Short History of Operating Systems

CS 4410

Operating Systems

Cornell CIS

COMPUTING AND INFORMATION SCIENCE

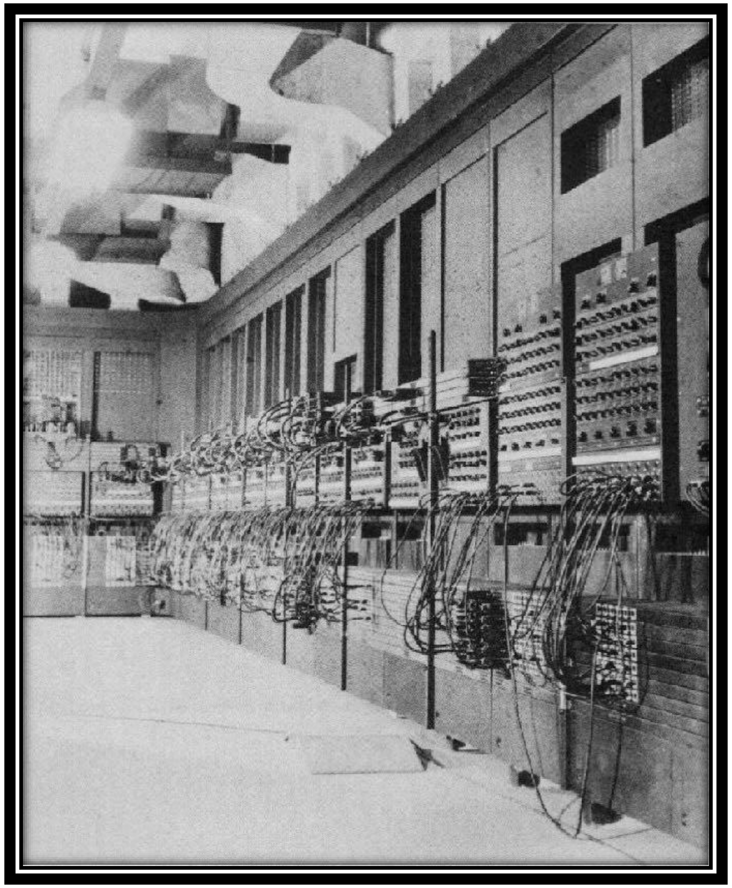
[R. Agarwal, L. Alvisi, A. Bracy, M. George,
F. B. Schneider, E. G. Sirer, R. Van Renesse]

PHASE 1 (1945 – 1975)

COMPUTERS EXPENSIVE, HUMANS CHEAP

Early Era (1945 – 1955):

- First computer: ENIAC
  - UPenn, 30 tons
  - Vacuum tubes
  - card reader/puncher
  - 100 KHz, 5000 additions/second
  - 100-word memory added in 1953
- Single User Systems
  - one app, then reboot
- "O.S" = loader + libraries
- Problem: Low utilization



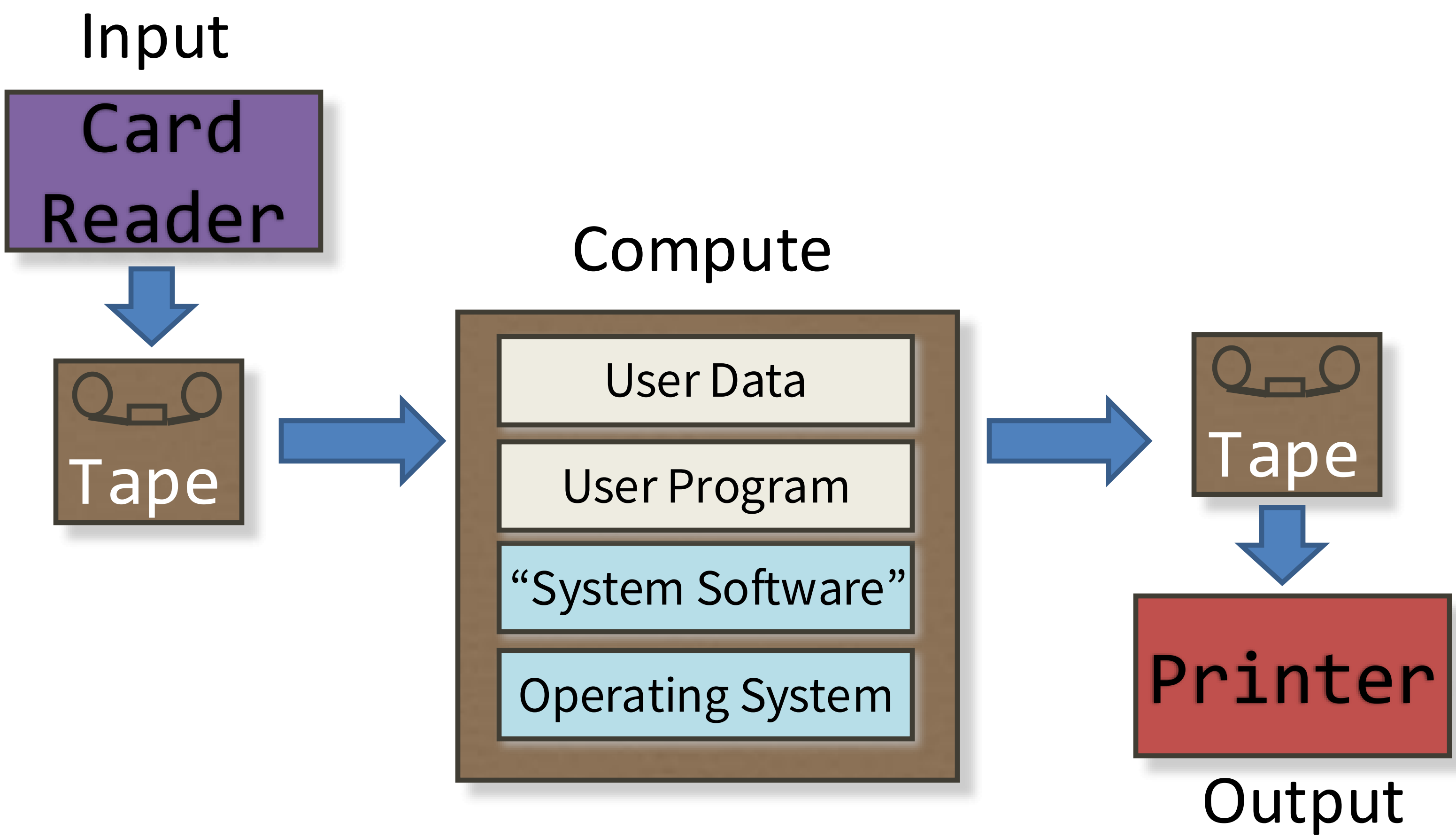
Batch Processing (1955 – 1960):

- First Operating System: GM-NAA-I/O
  - General Motors research division
  - North American Aviation
  - Input/Output

- Written for IBM 704 computer
  - 10 tons
  - 40KIPS (40,000 instructions / second)
  - 4K word memory (about 18 Kbyte)

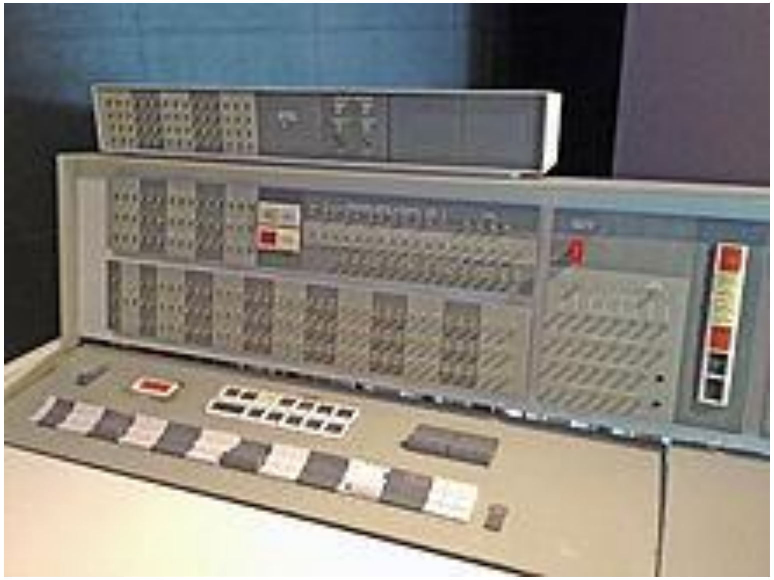
Batch Processing

- 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
  - (see Hidden Figures!!!)
- transistors!
- 500 KHz, 200 KIPS (68 KFLOPS)
- 32K word memory



