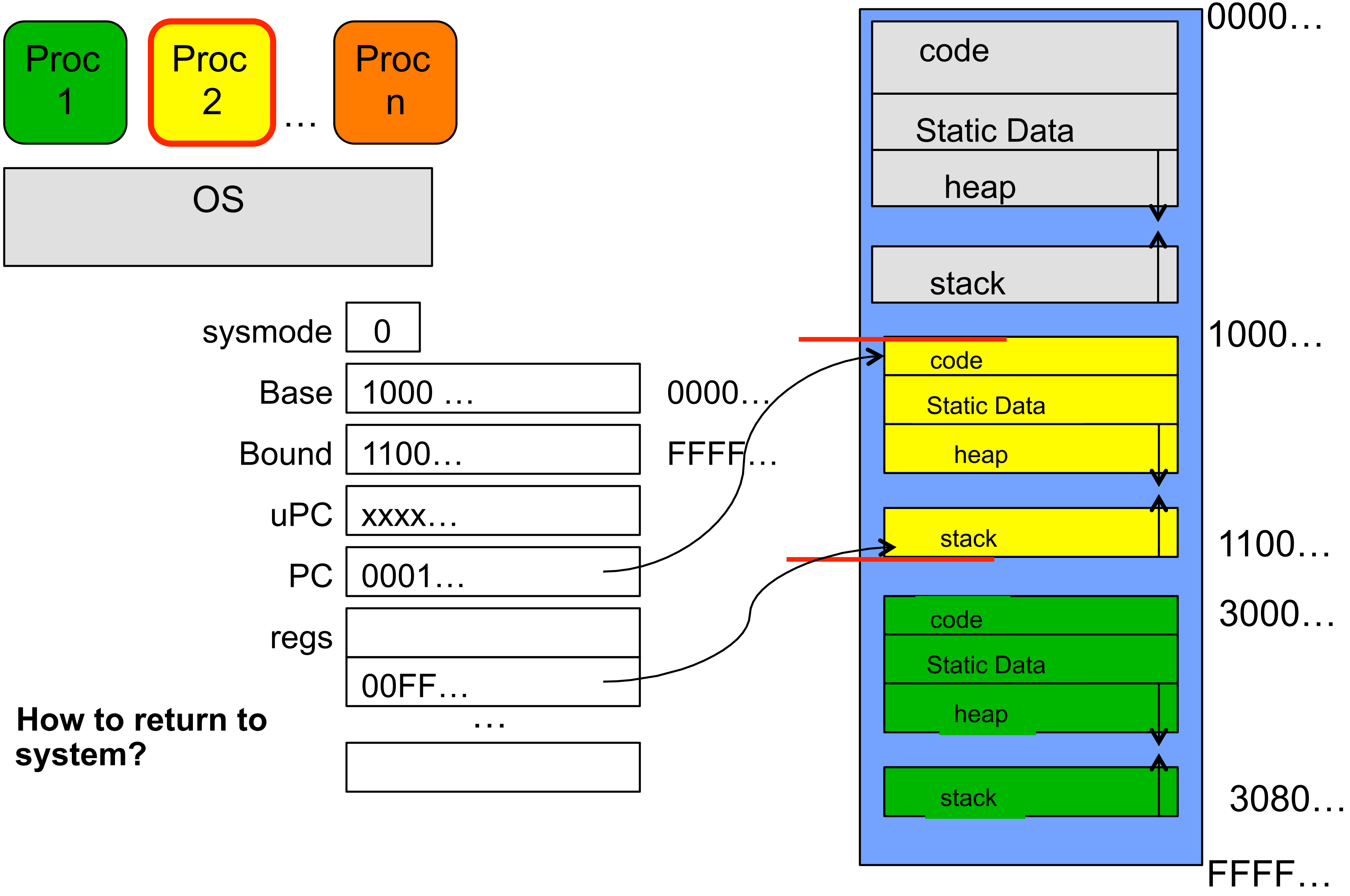
FFFF…

Simple B&B: OS gets ready to switch



- **Priv Inst: set special registers**
- **RTU**

Simple B&B: “Return” to User



- **How to return to system?**

Logistics Break

Recall: Getting started

- Start homework 0 immediately
  - Gets cs162-xx@cory.eecs.berkeley.edu (and other inst m/c)
  - Github account
  - Registration survey
  - Vagrant virtualbox – VM environment for the course
    - » Consistent, managed environment on your machine
  - icluster24.eecs.berkeley.edu is same
  - Get familiar with all the cs162 tools
  - Submit to autograder via git
- Go to section
- Waitlist ???
  - Pull hw0, can get inst acct
  - Will process at least weekly (thru early drop deadline)
  - Only registered students will form project groups
  - If cs162 is not for you now, please allow others to take it

Personal Integrity

- UCB Academic Honor Code: "As a member of the UC Berkeley community, I act with honesty, integrity, and respect for others."

