Short History of Operating Systems

CS 4410

Operating Systems

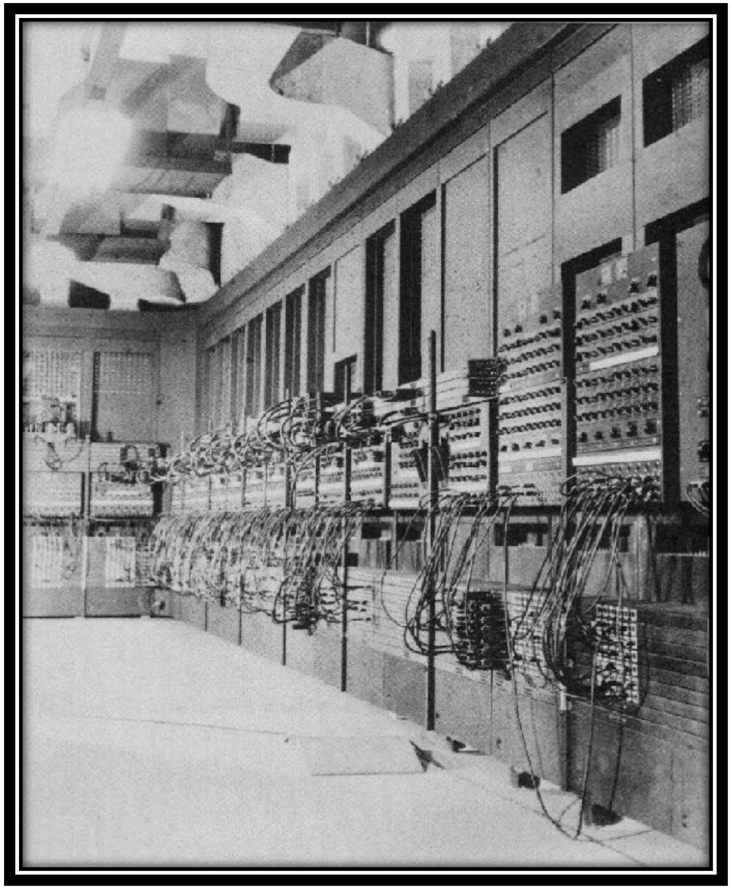
[R. Agarwal, L. Alvisi, A. Bracy, M. George,
F. B. Schneider, E. G. Sirer, R. Van Renesse]

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

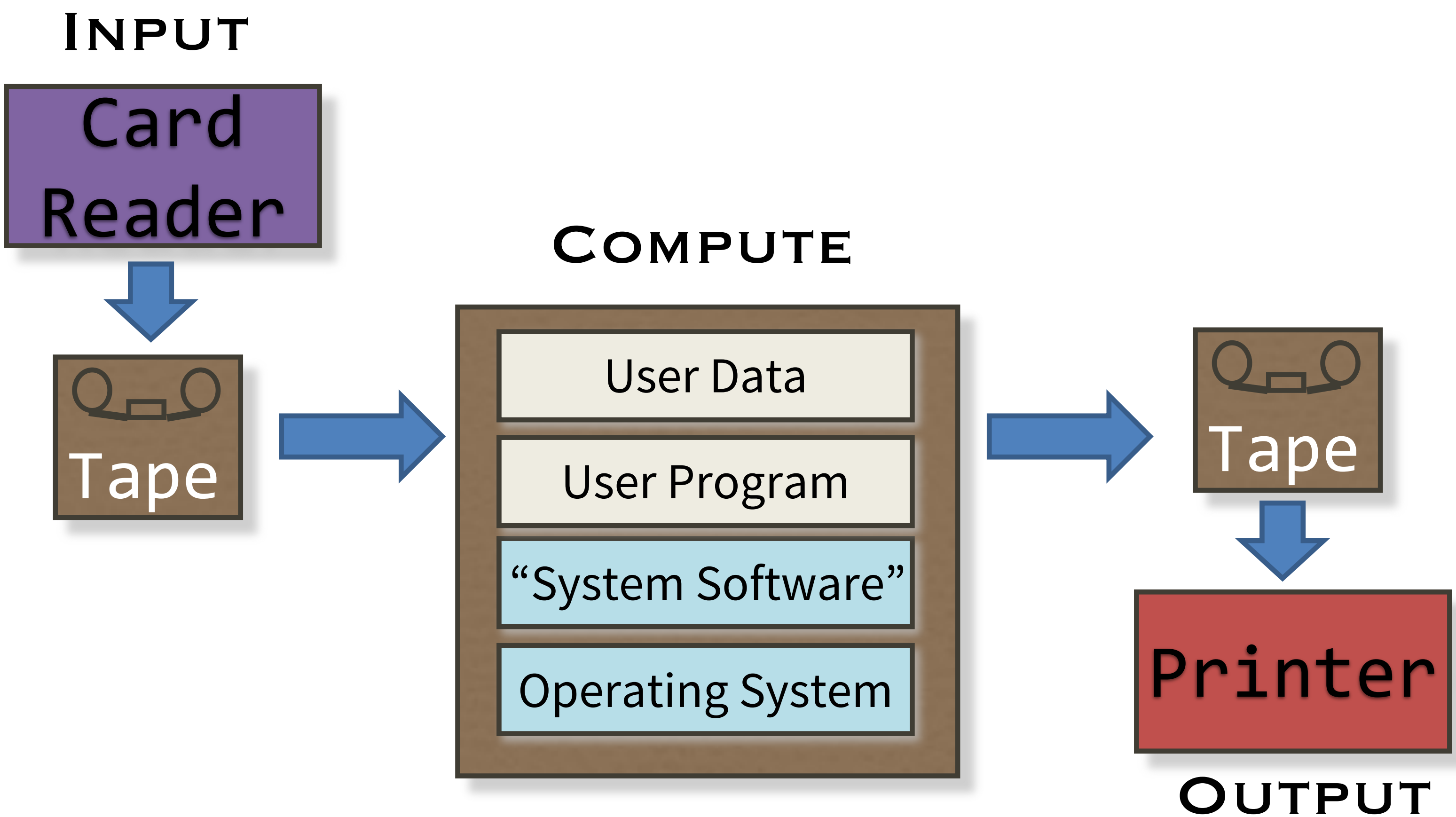


Batch Processing (1955 – 1960):

- 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)

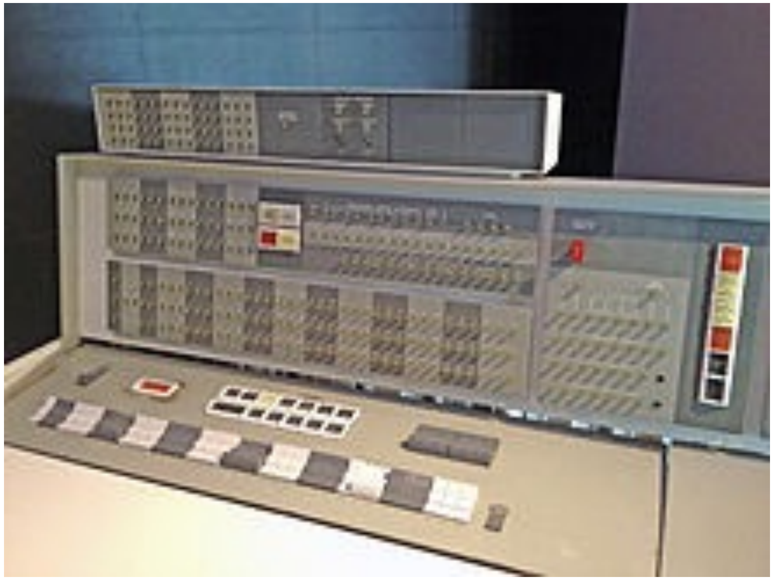
Batch Processing

- 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



Fernando J. Corbató (1926-2019)

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)

- Google Android (2006, based on Linux)
  - Android phones, tablets
- 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

Anatomy of a Computer (simplified)