Fernando J. Corbató (1926-2019)

Time-Sharing + Security (1965 –):

- Multics (MIT)
  - security rings
- GE-645 computer
  - 435 KIPS
  - hw-protected virtual memory
- Multics predecessor of
  - Unix (1970)
  - Minix (1987)
  - Linux (1990)
  - Android (2008)

PHASE 2 (1975 – 2020)

COMPUTERS CHEAP, HUMANS EXPENSIVE

Personal Computers (1975 –):

- 1975: IBM 5100 first “portable” computer
  - 55 pounds, 5’’ display (64x16 characters)
  - ICs, 16-64 KB, 1.9 MHz

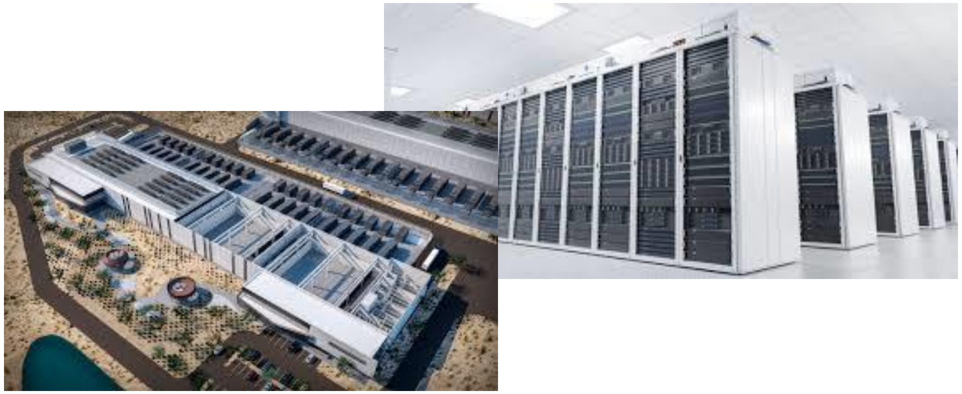
- 1977: RadioShack TRS-80 first “home” desktop
  - 12’’ display (64x16 characters as well)
  - 4-48 KB, 4 MHz

- 1981: Osborne 1 first “laptop”
  - 24.5 pounds, 5’’ display
  - 64 KB, 1.7 MHz

Modern Era (1990 –)

- Ubiquitous Computing / Internet-of-Things
  - Mark Weiser, 1988-ish

- Personal Computing
  - PDA ("PalmPilot") introduced in 1992
    - 512 KB, 16 MHz, 160x160 pixels
  - #computers / human >> 1
- Cloud Computing
  - Amazon EC2, 2006



Today's "winners" (by market share)

This is desktop + tablet + phone only

- 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
  - servers only a fraction of total computer market

PHASE 3 (2020 –)

COMPUTERS ←→ HUMANS ???

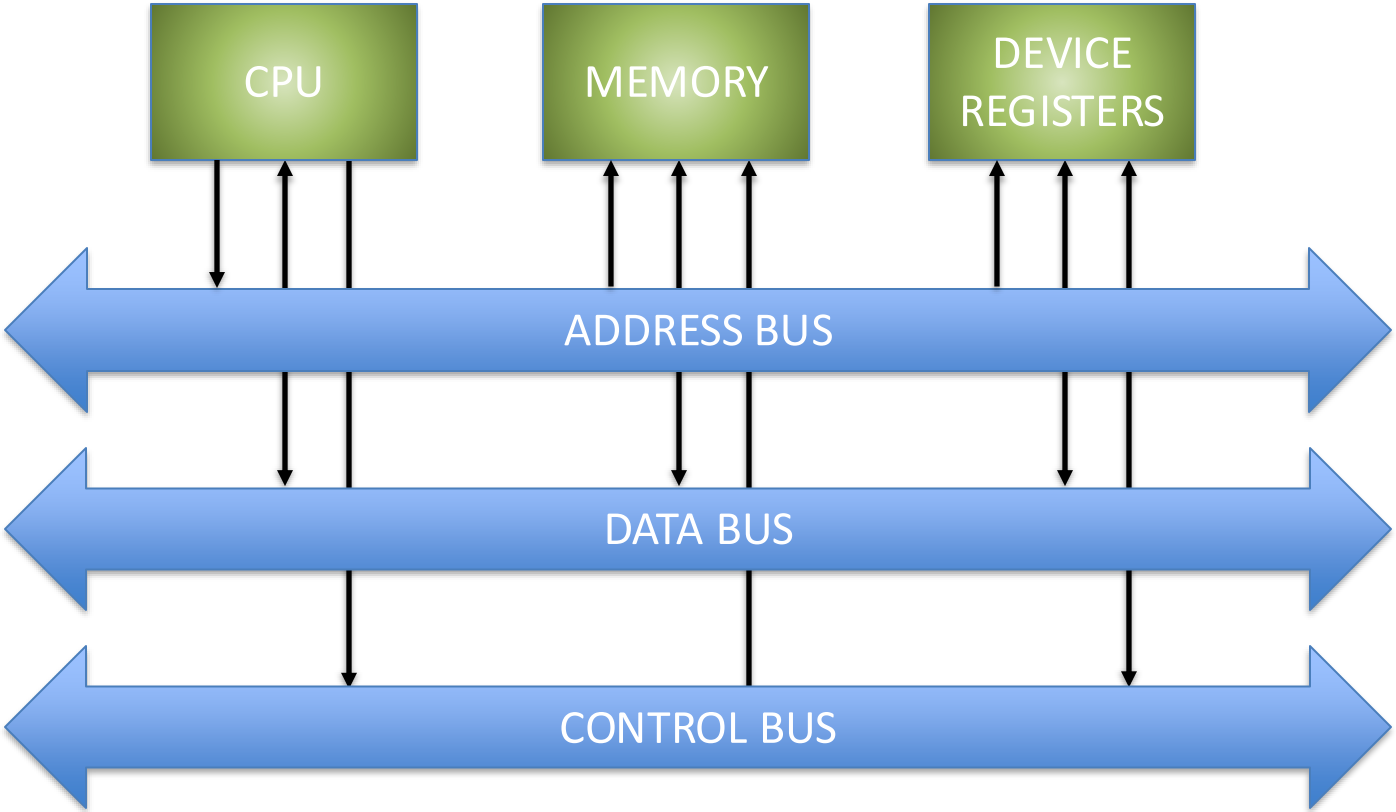
Anatomy of a Computer (simplified)