http://asuc.org/honorcode/resources/HC%20Guide%20for%20Syllabi.pdf

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

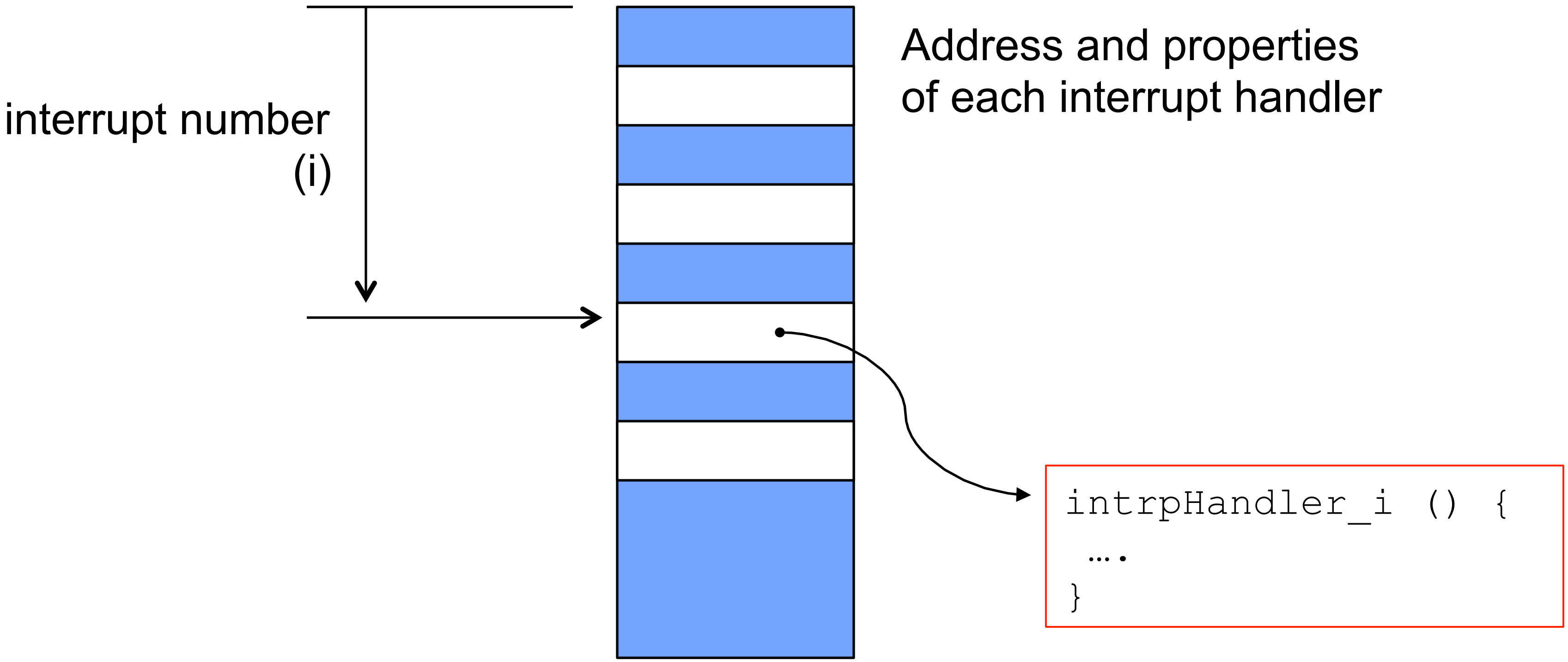
We compare all project submissions against prior year submissions and online solutions and will take actions (described on the course overview page) against offenders

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

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

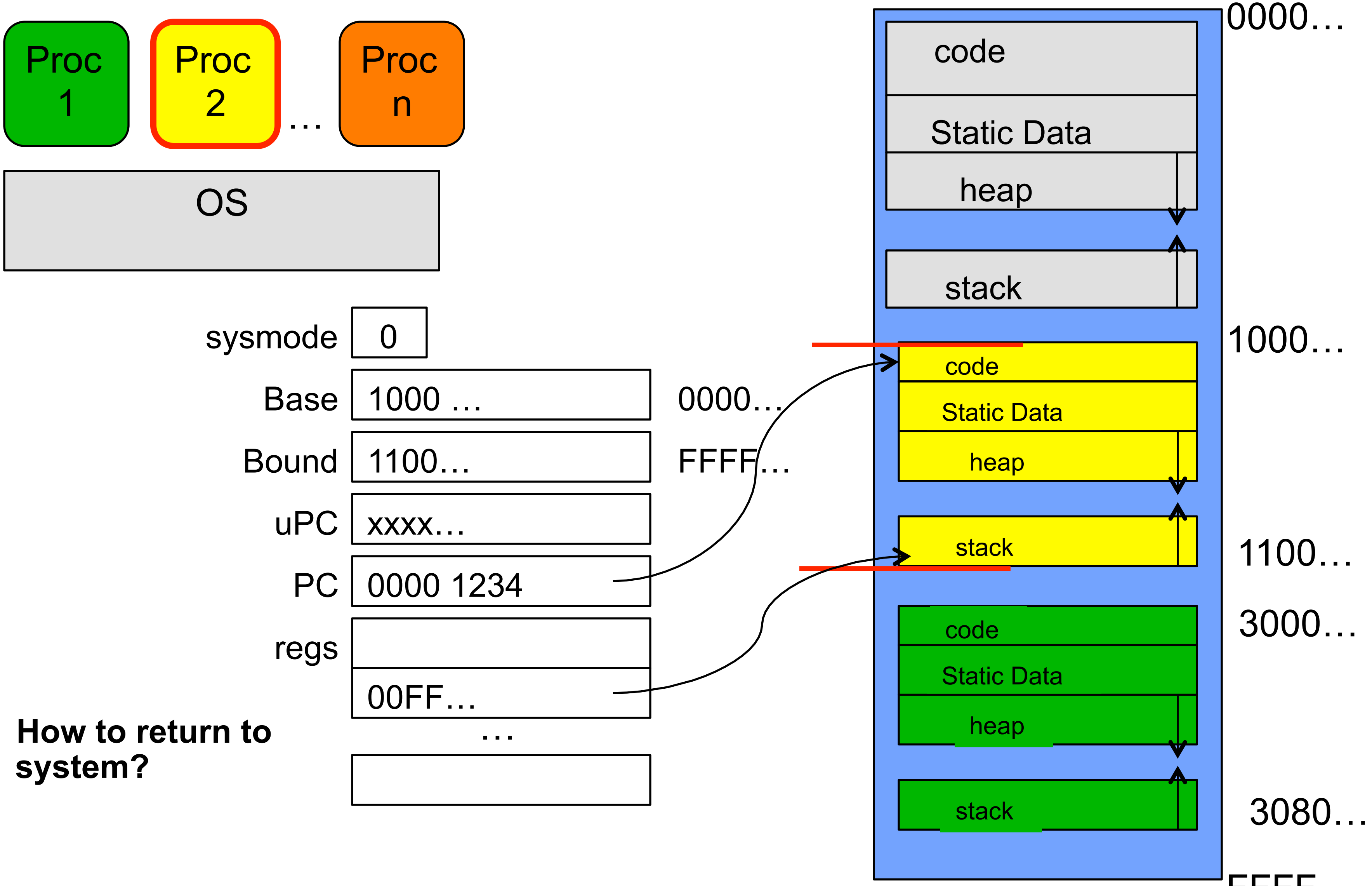
Interrupt Vector



```
intrpHandler_i () {
 ….
}
```

