CS 162: Operating Systems and Systems Programming

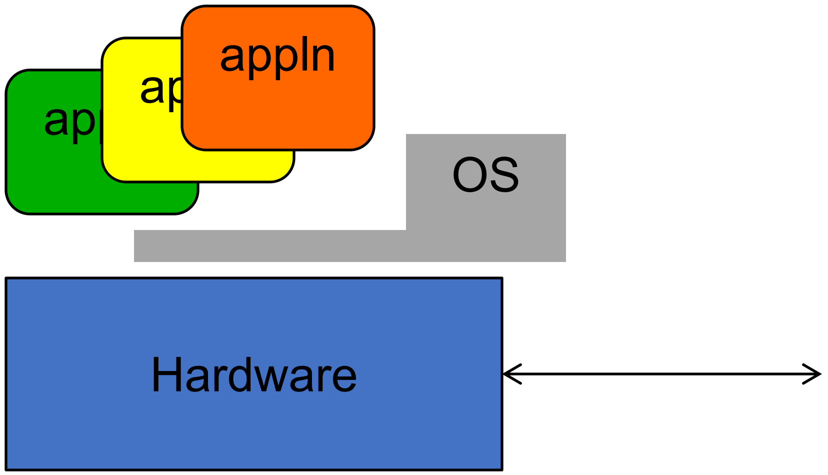
Lecture 2: Fundamental Concepts of Operating Systems

Sept 3, 2019
David Culler
https://cs162.eecs.berkeley.edu

Reading: A&D CH2.1-7, 2.10
8.1-2 (skim)
HW:0 due 9/6
Autograder Registration: TODAY

Recall: What is an operating system?

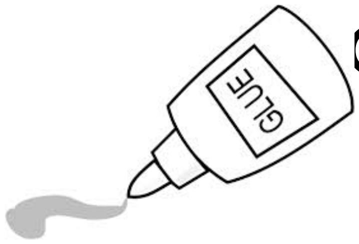
- Special layer of software that provides application software access to hardware resources
  - Convenient abstraction of complex hardware devices
  - Protected access to shared resources
  - Security, Authentication, Isolation, Fault tolerance
  - Communication amongst logical entities



Recall: 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: Protecting Processes

- Use two features offered by hardware:
  1. Dual Mode Operation
     - Processes run in *user* or *kernel* mode
     - Some operations prohibited in user mode
  2. Address Translation
     - Each process has a distinct and isolated *address space*
     - Hardware translates from virtual to physical addresses
- Policy: No program can read or write memory of another program or of the OS

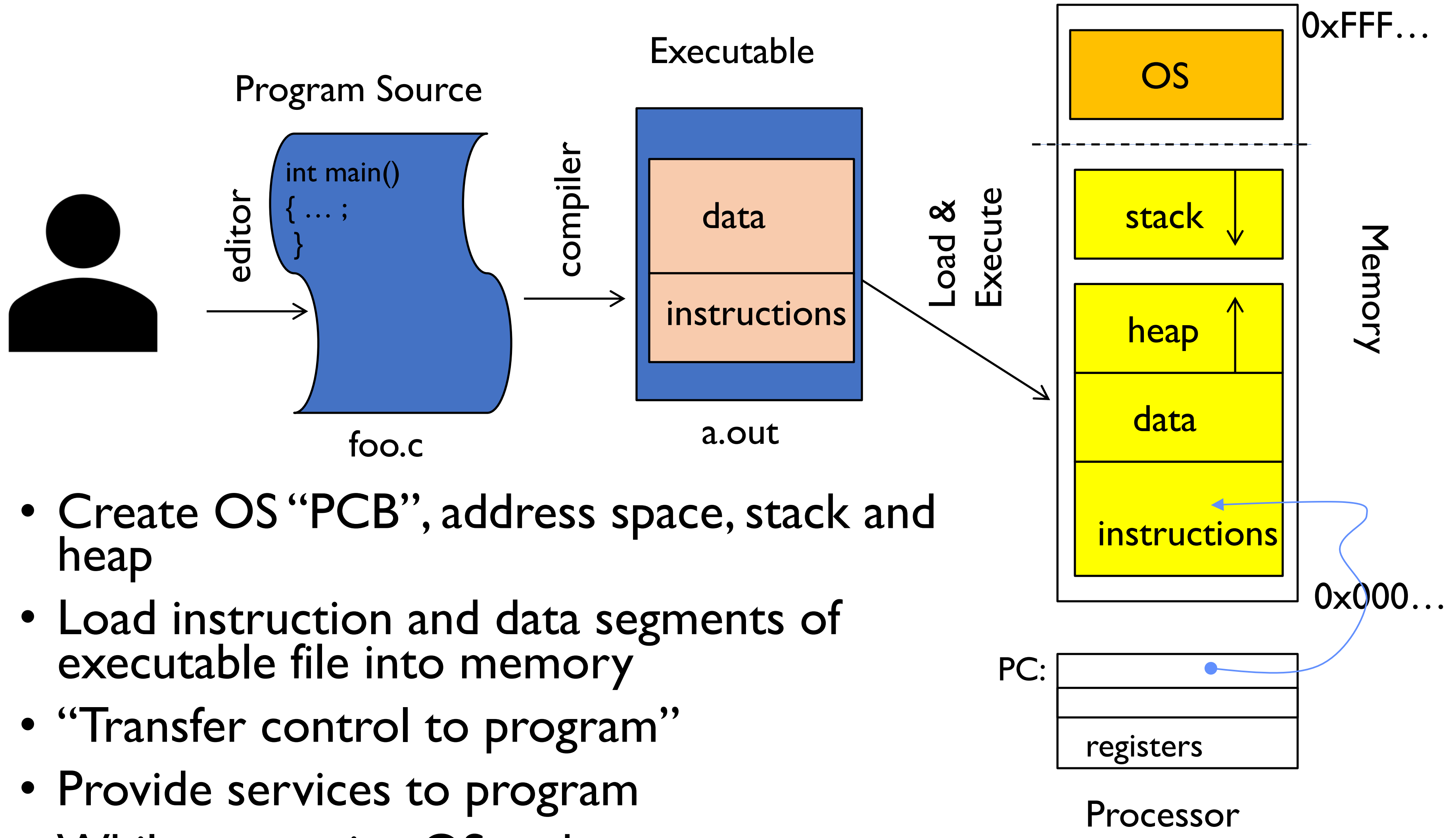
Recall: Virtual Machines

- Definition: Software emulation of physical machine
  - Programs believe *they own the machine*
  - Simulated "hardware" can have *features we want*
  - So perfect can run OS on it (guest OS)
- *Versus Processes and Containers*

Today: Four Fundamental OS Concepts

- **Thread: Execution Context**
  - Program Counter, Registers, Execution Flags, Stack
- **Address space** (with **translation**)
  - Program's view of memory is distinct from physical machine
- **Process: an instance of a running program**
  - Address Space + One or more Threads + …
- **Dual mode operation / Protection**
  - Only the "system" can access certain resources
  - Combined with translation, isolates programs from each other

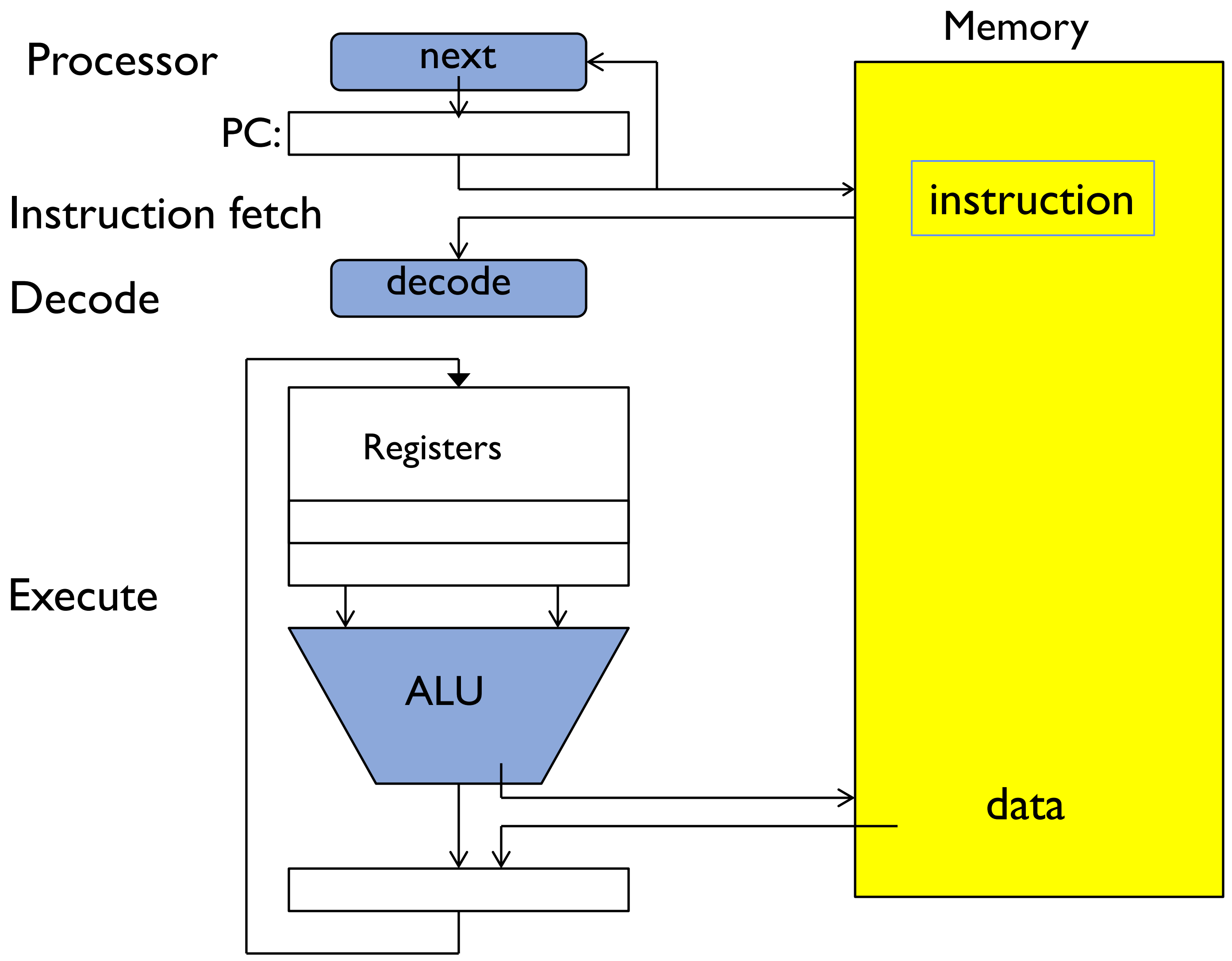
OS Bottom Line: Run Programs



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

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

The instruction cycle



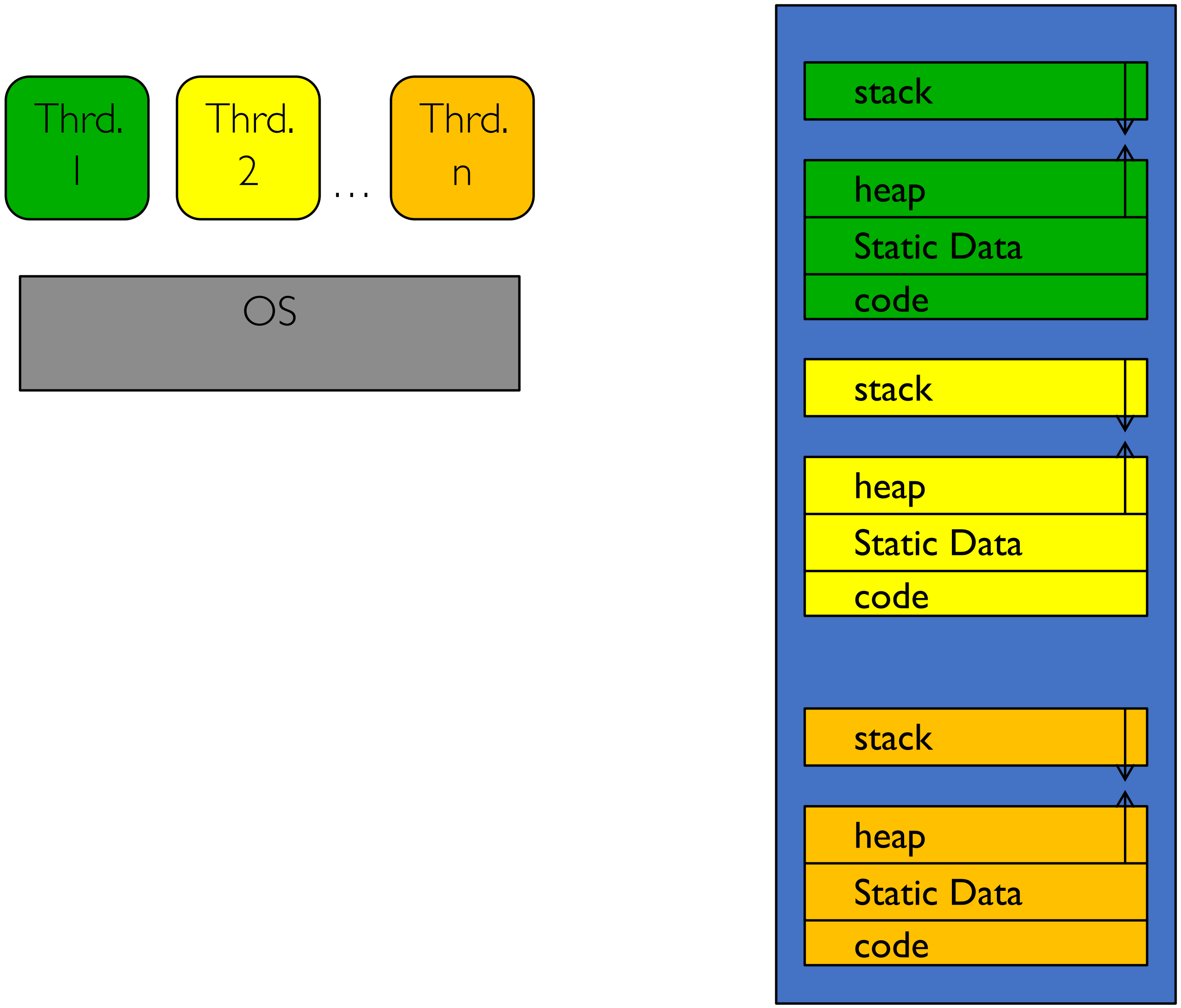
Today: Four Fundamental OS Concepts

- **Thread: Execution Context**
  - Program Counter, Registers, Execution Flags, Stack
- **Address space** (with **translation**)
  - Program's view of memory is distinct from physical machine
- **Process: an instance of a running program**
  - Address Space + One or more Threads
- **Dual mode operation / Protection**
  - Only the "system" can access certain resources
  - Combined with translation, isolates programs from each other

