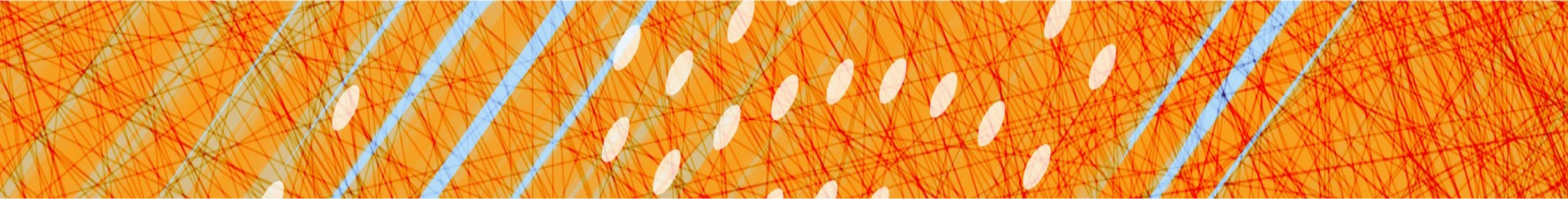


Short History of Operating Systems

CS 4410

Operating Systems

Cornell CIS

COMPUTING AND INFORMATION SCIENCE

[R. Agarwal, L. Alvisi, A. Bracy, M. George,
F. B. Schneider, E. G. Sirer, R. Van Renesse]

Find Study Partners!

Studying with peers is a great way to connect with other Cornell students and is a powerful tool for learning.

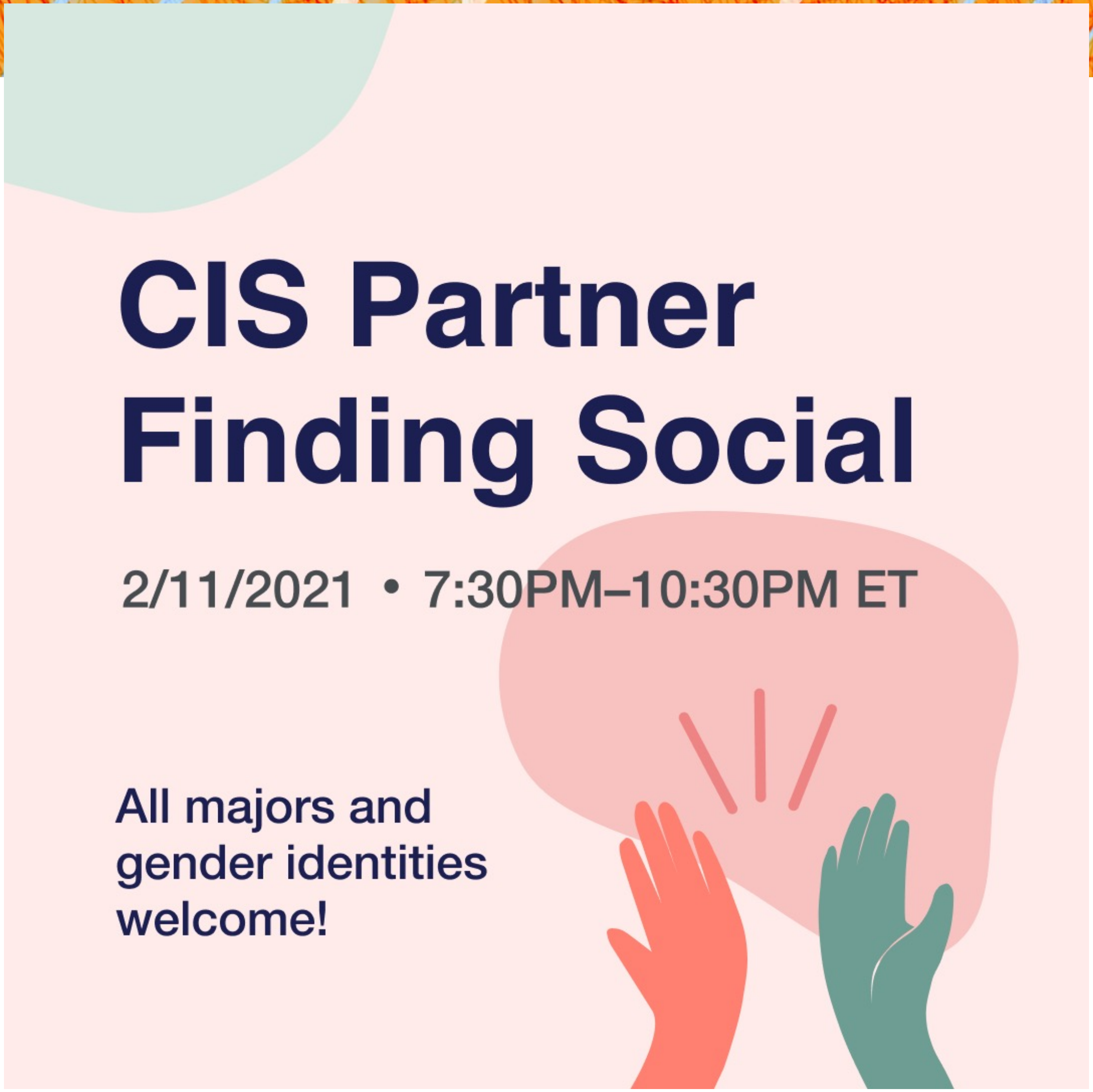
Cornell's Learning Strategies Center (LSC) helps match you with study partners.

To learn more, visit the LSC's Studying Together webpage or scan the code →

Scan the QR code to find out more about Study Partners

or visit http://lsc.cornell.edu/studying-together/

lsc.cornell.edu



CIS Partner Finding Social

2/11/2021 • 7:30PM–10:30PM ET

All majors and
gender identities
welcome!

https://www.cs.cornell.edu/information/news/newsitem11476/cis-partner-finding-social

Short History of Operating Systems

CS 4410

Operating Systems

Cornell CIS

COMPUTING AND INFORMATION SCIENCE

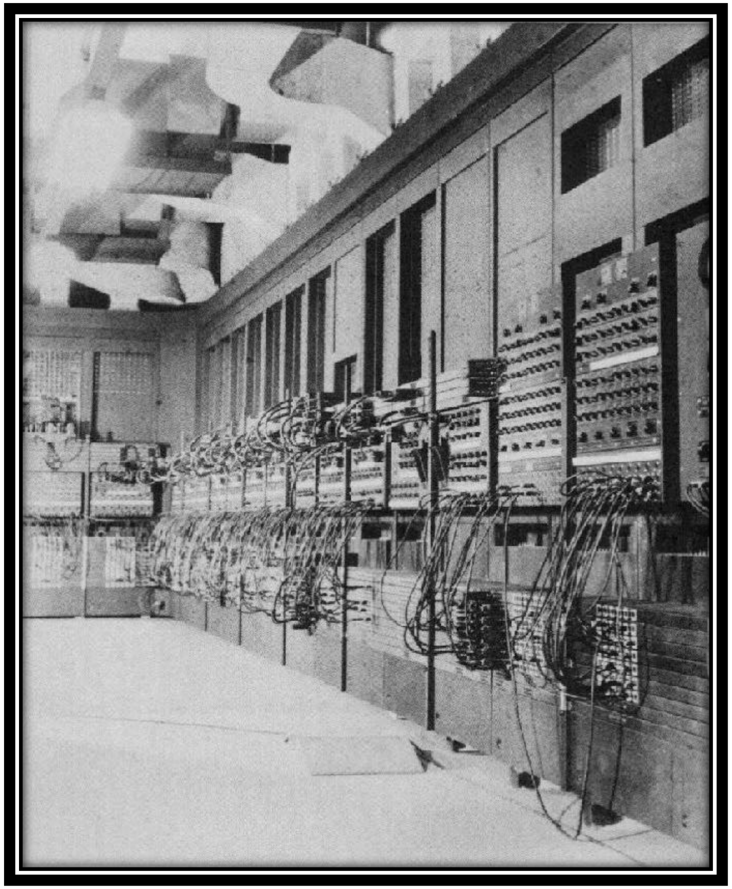
[R. Agarwal, L. Alvisi, A. Bracy, M. George,
F. B. Schneider, E. G. Sirer, R. Van Renesse]

PHASE 1 (1945 – 1975)

COMPUTERS EXPENSIVE, HUMANS CHEAP

Early Era (1945 – 1955):

- First computer: ENIAC
  - UPenn, 30 tons
  - Vacuum tubes
  - card reader/puncher
  - 100-word memory added in 1953
- Single User Systems
  - one app, then reboot
- “O.S” = loader + libraries
- Problem: Low utilization

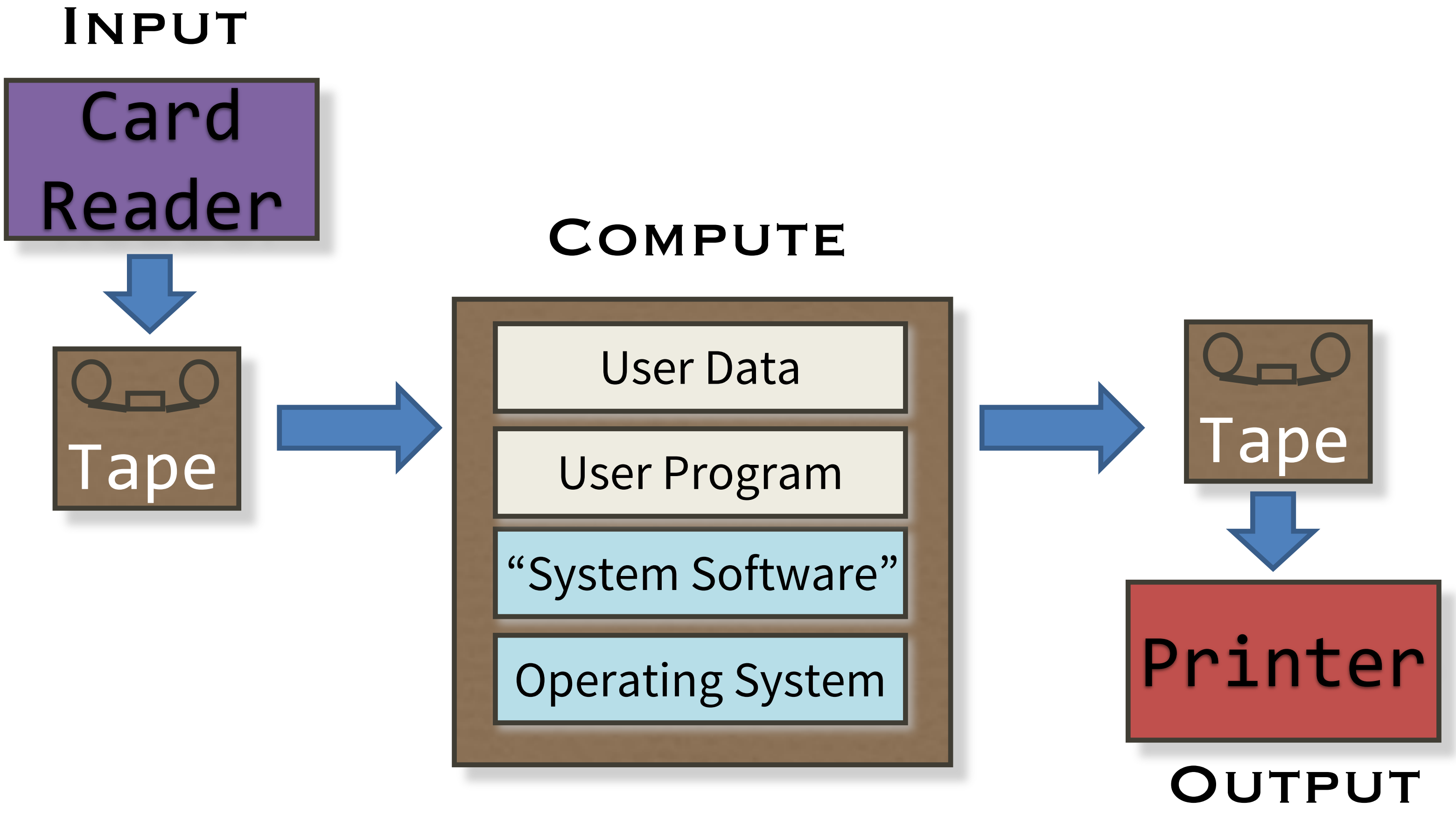


Batch Processing (1955 – 1960):

- First Operating System: GM-NAA-I/O
  - General Motors research division
  - North American Aviation
  - Input/Output
- Written for IBM 704 computer
  - 10 tons
  - Transistors
  - 4K word memory (about 18 Kbyte)

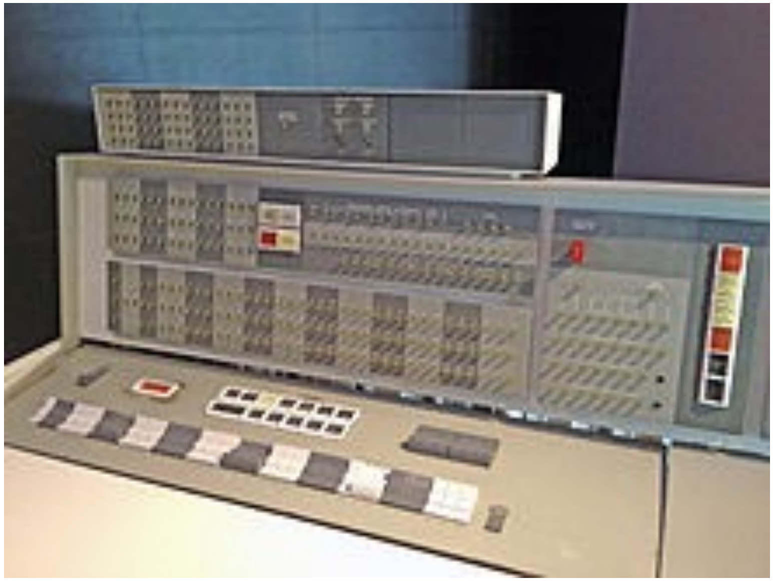
Batch Processing

- O.S = loader + libraries + sequencer
- Problem: CPU unused during I/O



Time-Sharing (1960 –):

- Multiplex CPU
- CTSS first time-sharing O.S.
  - Compatible Time-Sharing System
  - MIT Computation Center
  - predecessor of all modern O.S.'s
- IBM 7090 computer
- 32K word memory



Time-Sharing + Security (1965 –):

- Multics (MIT)
  - security rings
- GE-645 computer
  - hw-protected virtual memory
- Multics predecessor of
  - Unix (1970)
  - Linux (1990)
  - Android (2008)

PHASE 2 (1975 – TODAY)

COMPUTERS CHEAP, HUMANS EXPENSIVE

Personal Computers (1975 –):