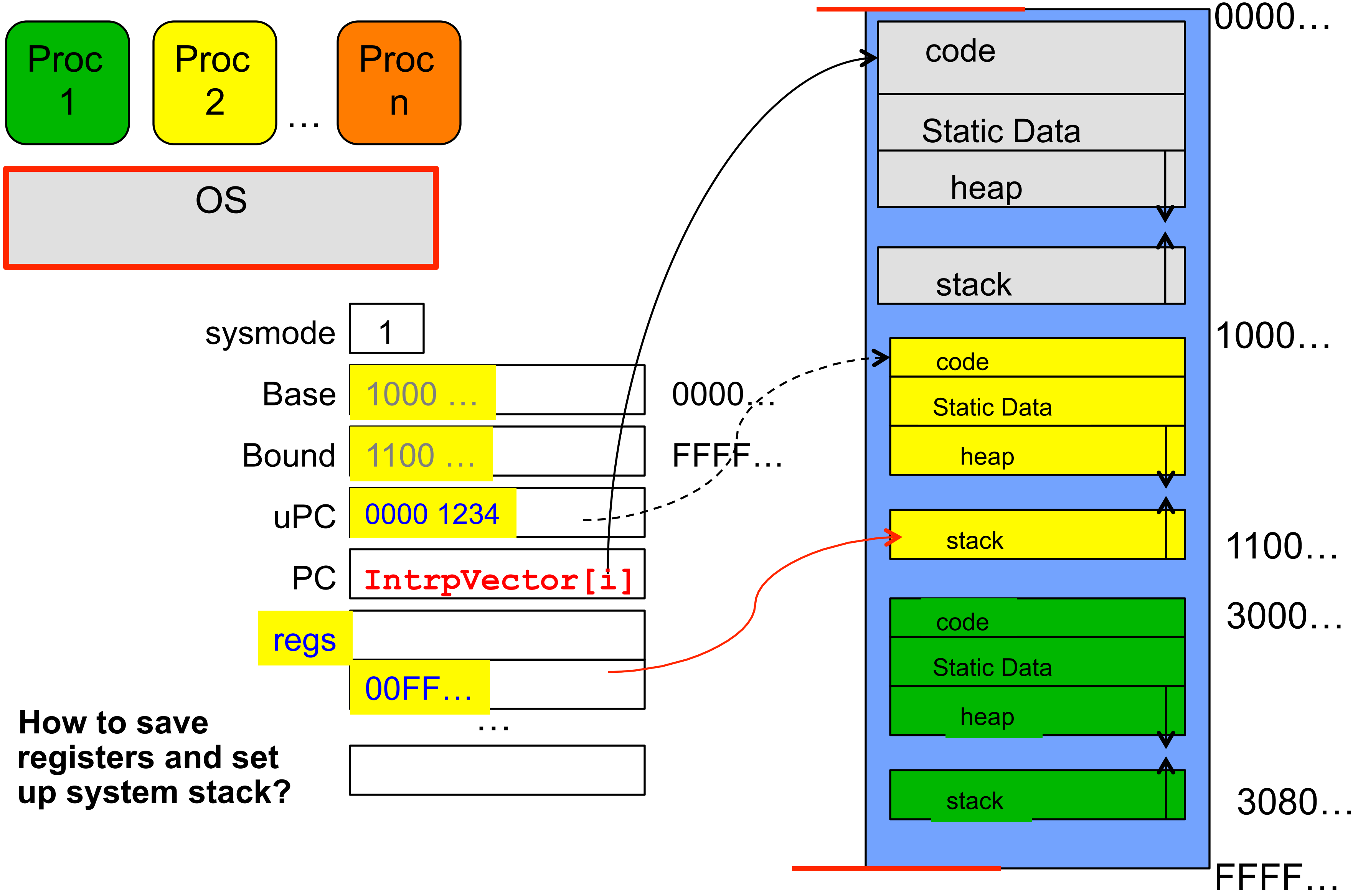
- **Where else do you see this dispatch pattern?**