CS 4410

Operating Systems

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

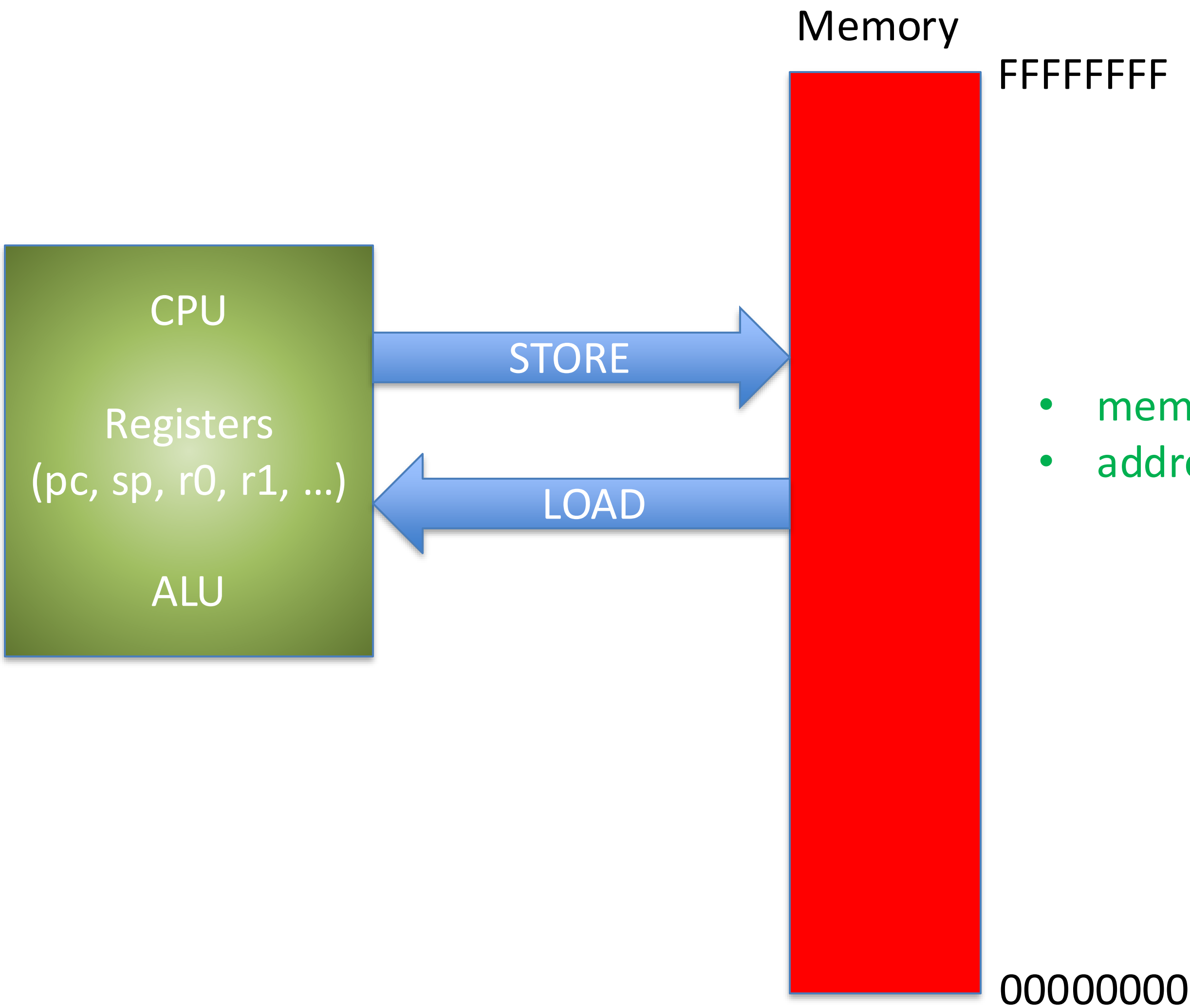
Architecture Diagram



“Bus”

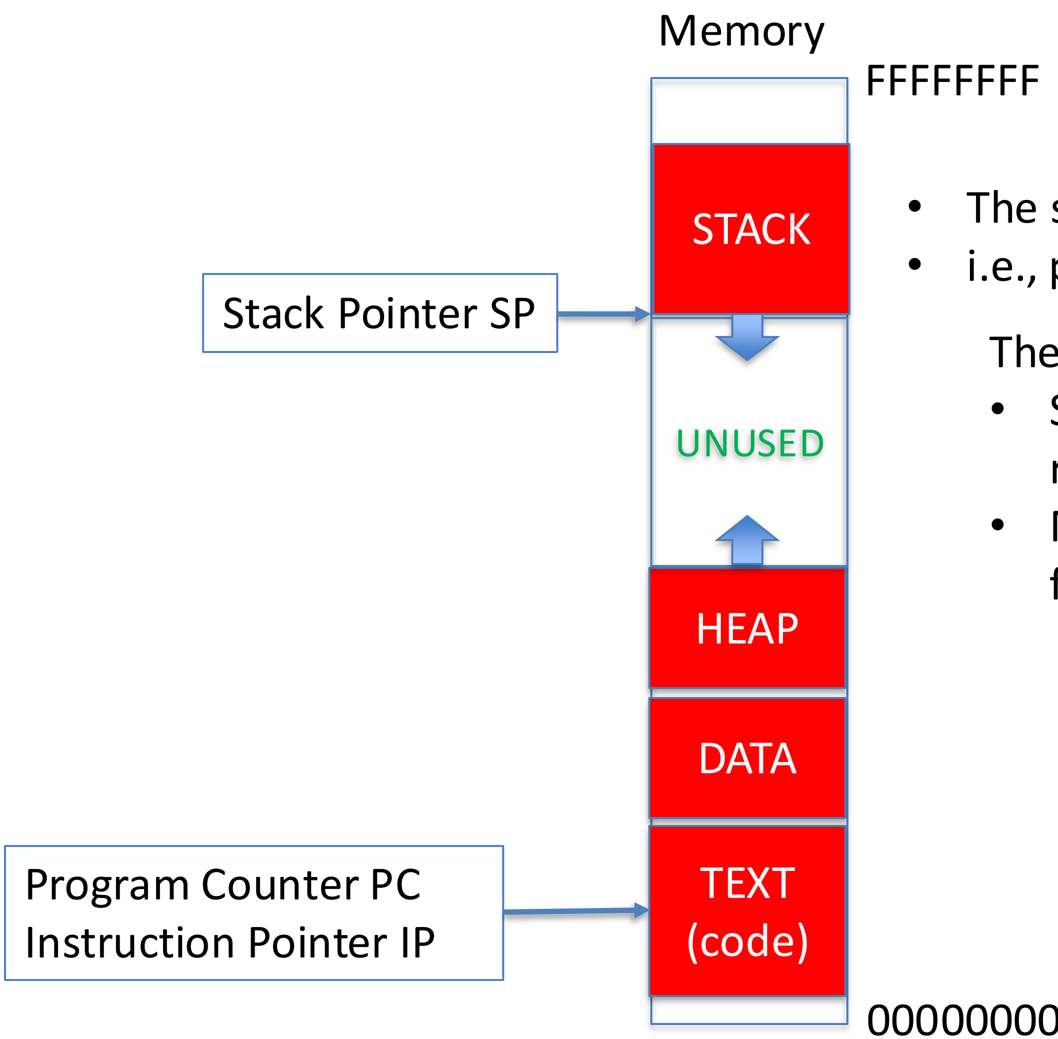
- Collection of “lines” (wires)
- Control bus: Load/Store/Interrupt/…
- Data bus: *x* lines (wires) → word is *x* bits
  - e.g: 32 lines: word is 32 bits (4 bytes)
- Address bus: *y* lines → address is *y* bits
  - process can address at most 2^y bytes

Logical View of CPU and Memory



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

Memory “segments”