- 1975: IBM 5100 first “portable” computer
  - 55 pounds…
  - ICs

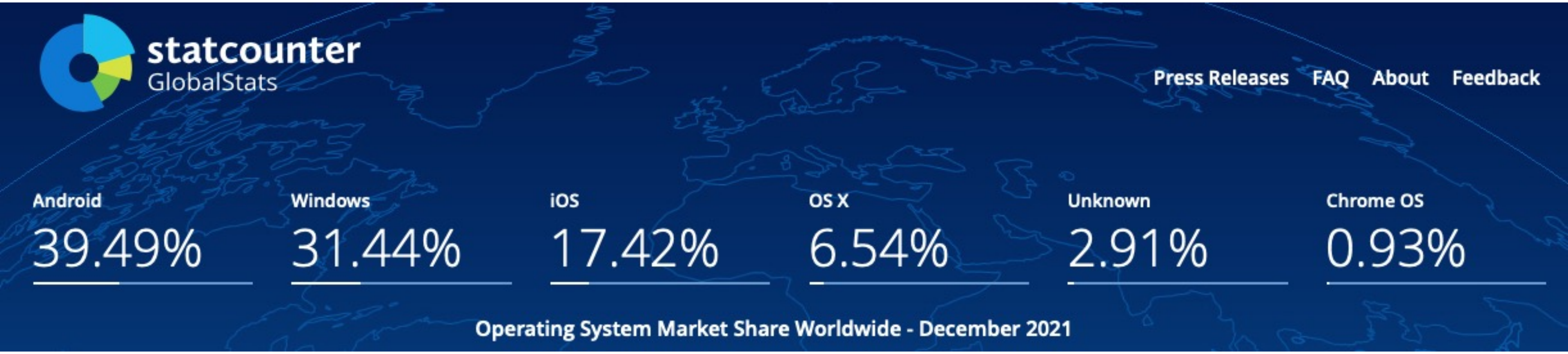
- 1977: RadioShack/Tandy TRS-80
  - first “home” desktop

- 1981: Osborne 1 first “laptop”
  - 24.5 pounds, 5’’ display

Modern Era (1990 –)

- Ubiquitous Computing / Internet-of-Things
  - – Mark Weiser, 1988-ish
- Personal Computing
  - – PDA (“PalmPilot”) introduced in 1992
  - – #computers / human >> 1
- Cloud Computing
  - – Amazon EC2, 2006

Today's "winners" (by market share)



Android	Windows	iOS	OS X	Unknown	Chrome OS
39.49%	31.44%	17.42%	6.54%	2.91%	0.93%

Operating System Market Share Worldwide - December 2021

- Google Android (2006, based on Linux)
  - Android phones
- Microsoft Windows NT (1993)
  - PC desktops, laptops, and servers
- Apple iOS (2007)
  - iPhones, iPads, …
- Apple Mac OS X (2001)
  - Apple Mac desktops and laptops
- Linux (1990)
  - Servers, laptops, IoT

Architectural Support for Operating Systems

(Chapter 2)

CS 4410

Operating Systems

[R. Agarwal, L. Alvisi, A. Bracy, M. George, E. Sirer, R. Van Renesse]

Outline

1. Support for Processes
2. Support for Devices
3. Booting an O.S.

SUPPORT FOR PROCESSES

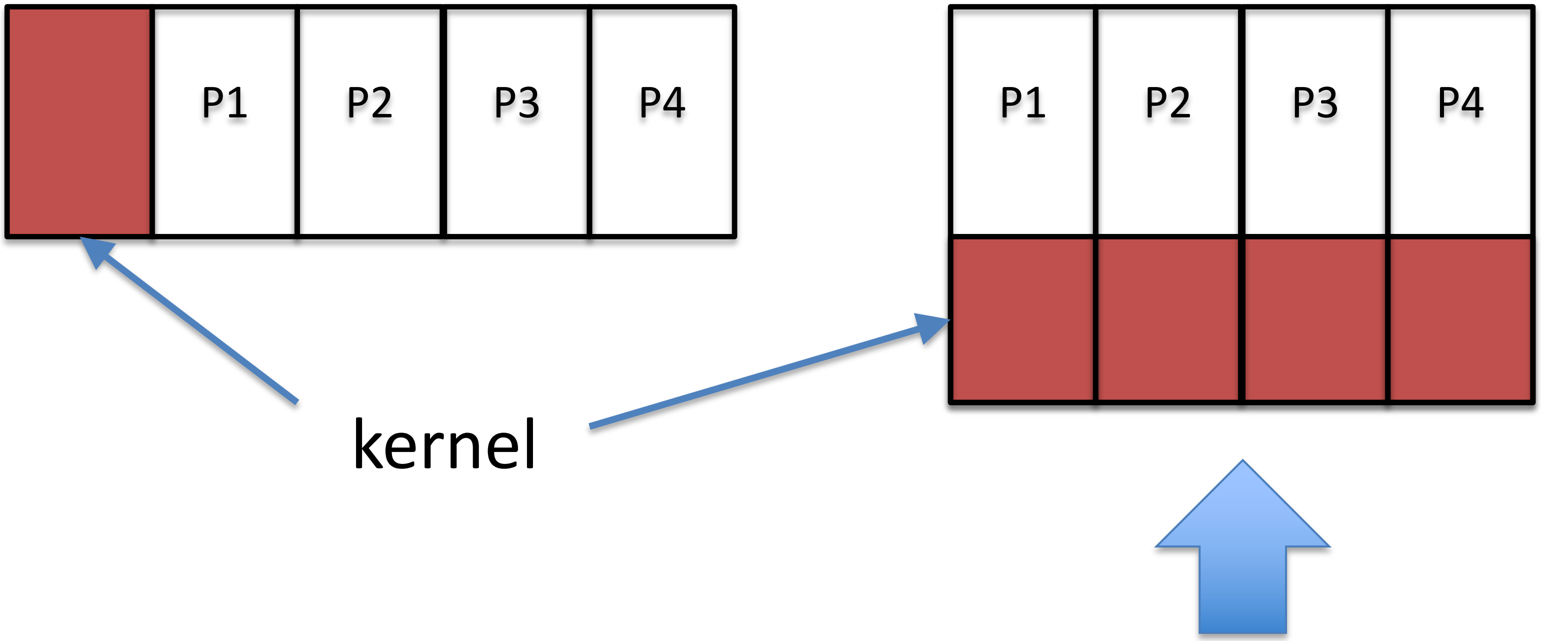
Hardware Support for Processes: *supervisor mode*

- One primary objective of an O.S. *kernel* is to manage and isolate multiple processes
  - Kernel runs in *supervisor mode (aka kernel mode)*
    - unrestricted access to all hardware
  - Processes run in *user mode*
    - restricted access to memory, devices, certain machine instructions, …
    - *other instructions run directly on the CPU*
      - no performance penalty
  - Kernel maintains a *Process Control Block* (PCB) for each process
    - holds page table and more

Two architectures of O.S. kernels

“kernel is a special process”

“kernel is a library”



kernel

most modern O.S.’s
(Linux, Windows, Mac OS X, …)

Comparison

Kernel is a process	Kernel is a library
Kernel has one interrupt stack. Each process has a user stack	Each process has a user stack and an interrupt stack (part of Process Control Block)
Kernel implemented using "event-based" programming (programmer saves/restores context explicitly)	Kernel implemented using "thread-based programming" (context handled by language run-time through "blocking")
Kernel has to translate between virtual and physical addresses when accessing user memory	Kernel can access user memory directly (through page table)

Which architecture do you like better? Why do you think most modern O.S.'s use the "kernel is a library" architecture?

How does the kernel get control?

- Boot (reset, power cycle, …)
  - kernel initializes devices, etc.
- Signals
  - user mode → supervisor mode

there is no “main loop”

(again: kernel more like a library
than a process)

Types of Signals

Exceptions (aka Faults)

- Synchronous / Non-maskable
- Process missteps (*e.g.*, div-by-zero)
- Privileged instructions

System Calls

- Synchronous / Non-maskable
- User program requests OS service

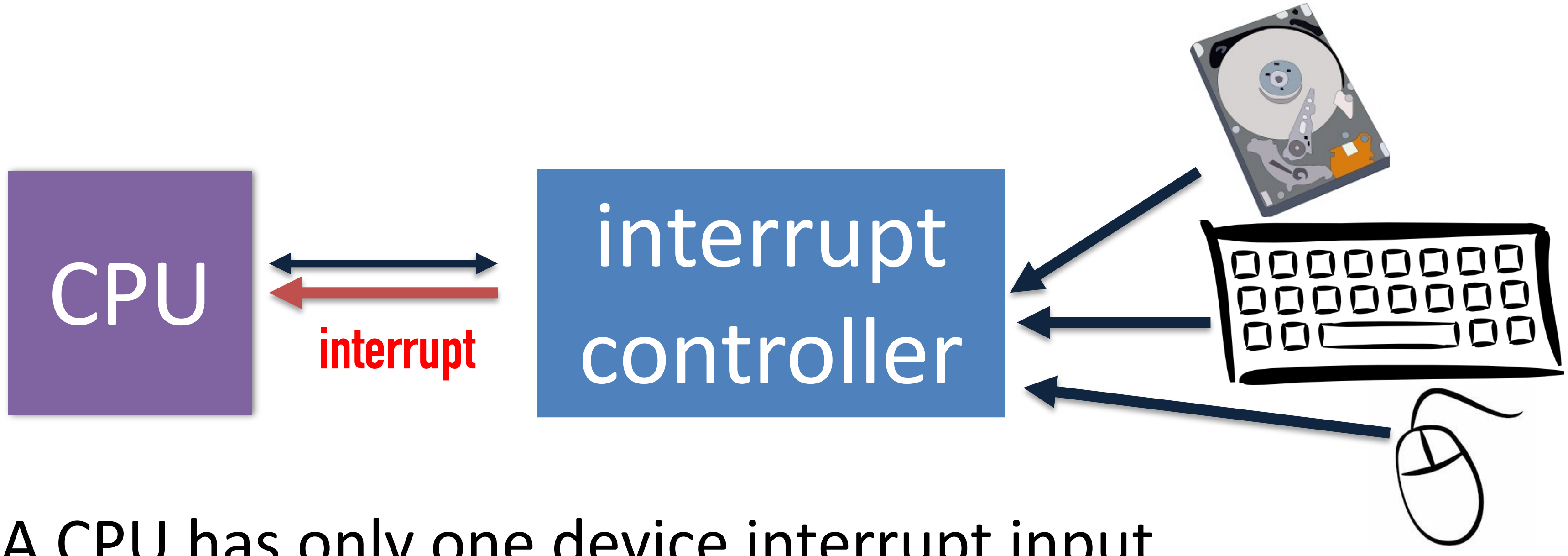
(Device or I/O) Interrupts

- Asynchronous / Maskable
- HW device requires OS service
  - timer, I/O device, inter-processor, …

Nomenclature warning

the term “interrupt” is often used synonymously with “signal”

H/W Interrupt Management



- A CPU has only one device interrupt input
- An *Interrupt Controller* manages interrupts from multiple devices:
  - Interrupts have descriptor of interrupting device
  - Priority selector circuit examines all interrupting devices, reports highest priority level to the CPU

Interrupt Handling

- Two objectives:
  1. handle the interrupt and remove its cause
  2. restore what was running before the interrupt
     - state may have been modified on purpose
- Two “actors” in handling the interrupt:
  1. the hardware goes first
  2. the kernel code takes control in *interrupt handler*

Interrupt Handling (conceptually)

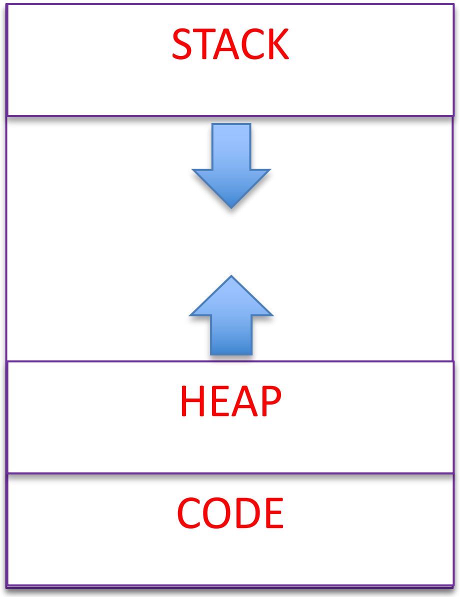
- There is a supervisor SP and a user SP
  - both called SP
  - determined by “supervisor mode bit”

WHY??
(next page)

- On signal, hardware:
  1. disables (“masks”) device interrupts
     - at least interrupts of the same device
  2. sets supervisor mode (if not set already)
  3. pushes PC (IP), SP, and PSW from before interrupt
  4. sets PC to “signal handler”
     - depends on signal type
     - signal handlers specified in “interrupt vector” initialized during boot:

Interrupt Vector
I/O interrupt handler
system call handler
page fault handler
…

Reasons for separating user SP / supervisor SP



- user SP may be illegal
  - badly aligned or pointing to unwritable memory
- user stack may be not be large enough and cause important data to be overwritten
  - remember: stack grows down, heap grows up
- user may use SP for other things than stack
- security risks if only one SP:
  - kernel could push sensitive data on user stack and unwittingly leave it there (pop does not erase memory)
  - process could corrupt kernel code or data by pointing SP to kernel address

Interrupt Handling, cont’d

PSW (Processor Status Word):

supervisor mode bit	interrupts enabled bit	condition codes

“return from interrupt” instruction:

- hardware pops PC, SP, and PSW
- depending on contents of PSW
  - switch to user mode
  - Re-enable interrupts
- partly privileged: process cannot switch to supervisor mode or disable interrupts this way
  - **WHY??**
  - **How can a process intentionally switch to supervisor mode?**