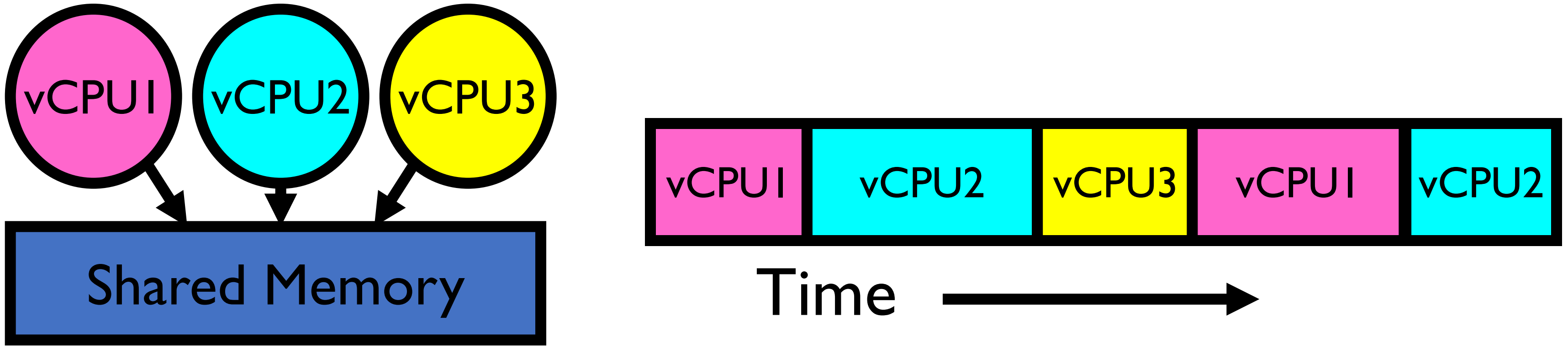
Thread of Control

- Definition: **A single, unique execution context**
    - Program counter, registers, stack
- A thread is *executing* on a processor (core) when it is resident in that processor's registers
- Registers hold the root state of the thread:
    - The rest is "in memory"
    - Including program counter & currently executing instruction
- Registers point to thread state in memory:
    - Stack pointer to the top of the thread's (own) stack

Multiprogramming - Multiple Threads of Control

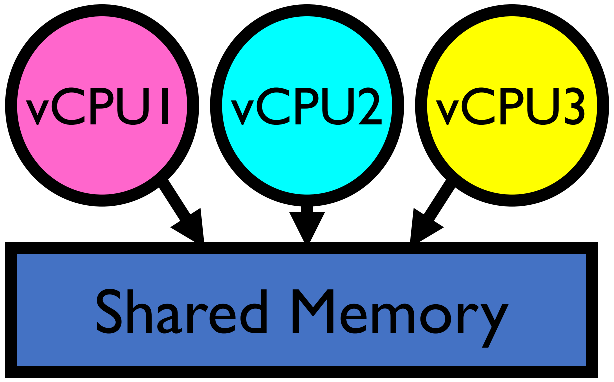


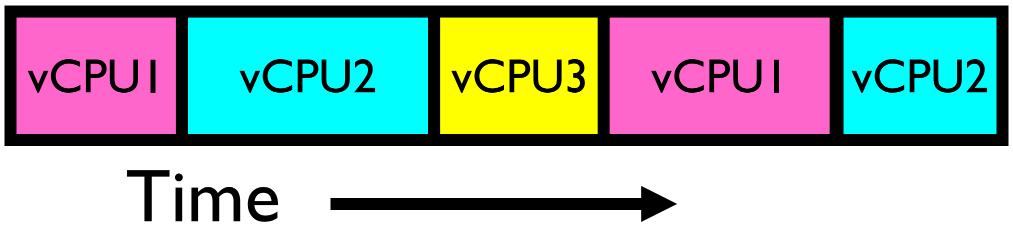
Illusion of Multiple Processors



- Threads are **virtual cores**
- Multiple threads: **Multiplex** hardware in time

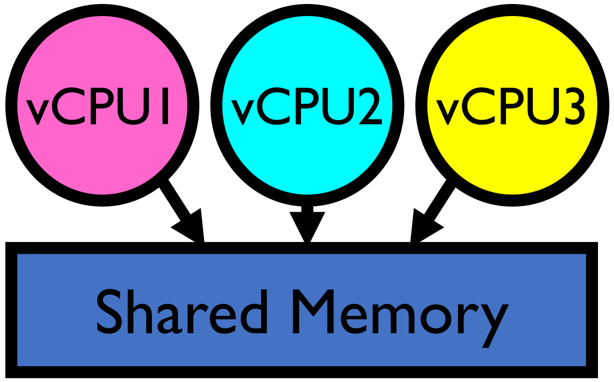
Illusion of Multiple Processors

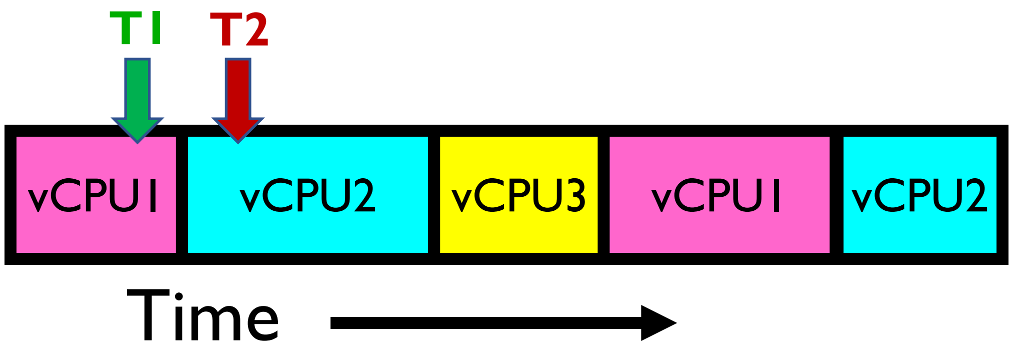




- Contents of virtual core (thread):
  - Program counter, stack pointer
  - Registers
- Where is it?
  - On the real (physical) core, or
  - Saved in memory – called the *Thread Control Block (TCB)*

Illusion of Multiple Processors





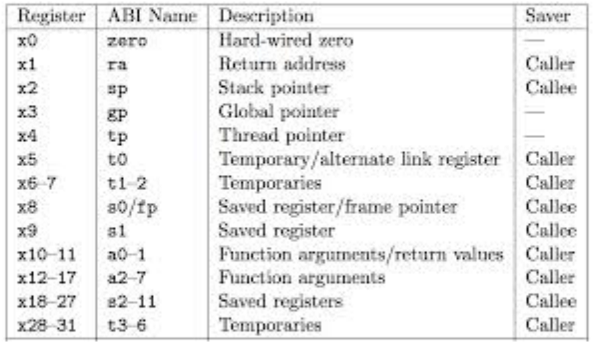
- At T1: vCPU1 on real core, vCPU2 in memory
- At t2: vCPU2 on real core, vCPU1 in memory
- What happened?
  - OS Ran [how?]
  - Saved PC, SP, … in vCPU1's thread control block (memory)
  - Loaded PC, SP, … from vCPU2's TCB, jumped to PC

OS object representing a thread?

- Traditional term: Thread Control Block (TCB)
- Holds contents of registers when thread is not running
- What other information?

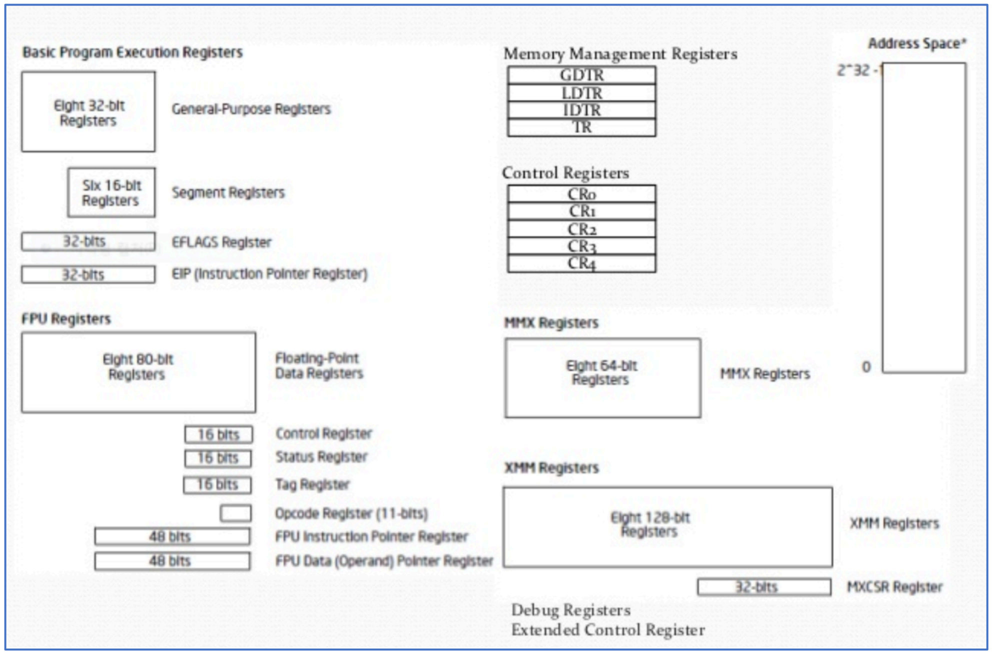
- PINTOS? – read thread.h and thread.c

Registers: RISC-V => x86



Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

Load/Store Arch with software conventions



Complex mem-mem arch with specialized registers and "segments"

- cs61C does RISC-V. Will need to learn x86…
- Section this week

Very Simple Multiprogramming

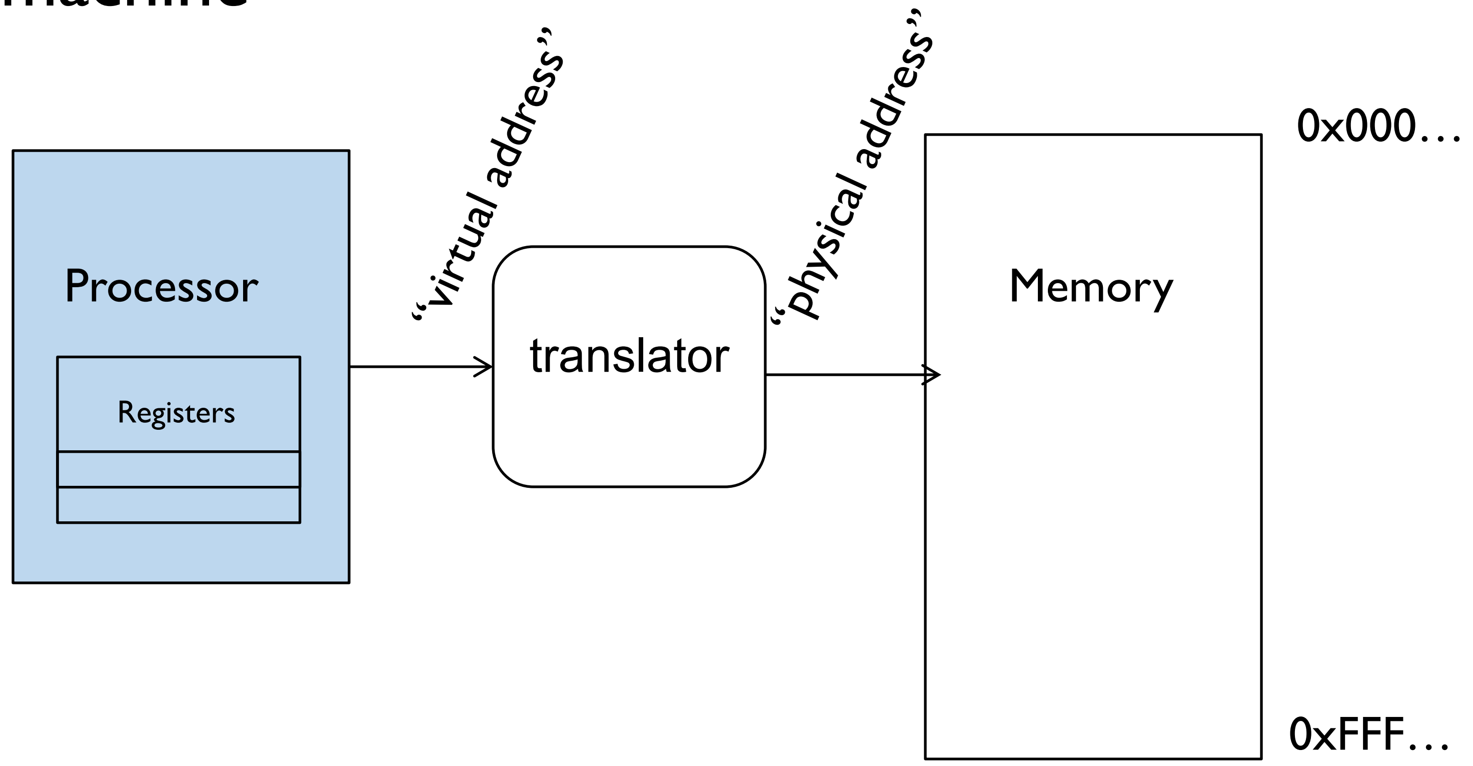
- All vCPU's share non-CPU resources
  - Memory, I/O Devices
- Each thread can **read/write memory**
  - Perhaps data of others
  - can overwrite OS ?
- Unusable?
- This approach is used in
  - Very early days of computing
  - Embedded applications
  - MacOS 1-9/Windows 3.1 (switch only with voluntary yield)
  - Windows 95-ME (switch with yield or timer)

Four Fundamental OS Concepts

- **Thread: Execution Context**
  - Program Counter, Registers, Execution Flags, Stack
- **Address space** (with **translation**)
  - Program's view of memory is distinct from physical machine
- **Process: an instance of a running program**
  - Address Space + One or more Threads
- **Dual mode operation / Protection**
  - Only the "system" can access certain resources
  - Combined with translation, isolates programs from each other