Simple B&B: User => Kernel



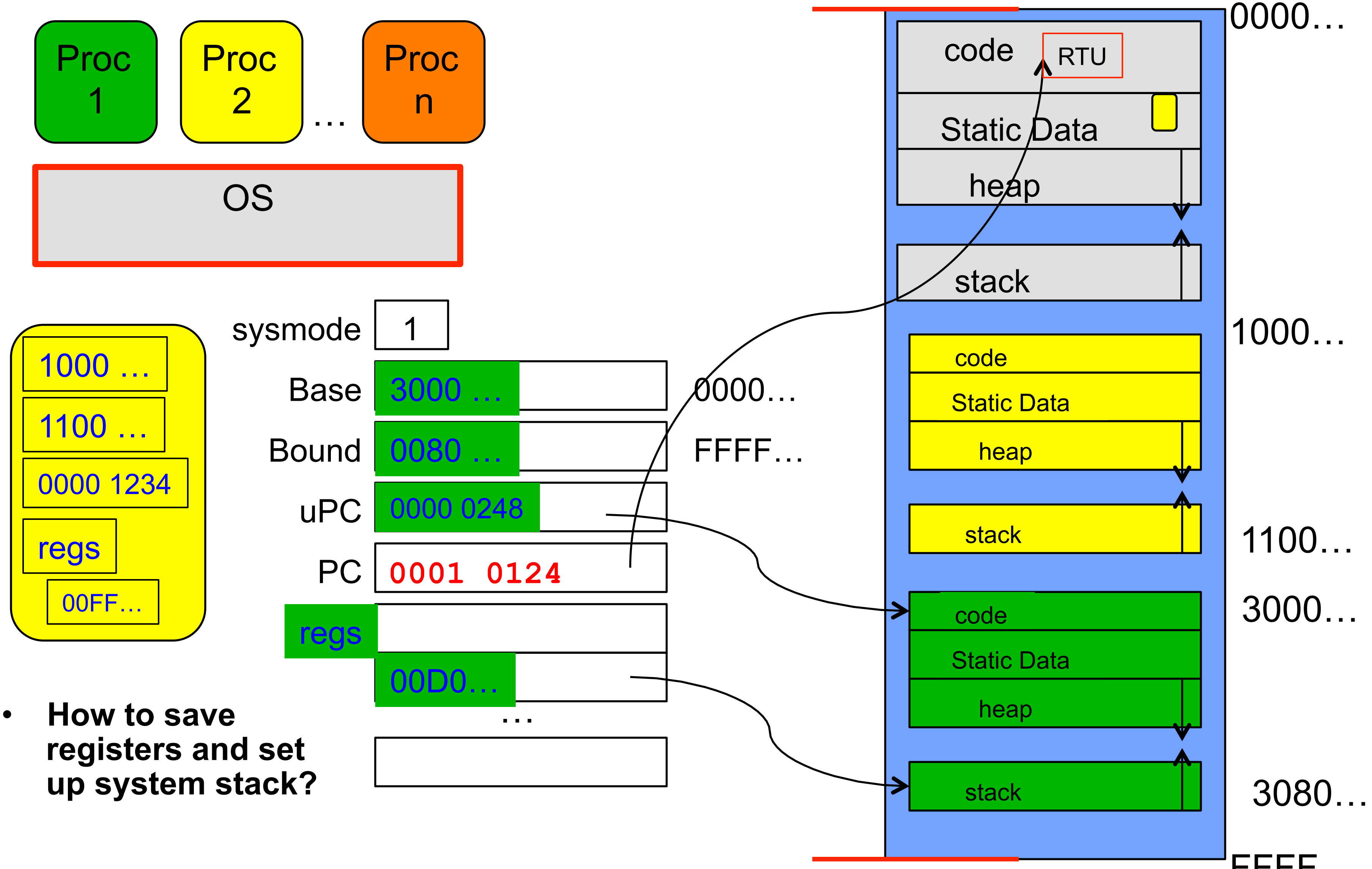
- **How to return to system?**

Simple B&B: Interrupt



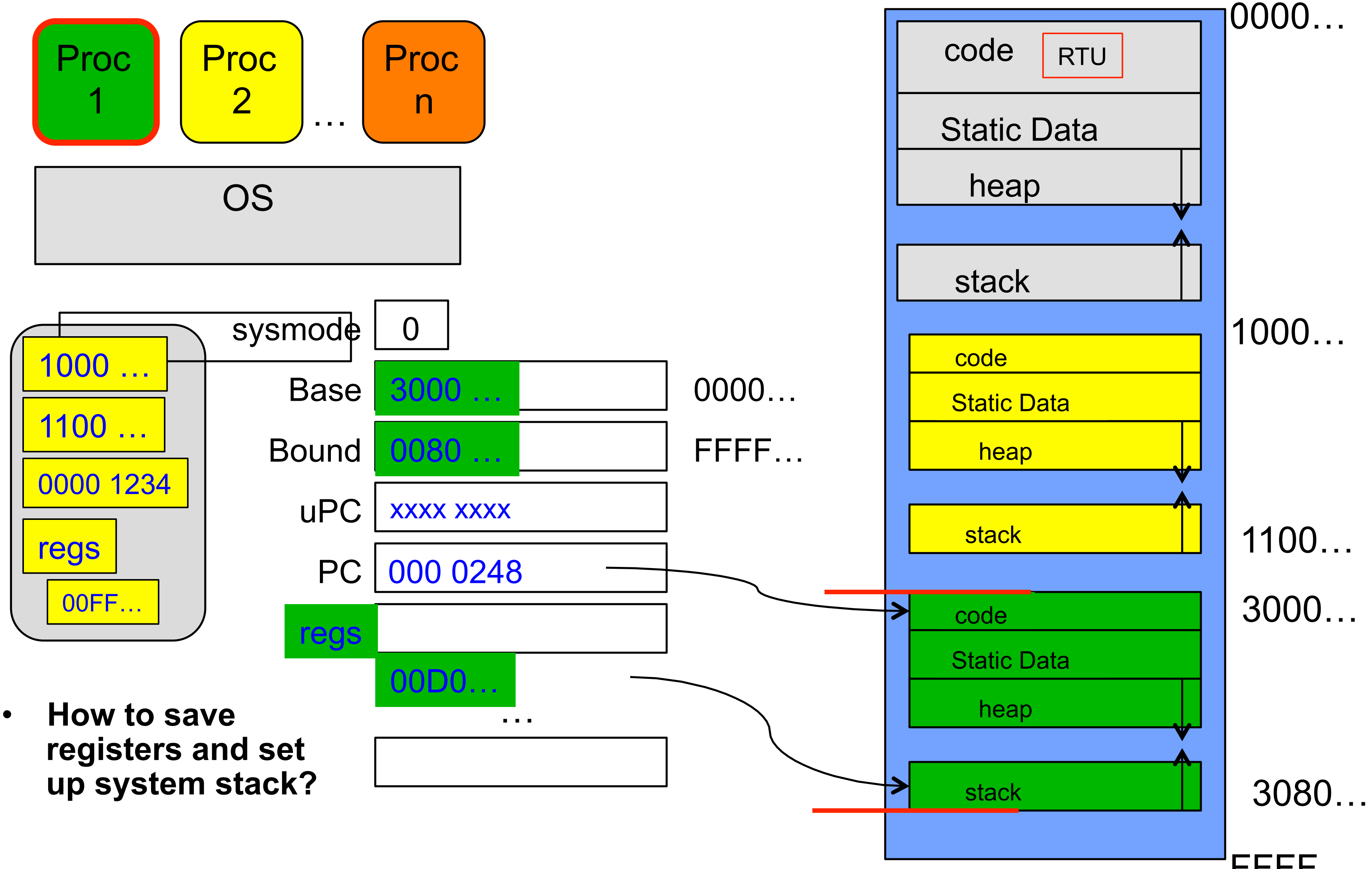
- **How to save registers and set up system stack?**