Interrupt Handling: software

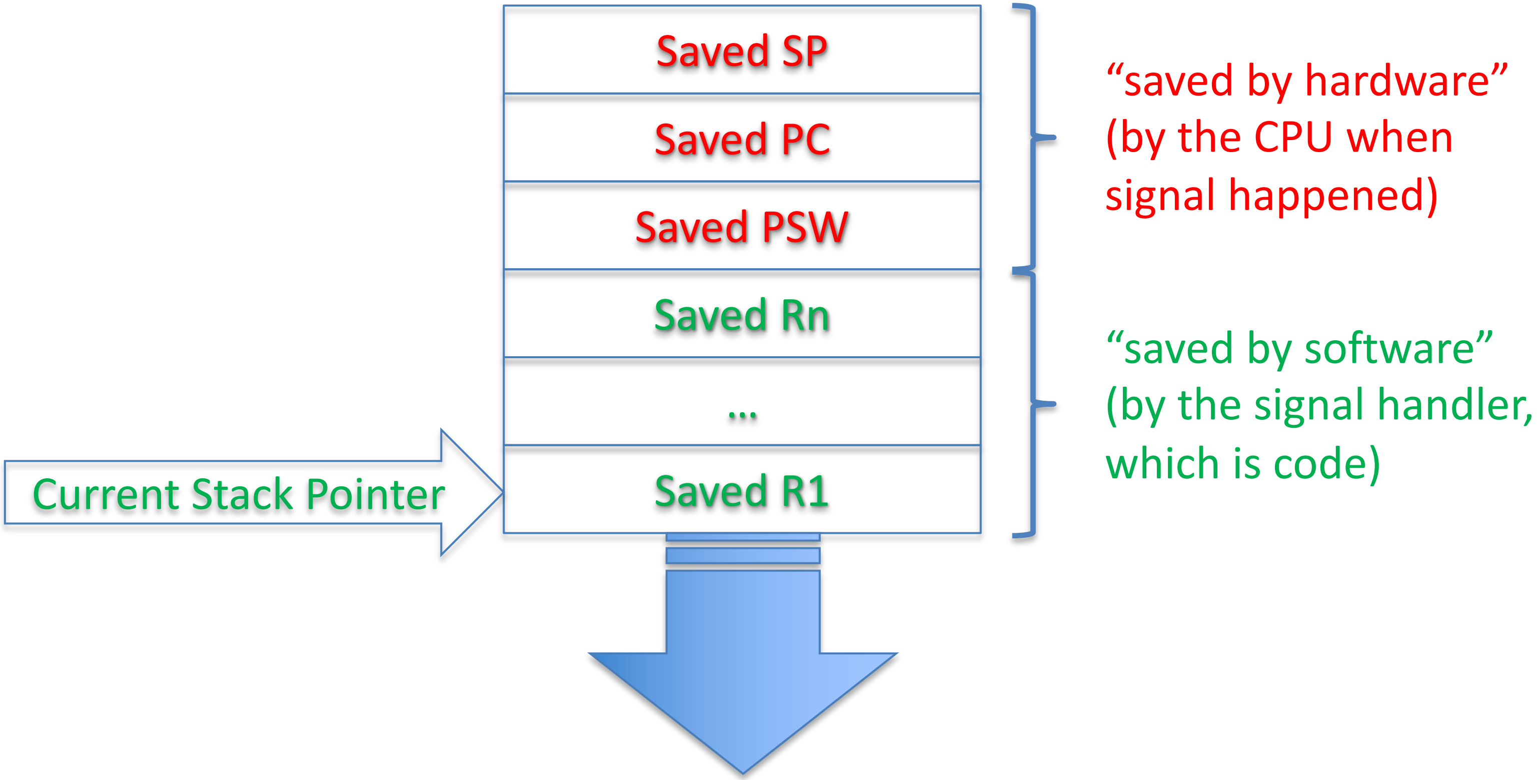
- Interrupt handler first pushes the registers onto the interrupt stack of the currently running process (part of PCB)
  - Why does it save the registers?
  - Why doesn’t the hardware do that?

answers on next page

Saving Registers

- On interrupt, the kernel needs to save the registers as the kernel code needs to use the registers to handle the interrupt
- Saving/restoring registers is expensive. Not all registers need be saved: the kernel uses only a subset, and most functions will already save and restore the registers that they use

Interrupt Stack



“saved by hardware” (by the CPU when signal happened)

“saved by software” (by the signal handler, which is code)

Typical Interrupt Handler Code

```
HandleInterruptX:
    PUSH %Rn
    …                  only need to save registers not
    PUSH %R1           saved by C functions
    CALL __handleX            // call C function handleX()
    POP %R1
    …                  restore the registers saved above
    POP %Rn
    RETURN_FROM_INTERRUPT
```

Example Clock Interrupt Handler in C

```
#define CLK_DEV_REG    0xFFFE0300

void handleClockInterrupt( ){
    int *cdr = (int *) CLK_DEV_REG;
    *cdr = 1;        // turn off clock interrupt
    scheduler()      // run another process?
}
```

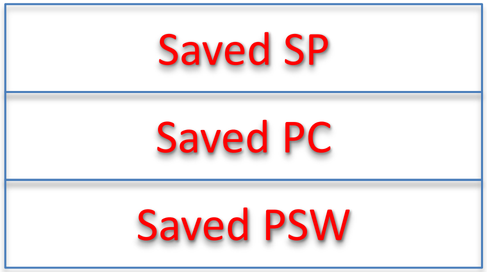
Example System Call Handler in C

```
struct pcb *current_process;

int handle_syscall(int type){
    switch (type) {
    case GETPID: return current_process->pid;
    …
    }
}
```

How Kernel Starts a New Process

1. allocate and initialize a PCB
2. set up initial page table
3. push process arguments onto user stack
4. *simulate an interrupt*
   - push initial PC, user SP
   - push PSW
     - with supervisor mode off and interrupts enabled
5. clear all other registers
6. return-from-interrupt



Saved SP

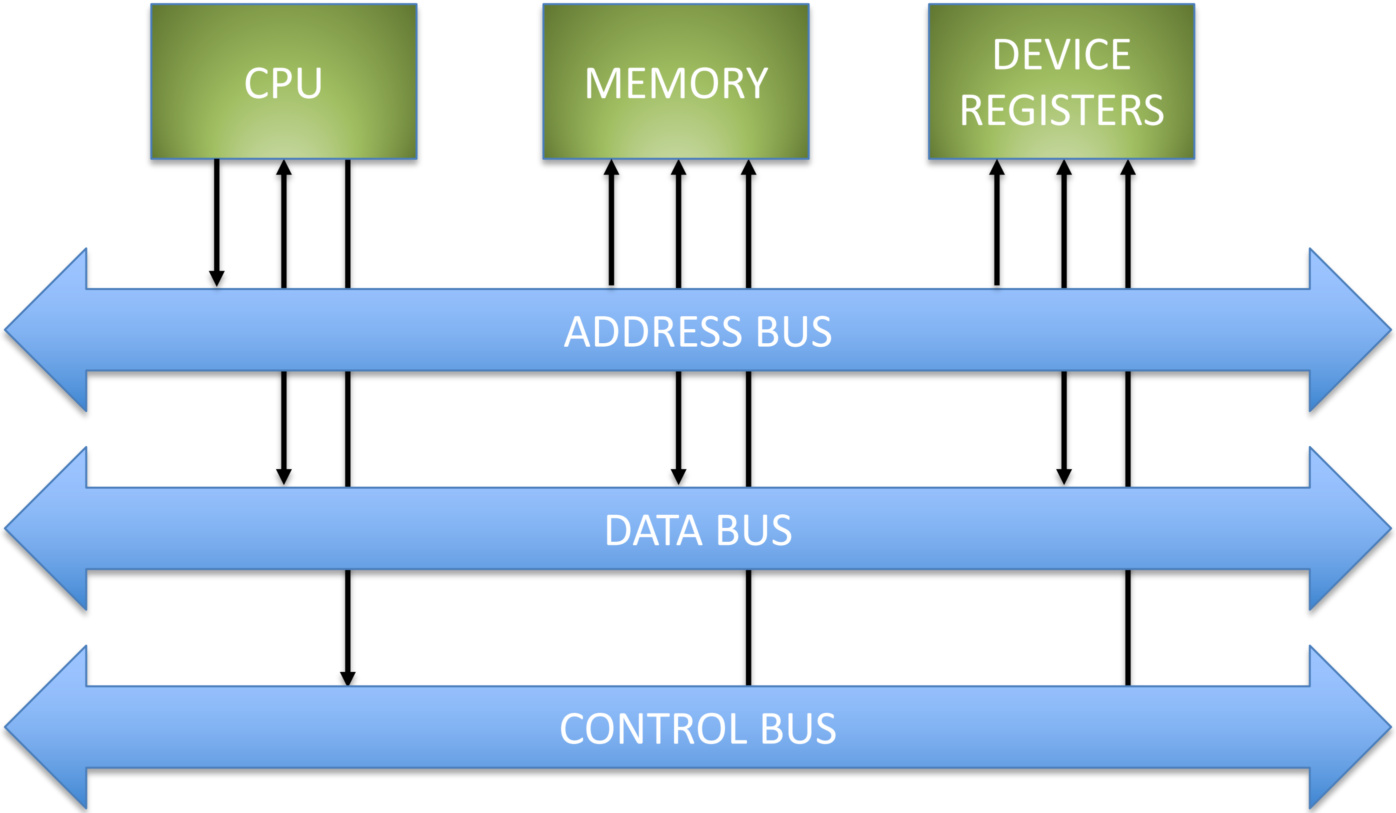
Saved PC

Saved PSW

新年快乐

ANATOMY OF A COMPUTER (SIMPLIFIED)

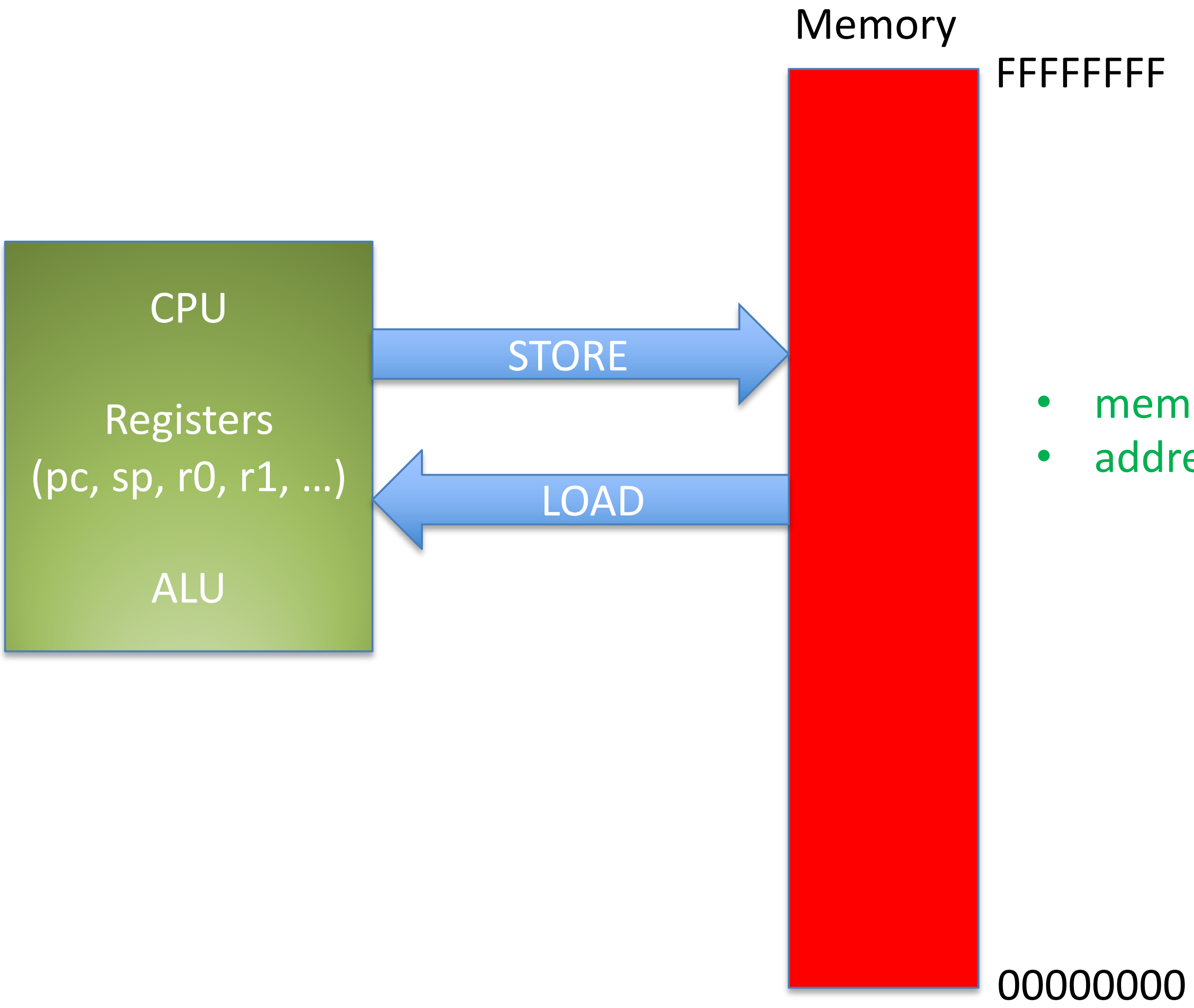
Architecture Diagram



“Bus”

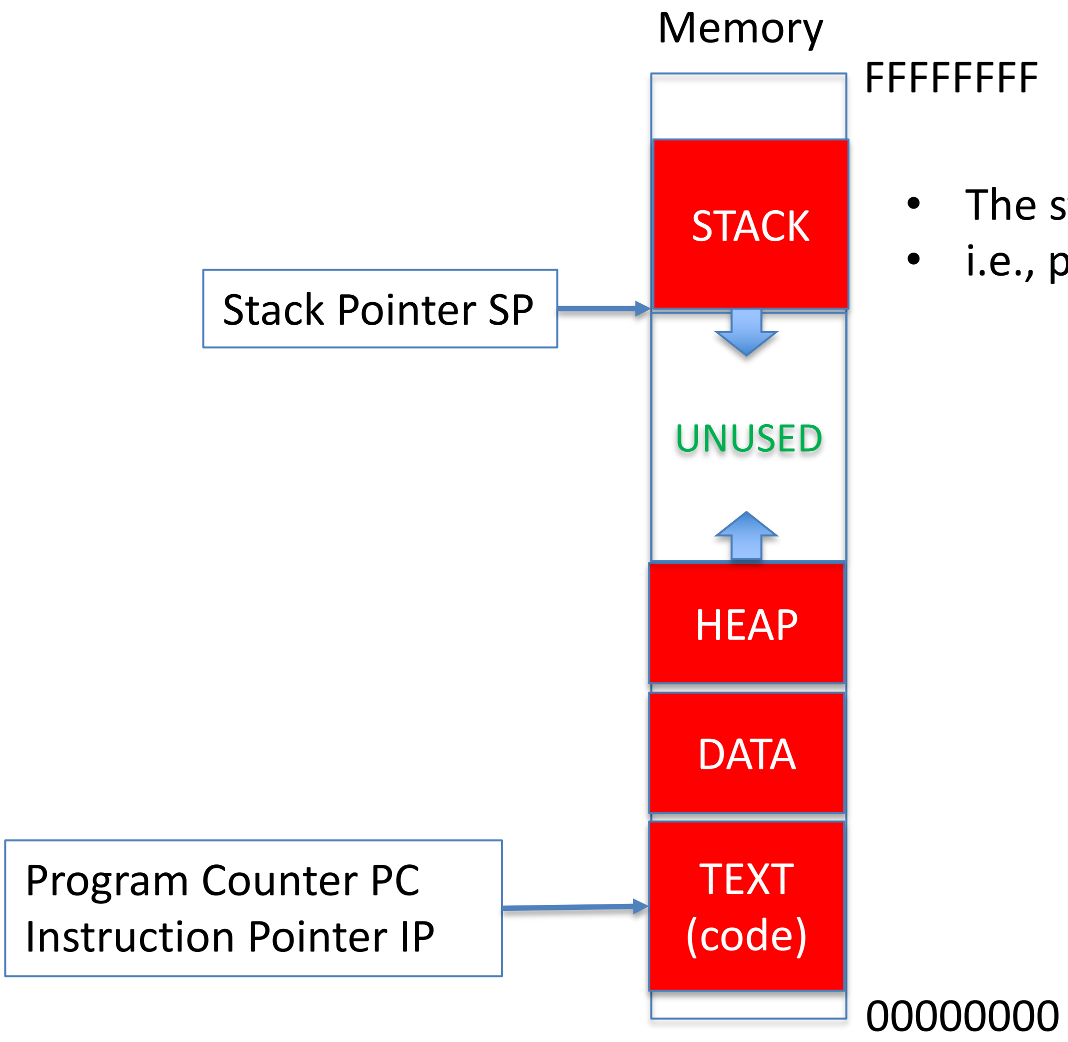
- Collection of “lines” (wires)
- Control bus: Load/Store/Interrupt/…
- Data bus: x lines → word is x bits
  - e.g: 32 lines: word is 32 bits (4 bytes)
- Address bus: y lines → address is y bits
  - process can address at most 2^y bytes