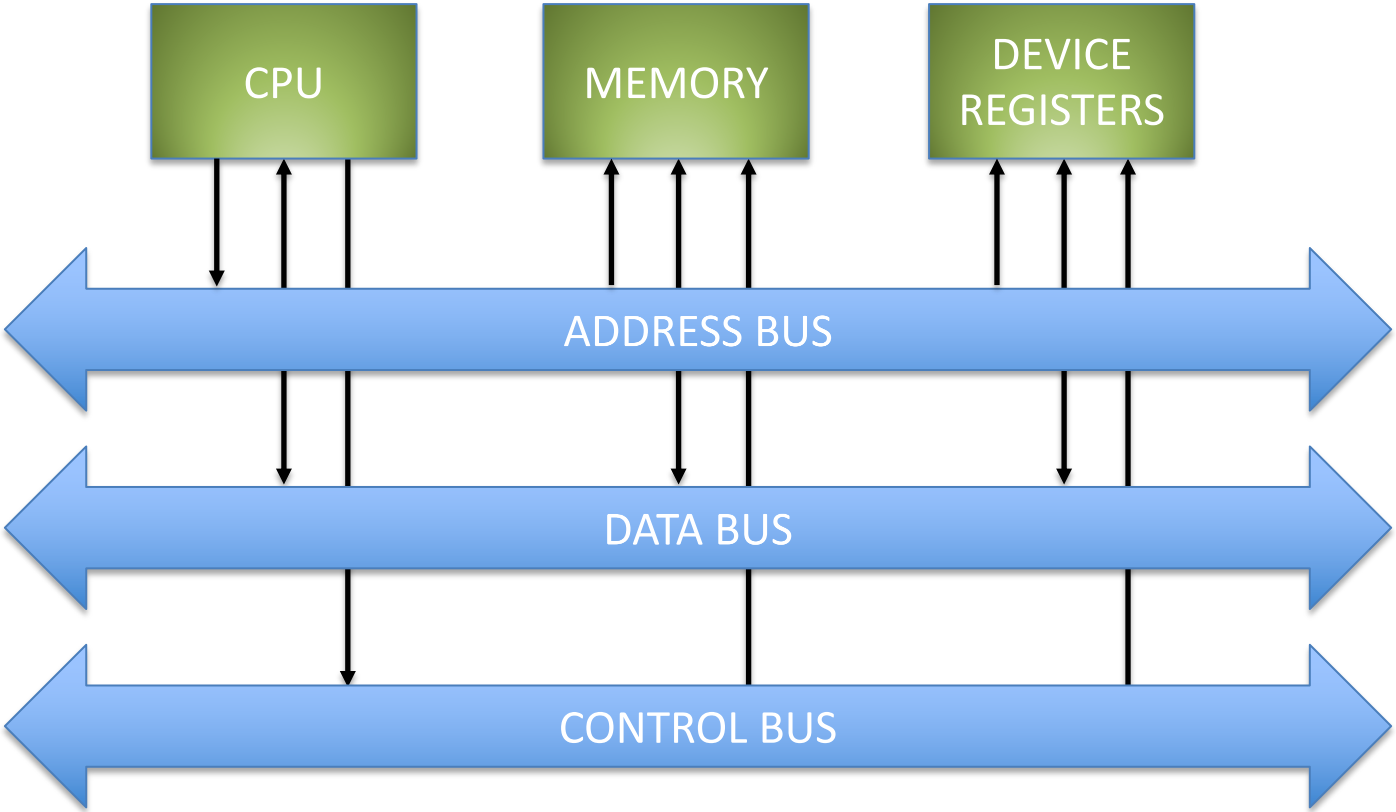
CS 4410

Operating Systems

Cornell CIS
COMPUTING AND INFORMATION SCIENCE

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

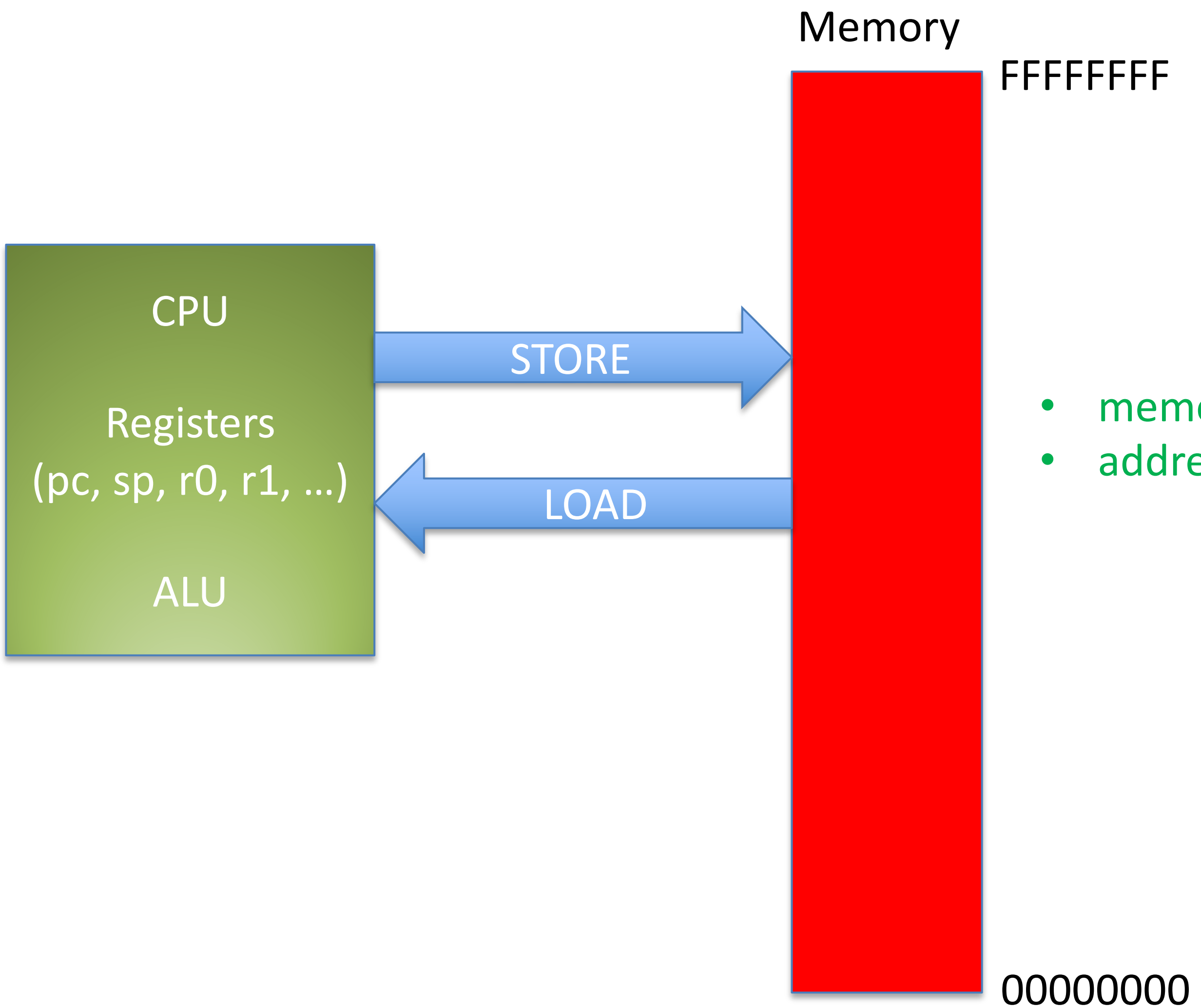
Architecture Diagram



“Bus”