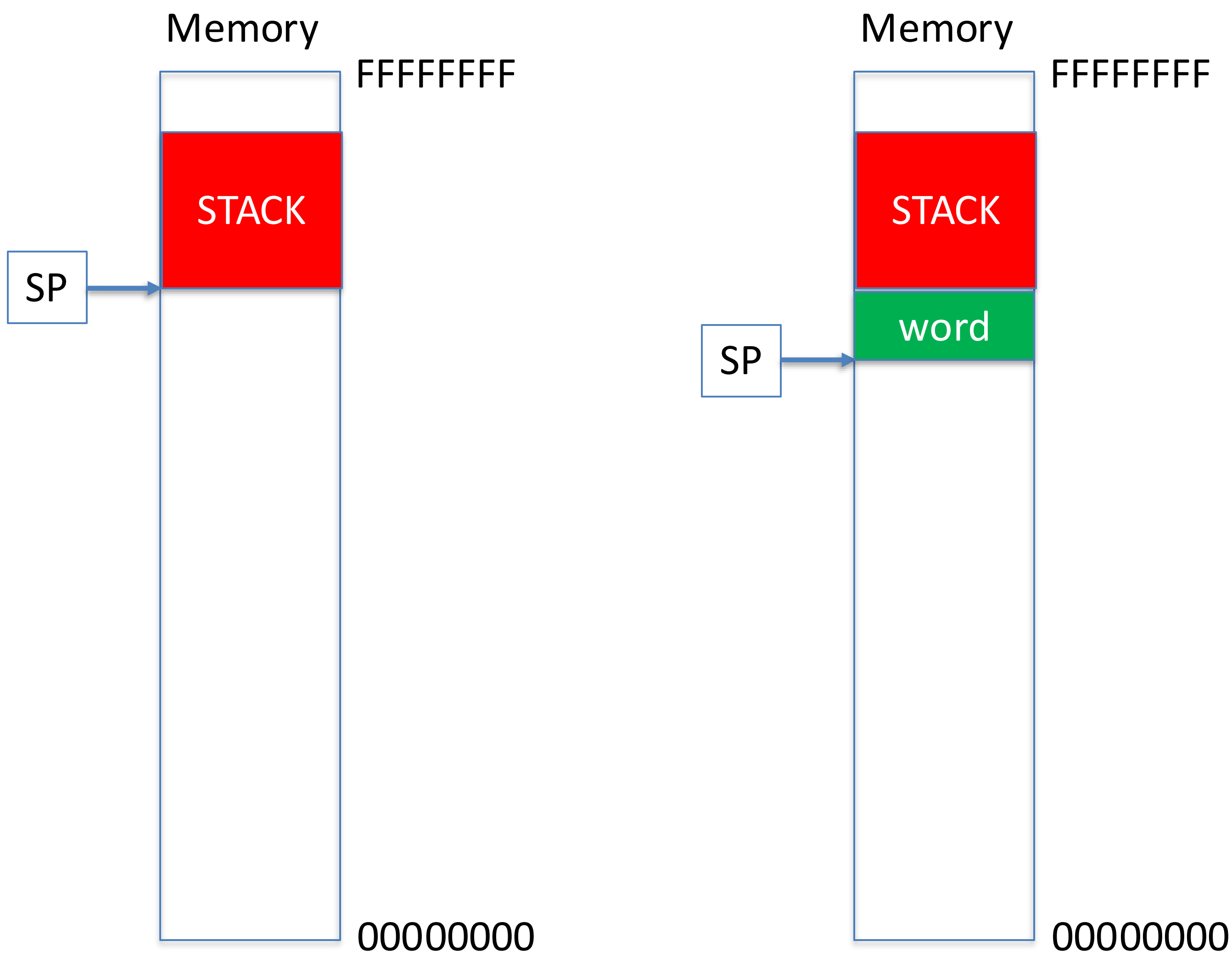


- The stack is usually word-aligned
- i.e., push and pop words

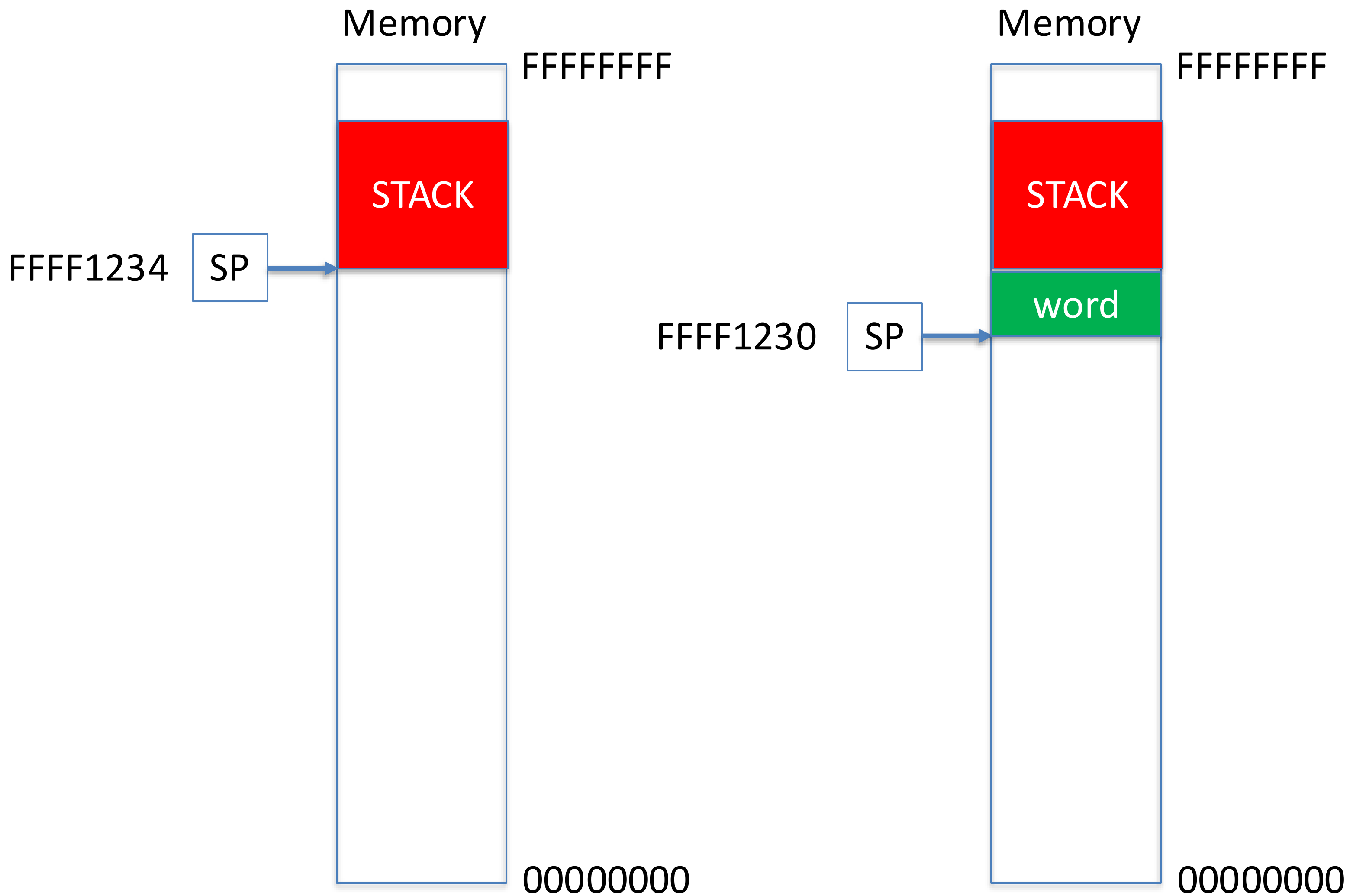
The stack is used for

- Storing intermediate results of computations
- Maintaining the control flow of the program