Logical View of CPU and Memory



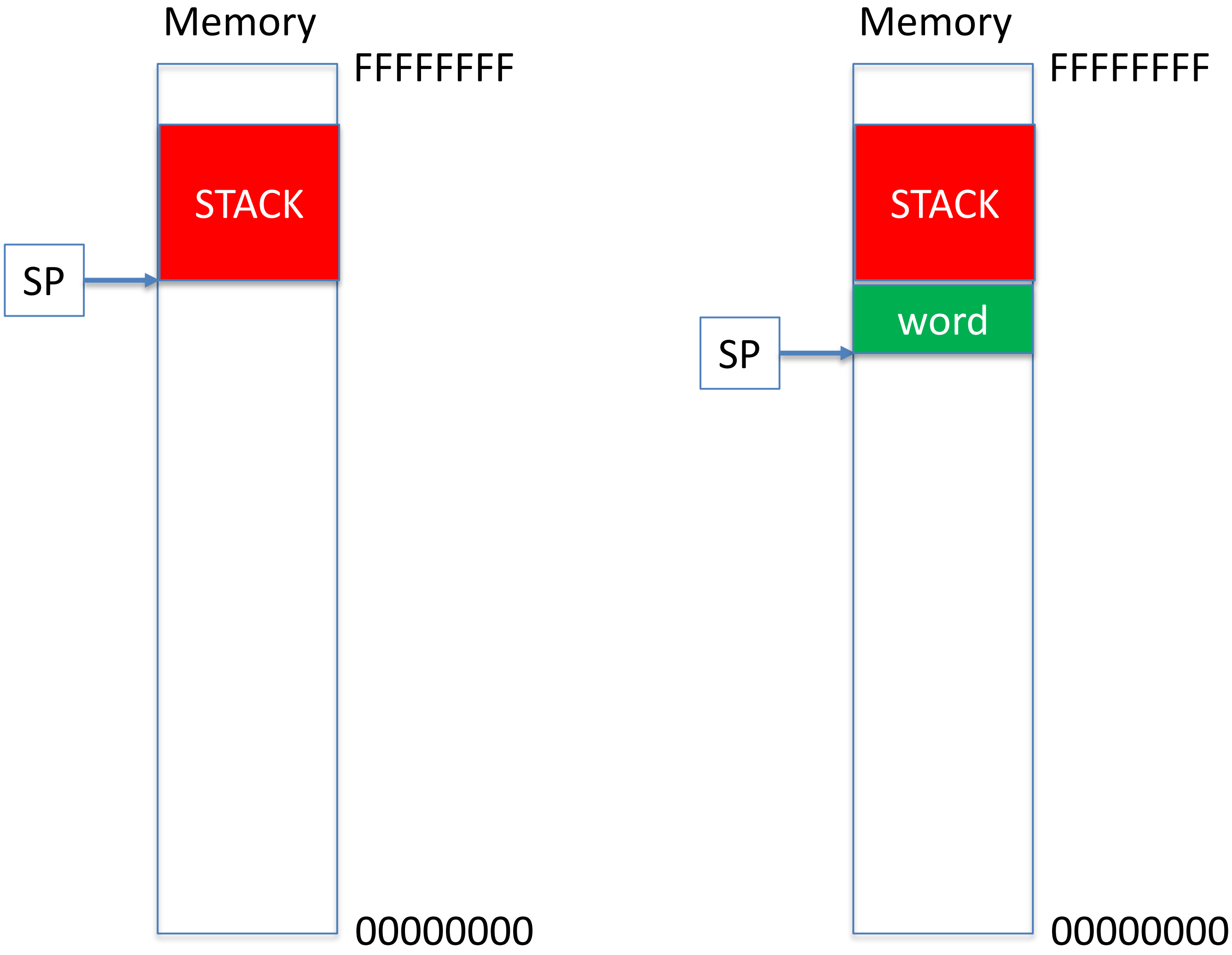
- memory is an array of bytes
- address is an index into the array

Memory "segments"