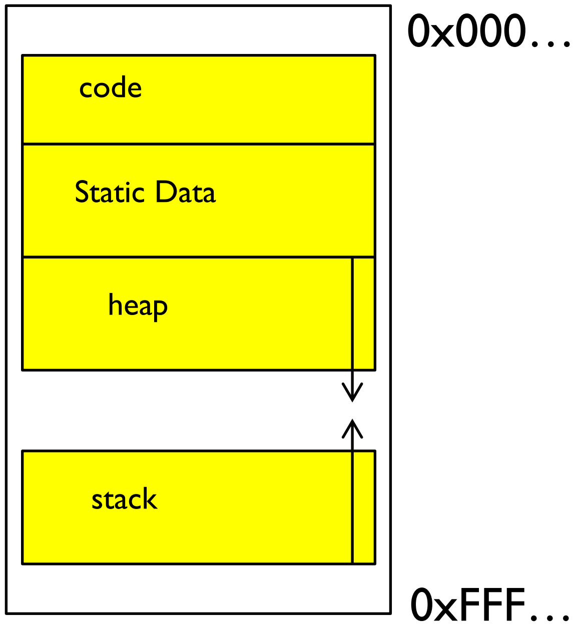
Key OS Concept: Address Space

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

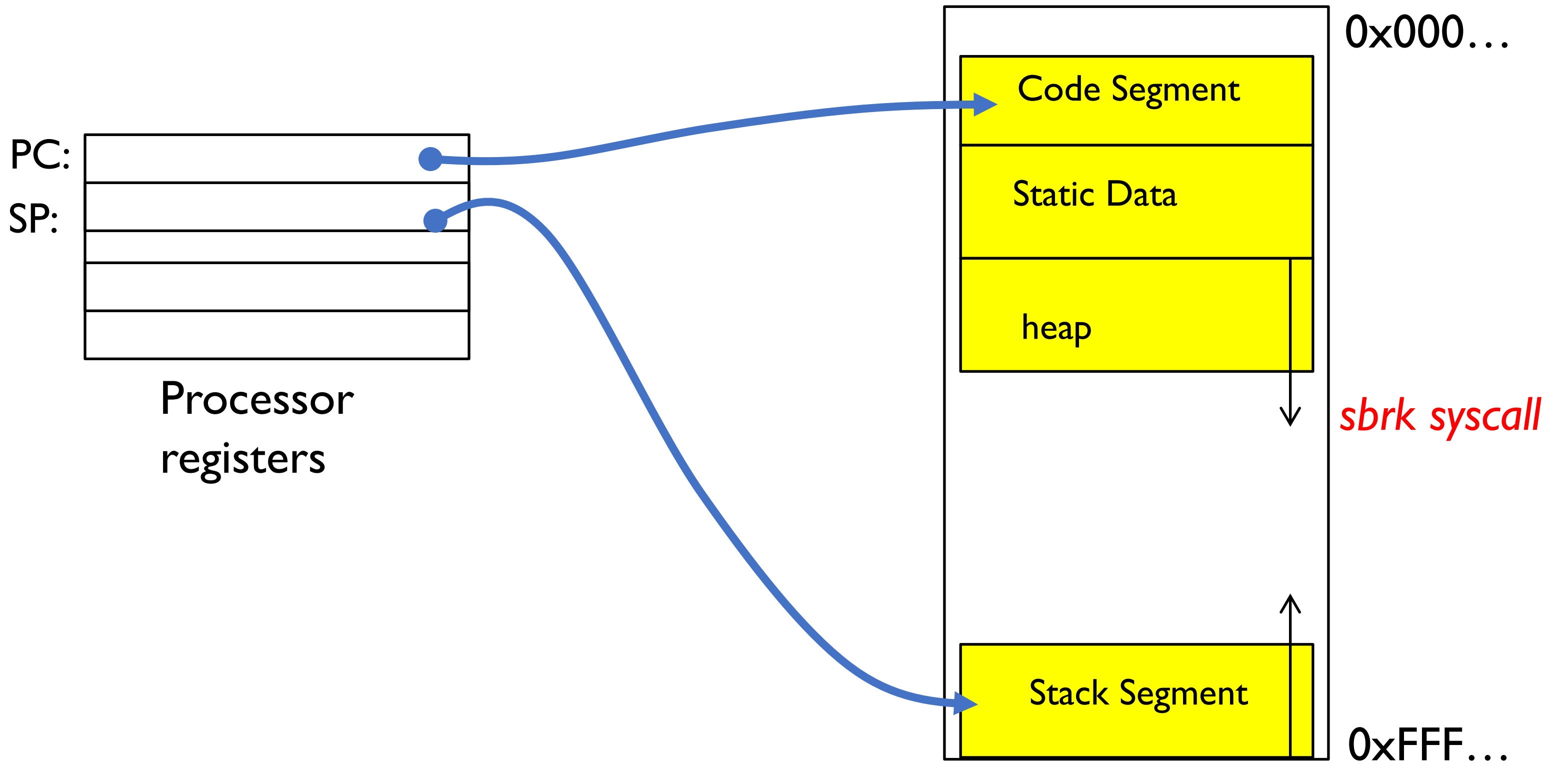


Address Space

- Definition: **Set of accessible addresses and the state associated with them**
  - 2^{32} = ~4 billion on a 32-bit machine
- What happens when you read or write to an address?
  - Perhaps acts like regular memory
  - Perhaps causes I/O operation
    - (Memory-mapped I/O)
  - Causes program to abort (segfault)?
  - Communicate with another program
  - …

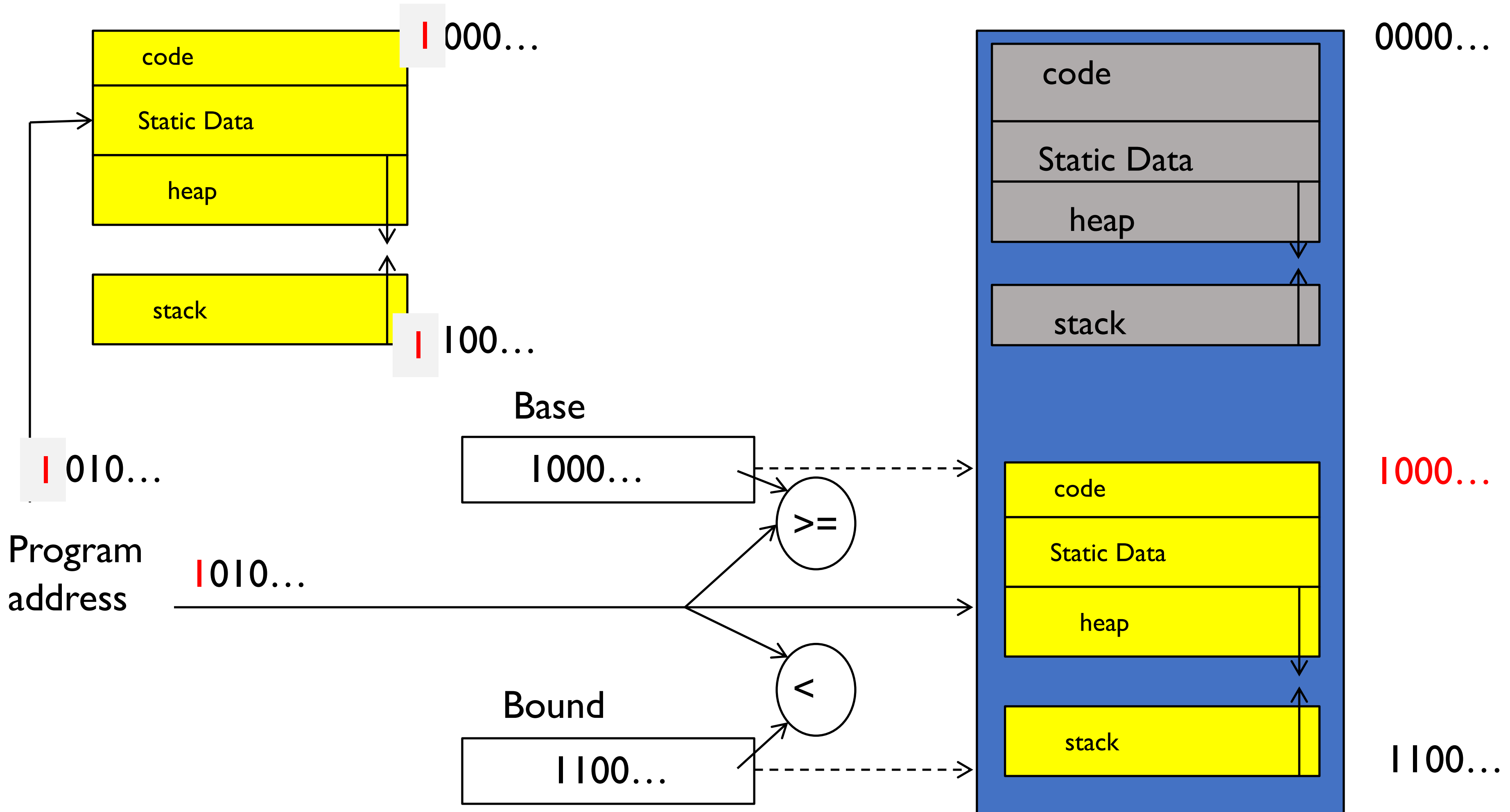


Address Space: typical structure



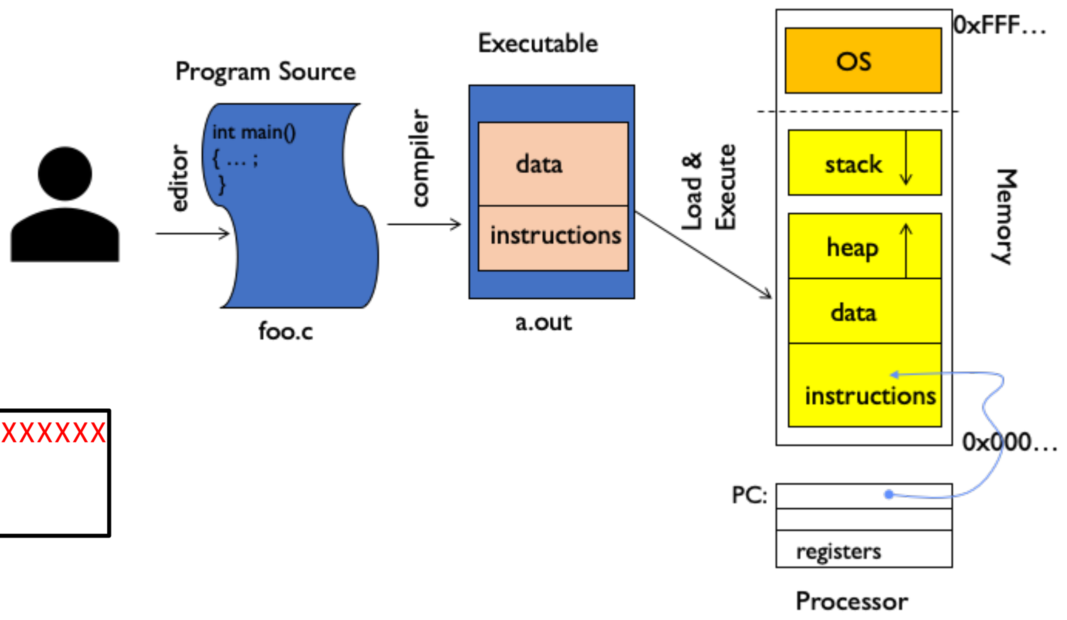
What can the hardware do to help the operating system protect itself from programs? Programs from others?

Strawman: Base and Bound (**no translation**)



- Protects OS and isolates program
- Requires *relocation* in loading

61C Review: Relocation



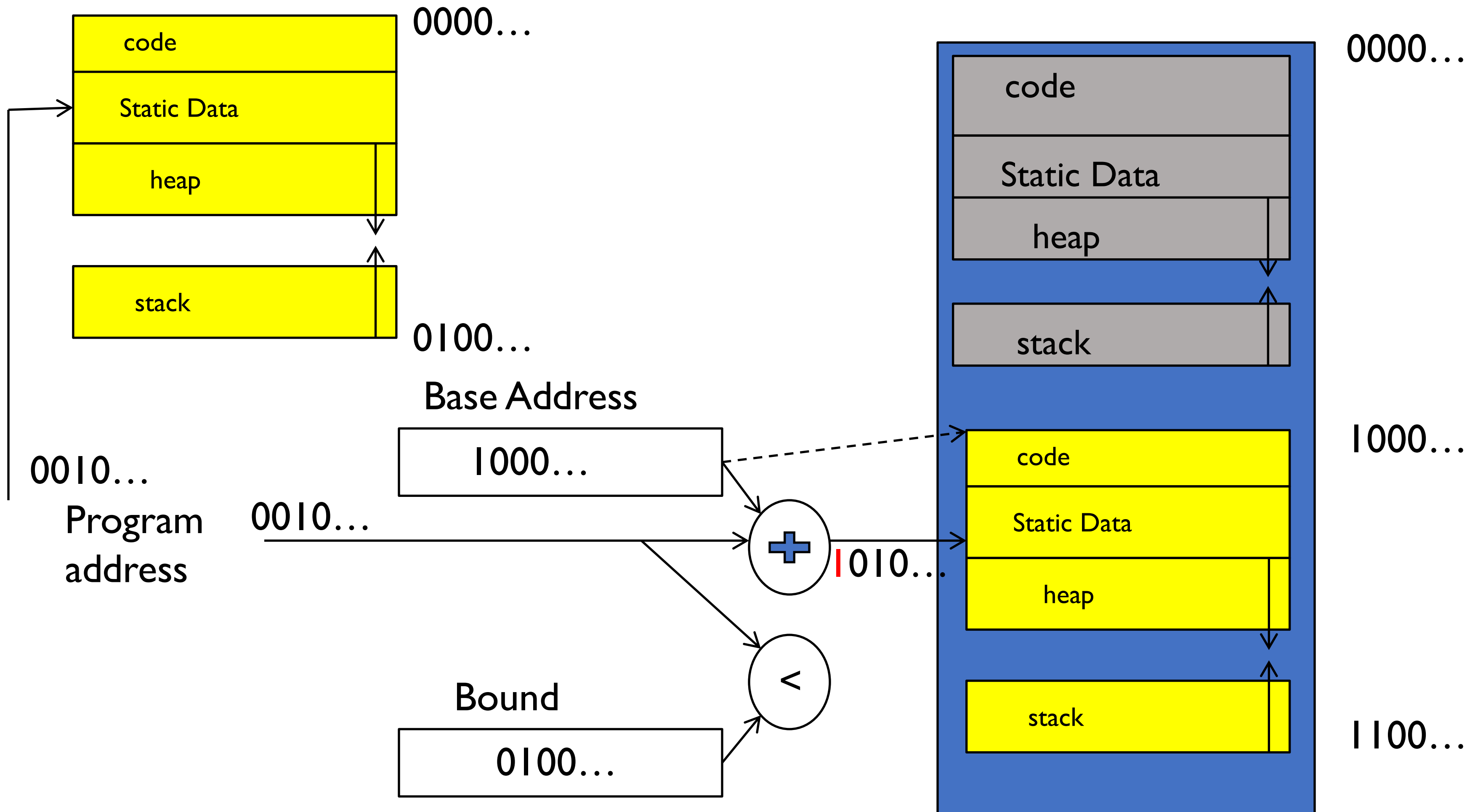
```
jal printf
```

```
000011 XX XXXXXXX XXXXXXX XXXXXXX
```