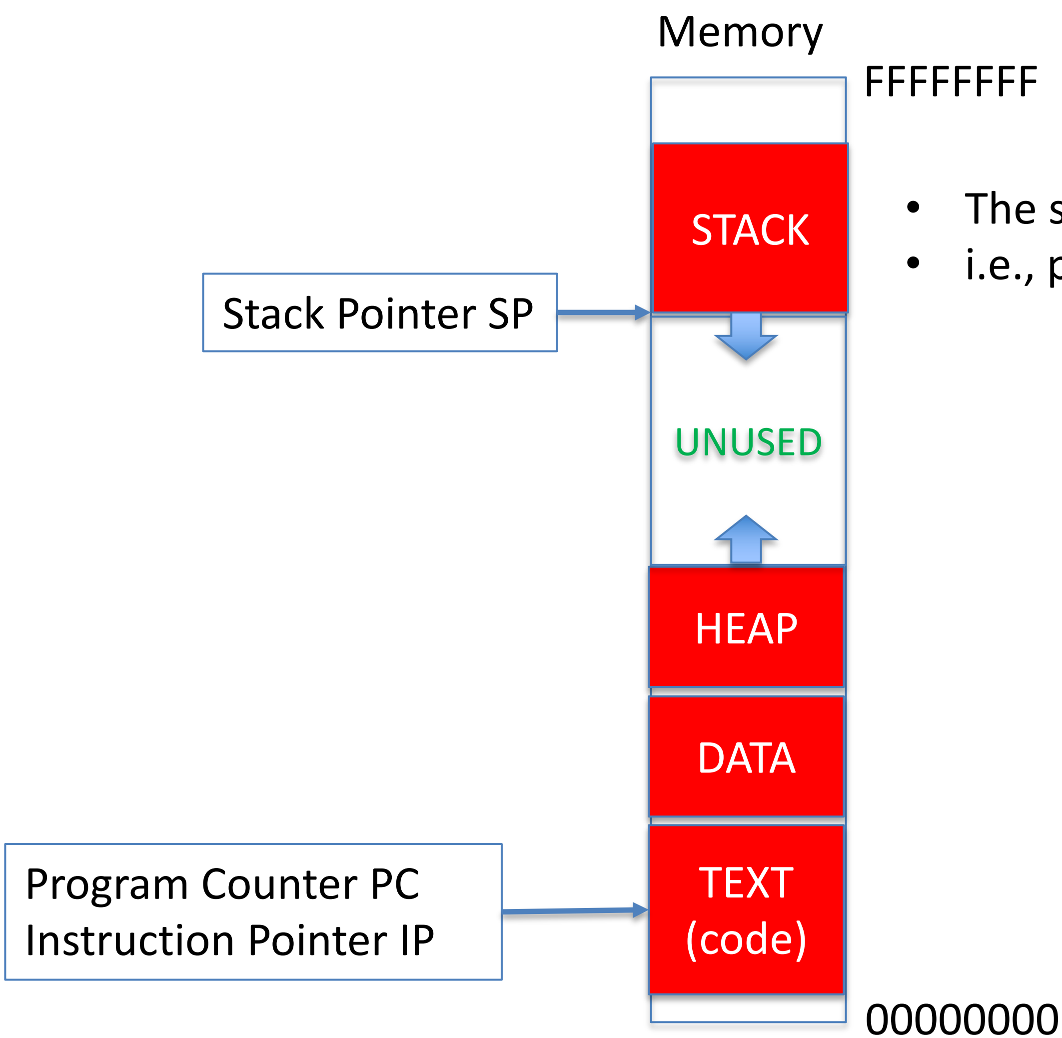
- Collection of “lines” (wires)
- Control bus: Load/Store/Interrupt/…
- Data bus: *x* lines → word is *x* bits
  - e.g: 32 lines: word is 32 bits (4 bytes)
- Address bus: *y* lines → address is *y* bits
  - process can address at most 2^y bytes

Logical View of CPU and Memory



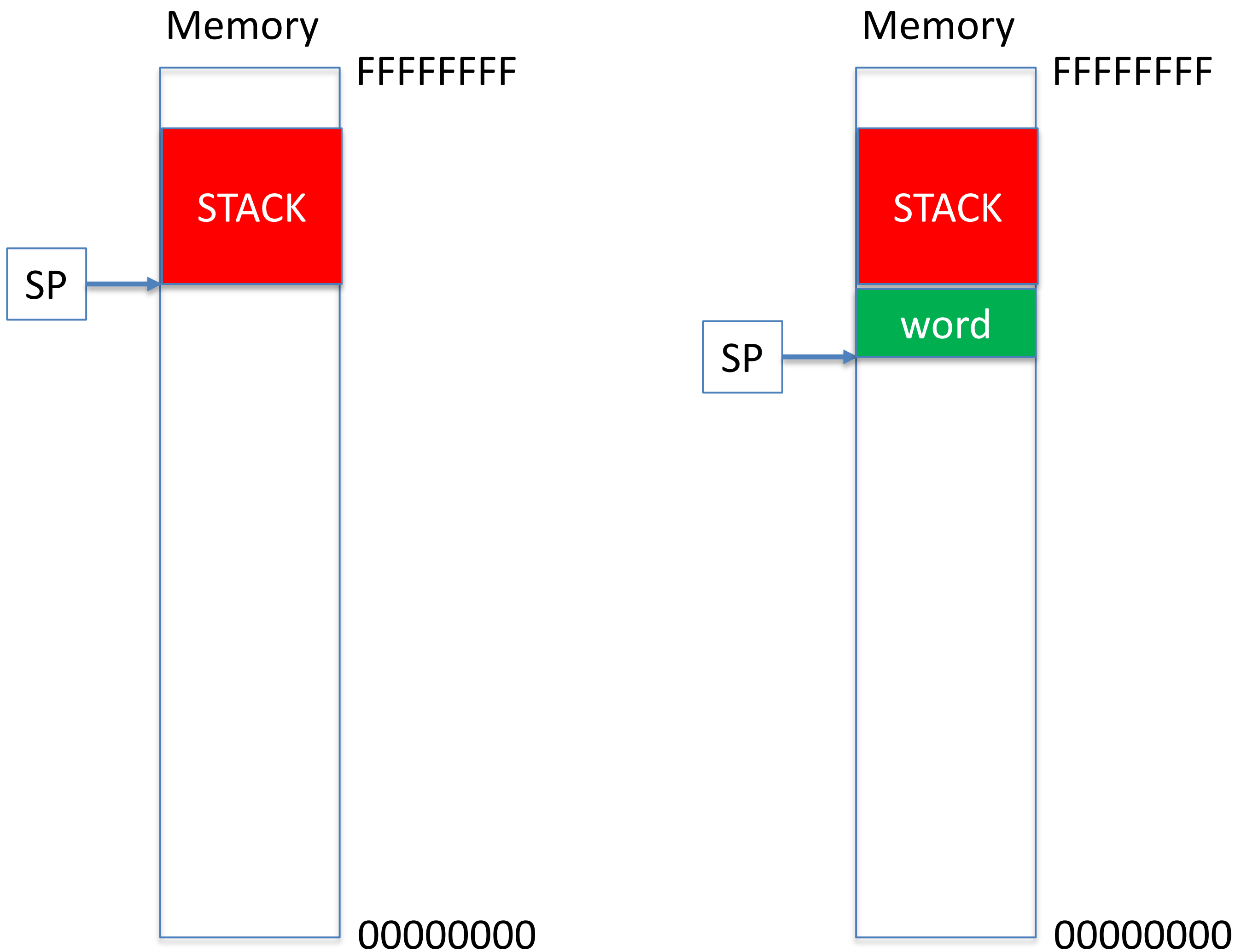
- memory is an array of bytes
- address is an index into the array

Memory "segments"