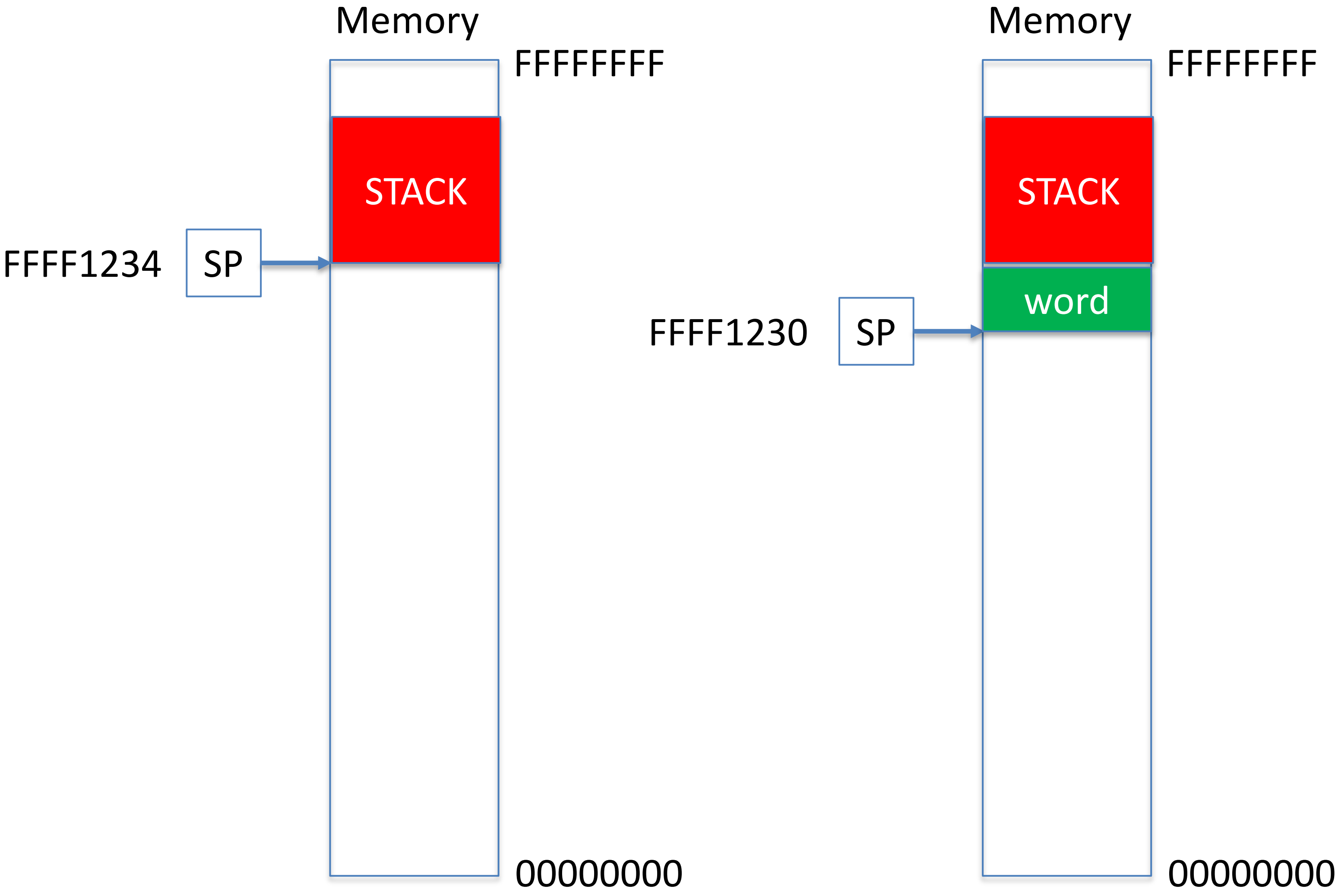


- The stack is usually word-aligned
- i.e., push and pop words

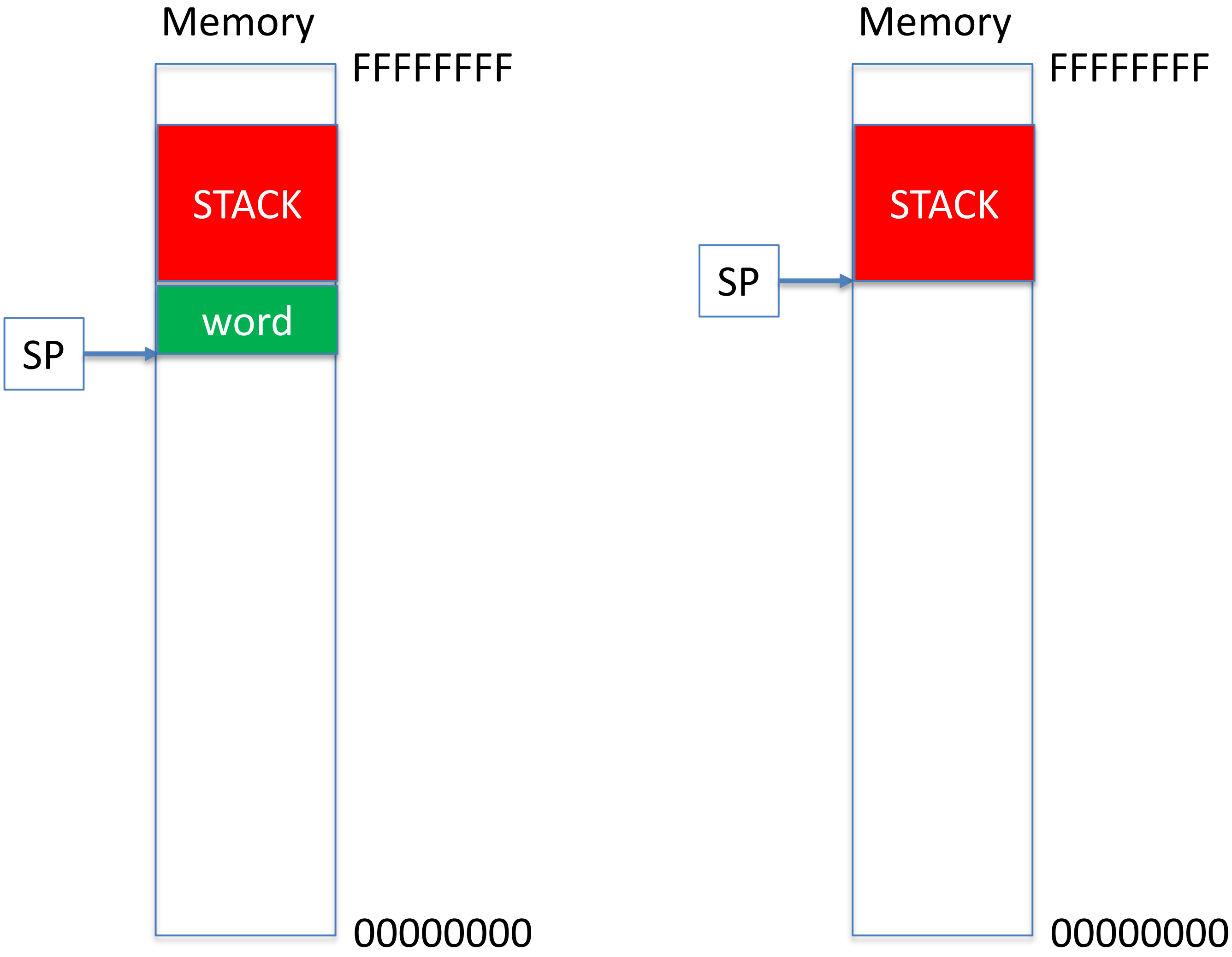
Stack before and after Push



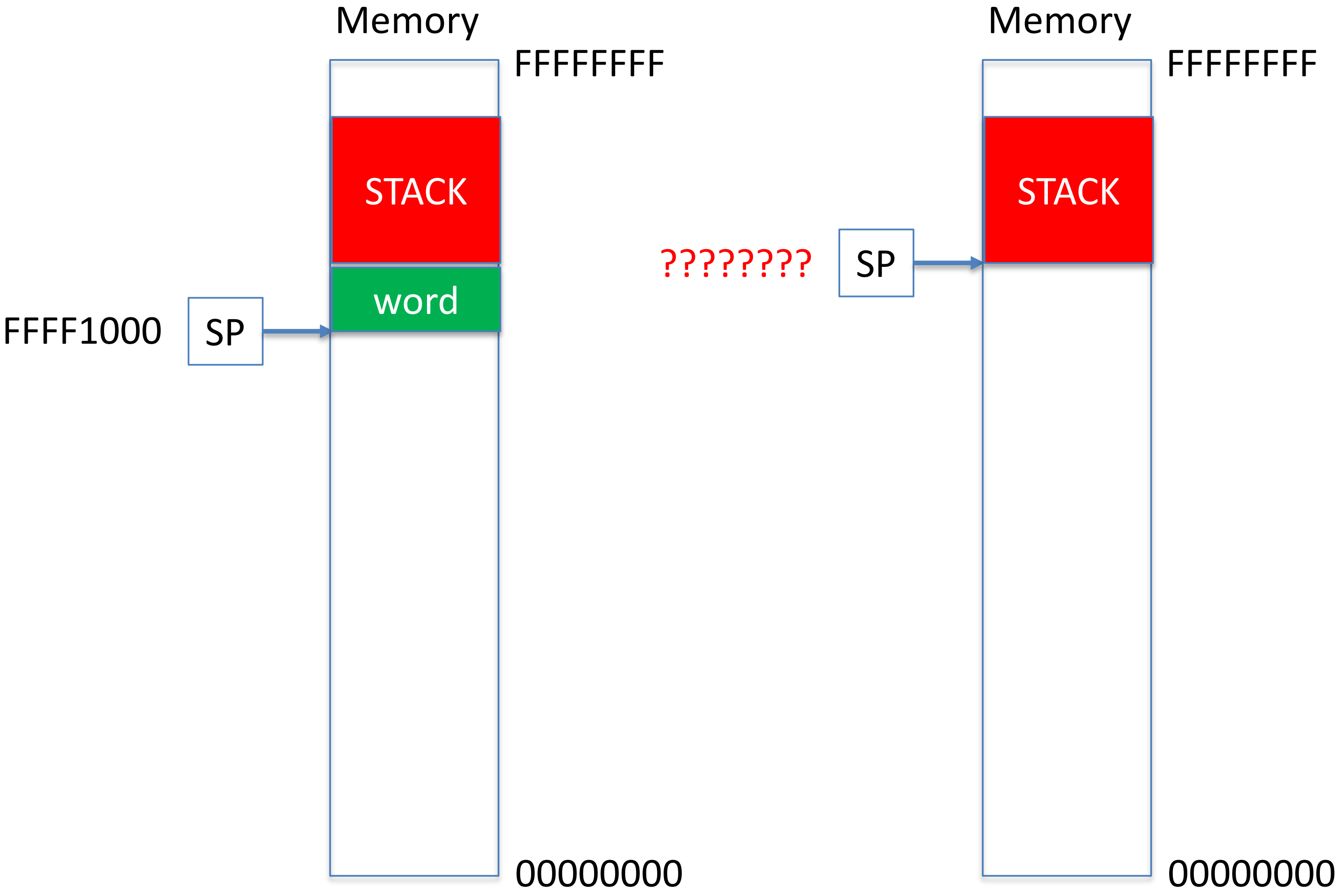
Stack before and after Push



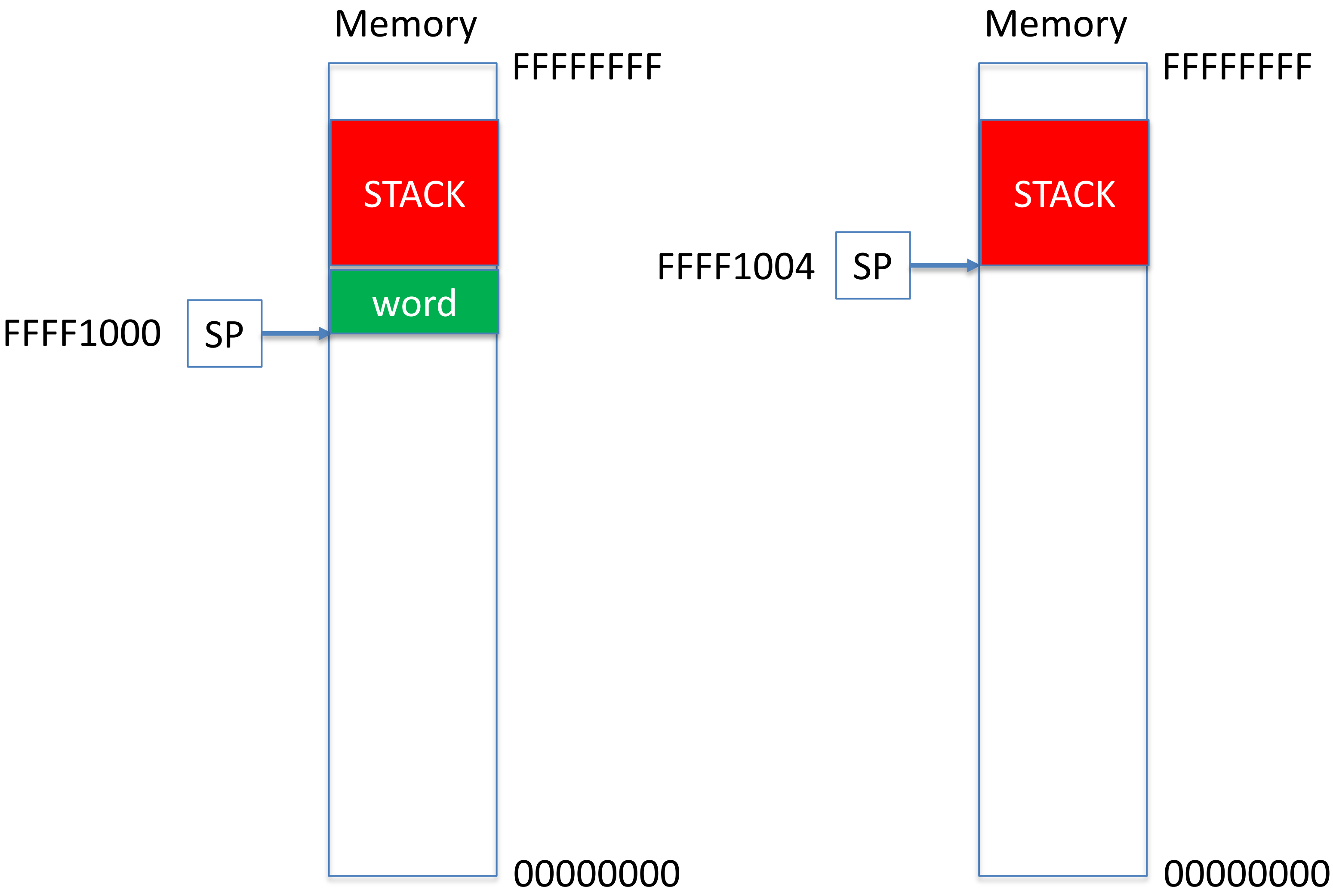
Stack before and after Pop



Stack before and after Pop



Stack before and after Pop



Control Flow and the Stack

- **call *f*:**
  - pushes return address onto the stack
  - sets program counter to address of *f*
  - *f* will typically start with saving registers that it wants to use and end with restoring them
- **return**
  - pops return address from the stack
    - and sets PC to the return address

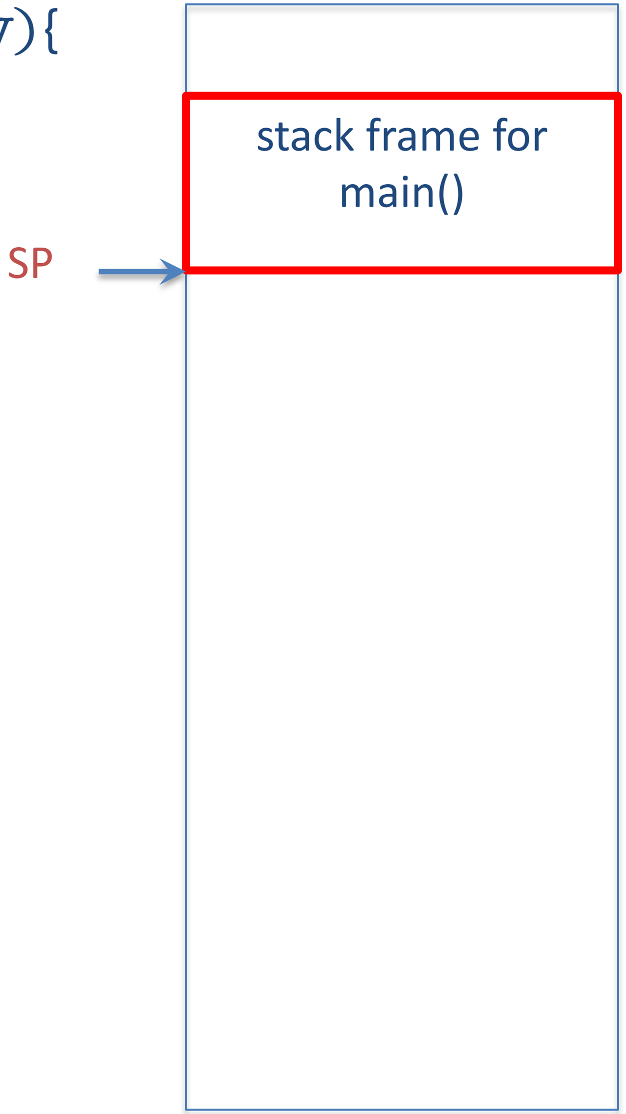
Control Flow

```
int main(argc, argv){
    …
    f(3.14)
    …
}

int f(x){
    …
    g();
    …
}

int g(y){
    …
}
```

PC/IP

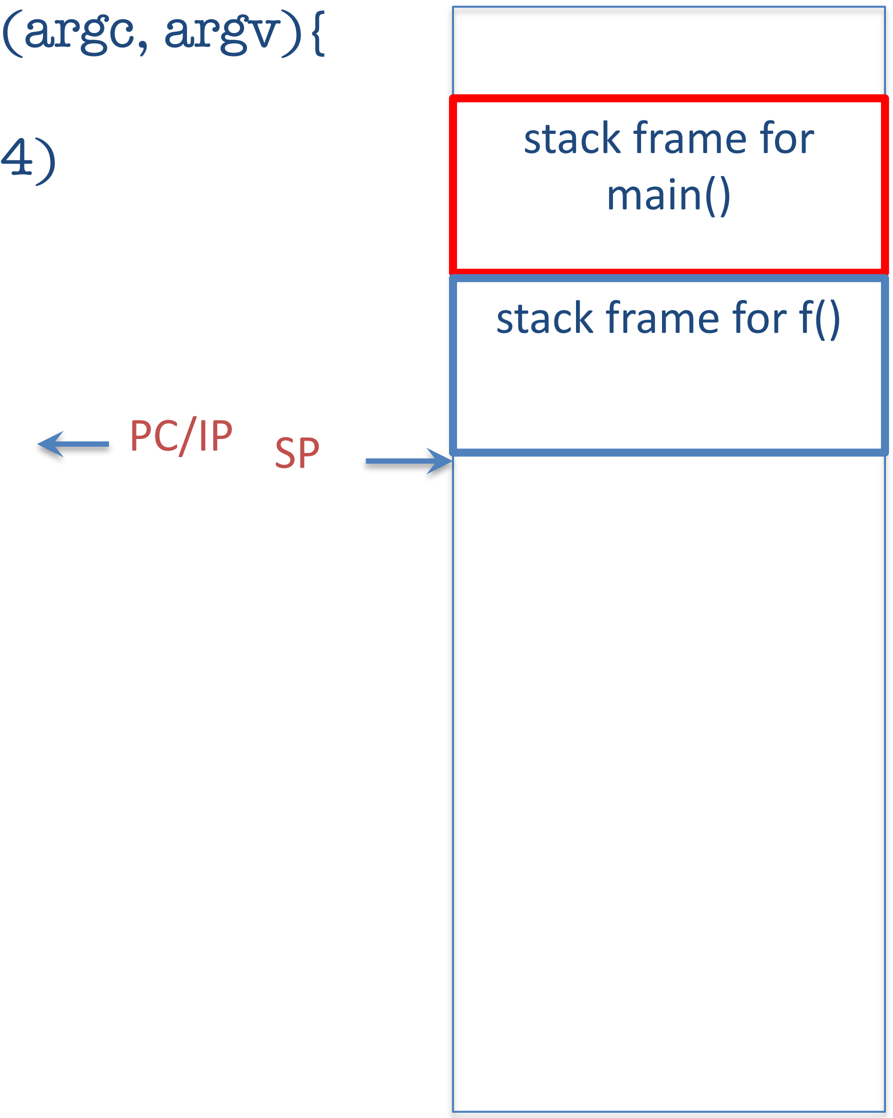


Control Flow

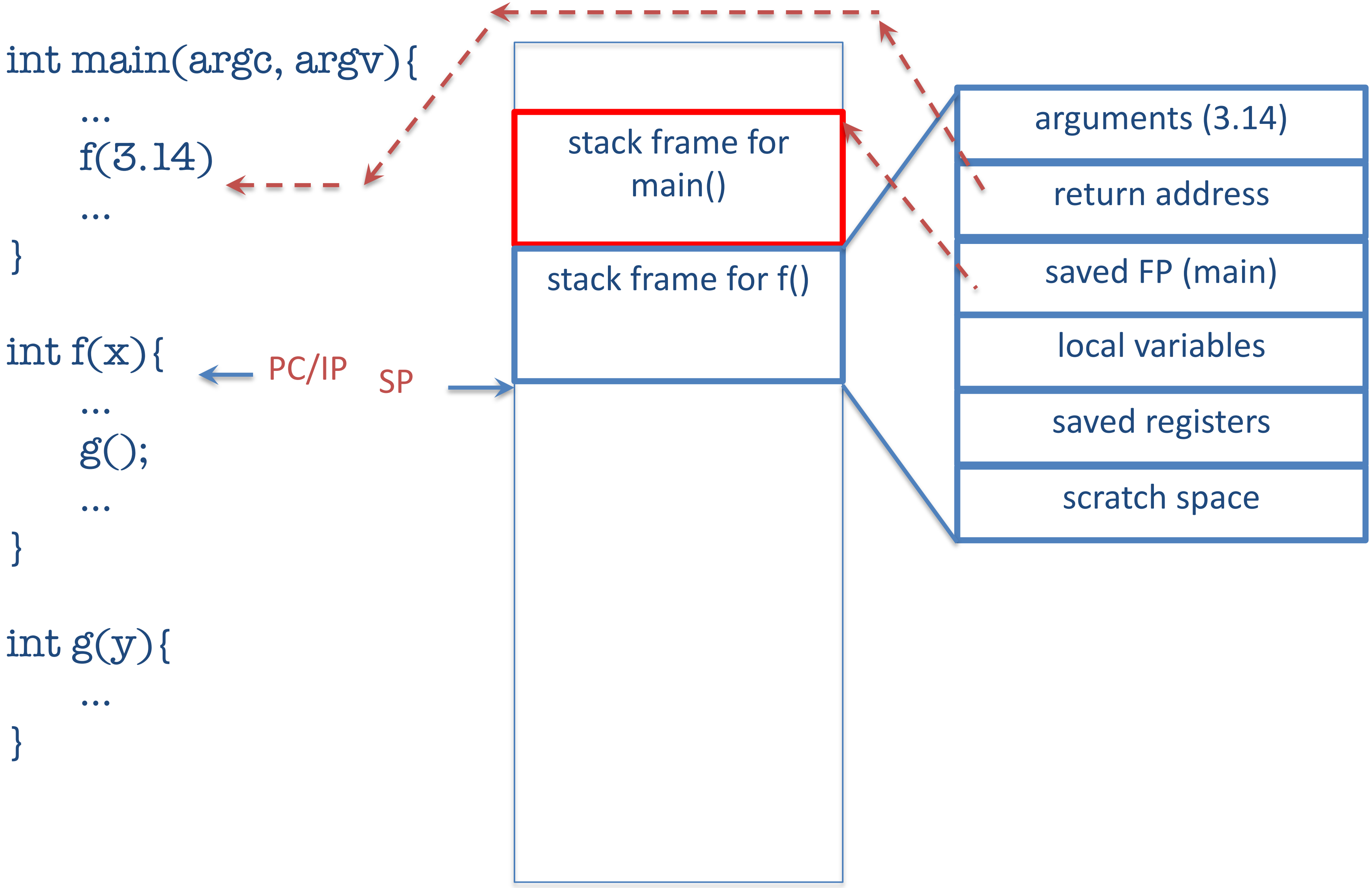
```
int main(argc, argv){
      …
      f(3.14)
      …
}

int f(x){
      …
      g();
      …
}

int g(y){
      …
}
```