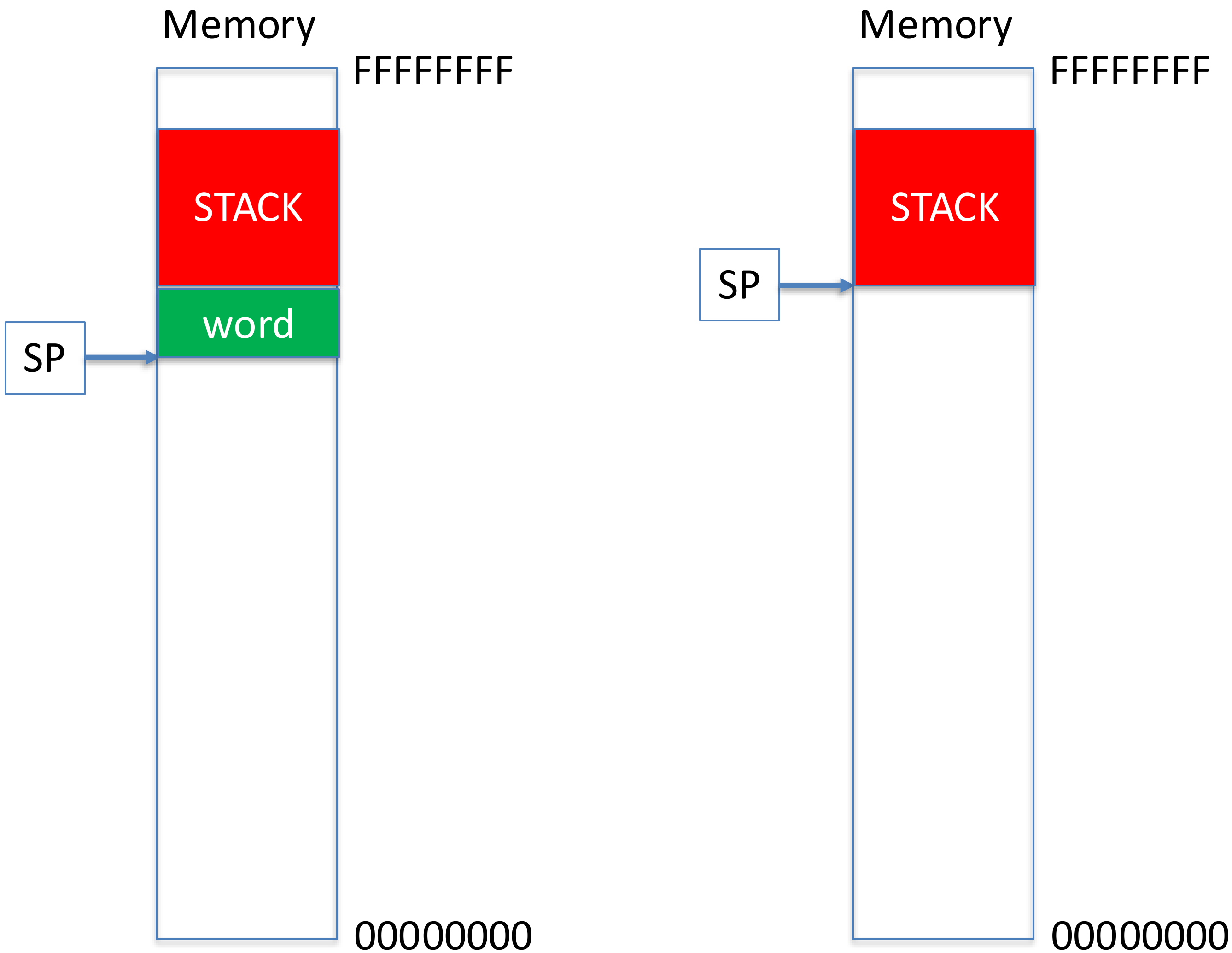
Stack before and after Push



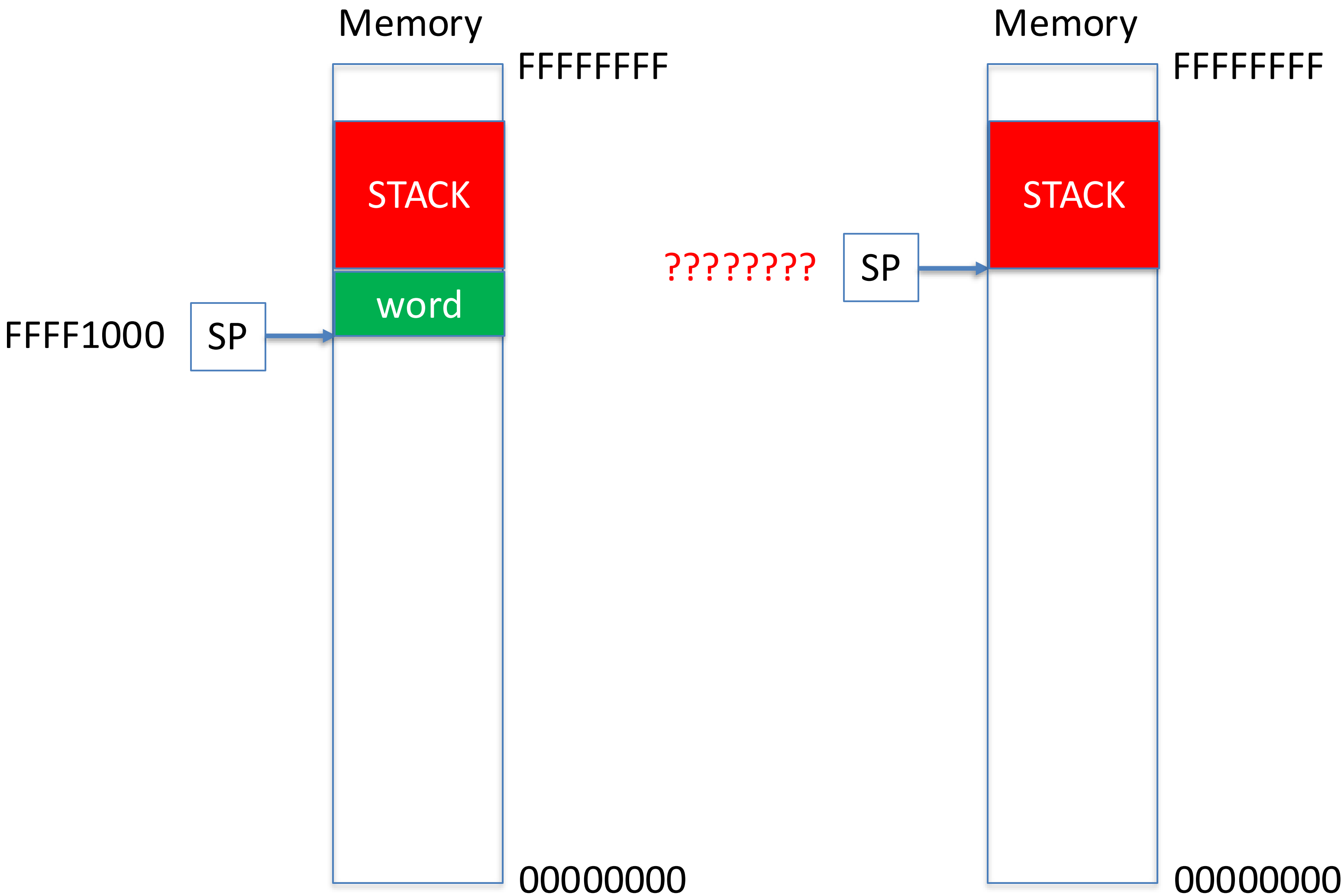
Stack before and after Push



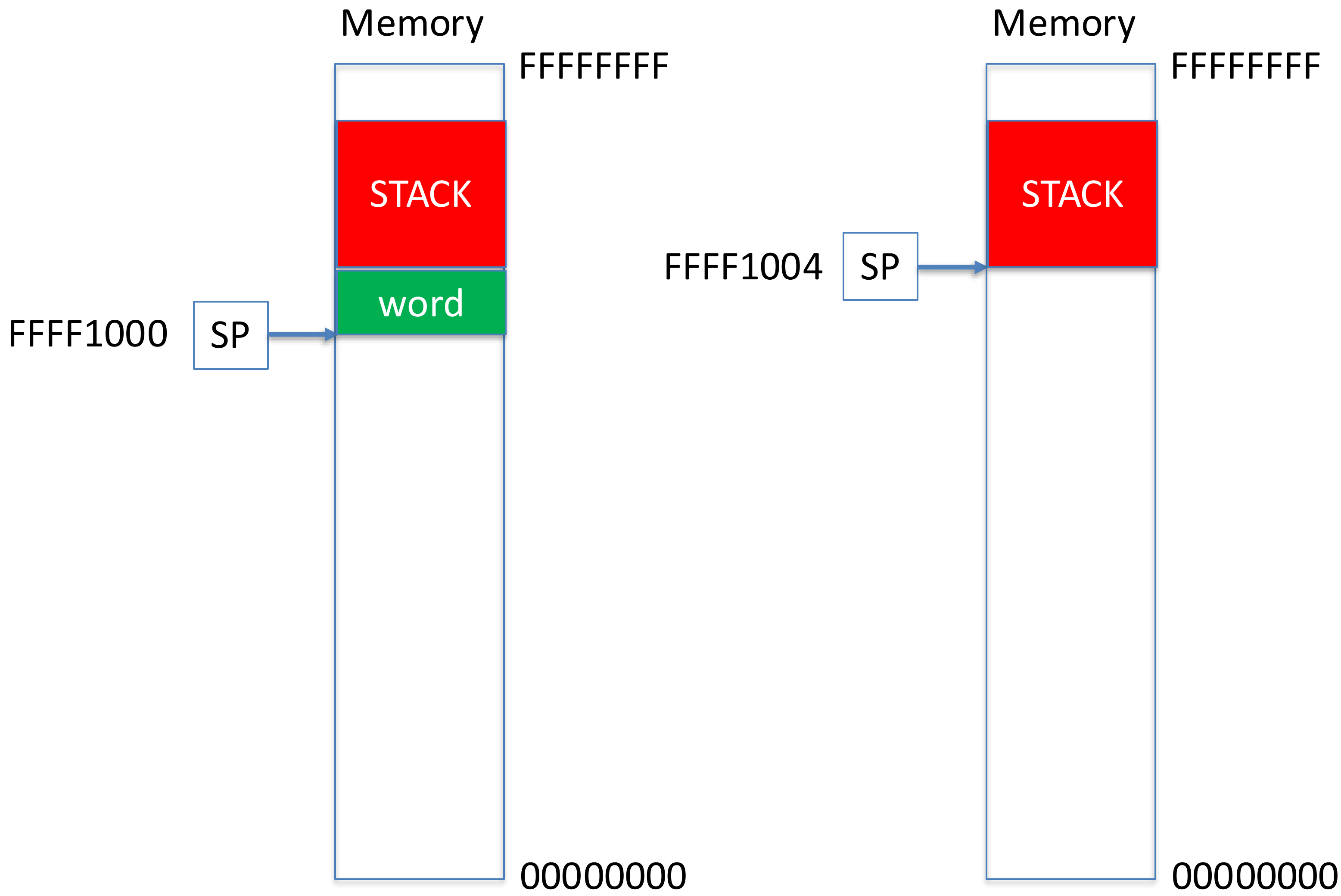
Stack before and after Pop



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 on the stack *if needed*

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