opcode for jal **address of printf (shifted right by 2)**

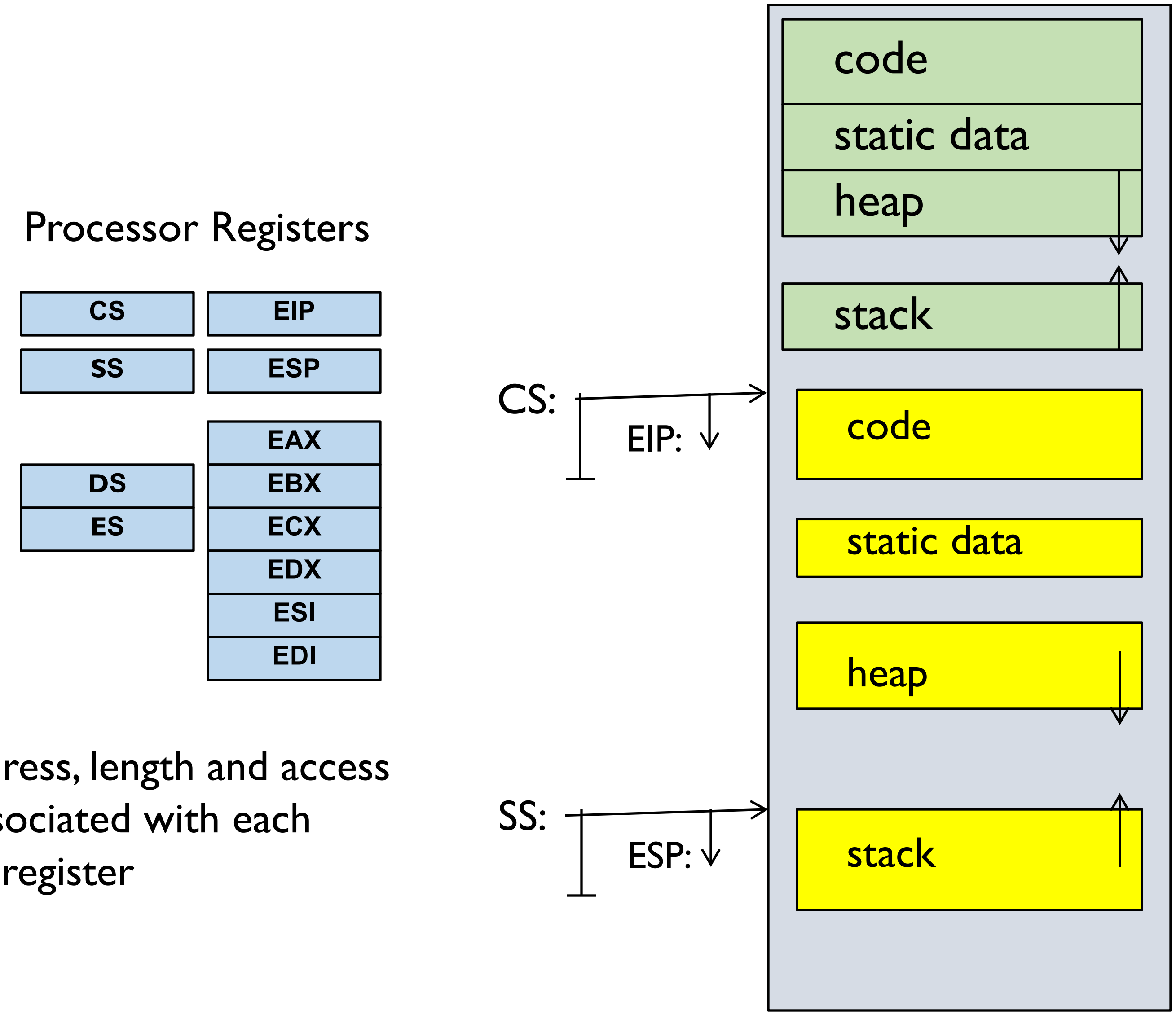
- Compiled .obj file linked together in an .exe
- All address in the .exe are as if it were loaded at memory address 00000000
- File contains a list of all the addresses that need to be adjusted when it is "relocated" to somewhere else.

What if we just added in the base?



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

x86 – segments and stacks

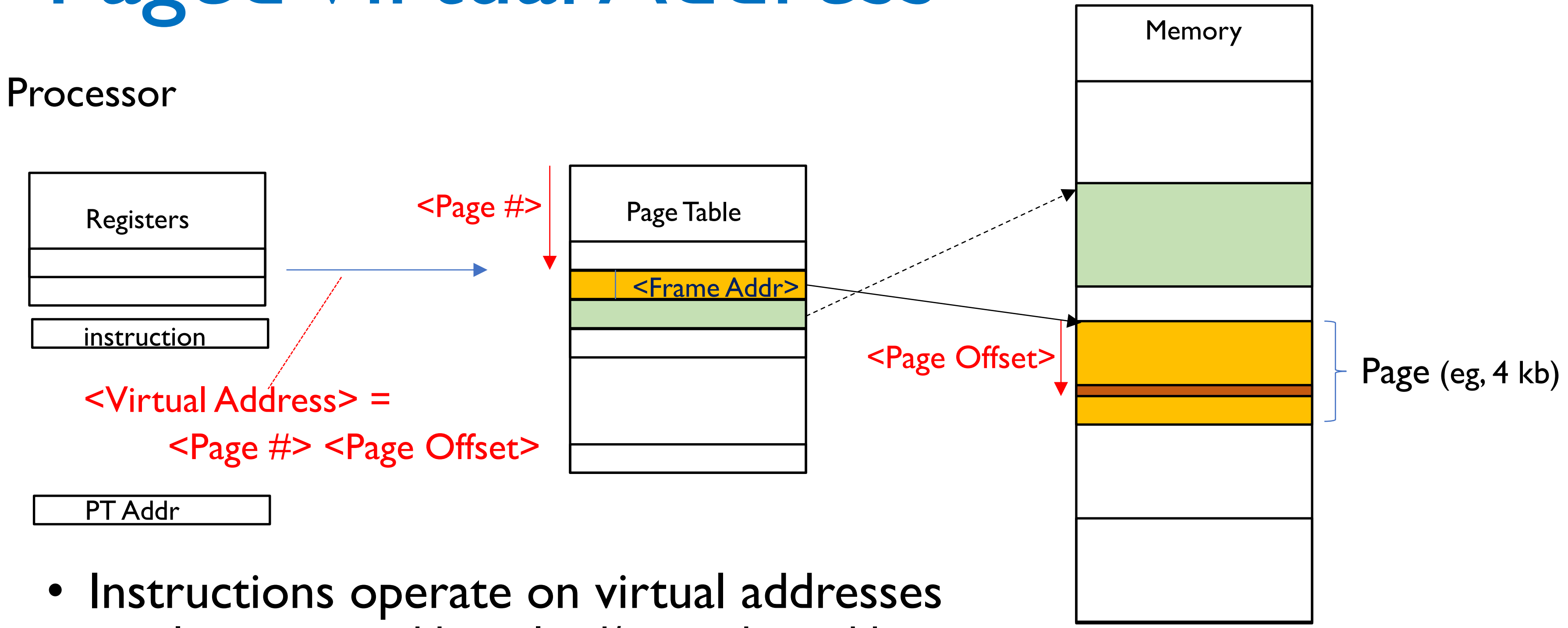


Start address, length and access rights associated with each segment register

Paged Virtual Address Space

- What if we break the entire virtual address space into equal size chunks (i.e., pages) have a base for each?
- Treat memory as page size frames and put any page into any frame …
- Another cs61C review

Paged Virtual Address



- Instructions operate on virtual addresses
  - Instruction address, load/store data address
- Translated to a physical address (or Page Fault) through a Page Table **by the hardware**
- Any Page of address space can be in any (page sized) frame in memory
  - Or not-present (access generates a page fault)
- Special register holds page table base address (of the process)

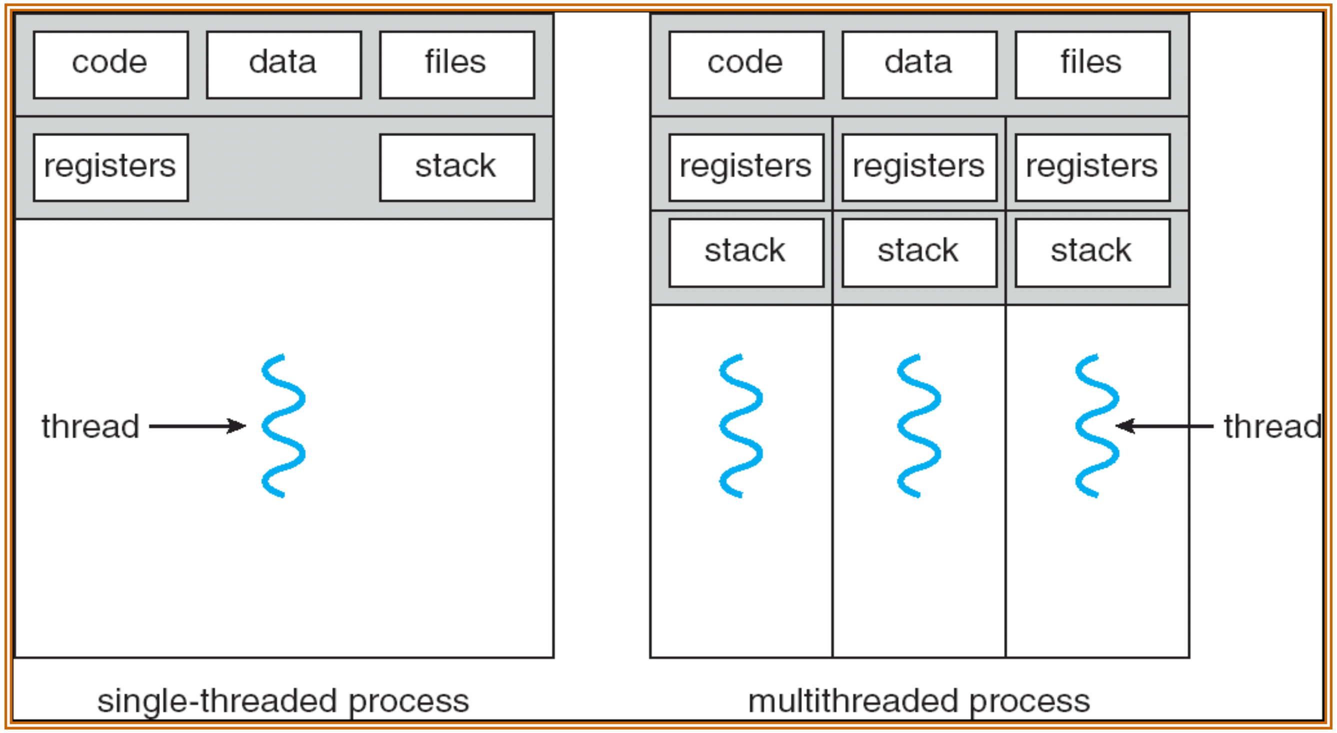
Four Fundamental OS Concepts

- **Thread: Execution Context**
  - Program Counter, Registers, Execution Flags, Stack
- **Address space** (with **translation**)
  - Program's view of memory is distinct from physical machine
- **Process: an instance of a running program**
  - Address Space + One or more Threads
- **Dual mode operation / Protection**
  - Only the "system" can access certain resources
  - Combined with translation, isolates programs from each other

The Process

- Definition: **execution environment with restricted rights**
  - Address Space with One or More Threads
  - Owns memory (mapped pages)
  - Owns file descriptors, file system context, …
  - Encapsulates 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.
  - Execute concurrently [trade-offs with threads? later]
  - Basic unit OS deals with

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?

Kernel code/data in process VAS?

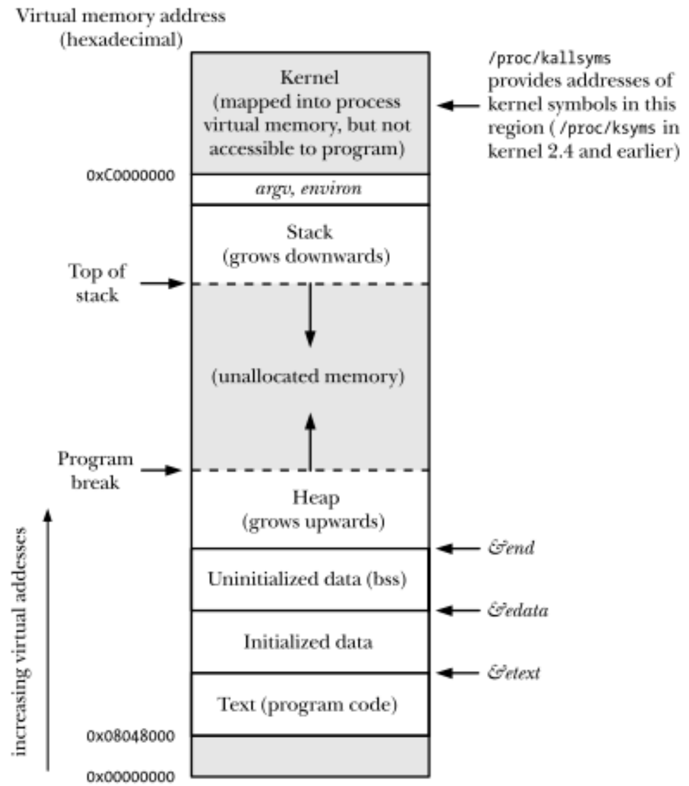

Virtual memory address (hexadecimal)
Kernel (mapped into process virtual memory, but not accessible to program)
/proc/kallsyms provides addresses of kernel symbols in this region (/proc/ksyms in kernel 2.4 and earlier)
0xC0000000
argv, environ
Stack (grows downwards)
Top of stack
(unallocated memory)
Program break
Heap (grows upwards)
&end
Uninitialized data (bss)
&edata
Initialized data
&etext
Text (program code)
0x08048000
0x00000000
increasing virtual addresses

Figure 6-1: Typical memory layout of a process on Linux/x86-32

How does PINTOS lay out the VAS?

- Unix: Kernel space is mapped in high - but inaccessible to user processes

Peer discussion: How does kernel access physical resources?