Control Flow



```
int main(argc, argv){
    ...
    f(3.14)
    ...
}

int f(x){
    ...
    g();
    ...
}

int g(y){
    ...
}
```

PC/IP

SP

stack frame for main()

stack frame for f()

arguments (3.14)

return address

saved FP (main)

local variables

saved registers

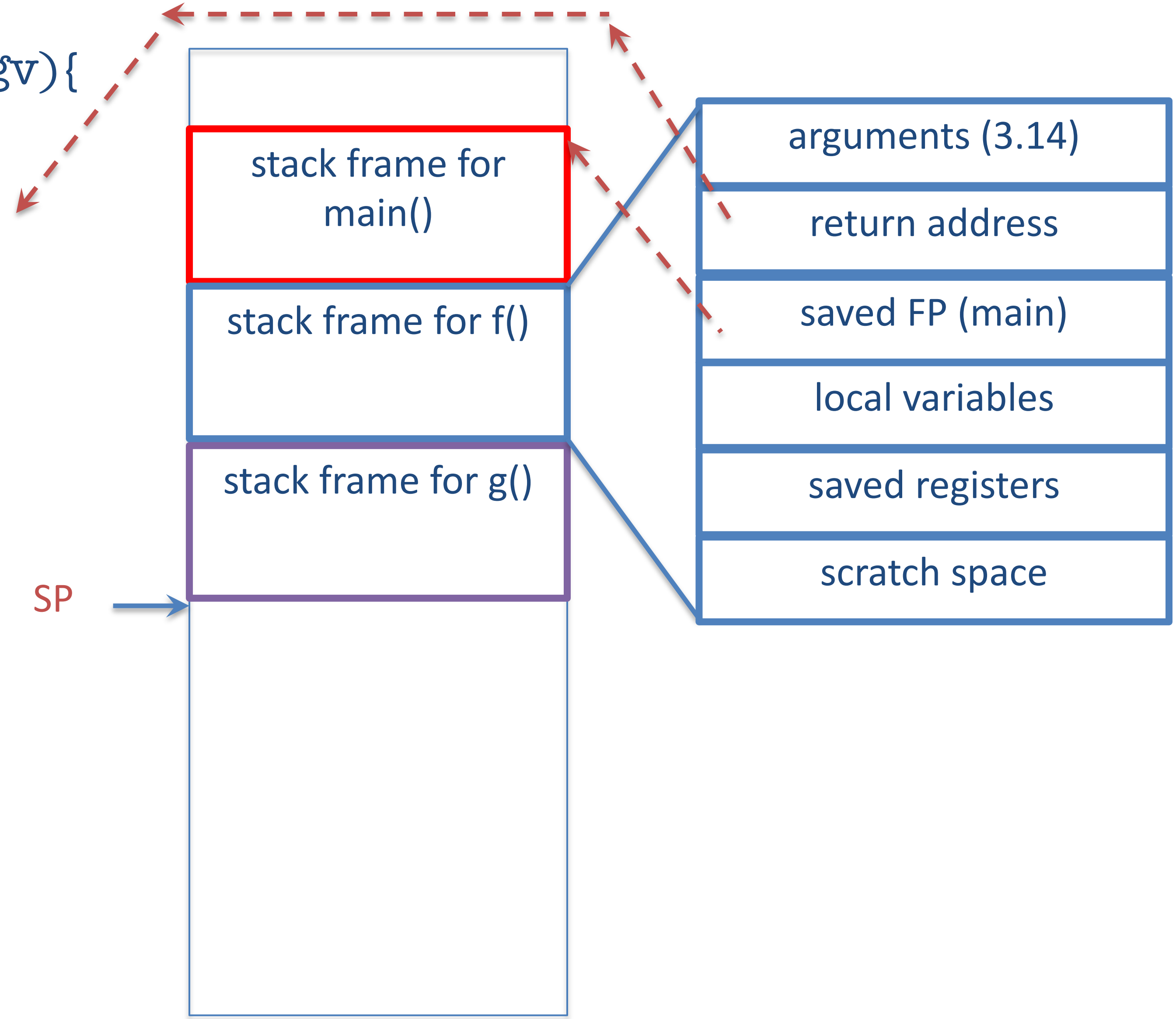
scratch space

Control Flow

```
int main(argc, argv){
    …
    f(3.14)
    …
}

int f(x){
    …
    g();
    …
}

int g(y){
    …
}
```

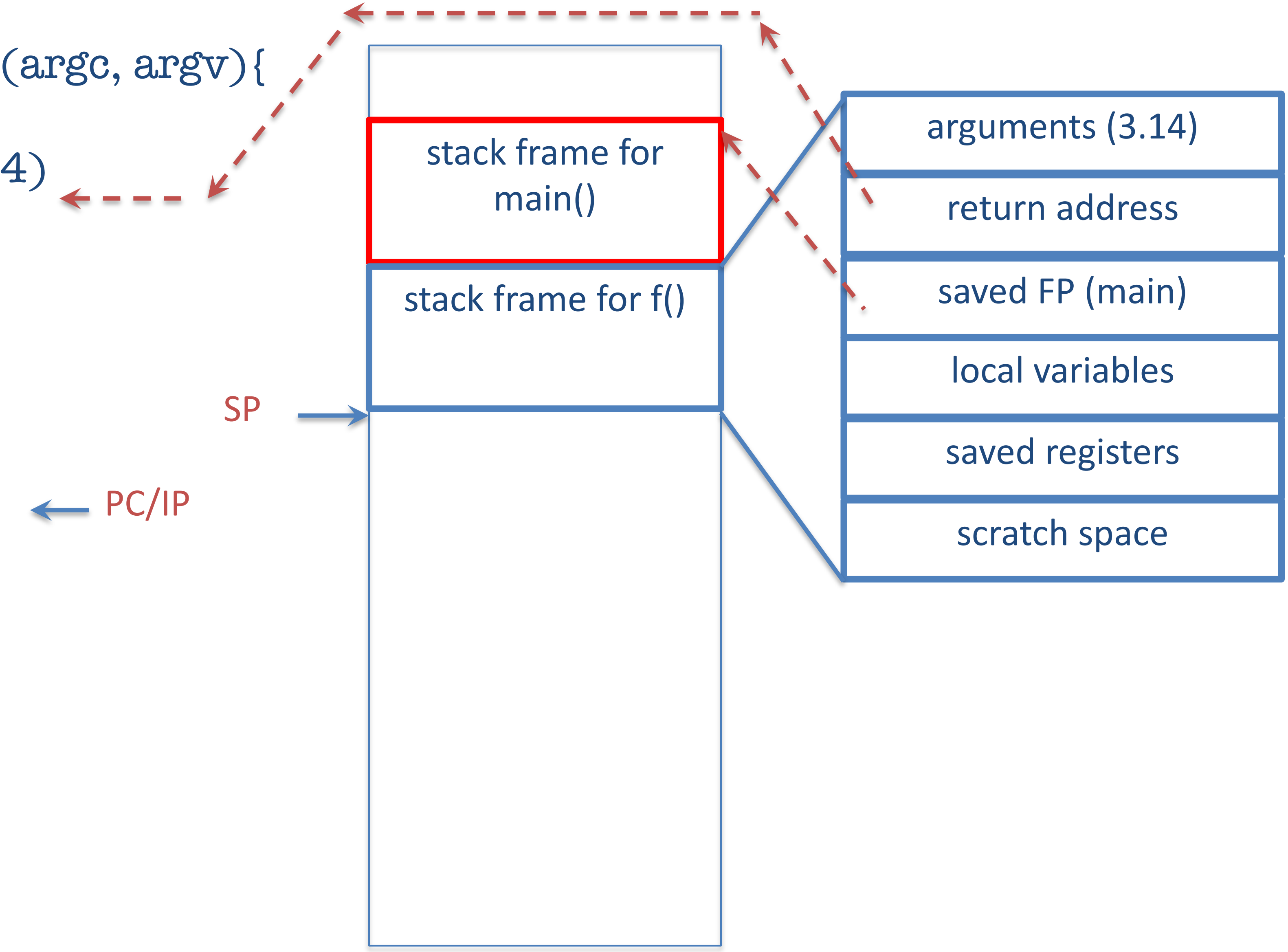


Control Flow

```
int main(argc, argv){
      …
      f(3.14)
      …
}

int f(x){
      …
      g();
      …
}

int g(y){
      …
}
```



SP

PC/IP

stack frame for main()

stack frame for f()

arguments (3.14)

return address

saved FP (main)

local variables

saved registers

scratch space

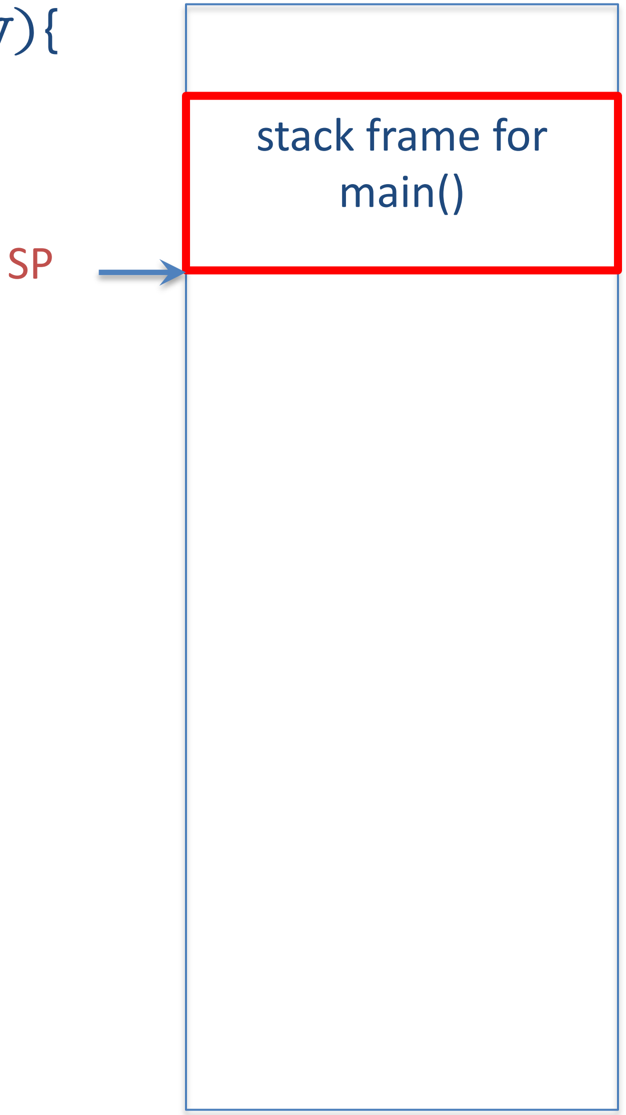
Control Flow

```
int main(argc, argv){
    …
    f(3.14)
    …
}

int f(x){
    …
    g();
    …
}

int g(y){
    …
}
```

PC/IP



Add “supervisor mode”

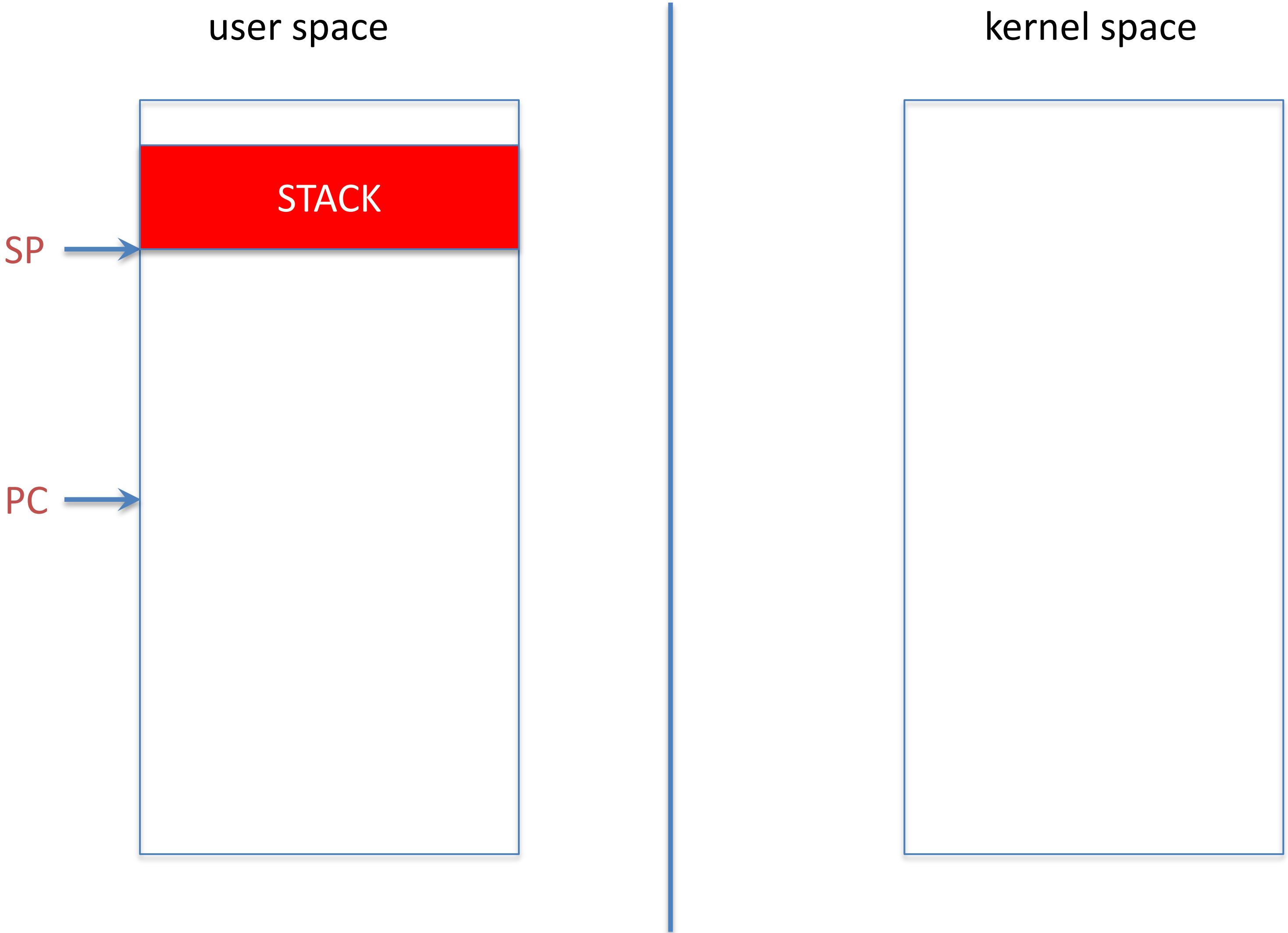
- The ”kernel” is code that runs in supervisor mode
- A ”process” is code that runs in user mode
- Each has its own segments (code, data, heap, stack) and its own registers (pc, sp, psw, r1, r2, …)
- How do you switch from one to the other?