Simple B&B: Switch User Process



- **How to save registers and set up system stack?**

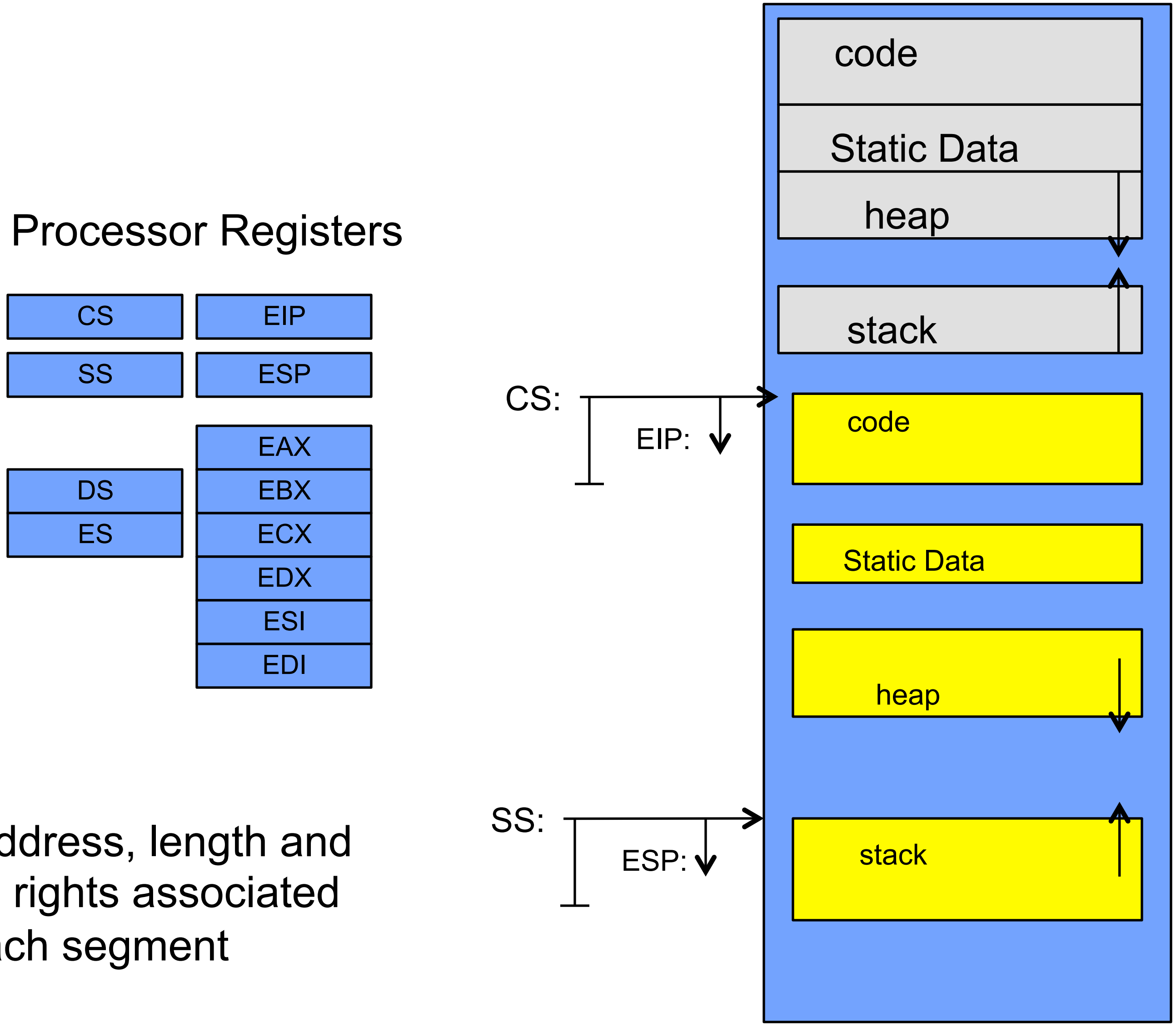
Simple B&B: “resume”



- **How to save registers and set up system stack?**

What’s wrong with this simplistic address translation mechanism?

x86 – segments and stacks



Start address, length and access rights associated with each segment

Virtual Address Translation

- Simpler, more useful schemes too!
- Give every process the illusion of its own BIG FLAT ADDRESS SPACE
  - Break it into pages
  - More on this later