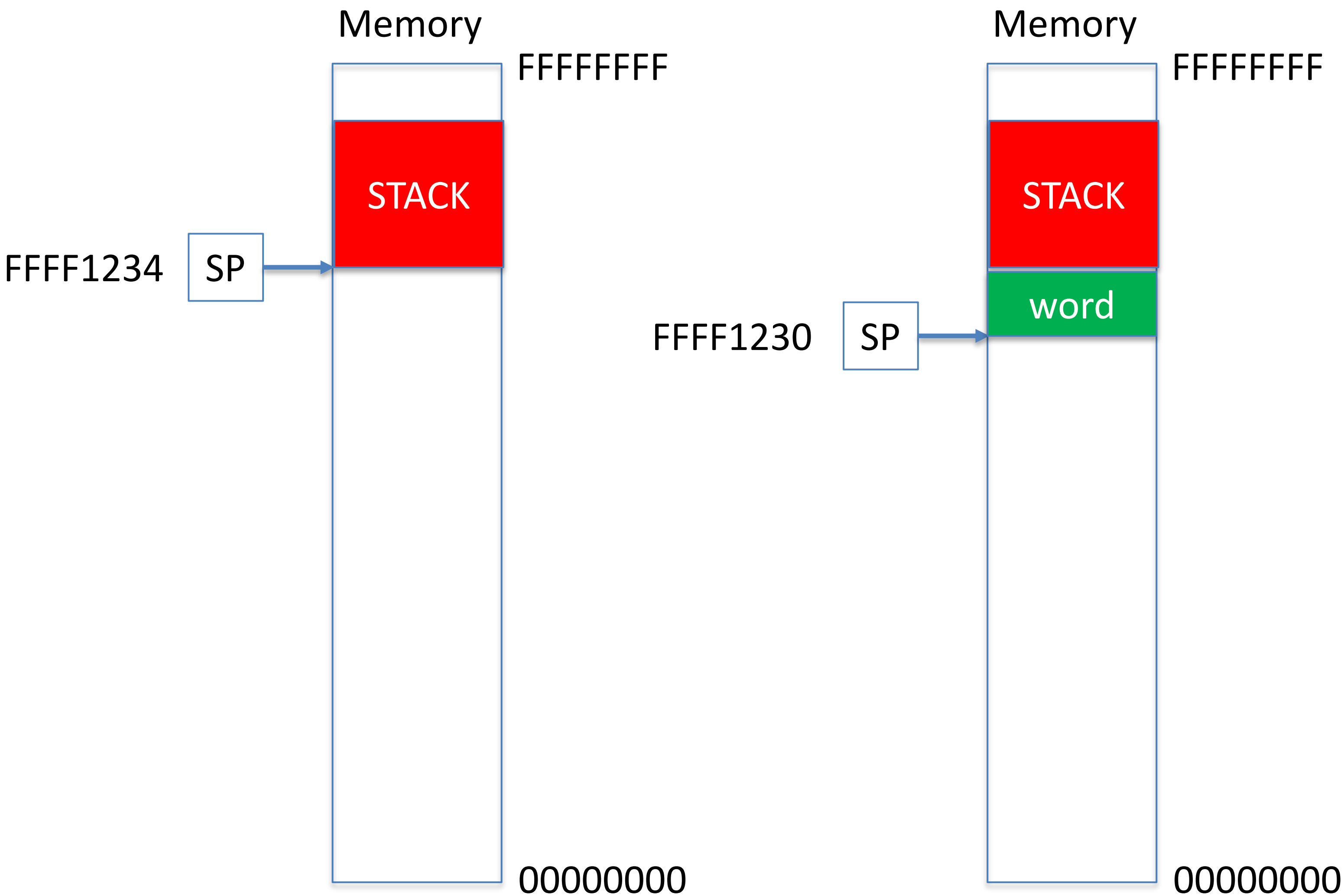


- The stack is usually word-aligned
- i.e., push and pop words

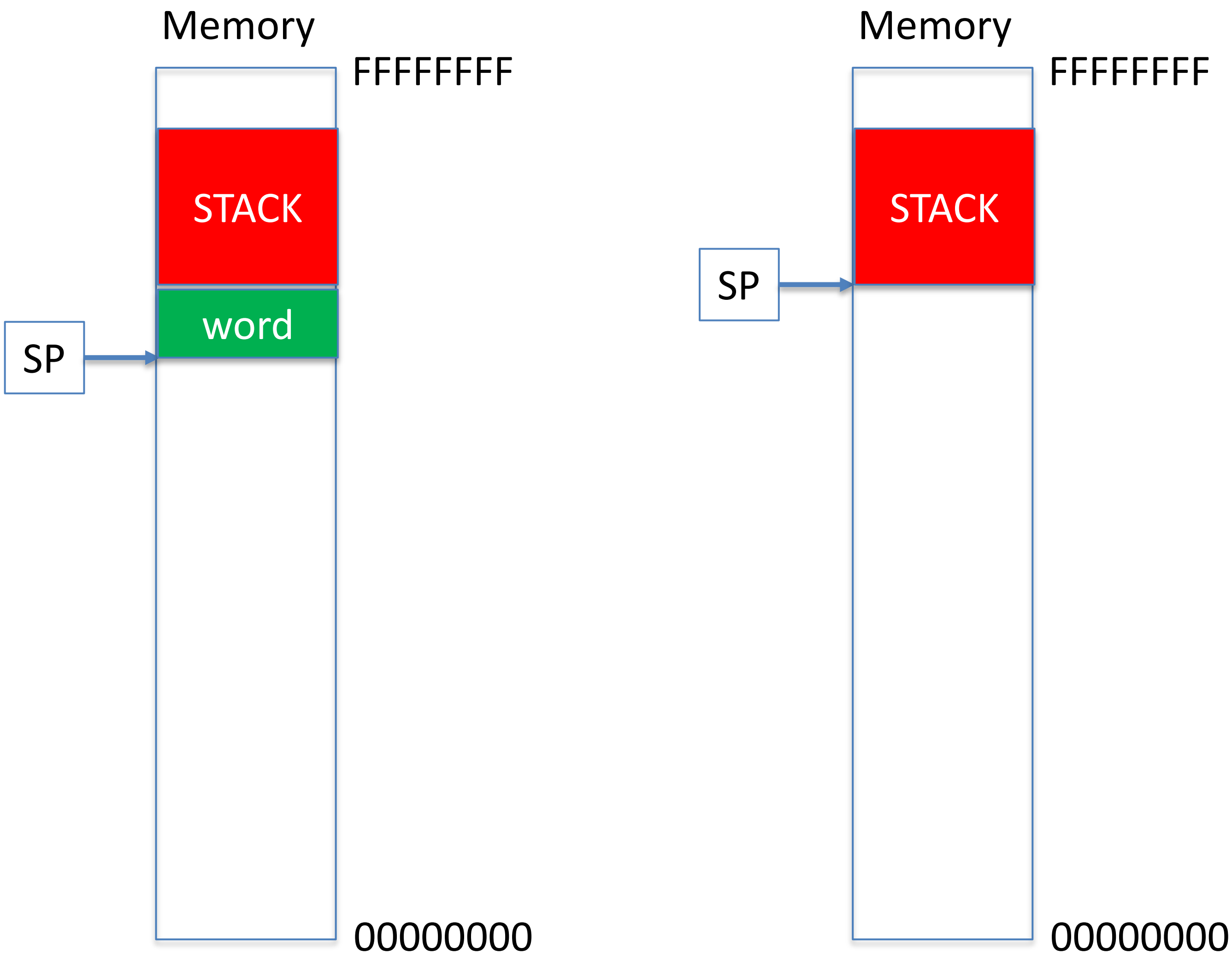
Stack before and after Push



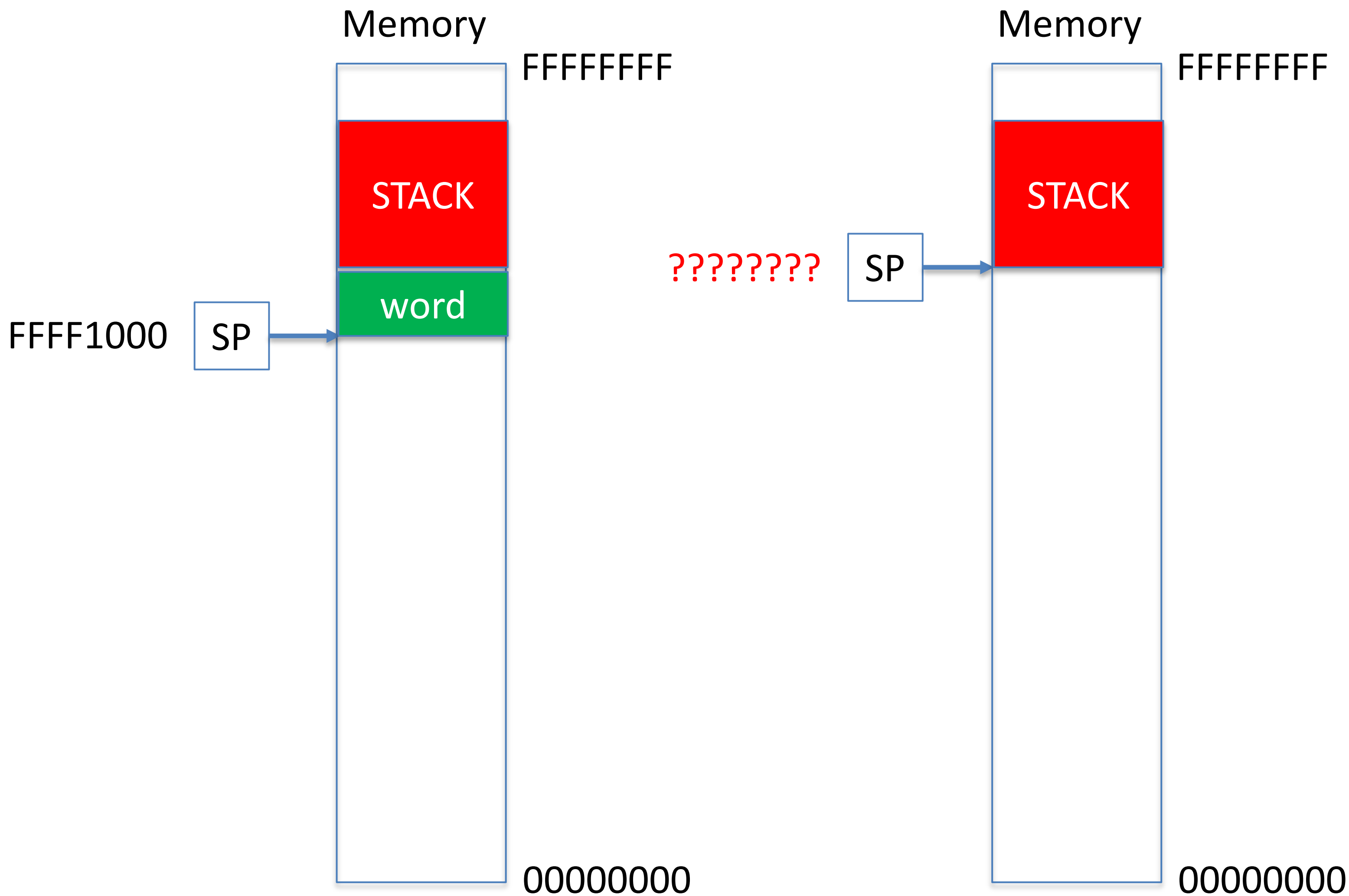
Stack before and after Push



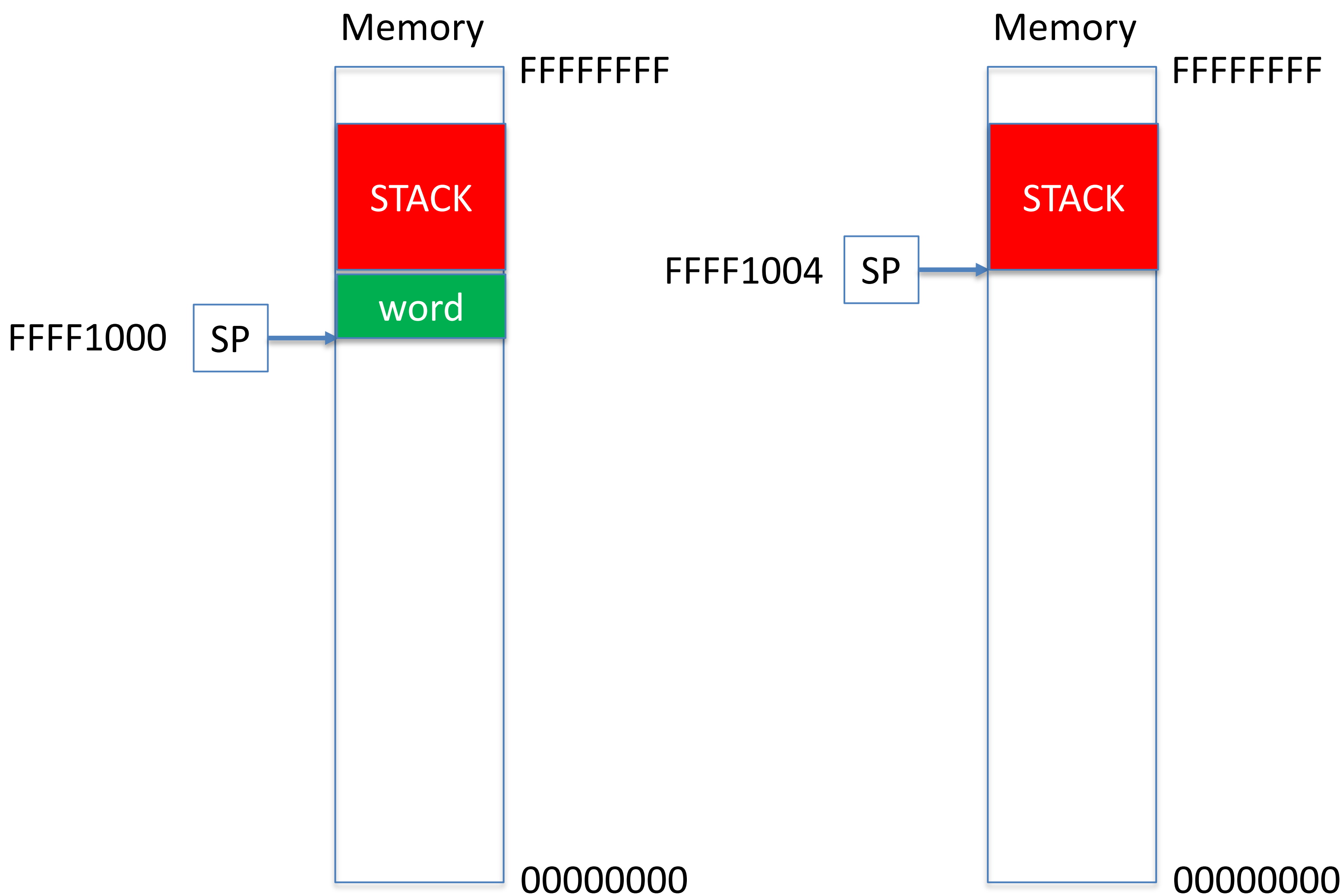
Stack before and after Pop



Stack before and after Pop



Stack before and after Pop



Control Flow and the Stack

- **call** f**:**
  - saves return address (where??)
  - 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**
  - restores return address (from where??)