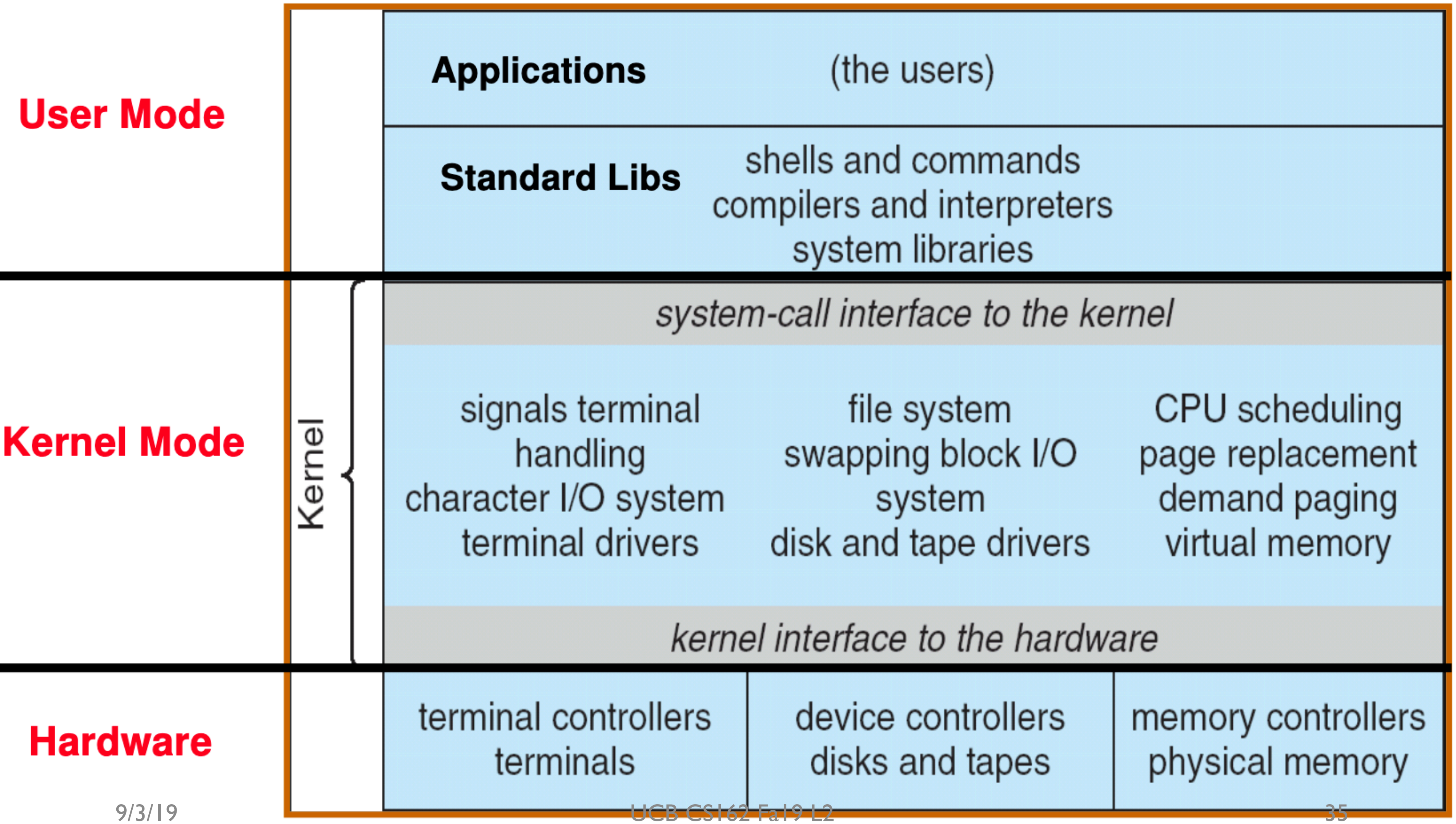
- The actual RAM
- (memory mapped) I/O devices

More cs61c review

Protection (Isolation)

- Why?
  - Reliability: buggy programs only hurt themselves
  - Security and privacy: trust programs less
  - Fairness: enforce shares of disk, CPU
- Mechanisms:
  - Address translation: address space only contains its own data
  - Syscall processing (e.g., enforce file access rights)
    - Thread "traps" to the OS
  - Privileged instructions, registers (???)

Typical system (Unix) structure – what happens where?



Four Fundamental OS Concepts

- **Thread: Execution Context**
  - Program Counter, Registers, Execution Flags, Stack
- **Address space** (with **translation**)
  - Program's view of memory is distinct from physical machine
- **Process: an instance of a running program**
  - Address Space + One or more Threads
- **Dual mode operation / Protection**
  - Only the "system" can access certain resources
  - Combined with translation, isolates programs from each other

Dual Mode Operation: HW Support

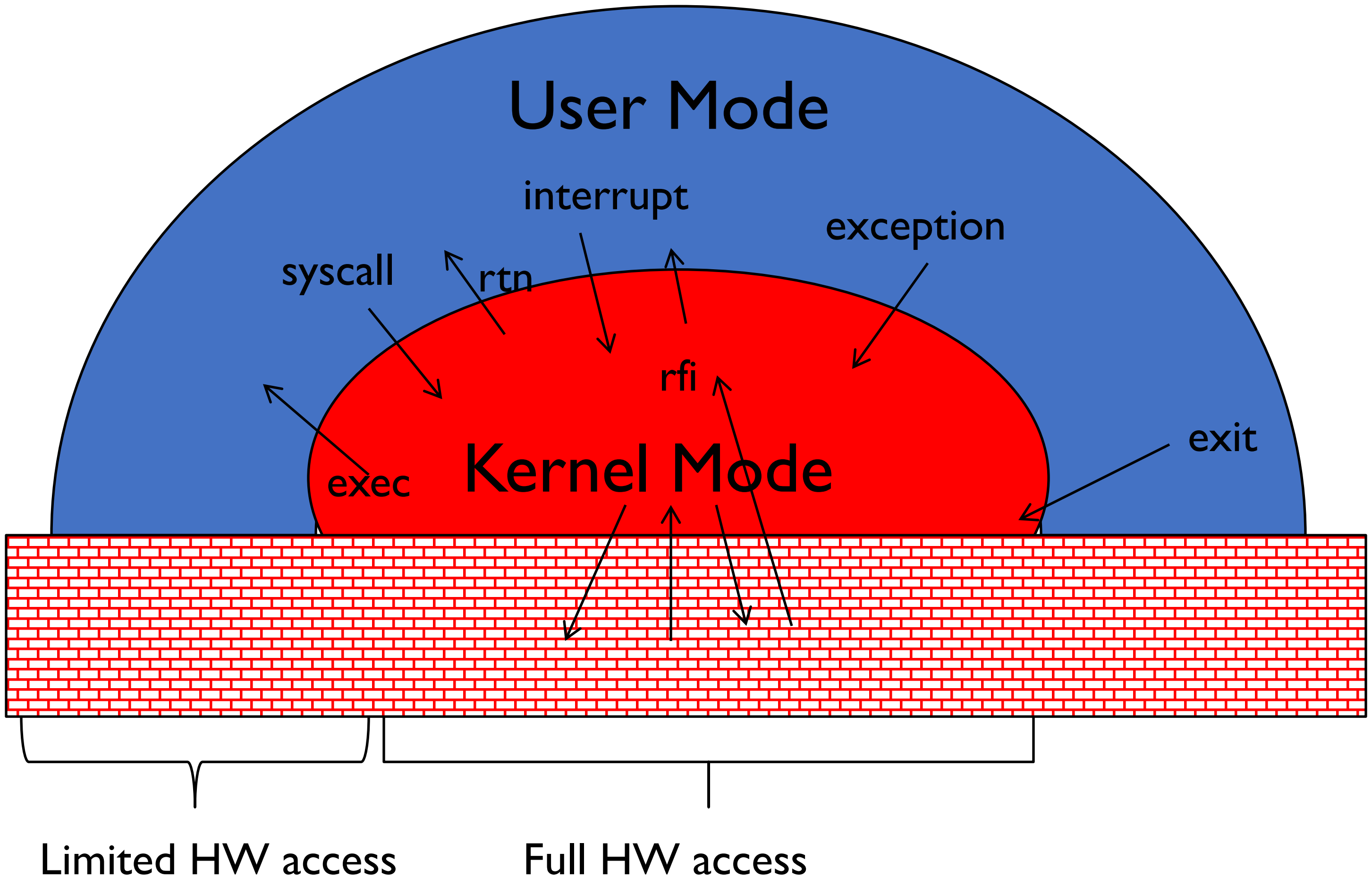
- **bit** of state (user/kernel mode)
- Certain actions only permitted in kernel mode
  - Which page table entries can be accessed
- User => Kernel mode sets kernel mode, saves user PC
  - Only transition to *OS-designated addresses*
  - OS can save the rest of the user state if necessary
- Kernel => User mode sets user mode, restores user PC, …

Logistics

- Course expanded to 487 students
  - Waitlist order
- HW0 due on Friday 9/6, 9:00 PM
- C & 61C Review tonight at 7
- Project 1 Pre-Released
- HW1 release 9/7
- Group formation enabled on autograder
  - Opens 9/7, deadline 9/11
- Requested and in process
  - Alternate Midterm: Thurs 10/10 4-6 pm
  - Alternate Final: Mon 12/16 3-6pm

Break

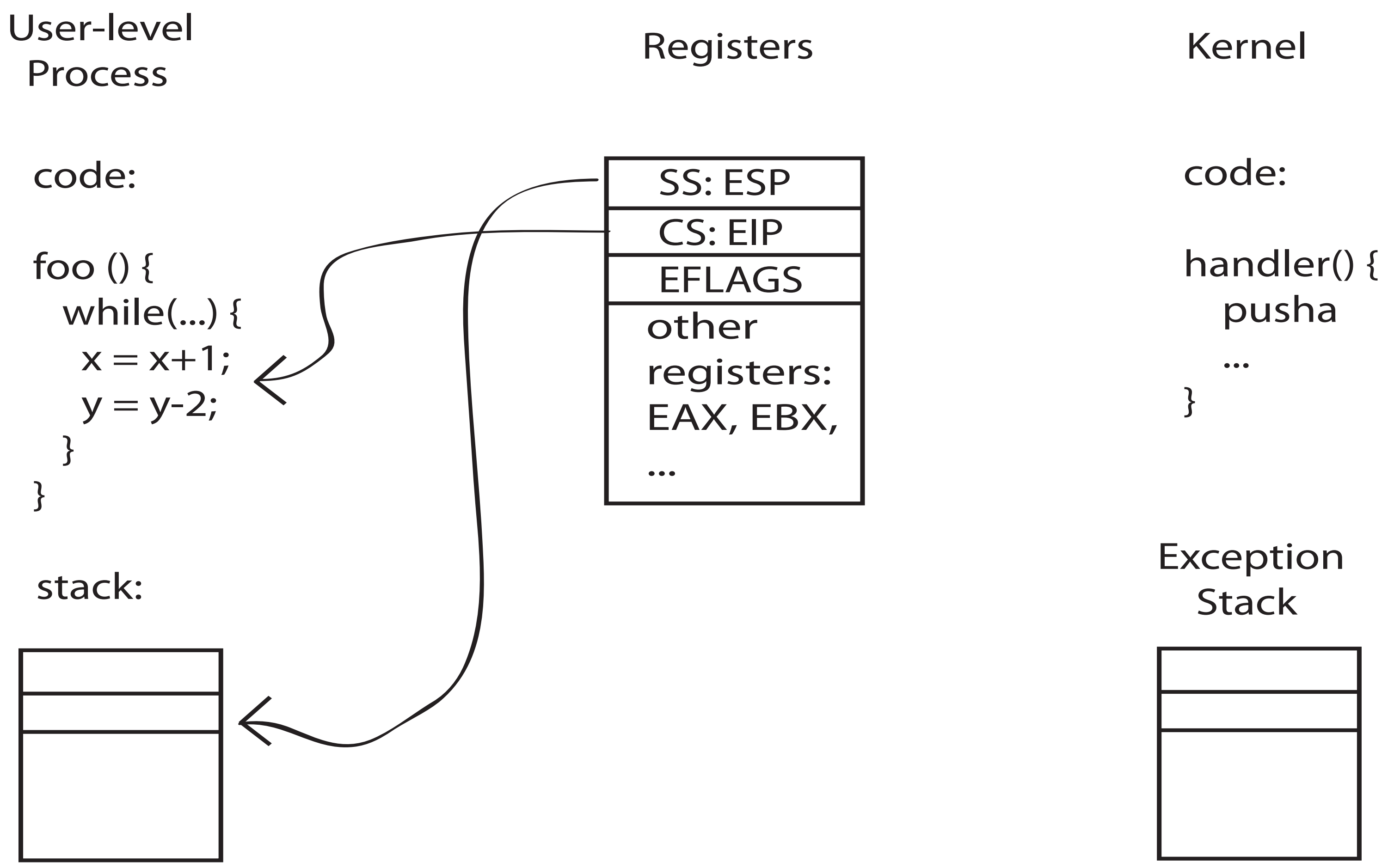
User/Kernel (Privileged) Mode



3 types of U→K 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
  - Internal synchronous event in process triggers context switch
  - e.g., Protection violation (segmentation fault), Divide by zero, …
- All 3 *exceptions* are an UNPROGRAMMED CONTROL TRANSFER
  - Where does it go?