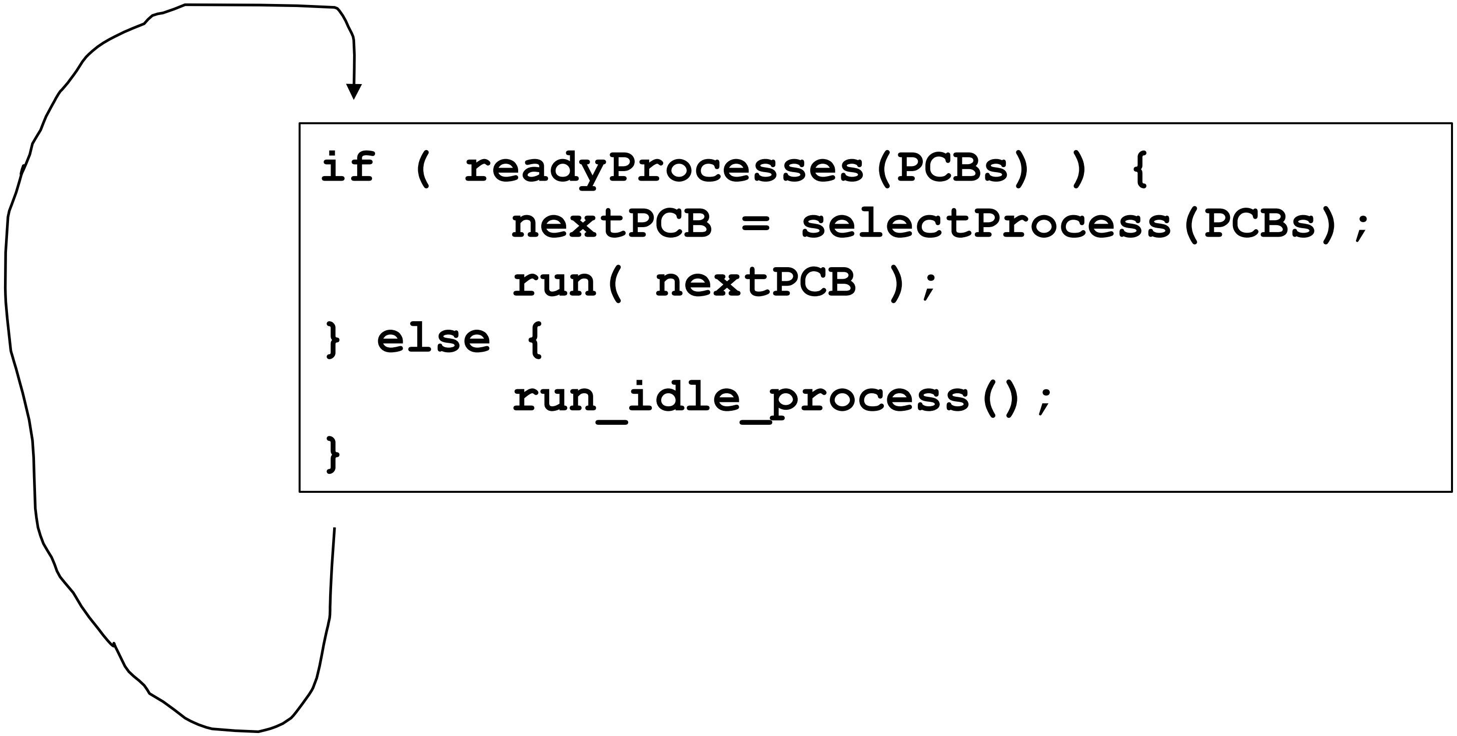
Running Many Programs ???

- 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

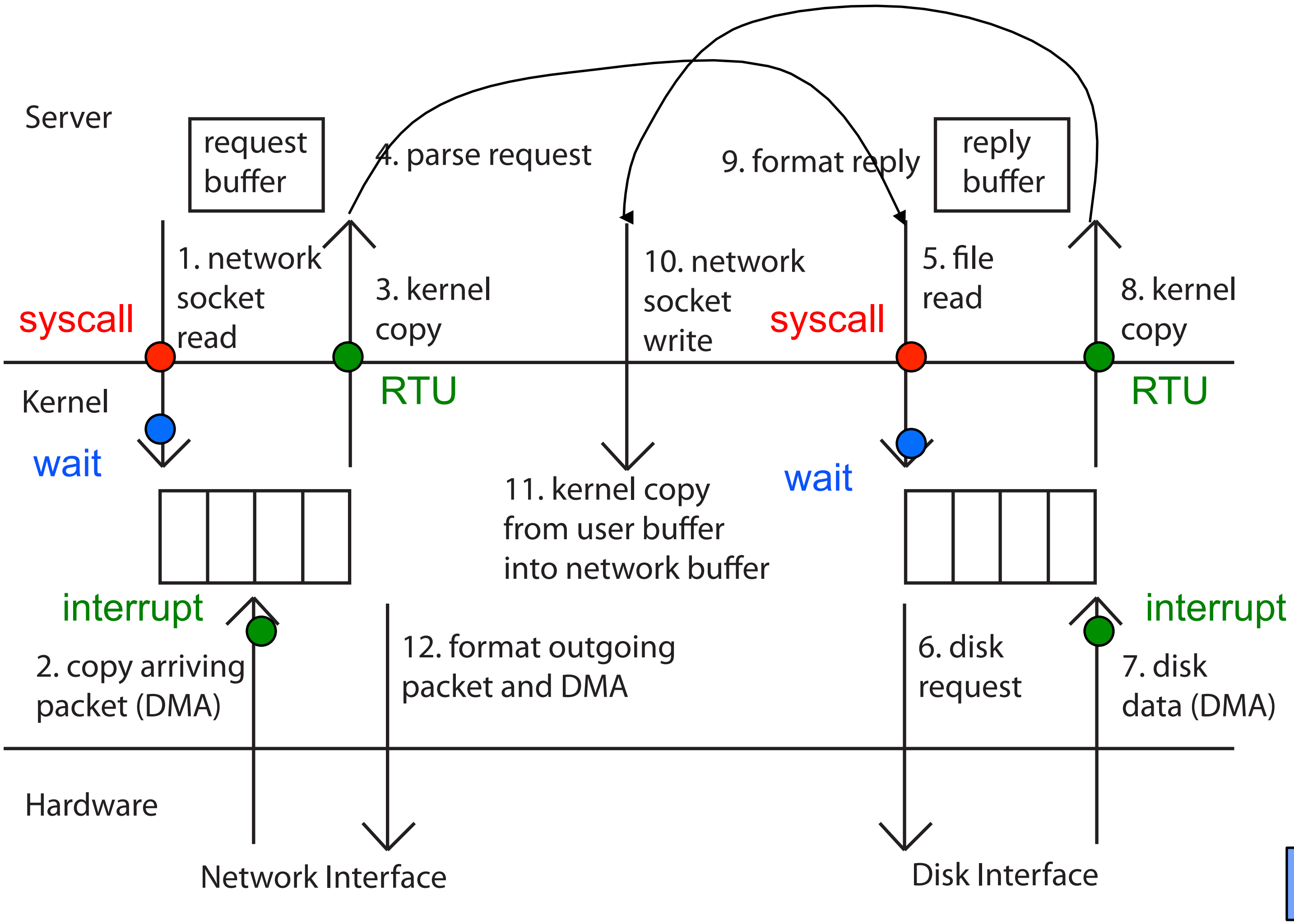
- Kernel represents each process as a process control block (PCB)
  - Status (running, ready, blocked, …)
  - Register state (when not ready)
  - Process ID (PID), User, Executable, Priority, …
  - Execution time, …
  - Memory space, translation, …
- Kernel Scheduler maintains a data structure containing the PCBs
- Scheduling algorithm selects the next one to run