Return Address

- x86: return address pushed onto stack
  - allows for nested calls automatically
- RISC-V, ARM: saved in special register
  - caller is responsible for saving and restoring the register *if needed*, which it usually does on the stack

Net result is the similar for nested calls!

Arguments and Return values

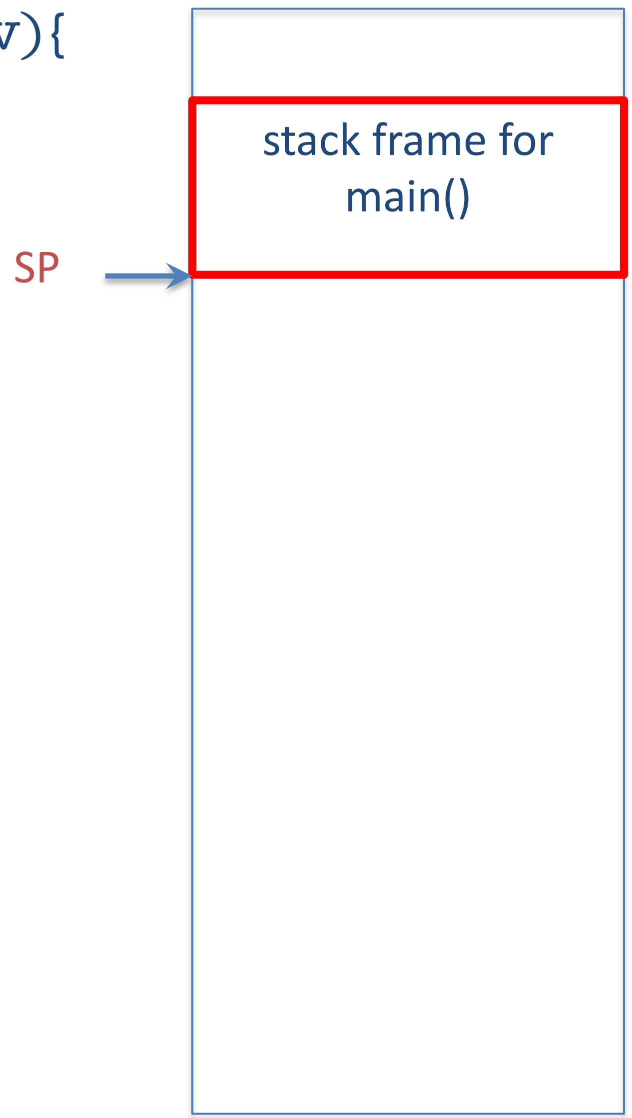
- Arguments are usually passed in registers for efficiency
- If there are too many arguments, rest is passed by pushing them onto the stack
- The return value is usually stored in a dedicated register