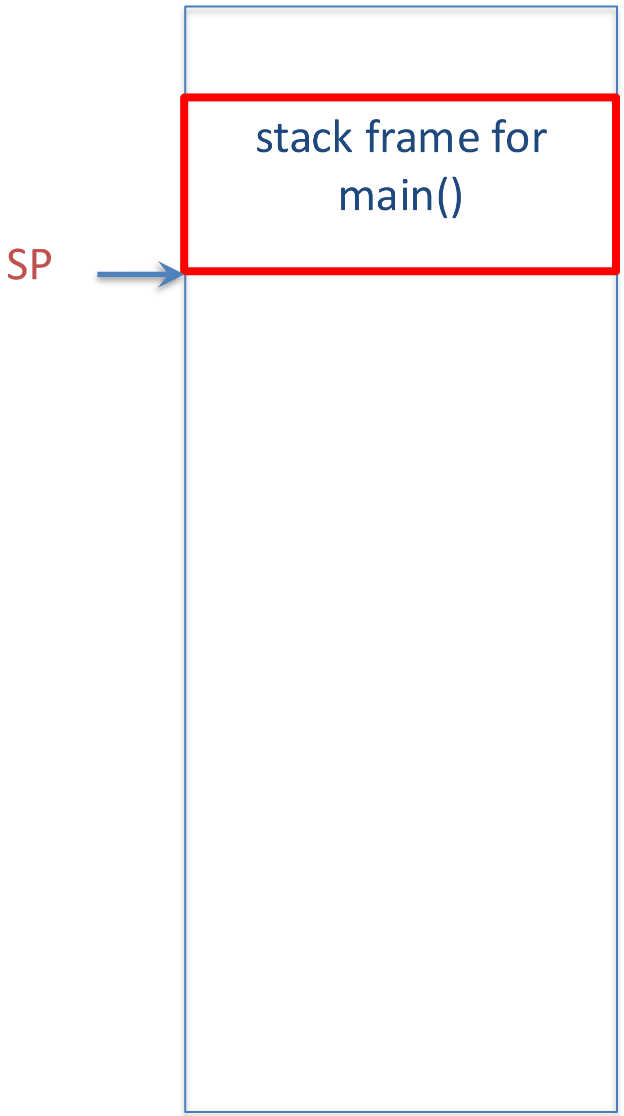
Control Flow

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

int f(x){
    …
    g();
    …
}

int g(y){
    …
}
```

PC/IP

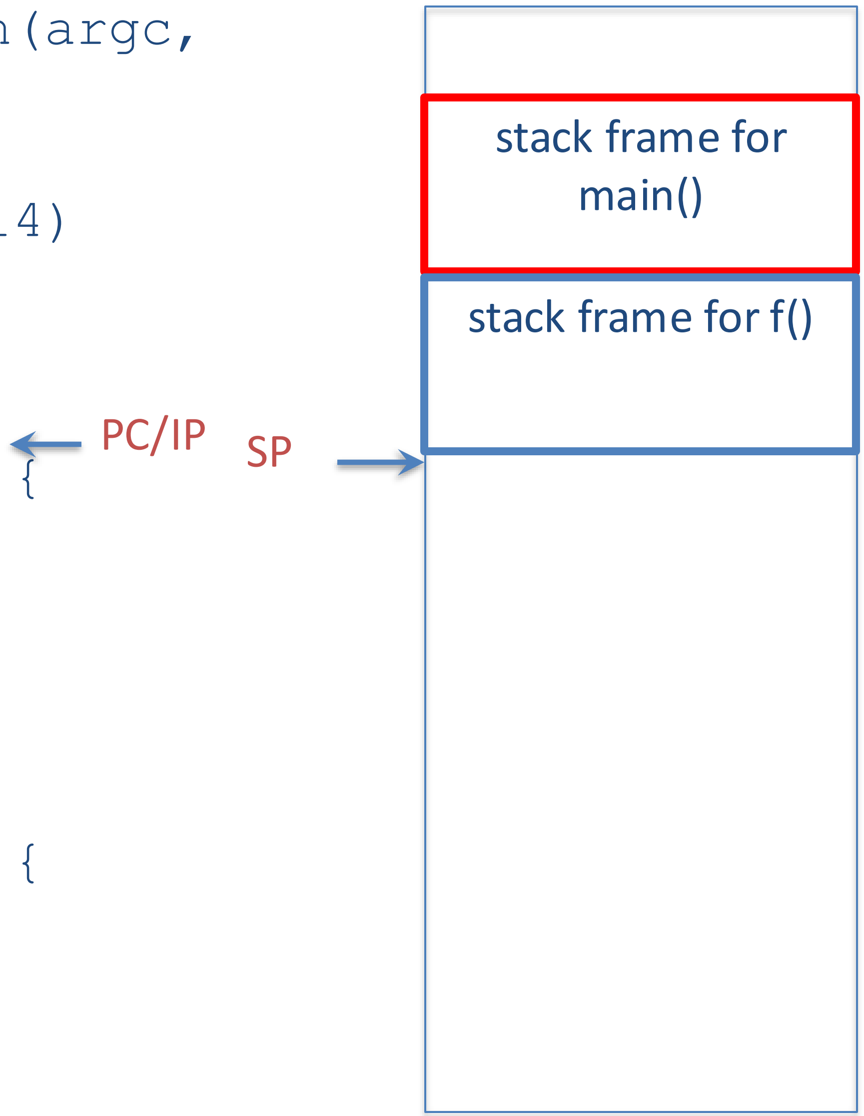


Control Flow

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

int f(x){
    …
    g();
    …
}

int g(y){
    …
}
```