Scheduler



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

Putting it together: web server



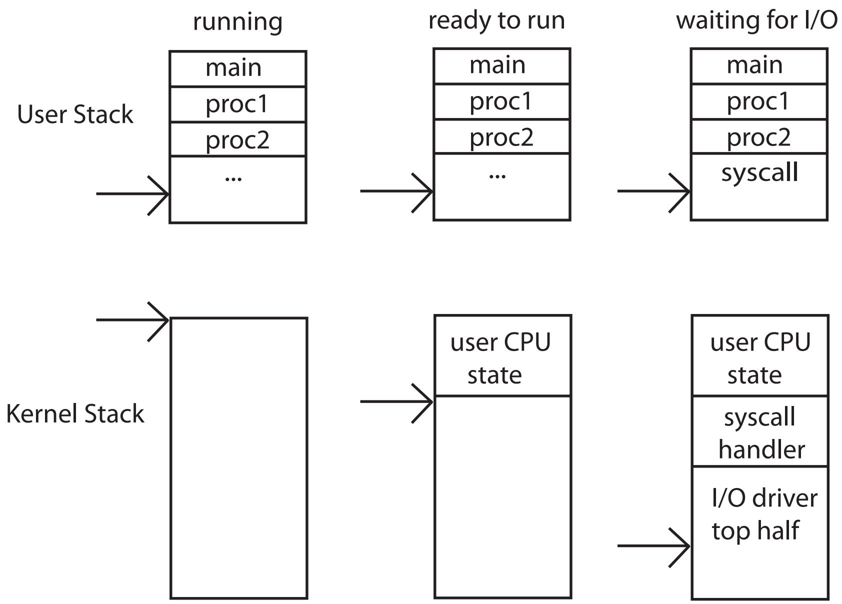
Digging Deeper: Discussion & Questions

Implementing Safe Mode Transfers

- Carefully constructed kernel code packs up the user process state an 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.
- Interrupt processing must not be visible to the user process (why?)
  - Occurs between instructions, restarted transparently
  - No change to process state
  - What can be observed even with perfect interrupt processing?

Kernel Stack Challenge

- 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 (???)