Add “supervisor mode”

- The ”kernel” is code that runs in supervisor mode
- A ”process” is code that runs in user mode
- Each has its own segments (code, data, heap, stack) and its own registers (pc, sp, psw, r1, r2, …)
- How do you switch from one to the other?
  - user mode → supervisor mode
    - signal: interrupt, system call, fault
  - supervisor mode → user mode
    - return-from-interrupt instruction

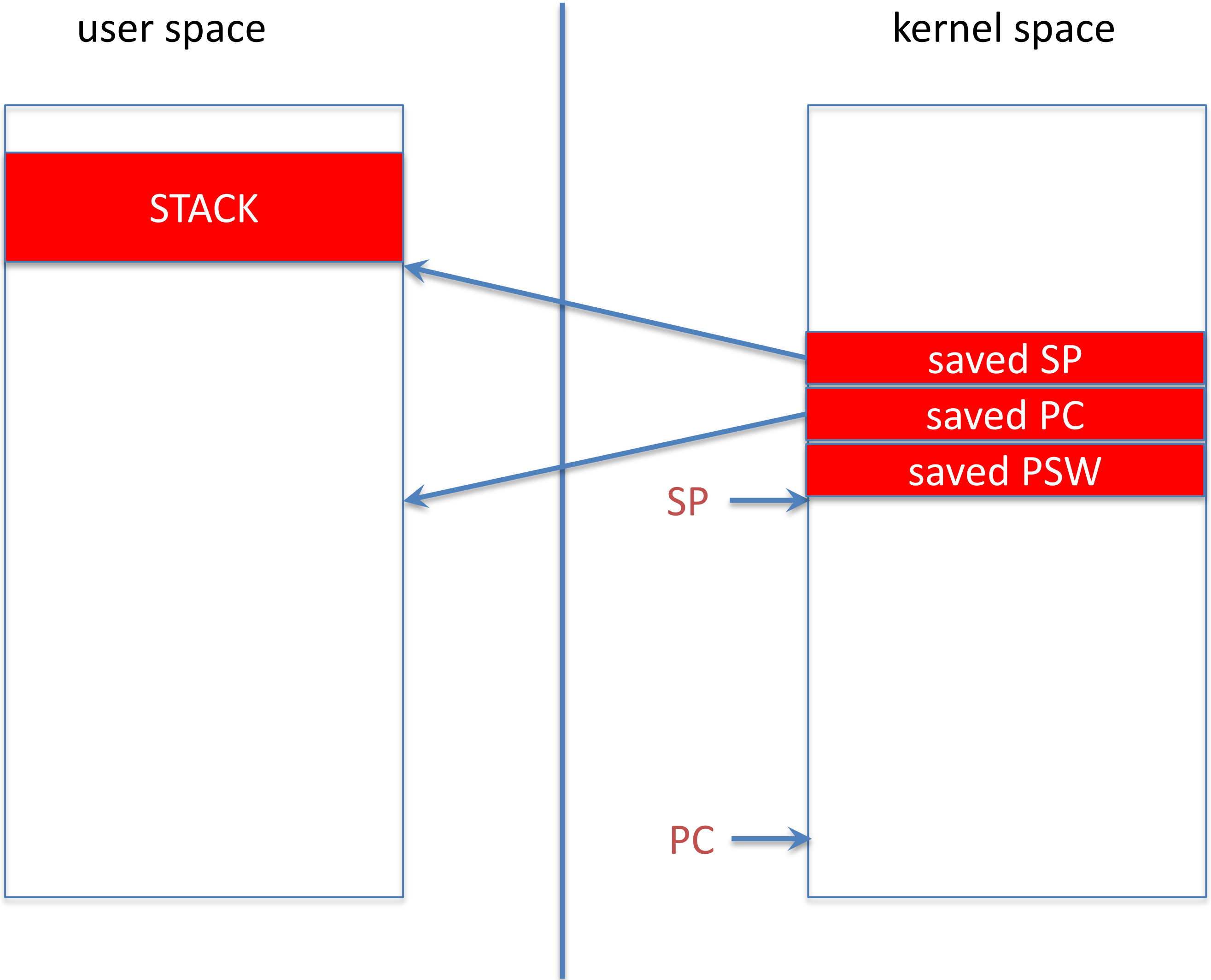
user mode → supervisor mode

In user mode, before interrupt



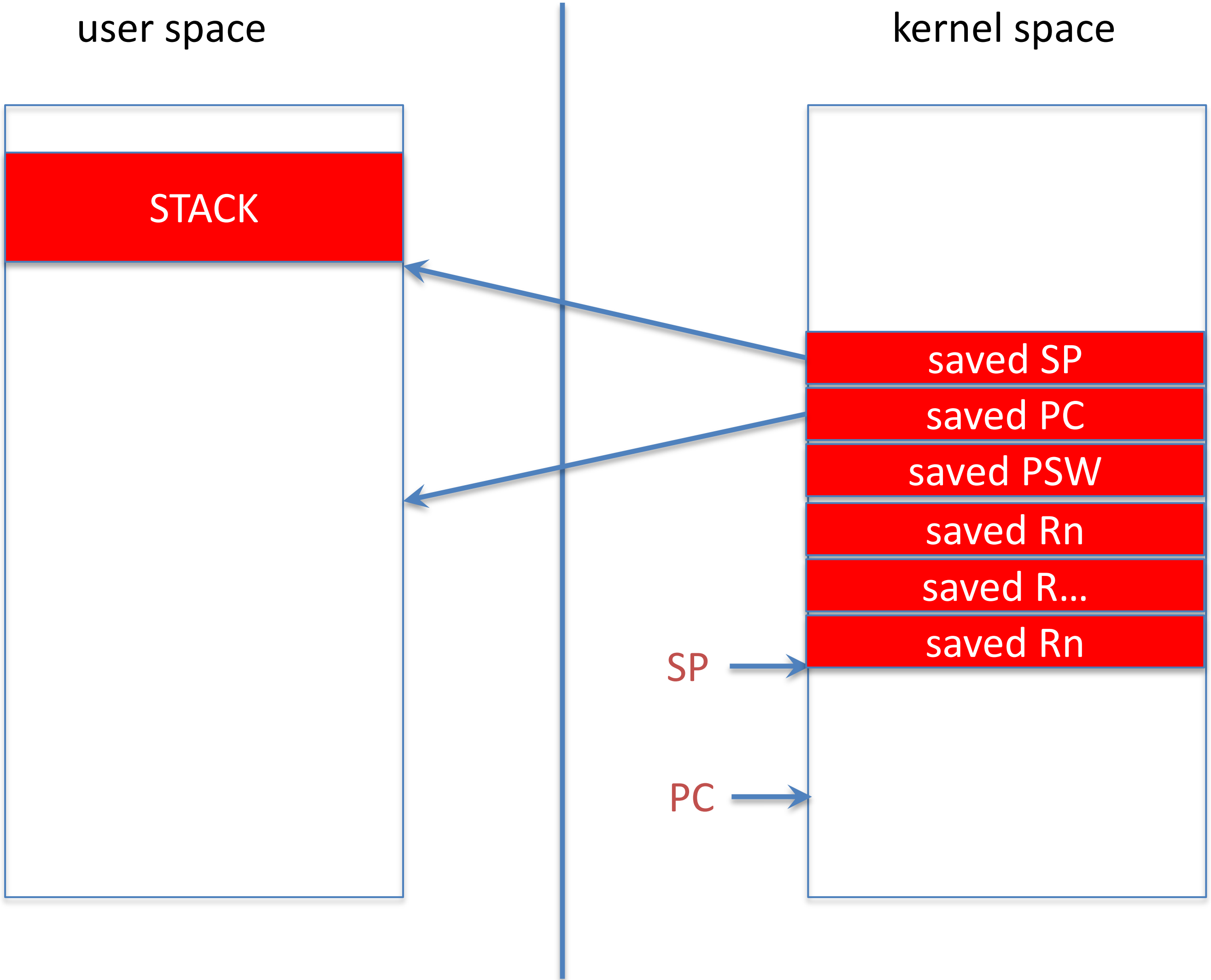
user mode → supervisor mode

In supervisor mode, right after interrupt



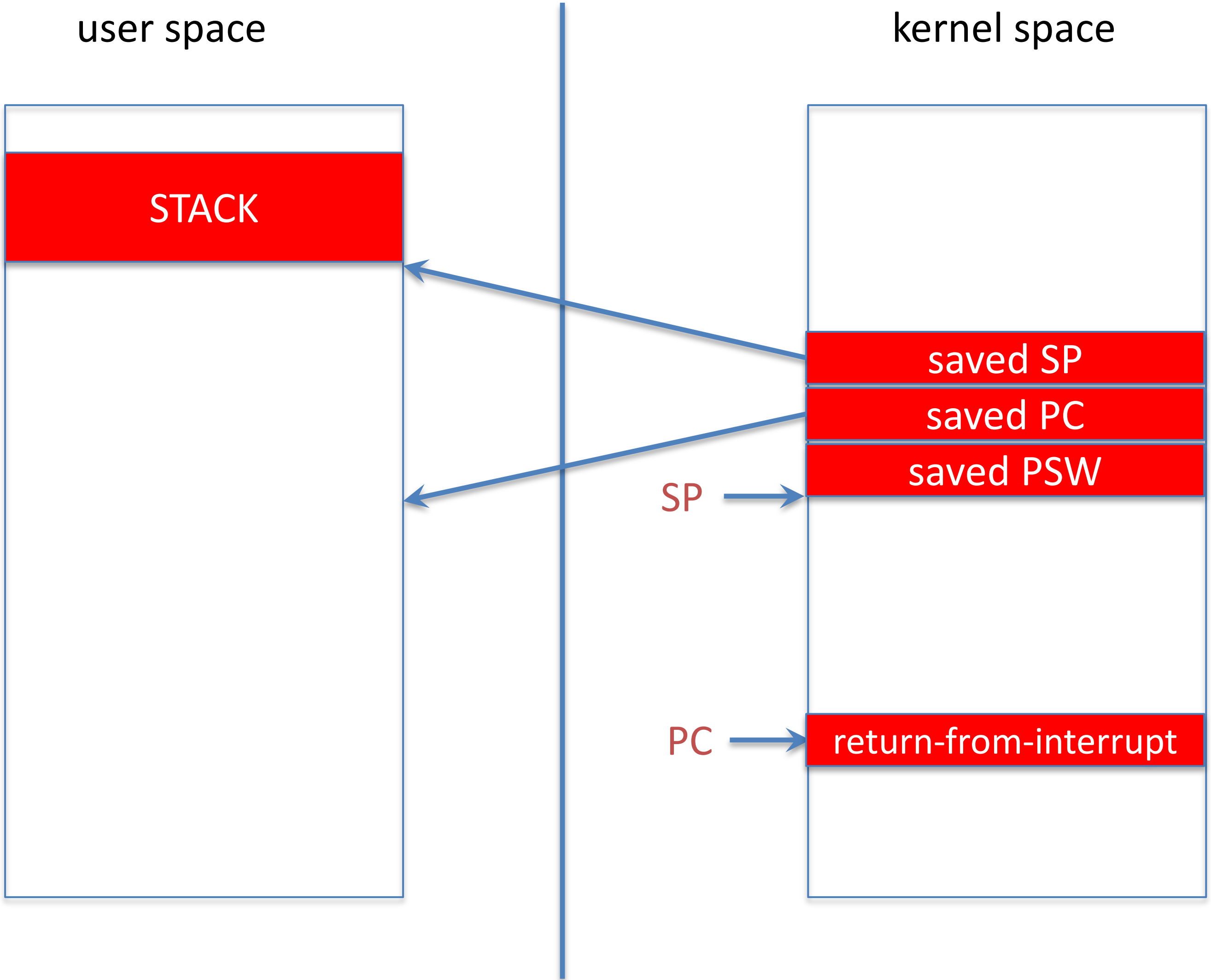
register save/restore

In supervisor mode, after saving registers



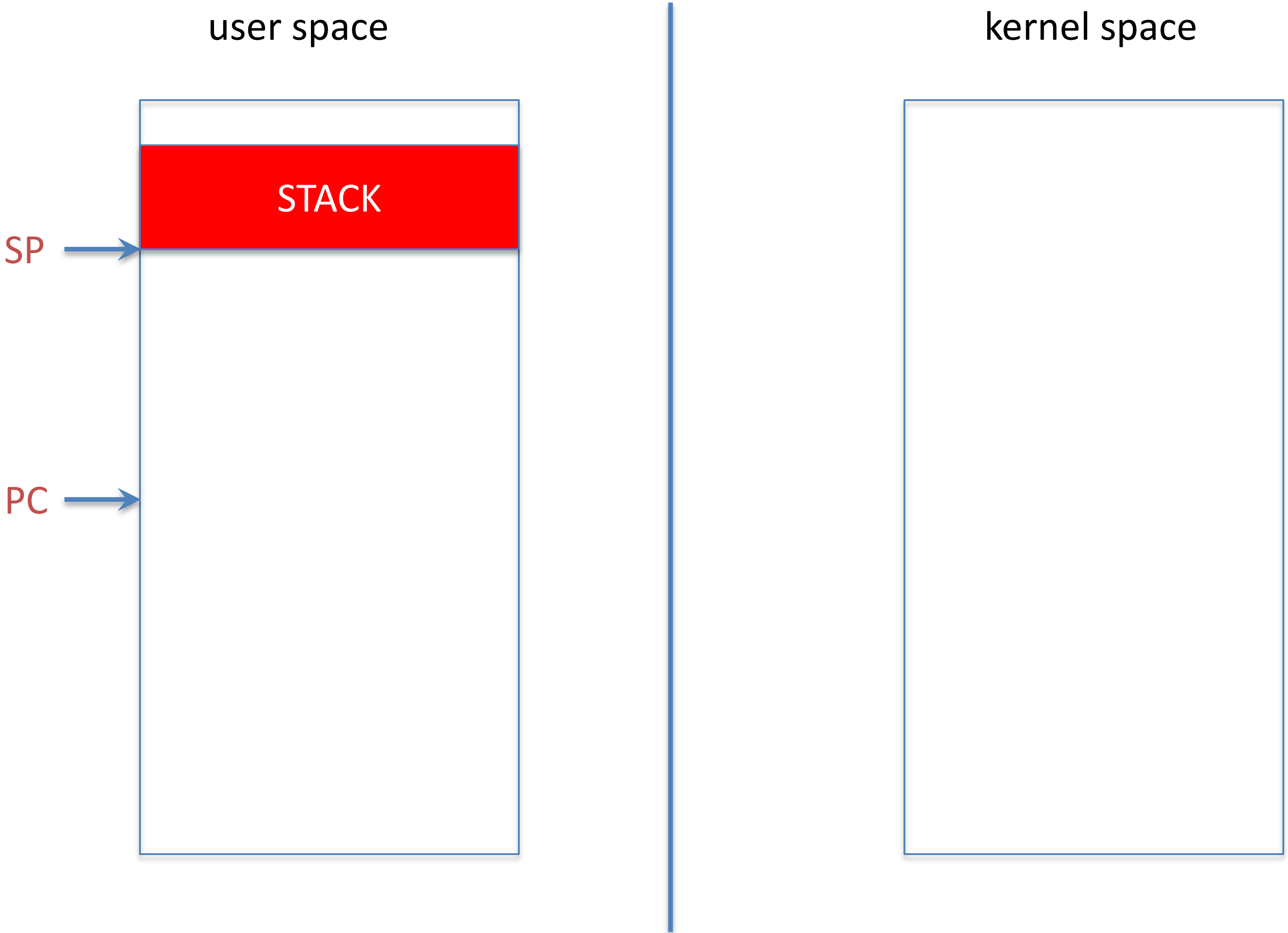
supervisor mode → user mode

In supervisor mode, just before "return-from-interrupt"