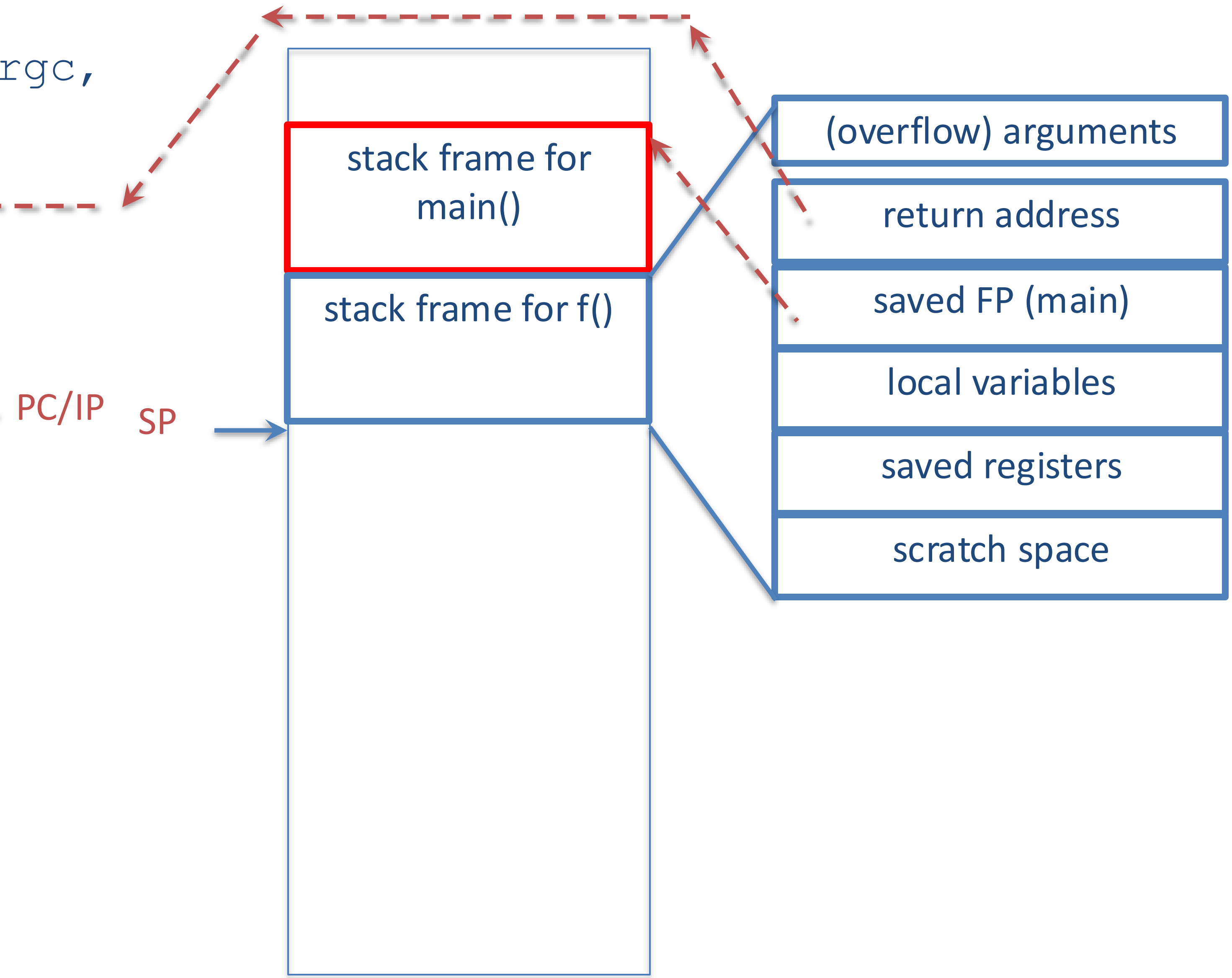


Control Flow

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

int f(x){
    …
    g();
    …
}

int g(y){
    …
}
```



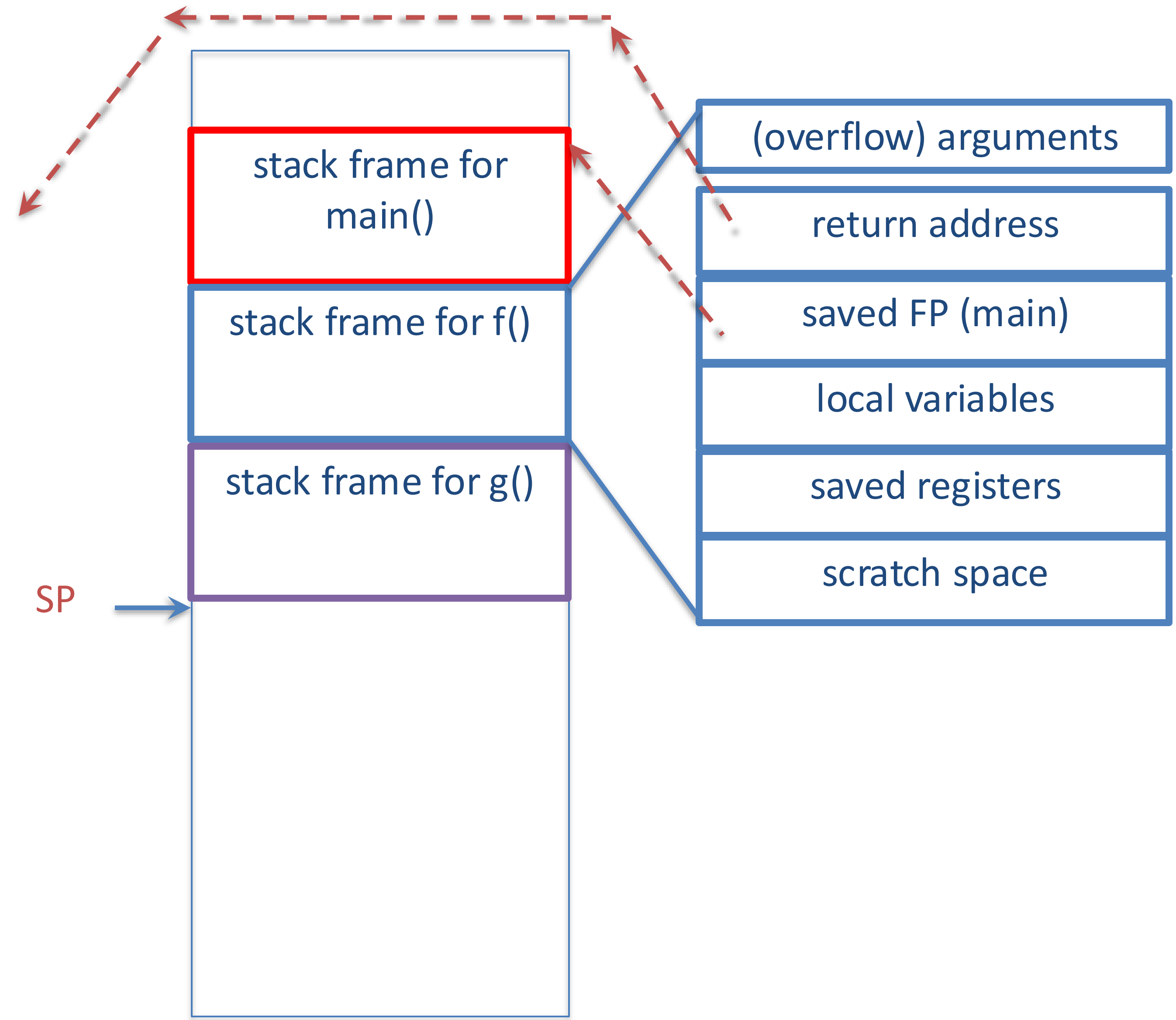
Control Flow

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

int f(x){
    …
    g();
    …
}

int g(y){
    …
}
```