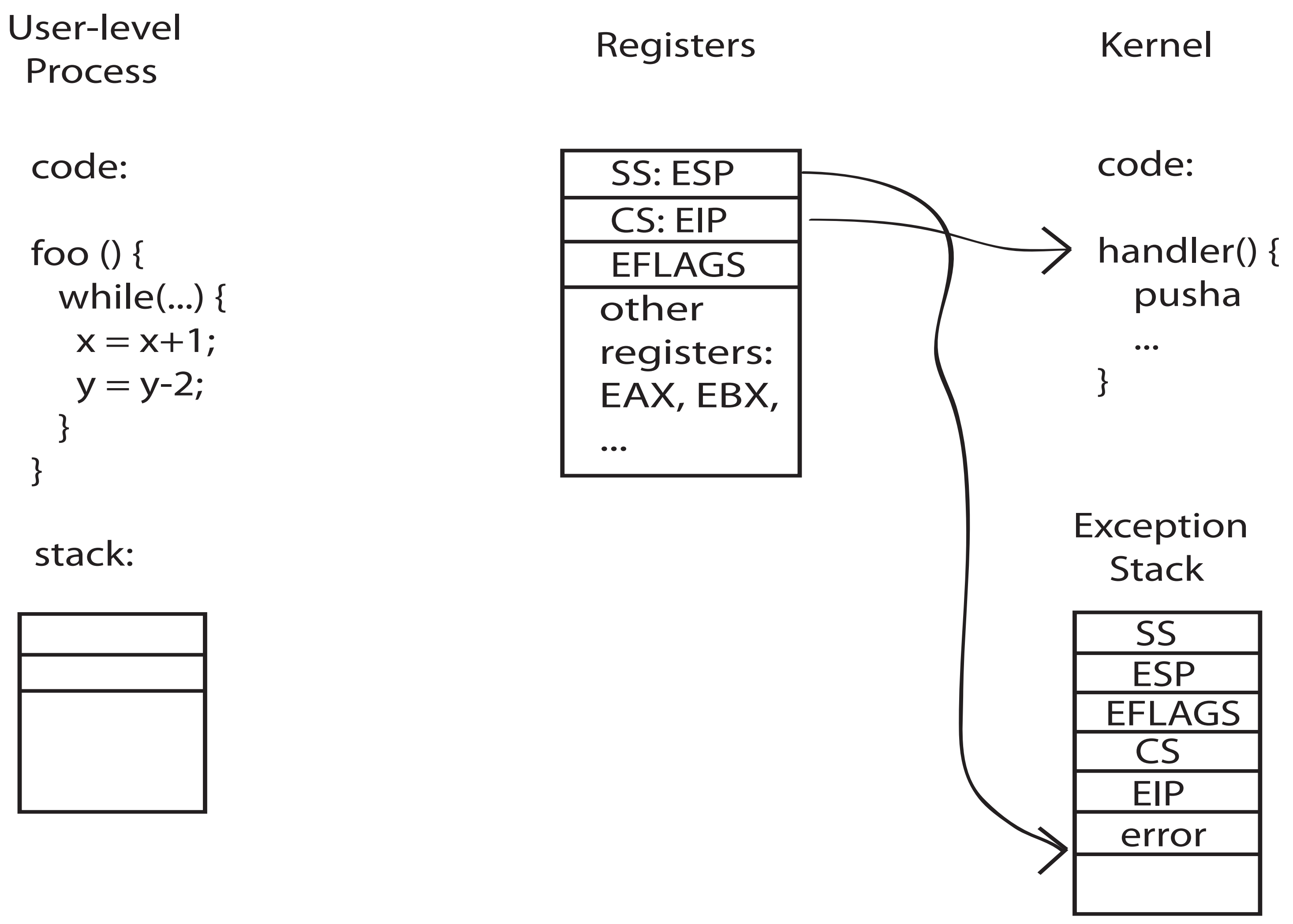
Before Exception (Interrupt / Trap)



```
code:

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

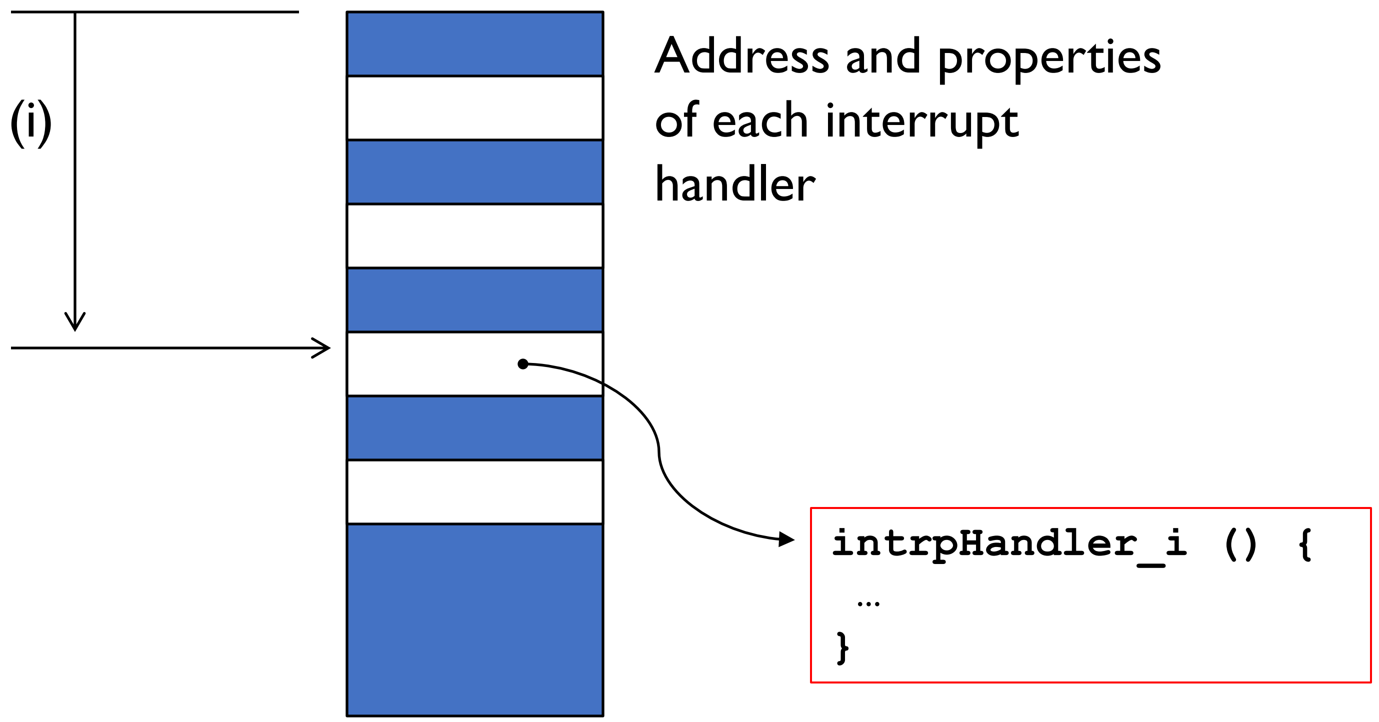
```
code:

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

During Exception



Where do U→K mode transfers go?

- Cannot let user program specify (why?)
- Solution: ***Interrupt Vector***



```
intrpHandler_i () {
  …
}
```

Implementing Safe Kernel Mode Transfers

- *Carefully* constructed kernel code packs up the user process state and sets it aside
- Must handle weird/buggy/malicious user state
  - Syscalls with null pointers
  - Return instruction out of bounds
  - User stack pointer out of bounds
- Should be impossible for buggy or malicious user program to cause the kernel to corrupt itself
- User program should not know interrupt has occurred *(transparency)*

How do K→U transfers go back?

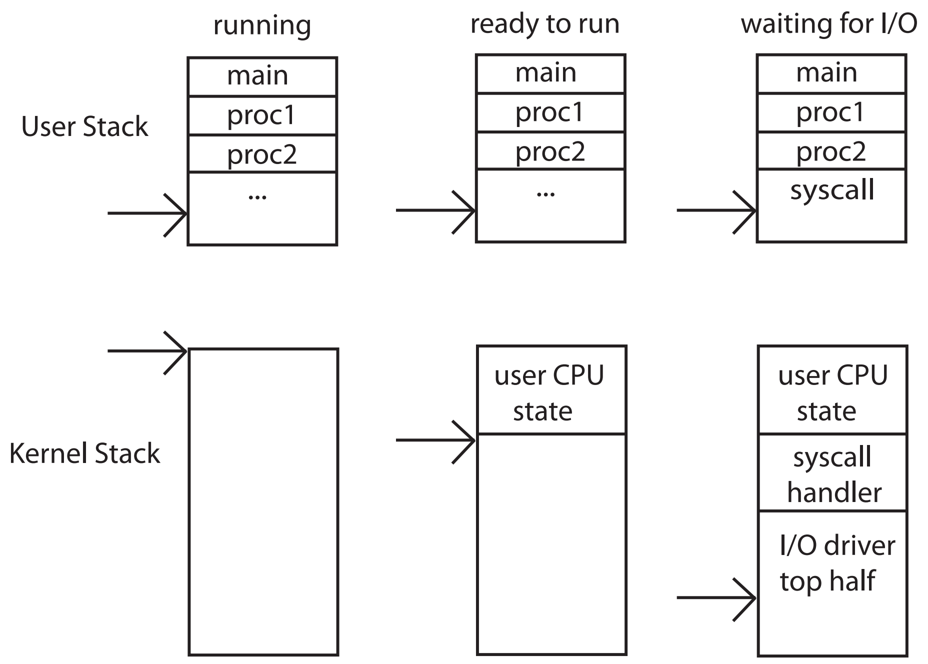
- “Return from interrupt” instruction
- Drops mode from kernel to user priviledge
- Restores user PC (possibly stack)

The Kernel Stack

- Interrupt handlers need a stack
- System call handlers need a stack
- Can't just use the user stack [why?]

The Kernel Stack

- Solution: two-stack model
  - Each OS thread has kernel stack (located in kernel memory) plus user stack (located in user memory)
- Place to save user registers during interrupt



Kernel System Call Handler

- Vector through well-defined syscall entry points!
  - Table mapping system call number to handler
- Locate arguments
  - In registers or on user (!) stack
- Copy arguments
  - From user memory into kernel memory – carefully checking locations!
  - Protect kernel from malicious code evading checks
- Validate arguments
  - Protect kernel from errors in user code
- Copy results back
  - Into user memory – carefully checking locations!

Hardware support: Interrupt Control

- Interrupt processing not visible to the user process:
  - Occurs between instructions, restarted transparently
  - No change to process state
  - What can be observed even with perfect interrupt processing?
- Interrupt Handler invoked with interrupts 'disabled'
  - Re-enabled upon completion
  - Non-blocking (run to completion, no waits)
  - Pack up in a queue and pass off to an OS thread for hard work
    - wake up an existing OS thread

Hardware support: Interrupt Control

- 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 interrupts
  - Mask off (disable) certain interrupts, eg., lower priority
  - Certain Non-Maskable-Interrupts (NMI)
    - e.g., kernel segmentation fault
    - Also: Power about to fail!

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

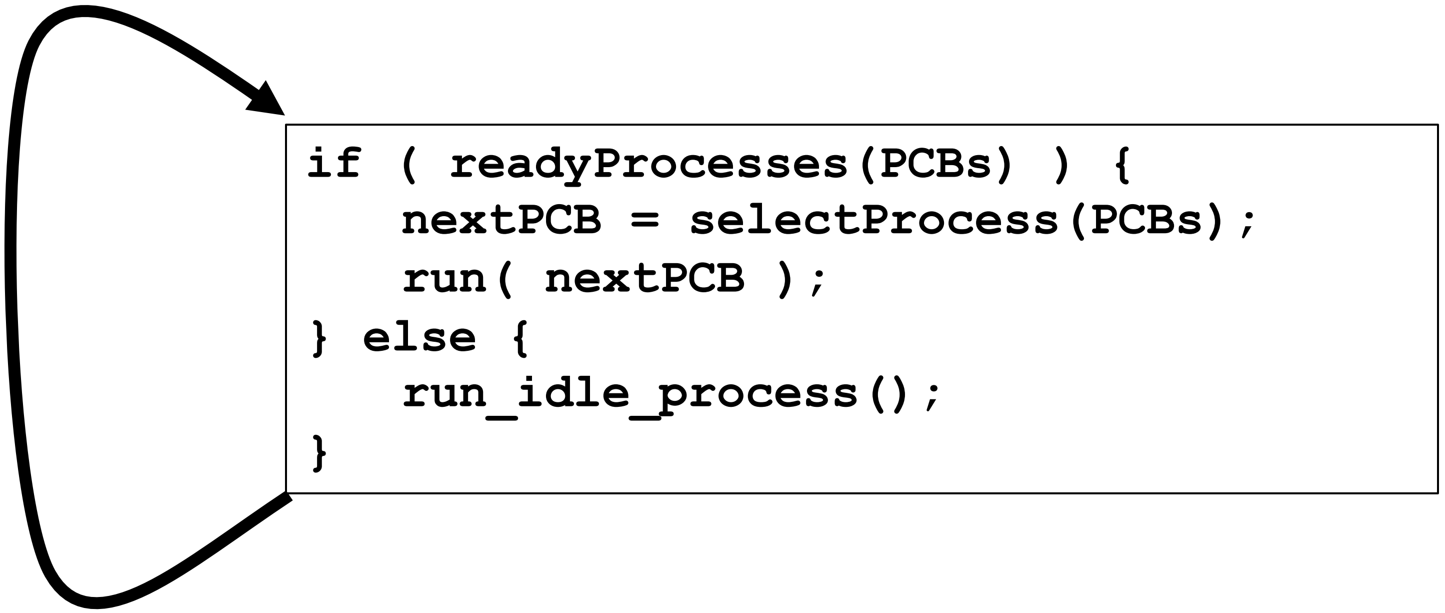
Four Fundamental OS Concepts

- **Thread: Execution Context**
  - Program Counter, Registers, Execution Flags, Stack
- **Address space** (with **translation**)
  - Program's view of memory is distinct from physical machine
- **Process: an instance of a running program**
  - Address Space + One or more Threads
- **Dual mode operation / Protection**
  - Only the "system" can access certain resources
  - Combined with translation, isolates programs from each other

Process Control Block

- Kernel representation of each process
  - Status (running, ready, blocked)
  - Register state (if not running)
  - Thread control block(s)
  - Process ID
  - Execution time
  - Memory translations
- **Scheduler** maintains a data structure of PCBs
  - Or threads
- *Scheduling algorithm:* Which thread/process should the OS run next?

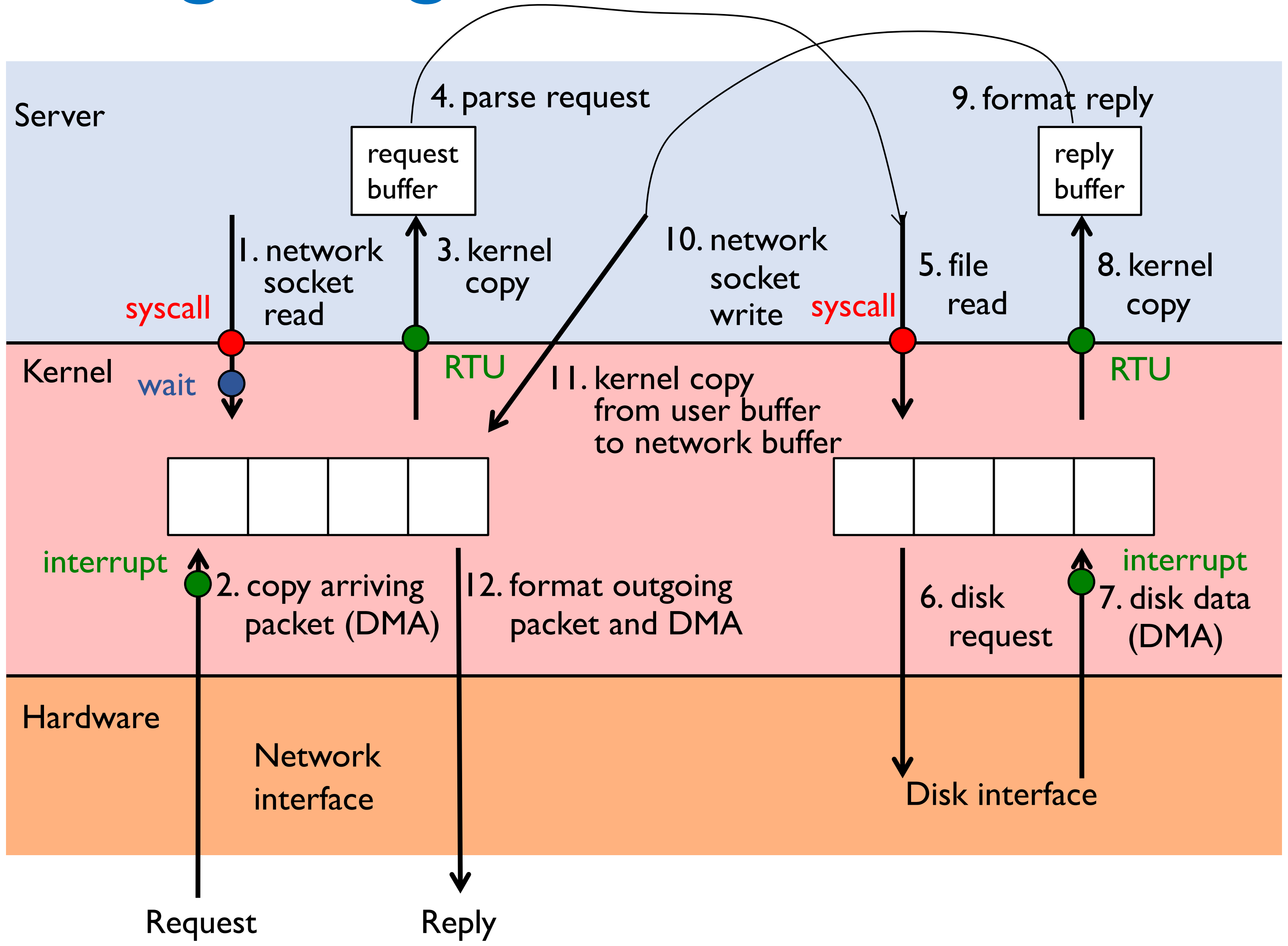
Scheduler



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

- Scheduling: Mechanism for deciding which processes/threads receive the CPU
- Lots of different scheduling policies provide …
    - Fairness or
    - Realtime guarantees or
    - Latency optimization or ..