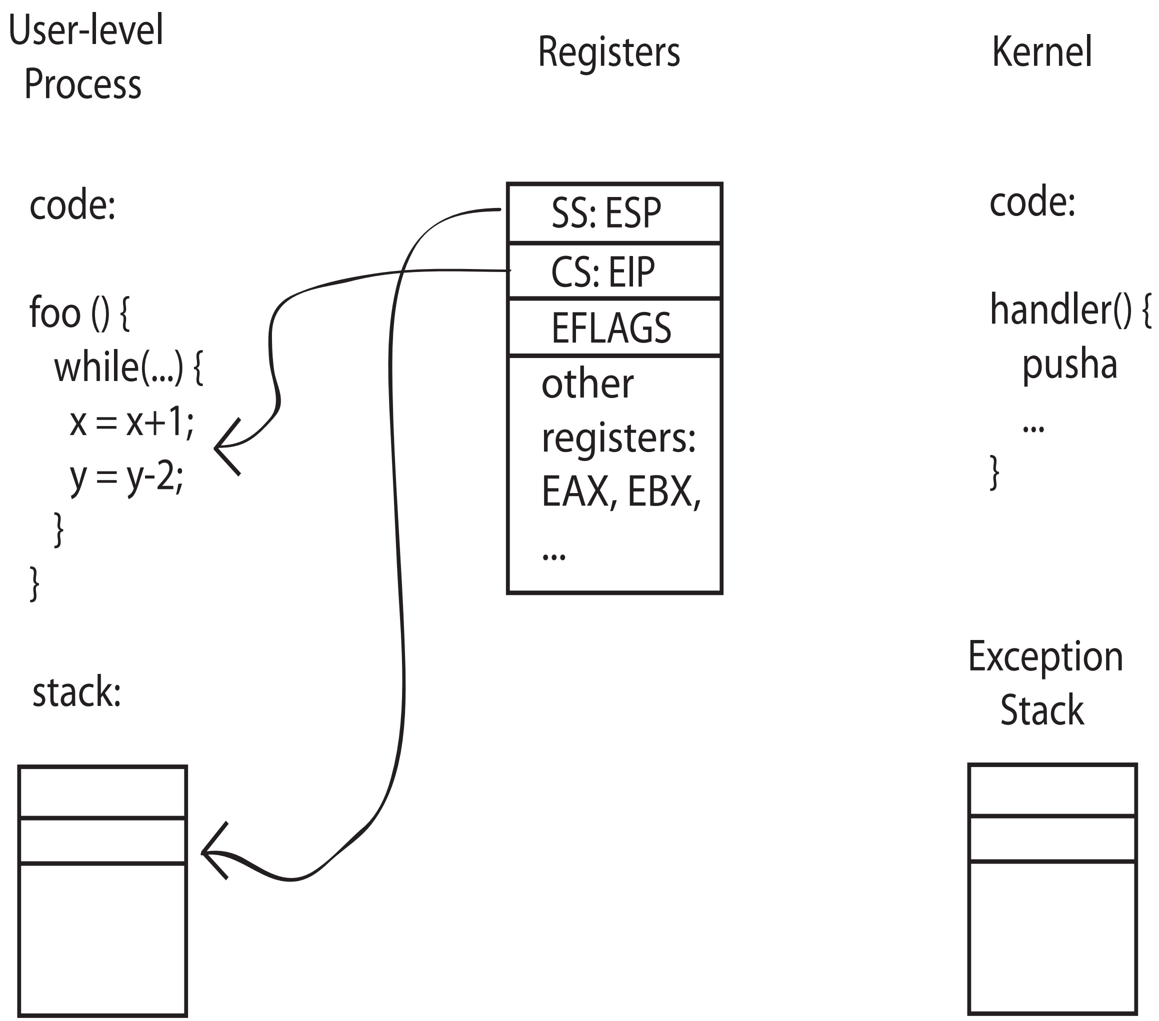
Hardware support: Interrupt Control

- Interrupt Handler invoked with interrupts ‘disabled’
  - Re-enabled upon completion
  - Non-blocking (run to completion, no waits)
  - Pack it up in a queue and pass off to an OS thread to do the 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

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

Before



User-level Process

code:

```
foo () {
  while(...) {
    x = x+1;
    y = y-2;
  }
}
```

stack:

Registers

SS: ESP

CS: EIP

EFLAGS

other registers: EAX, EBX, ...

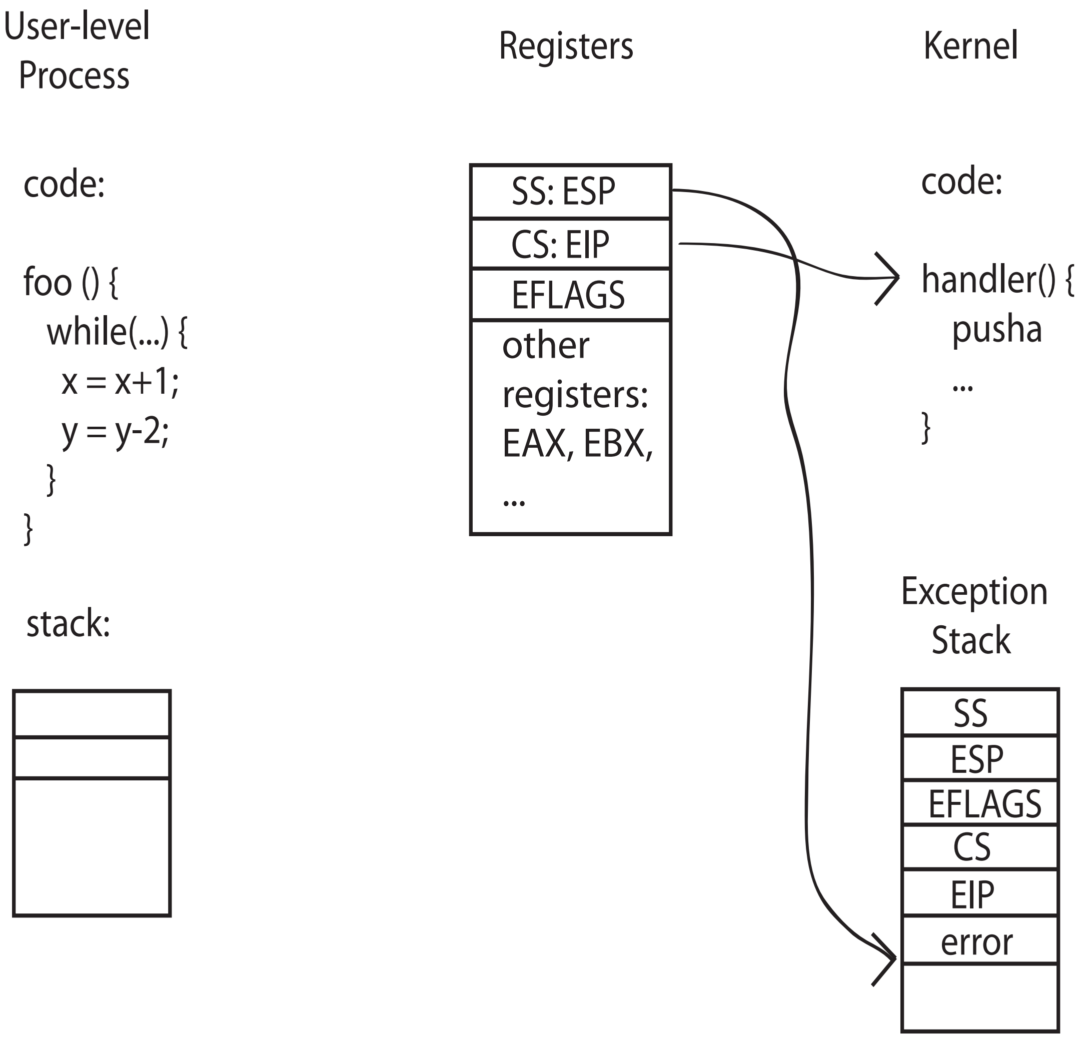
Kernel

code:

```
handler() {
   pusha
   ...
}
```

Exception Stack

During



User-level Process

code:

```
foo () {
  while(...) {
    x = x+1;
    y = y-2;
  }
}
```

stack:

Registers

SS: ESP
CS: EIP
EFLAGS
other registers: EAX, EBX, ...

Kernel

code:

```
handler() {
   pusha
   ...
}
```

Exception Stack

SS
ESP
EFLAGS
CS
EIP
error

Kernel System Call 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

Multiprocessors - Multicores – Multiple Threads

- What do we need to support Multiple Threads
  - Multiple kernel threads?
  - Multiple user threads in a process?
- What if we have multiple Processors / Cores

Idle Loop & Power

- Measly do-nothing unappreciated trivial piece of code that is central to low-power

Performance

- Performance = Operations / Time

- How can the OS ruin application performance?
- What can the OS do to increase application performance?

4 OS concepts working together

- Privilege/User Mode
  - The hardware can operate in two modes, with only the “system” mode having the ability to access certain resources.
- Address Space
  - 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*
- Protection
  - 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