Control Flow

```
int main(argc, argv){
      ...          ← PC/IP
      f(3.14)
      ...
}

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

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

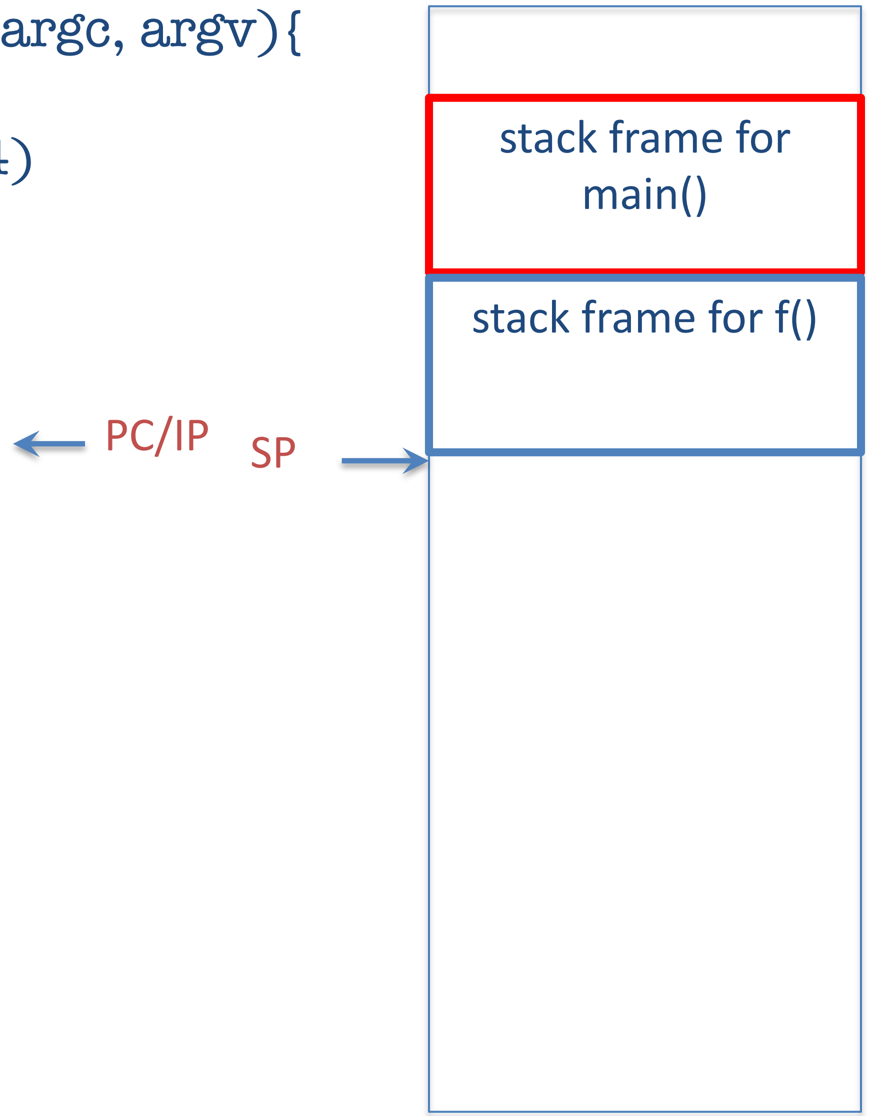


Control Flow

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

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

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

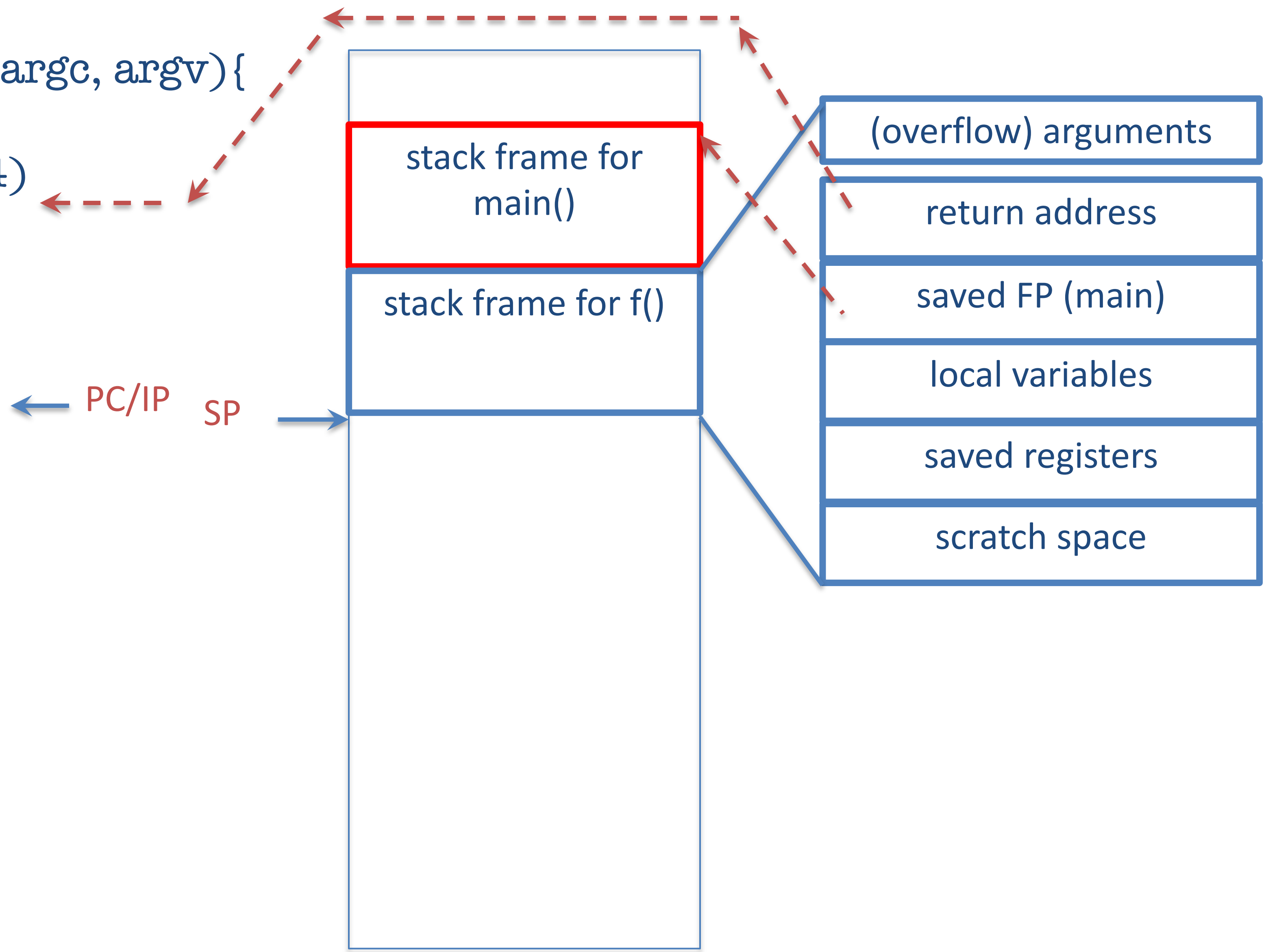


Control Flow

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

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

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

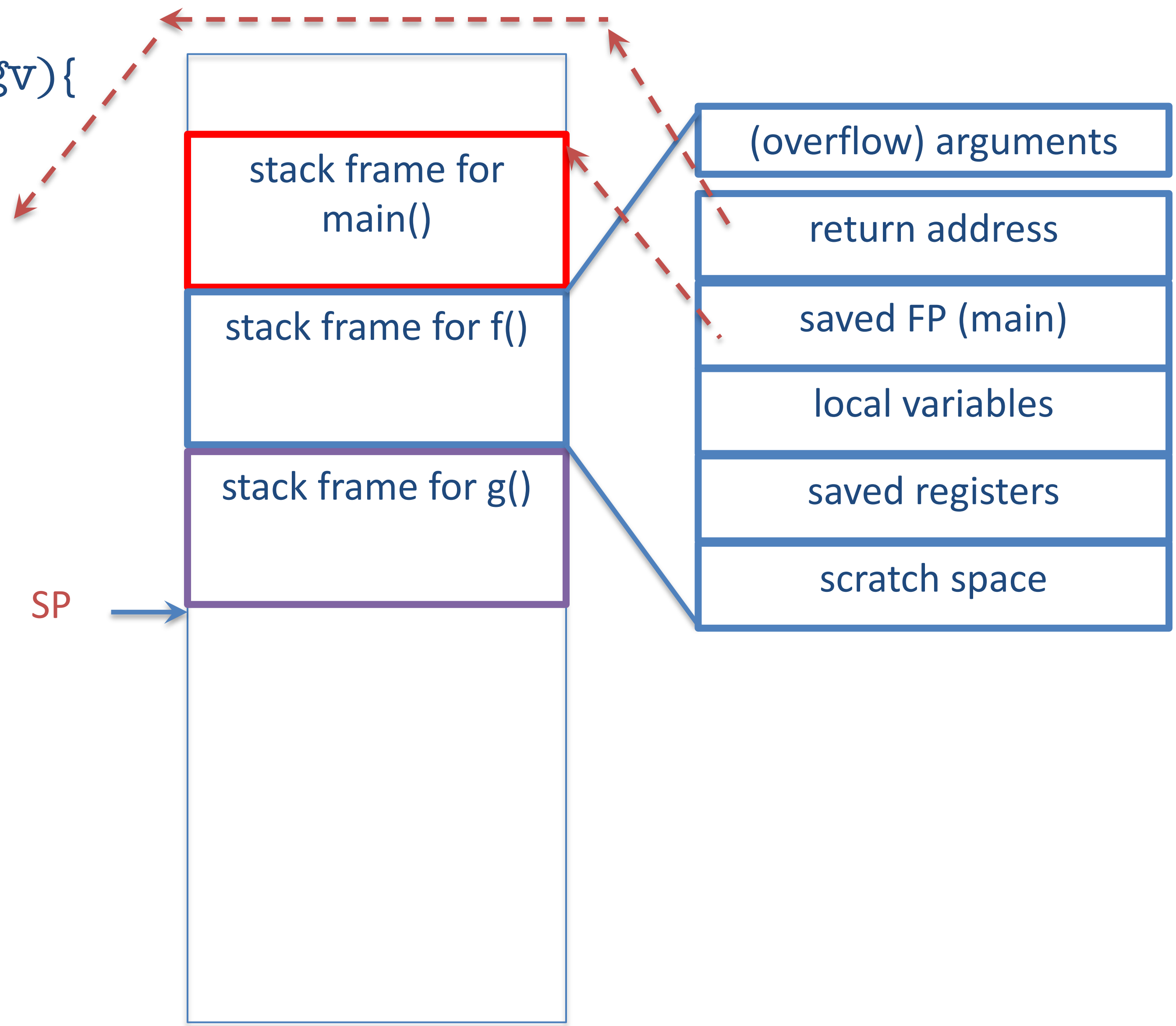


Control Flow

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

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

int g(y){
    ...        ← PC/IP
}
```