Putting it together: web server



Conclusion: Four OS Concepts

- **Thread: Execution Context**
  - Program Counter, Registers, Execution Flags, Stack
- **Address space** (with **translation**)
  - Program's view of memory is distinct from physical machine
- **Process: an instance of a running program**
  - Address Space + One or more Threads
- **Dual mode operation / Protection**
  - Only the "system" can access certain resources
  - Combined with translation, isolates programs from each other

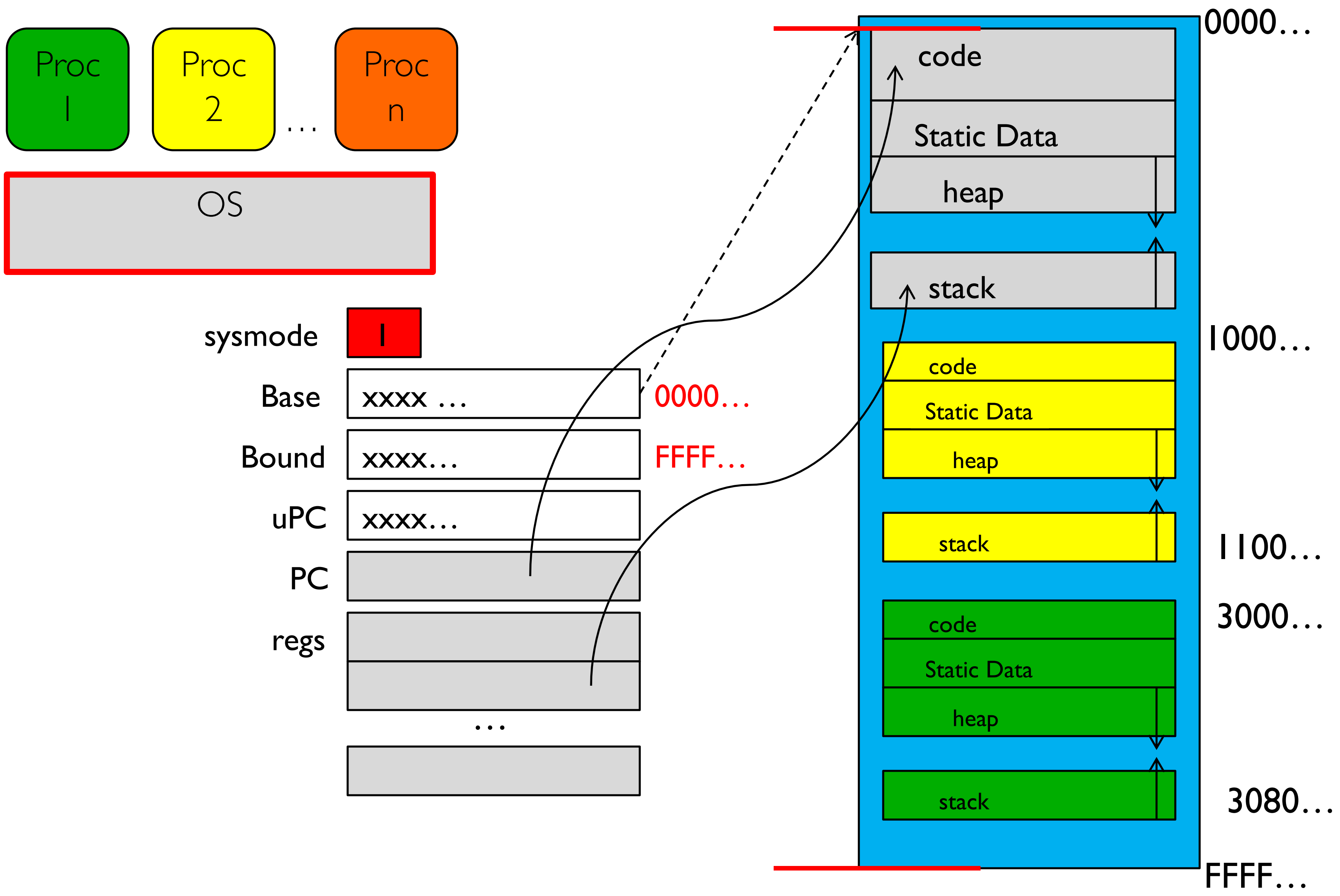
Additional Material

- Explore on your own

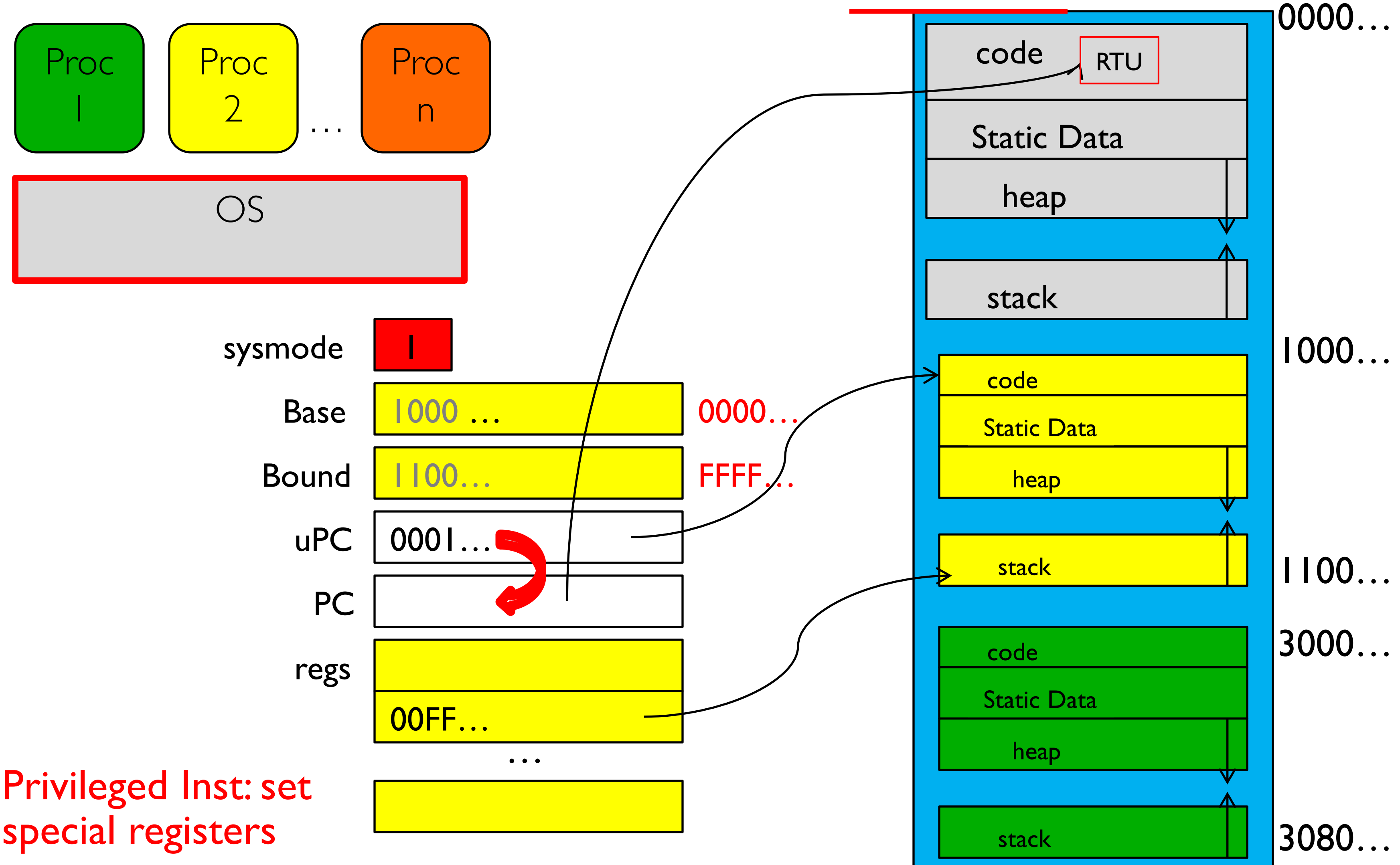
Putting It Together: Mode Transfer & Translation

- Mode transfer should change address translation mapping
- Examples:
  - Ignore base and bound in kernel mode
  - Page tables with "kernel mode only" bits