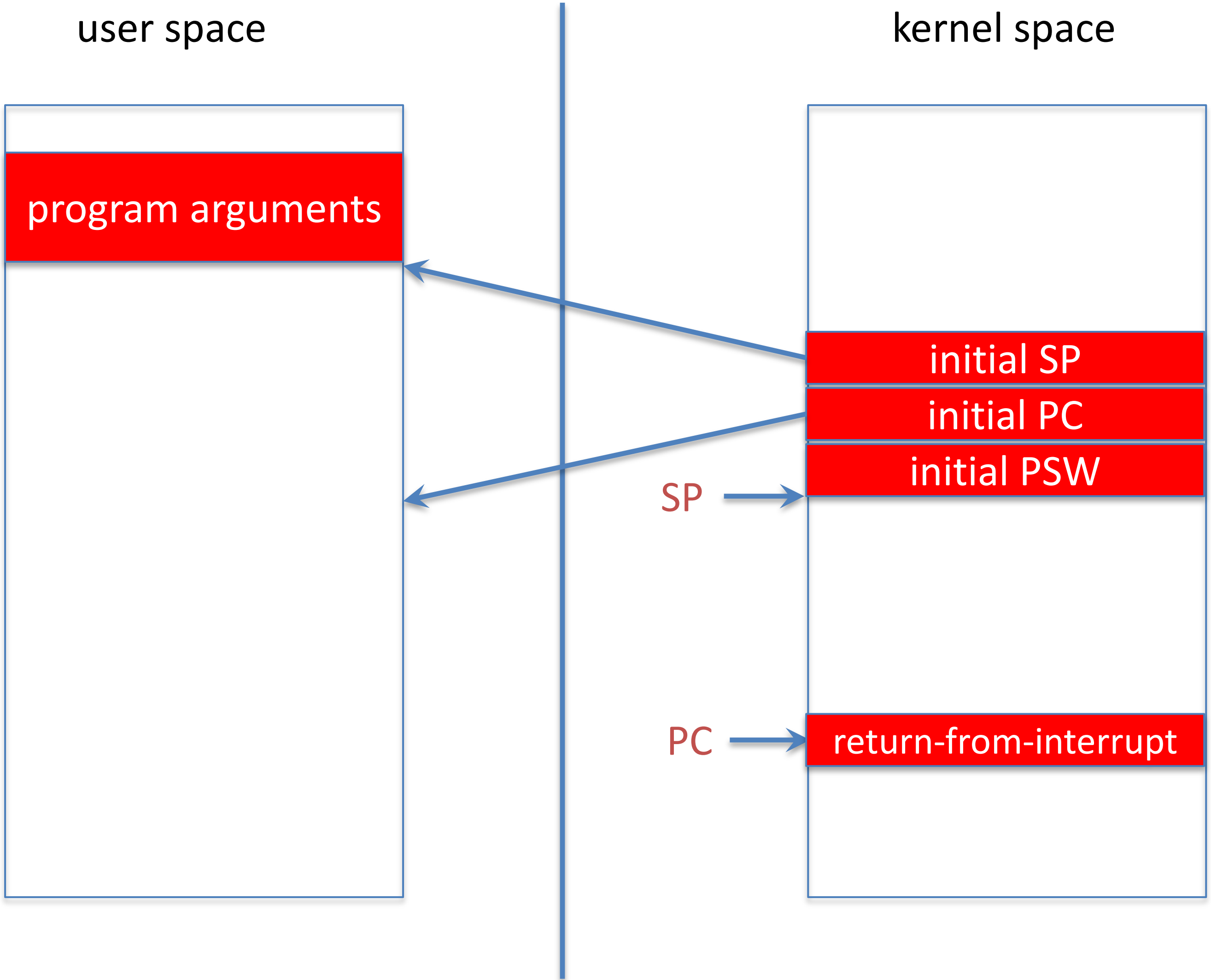
supervisor mode → user mode

In user mode, right after return-from-interrupt



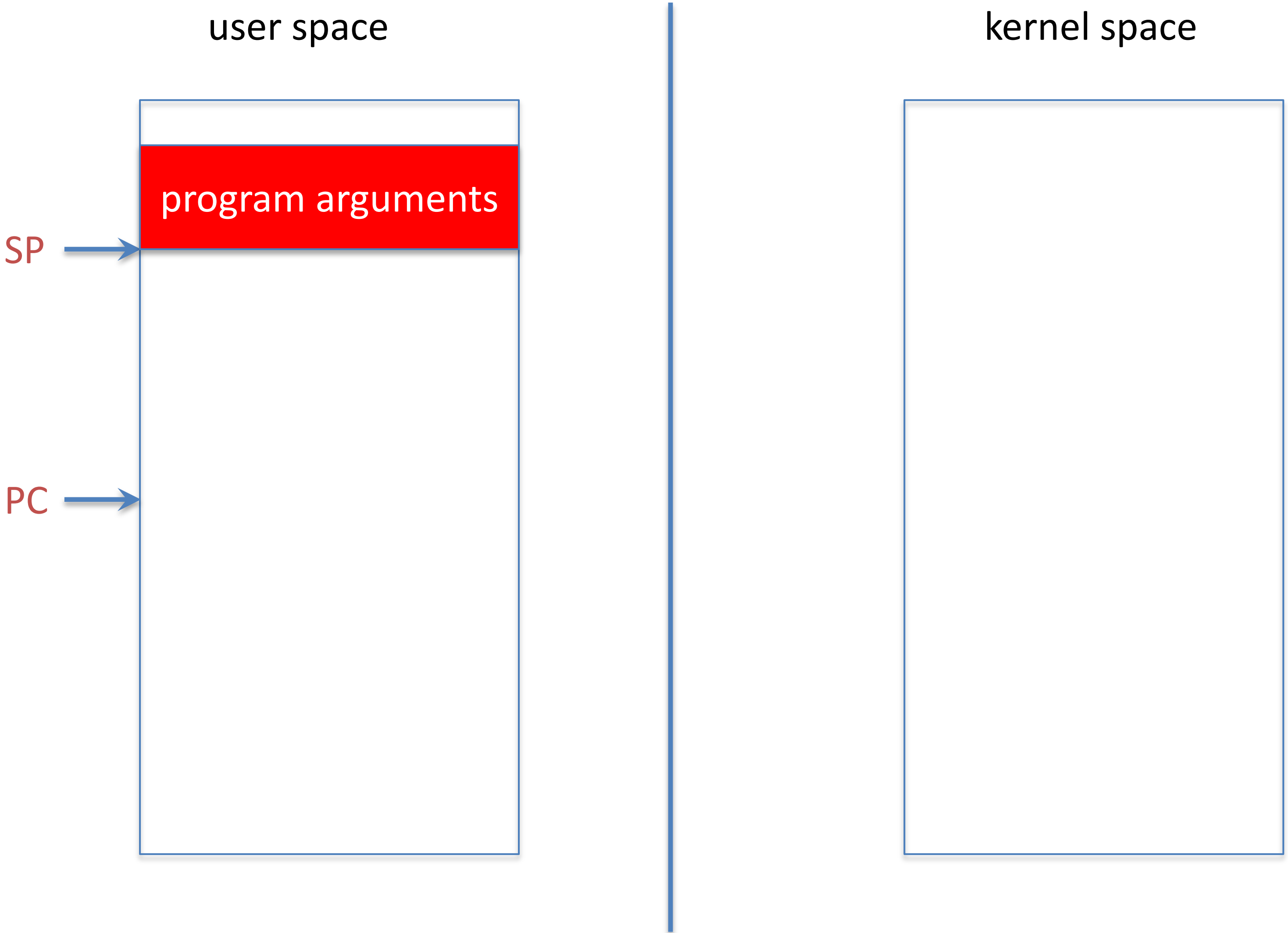
Starting a new process

In supervisor mode, just before "return-from-interrupt"



Starting a new process

In user mode, right after return-from-interrupt



SUPPORT FOR DEVICES

Device Management

- Another primary objective of an O.S. kernel is to manage and multiplex devices
- Example devices:
    - screen
    - keyboard
    - mouse
    - camera
    - microphone
    - printer
    - clock
    - disk
    - USB
    - Ethernet
    - WiFi
    - Bluetooth

Device Registers

- A device presents itself to the CPU as (pseudo)memory
- Simple example:
  - each pixel on the screen is a word in memory that can be written
- Devices define a range of *device registers*
  - accessible through LOAD and STORE operations

Example: Disk Device (simplified)

- can only read and write blocks, not words
- registers:
  1. block number: which block to read or write
  2. memory address: where to copy block from/to
  3. command register: to start read/write operations
     - device interrupts CPU upon completion
  4. interrupt ack register: to tell device interrupt received
  5. status register: to examine status of operations

Example: Network Device (simplified)

- registers:
  1. receive memory address: for incoming packets
  2. send memory address: for outgoing packets
  3. command register: to send/receive packet
     - device interrupts CPU upon completion
  4. interrupt ack register: to tell device interrupt received
  5. status register: to examine status of operations

Device Drivers

- *Device Driver*: a code module that deals with a particular brand/model of hardware device
  - initialization
  - starting operations
  - interrupt handling
  - error handling
- An O.S. has many disk drivers, many network drivers, etc.
  - >90% of an O.S. code base
  - huge security issue… **WHY??**
- But all disk drivers have a common API
  - disk_init(), read_block(), write_block(), etc.
- So do all network drivers
  - net_init(), receive_packet(), send_packet()

O.S. support for device drivers

- kernels provide many functions for drivers:
  - interrupt management
  - memory allocation
  - queues
  - copying between user space/kernel space
  - error logging
  - …

BOOTING AN O.S.

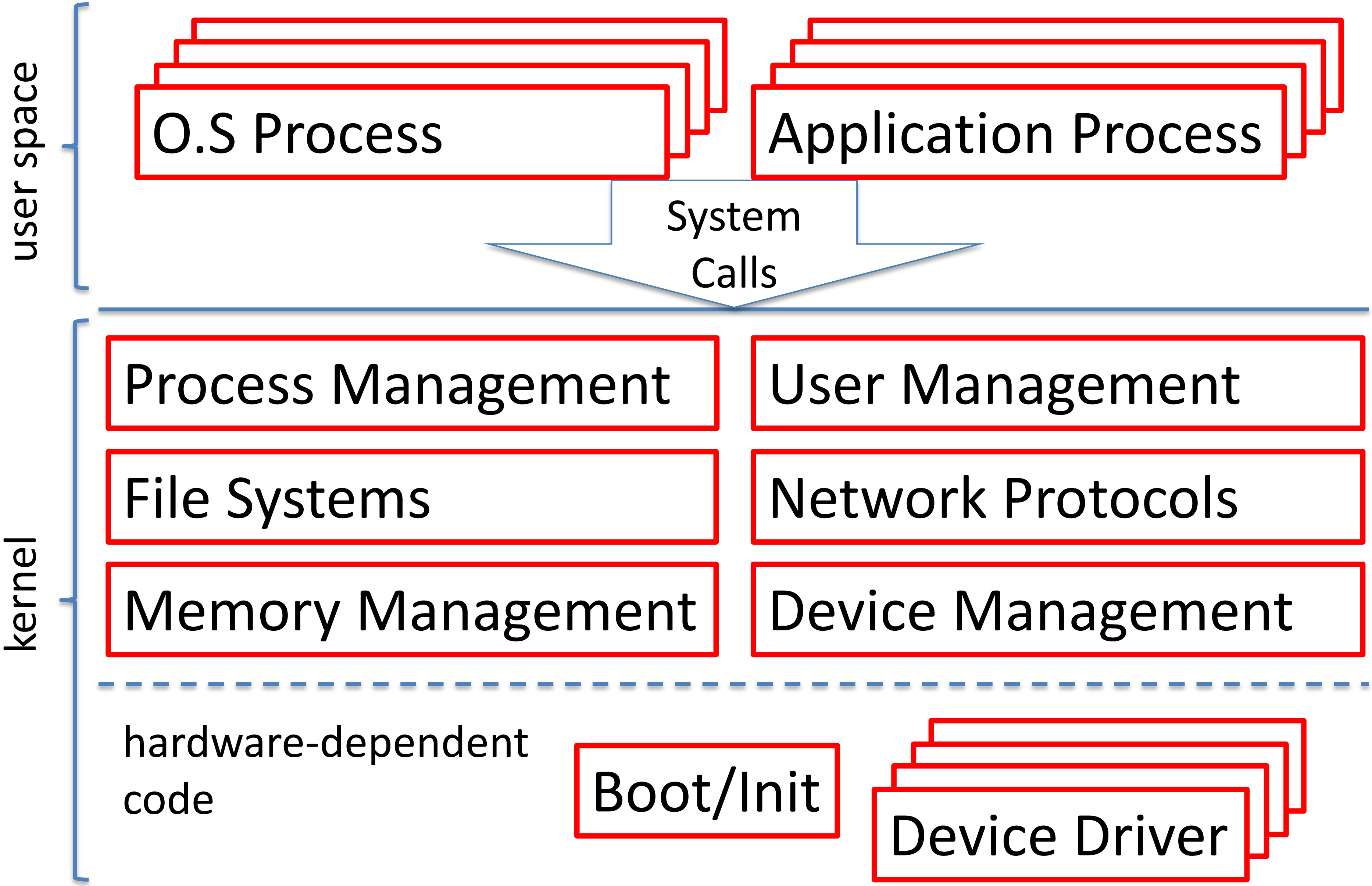
Booting an O.S.

- “pull oneself over a fence by one's bootstraps”
- Steps in booting an O.S.:
    1. CPU starts at fixed address
        - in supervisor mode with interrupts disabled
    2. BIOS (in ROM) loads “boot loader” code from specified storage or network device into memory and runs it
    3. boot loader loads O.S. kernel code into memory and runs it

O.S. initialization

1. determine location/size of physical memory
2. set up initial MMU / page tables
3. initialize the interrupt vector
4. determine which devices the computer has
   - invoke device driver initialization code for each
5. initialize file system code
6. load first process from file system
7. start first process

O.S. Code Architecture



user space

O.S Process

Application Process

System Calls

kernel

Process Management

User Management

File Systems

Network Protocols

Memory Management

Device Management

hardware-dependent code

Boot/Init

Device Driver