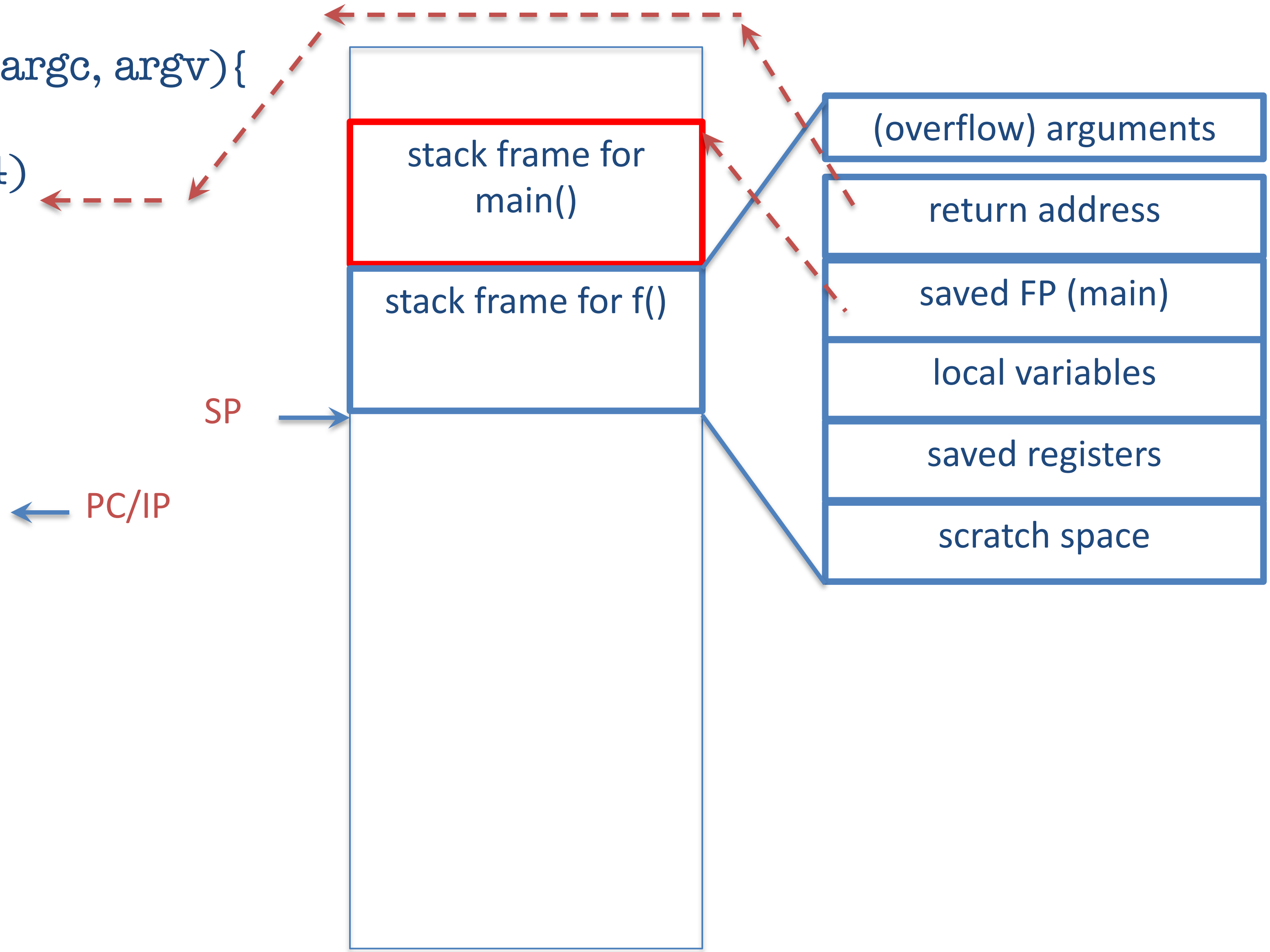


Control Flow

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

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

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



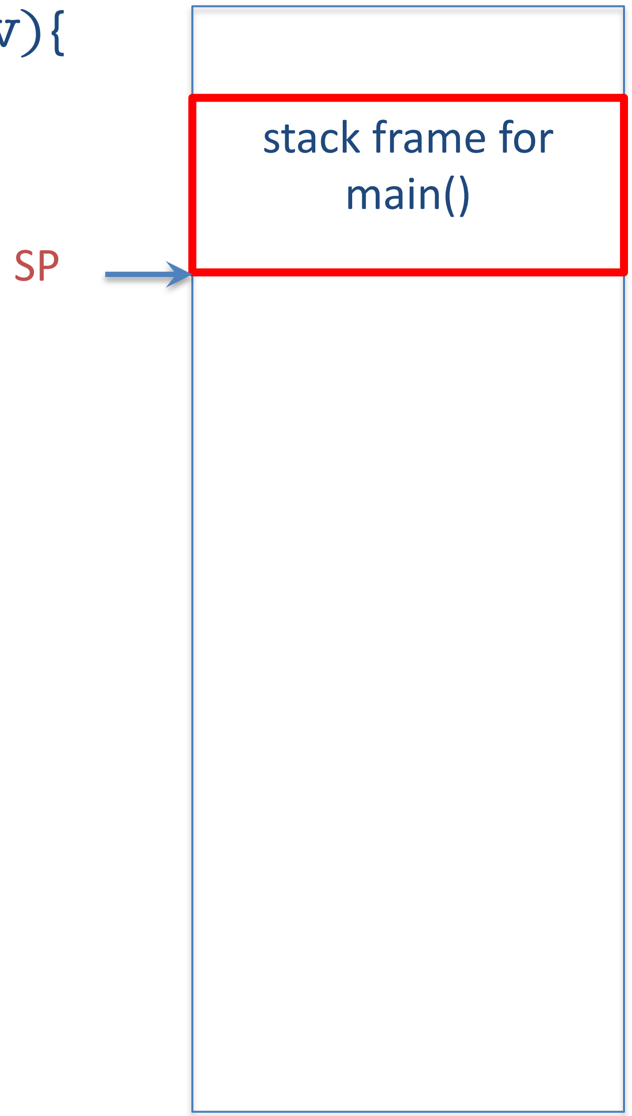
Control Flow

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

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

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

PC/IP



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

How does the kernel get control?