Base & Bound: OS Loads Process

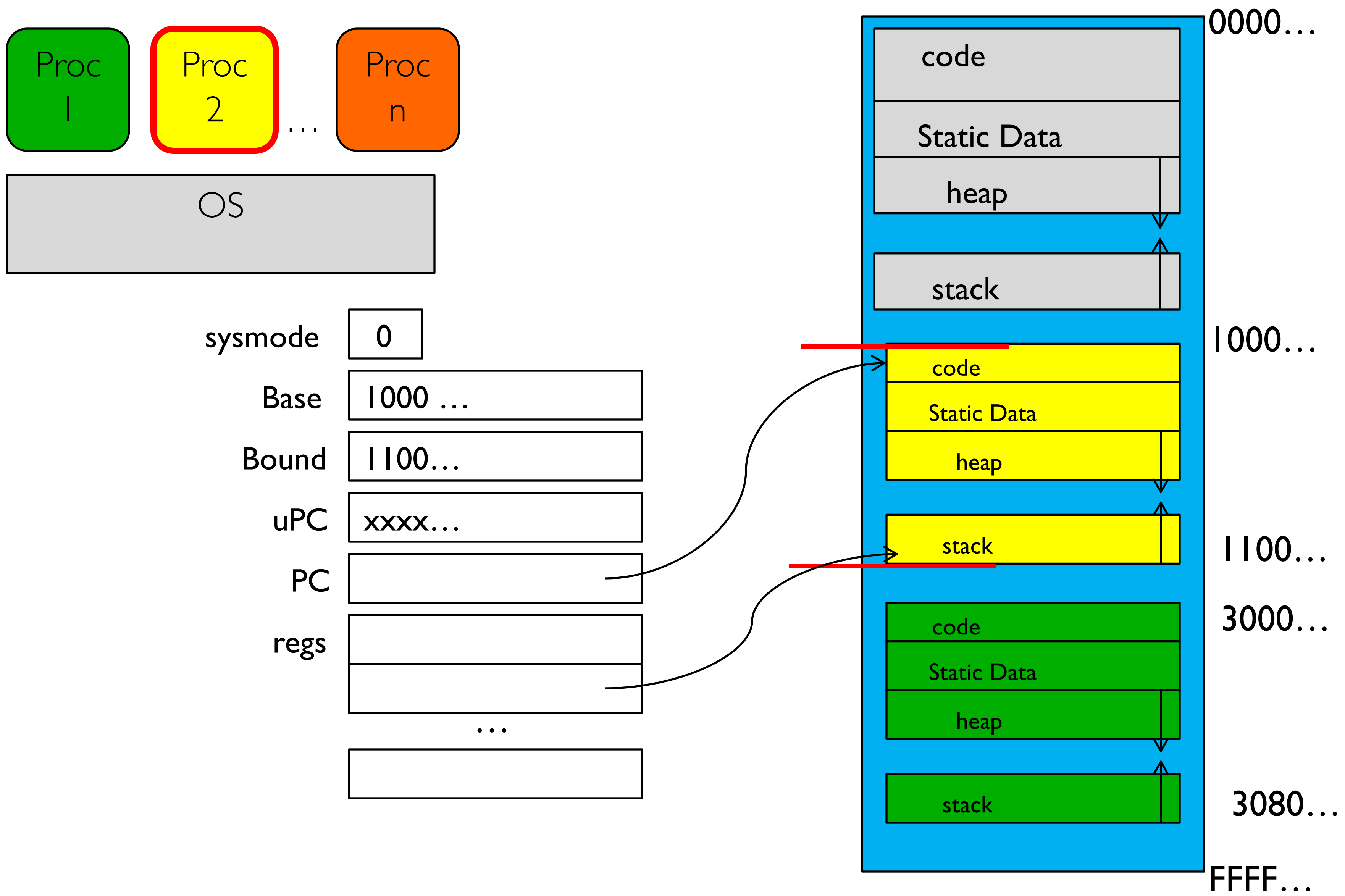


Base & Bound: About to Switch

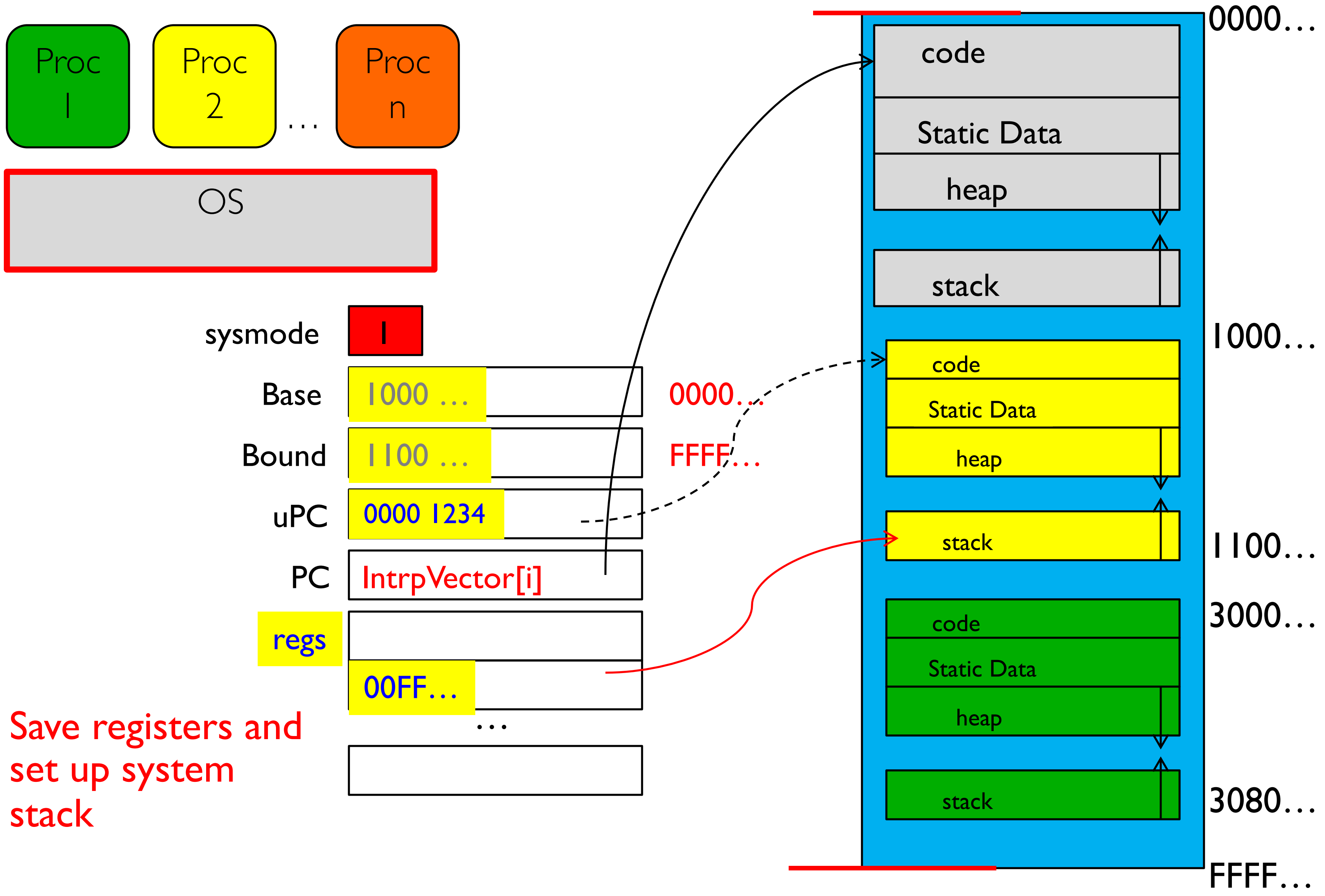


- Privileged Inst: set special registers
- RTU

Simple B&B: User Code Running



Base & Bound: Handling Interrupt

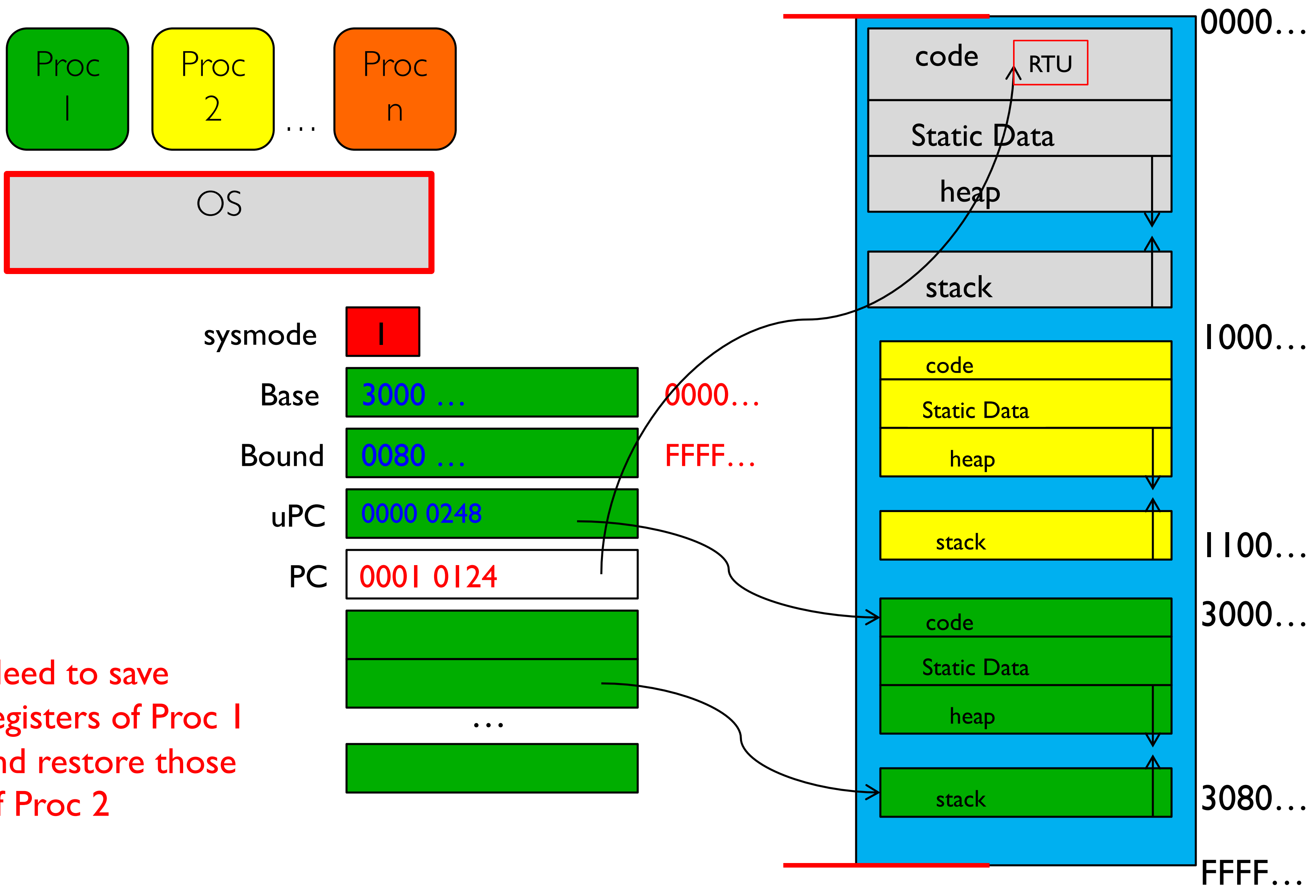


- Save registers and set up system stack

How do we switch between processes?

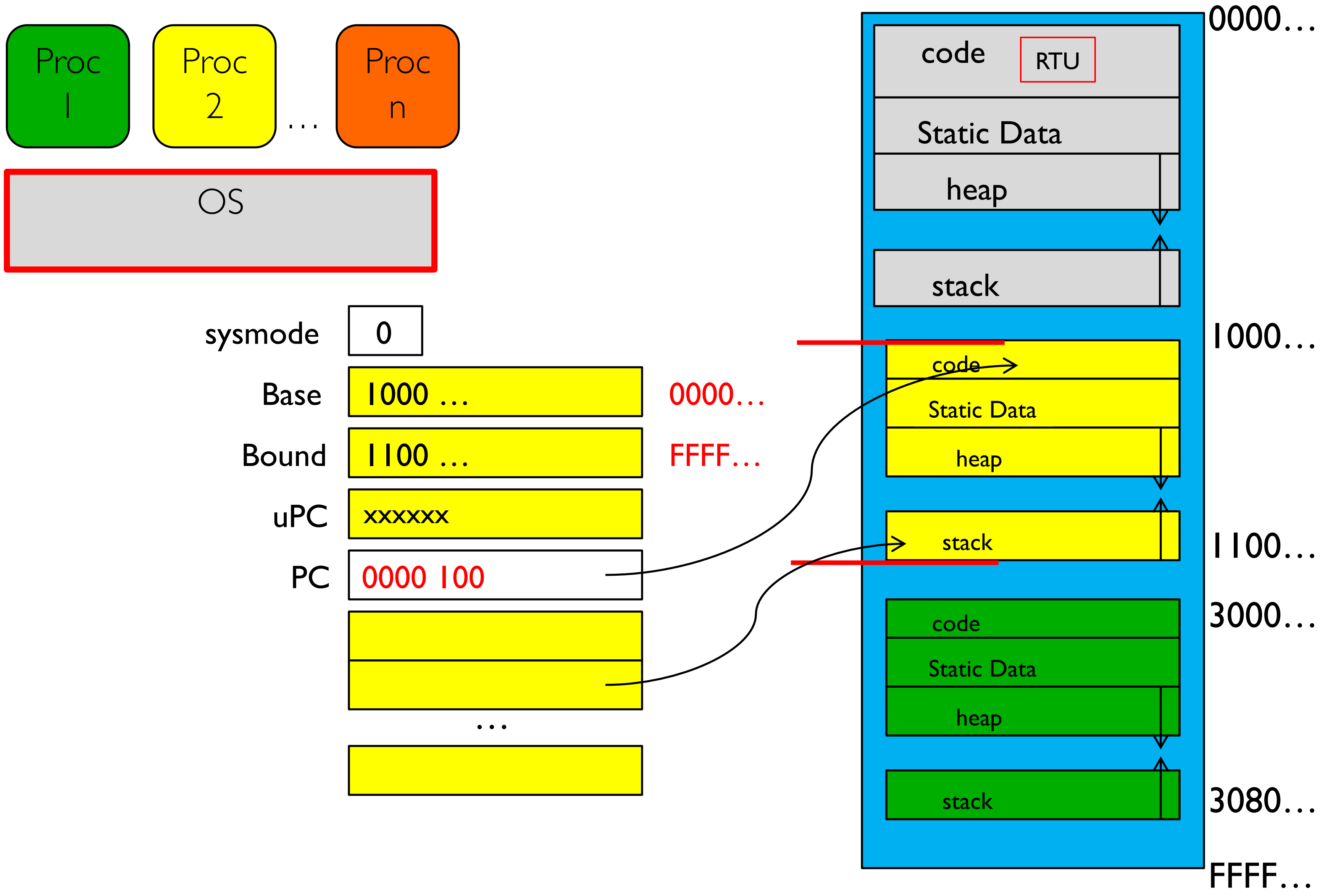
- We already have all the necessary machinery!
- Just requires two mode transfers

Base & Bound: Switch User Process

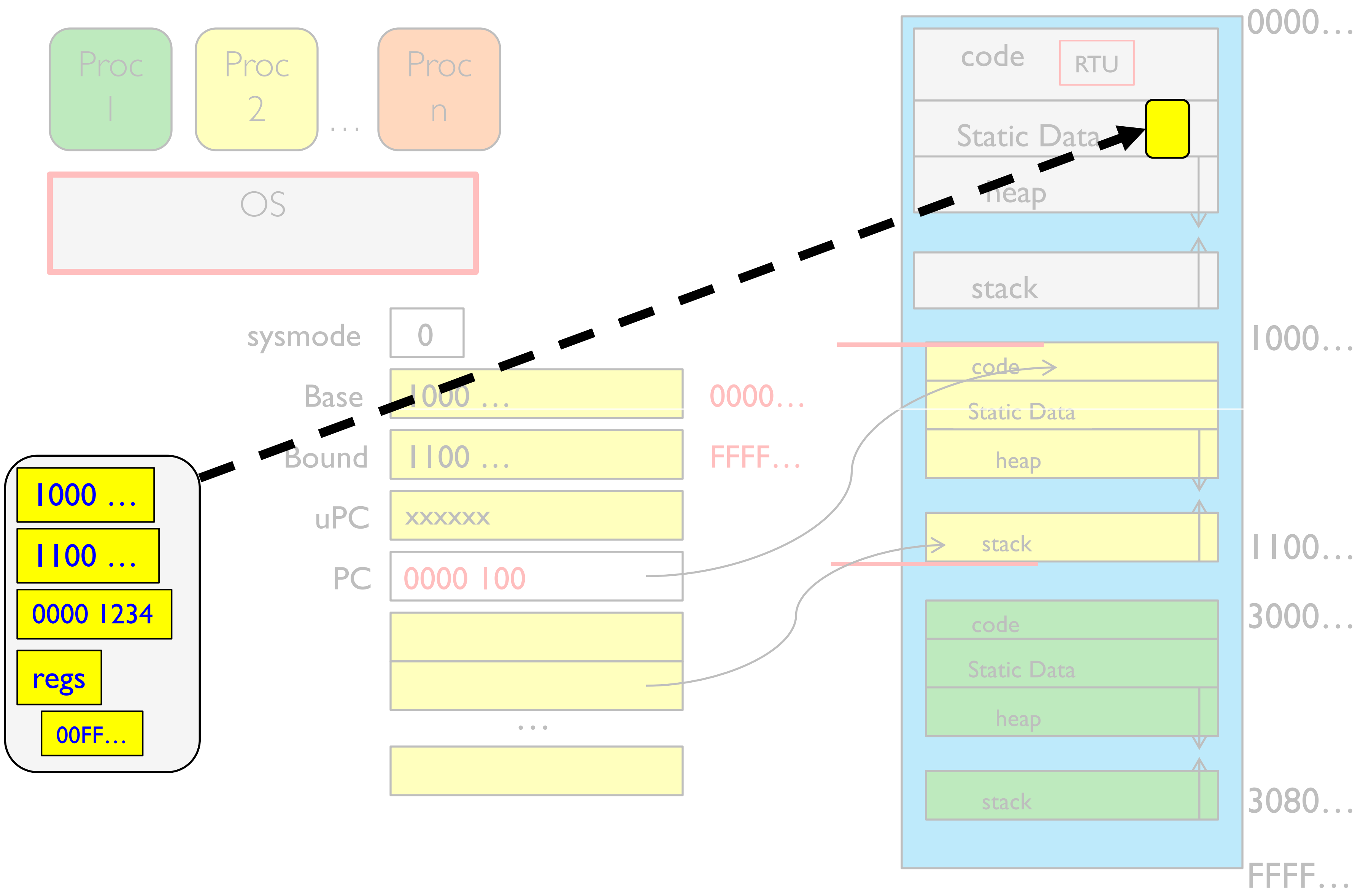


Need to save registers of Proc 1 and restore those of Proc 2

Base & Bound: Switch User Process



Representing Processes: PCB


Proc
1
Proc
2
…
Proc
n
OS
sysmode
0
Base
1000 …
0000…
Bound
1100 …
FFFF…
uPC
xxxxxx
PC
0000 100
…
1000 …
1100 …
0000 1234
regs
00FF…
code
RTU
Static Data
heap
stack
code
Static Data
heap
stack
code
Static Data
heap
stack
0000…
1000…
1100…
3000…
3080…
FFFF…