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

there is no “main loop”

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

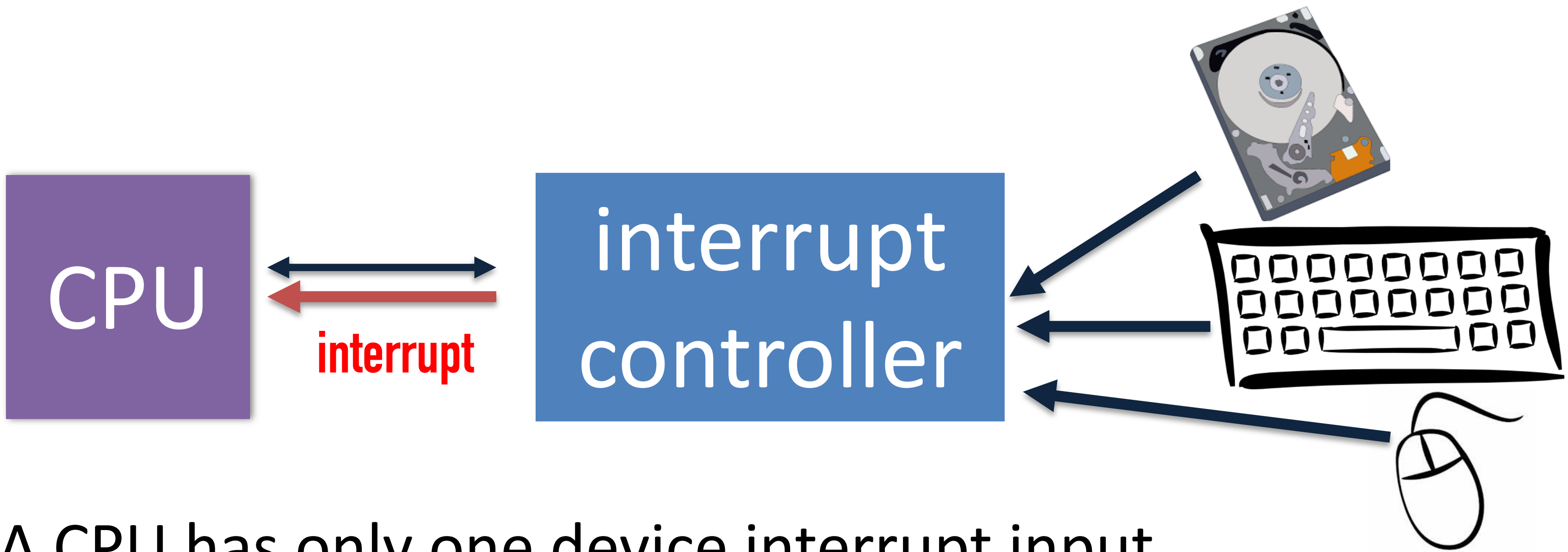
(Device or I/O) Interrupts

- 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)

- On signal, hardware:
  1. Saves certain state that is modified by the interrupt
     - **program counter and mode**
     - *where?* (depends on hardware)
  2. disables ("masks") device interrupts
     - at least interrupts of the same device
  3. sets supervisor mode (if not set already)
  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
…

Interrupt Handling, cont'd

"return from interrupt" instruction:

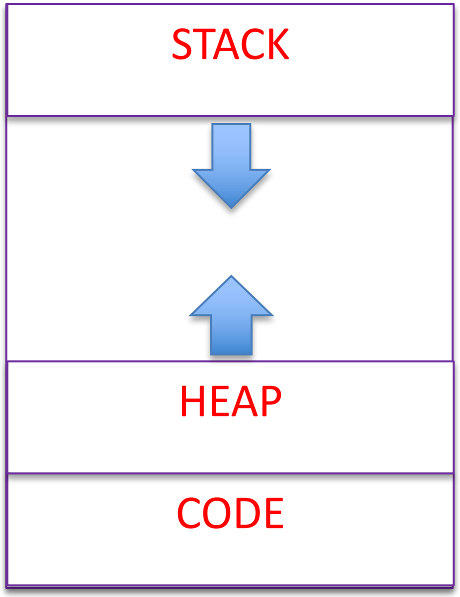
- Restores mode
- Restores program counter / instruction pointer
- Re-enables interrupts

Two Stacks involved!

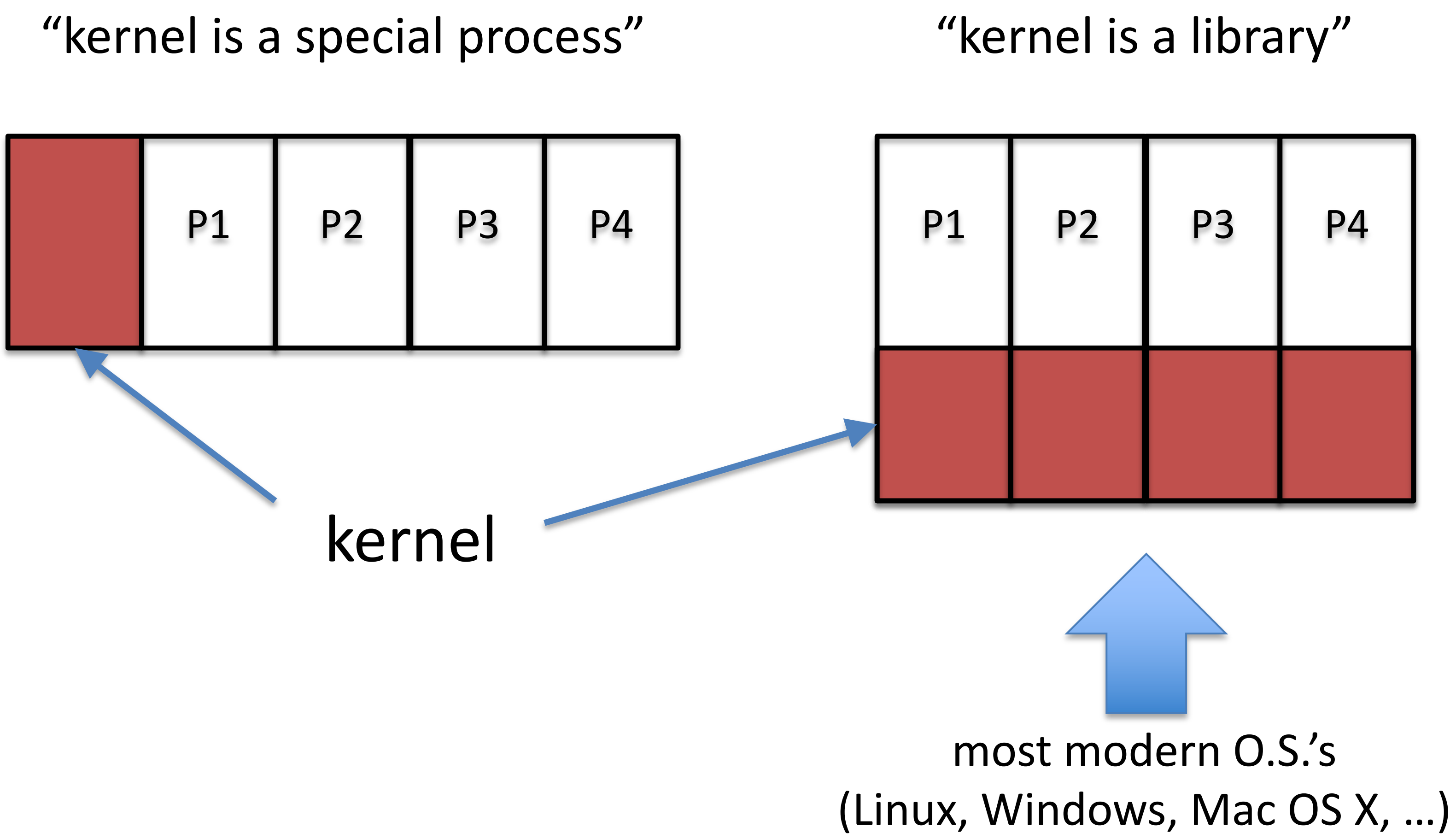
- User process has a stack to maintain control flow
- Kernel has its own control flow thus also needs a stack
- Why can’t we use the same one?

Reasons for separating user stack / supervisor stack

- 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 stack:
  - 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



Two architectures of O.S. kernels



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 process state explicitly)	Kernel implemented using "thread-based programming" (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?

Summary

- The ”kernel” is code that runs in supervisor mode
- A ”process” is code that runs in user mode
  - always with interrupts enabled
- Each has its own segments (code, data, heap, stack) and its own registers (pc, sp, r1, r2, …)
  - both are “virtual” (CPU does not know)
- Switching between modes
  - user mode → supervisor mode
    - signal: interrupt, system call, exception/fault
  - supervisor mode → user mode
    - return-from-interrupt instruction

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 process registers as the kernel code needs to use the registers to handle the interrupt
- Registers are typically saved on the interrupt stack but can be stored anywhere in the PCB
- 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

Typical Interrupt Handler Code

```
HandleInterruptX:
    PUSH %Rn         only need to save registers not
    …                saved by C functions
    PUSH %R1
    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;
    …
    }
}
```

Signal handling: View from the process

- (Device) Interrupts
  - usually invisible to running process. Process is restored to its prior state, including program counter
  - certain interrupts may be passed on to process
    - <control>C
    - process that has requested a timer interrupt
- System calls
  - process is usually modified in some ways
    - dedicated register contains result of system call
    - memory may have been modified (e.g., when reading from file)
- Exceptions (divide-by-zero, illegal address, etc.)
  - process is usually terminated
  - process can set up a handler if it so desires

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*
   - *"save" program counter, interrupts enabled bit (enabled), supervisor mode bit (user mode)*
5. clear all other registers
6. 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