← PC/IP

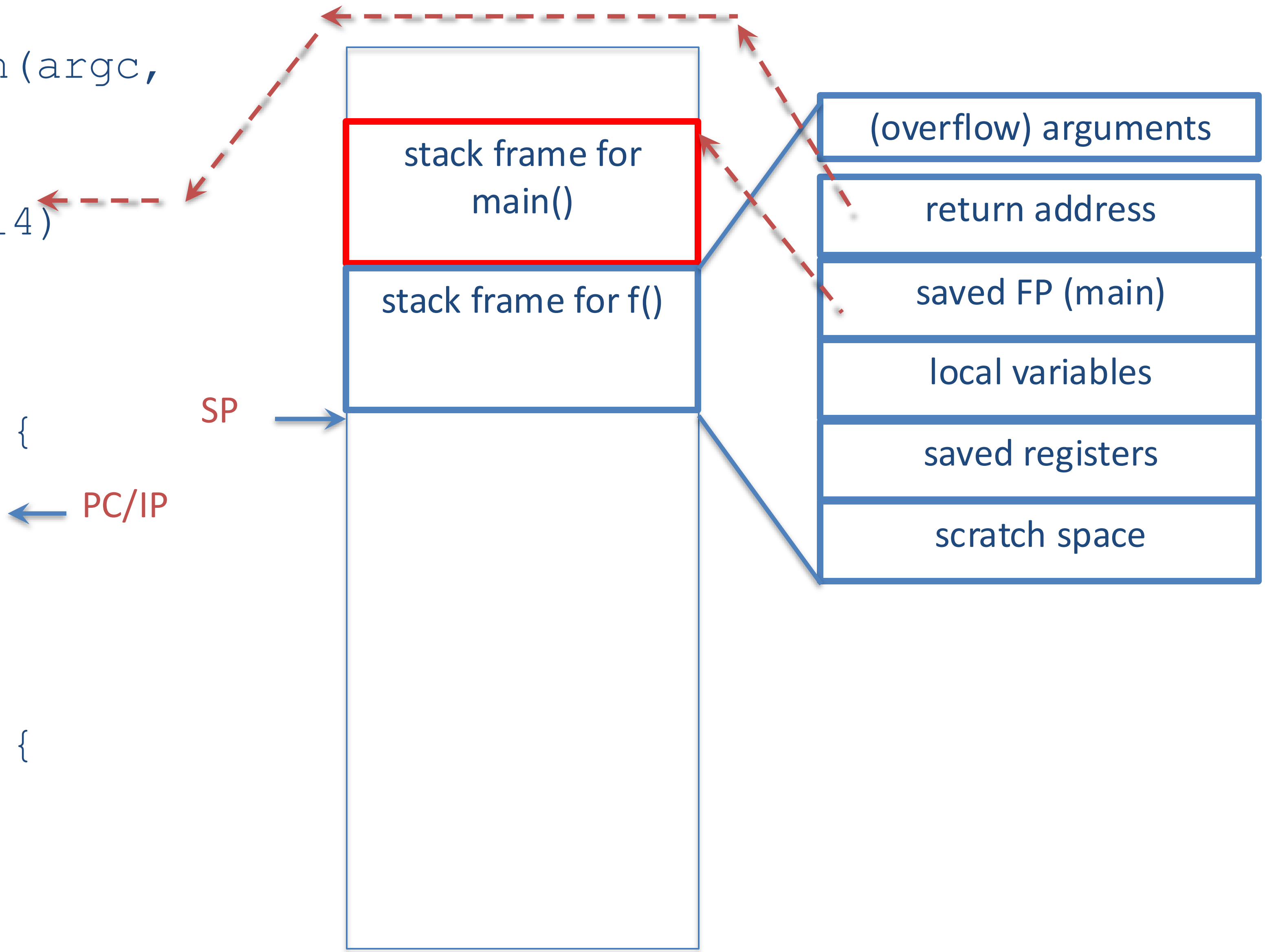


Control Flow

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

int f(x){
    …
    g();
    …
}

int g(y){
    …
}
```

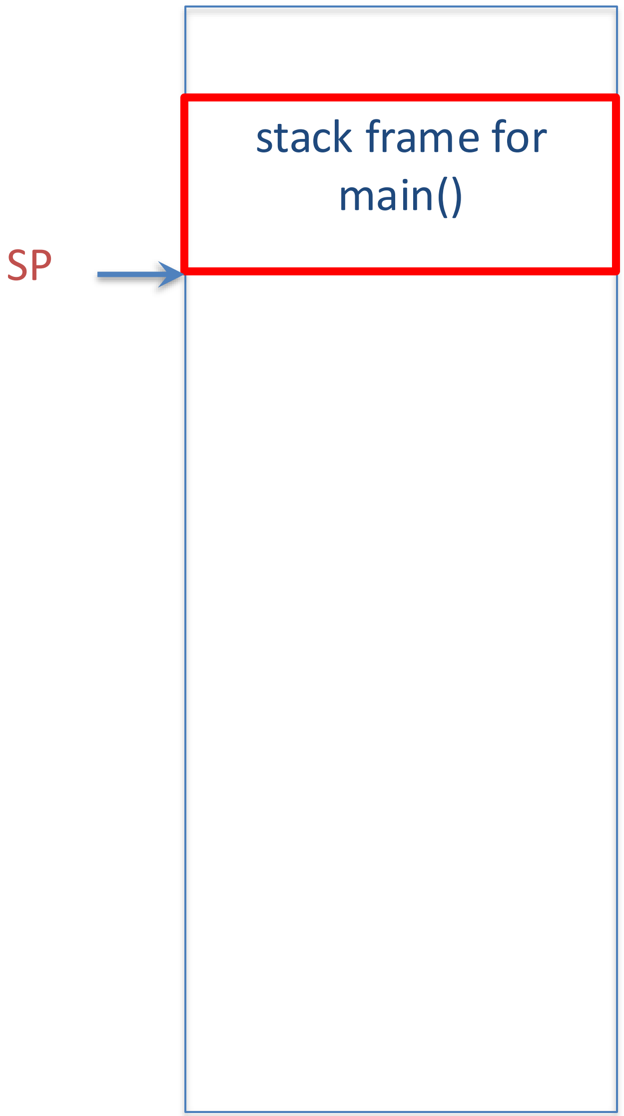


Control Flow

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

int f(x){
    …
    g();
    …
}

int g(y){
    …
}
```



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.

ARCHITECTURAL SUPPORT FOR PROCESSES

Hardware Support for Processes: *supervisor mode (bit in the CPU)*

- One primary objective of an O.S. *kernel* is to manage and isolate multiple processes
  - Kernel code runs in *supervisor mode (aka kernel mode)*
    - unrestricted access to all hardware
  - Process code runs 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, process's kernel stack, and more

How does the kernel get control?

- Boot (reset, power cycle, …)
  - CPU is initially in supervisor mode
  - 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

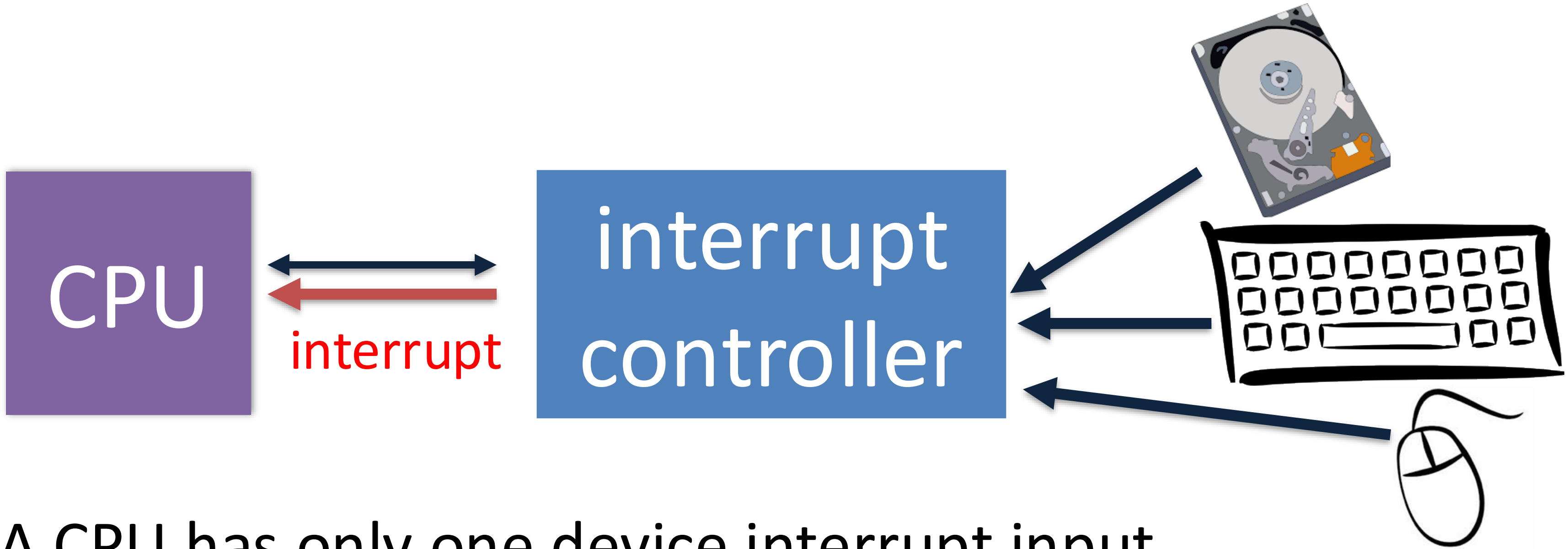
(Device or I/O) Interrupts

- 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 Processing

- Three objectives:
  1. handle the interrupt
  2. remove its cause
  3. 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 Processing (conceptually)

- On signal, CPU:
  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 Processing, cont'd

"return from interrupt" instruction:

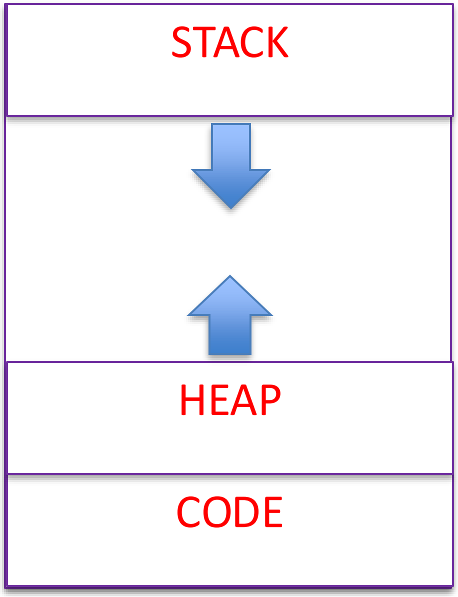
- Restores (kernel/user) mode
- Restores program counter (aka instruction pointer)
- Re-enables interrupts

Two Stacks involved!

- Process has a stack to maintain its control flow
  - “user stack”
- Kernel has its own control flow thus also needs a stack
  - “kernel stack” or “interrupt stack”
- Why can’t we use the same one?

Reasons for separating user stack / kernel stack

- user SP may be illegal
  - badly aligned or pointing to unwritable memory
  - would make it impossible to service interrupt
- user stack may 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
  - interrupt could overwrite object in use
- security risks if only one stack:
  - kernel could push sensitive data on user stack and unwittingly leave it there (stack pop does not erase memory)
  - process could corrupt kernel code or data by pointing SP to kernel address

Summary

- Kernel code runs in supervisor mode
- Application (process) code runs in user mode
  - always with interrupts enabled (why?)
- Each process has its own segments (code, data, heap, stack) and its own registers (pc, sp, r1, r2, …)
  - both are "virtual"
    - CPU does not know, process does not know
- Kernel maintains a PCB (incl. kernel stack) for each process
- 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 automatically invoked by hardware upon an interrupt
- Interrupt handler first pushes the registers onto the kernel stack of the currently running process (part of PCB)
  - Why does it save the registers?
  - Why doesn't the hardware do that automatically?

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 kernel stack but can be stored anywhere in the PCB
- Saving/restoring registers is expensive. Not all registers need to be saved: the kernel uses only a subset, and most functions will already save and restore the registers that they use
- Note: not that different from reasons for saving registers in function calls

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;
    …
    }
}
```

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 (why?)
6. return-from-interrupt instruction

RISC-V interrupts

- A RISC-V processor has some important "control and status registers" (CSRs):
  - mtvec: interrupt handler/vector
  - mstatus:
    - has bits for supervisor mode and interrupt enable
    - has bit that saves supervisor mode bit on interrupt
  - mepc: where PC is saved on interrupt
  - mcause: info about cause of interrupt
- mret instruction restores PC and mode

Processing RISC-V interrupts

- Save mepc, mstatus, and SP in PCB
- Set SP to top of kernel stack (in PCB)
- Push registers to save them
  – "callee-saved" registers can be skipped
- Call C interrupt handler function
- Pop registers
- Restore user SP from PCB
- Restore mepc and mstatus from PCB
- mret instruction
  – restores PC, pre-interrupt supervisor mode, and enables interrupts

Starting a new process with RISC-V

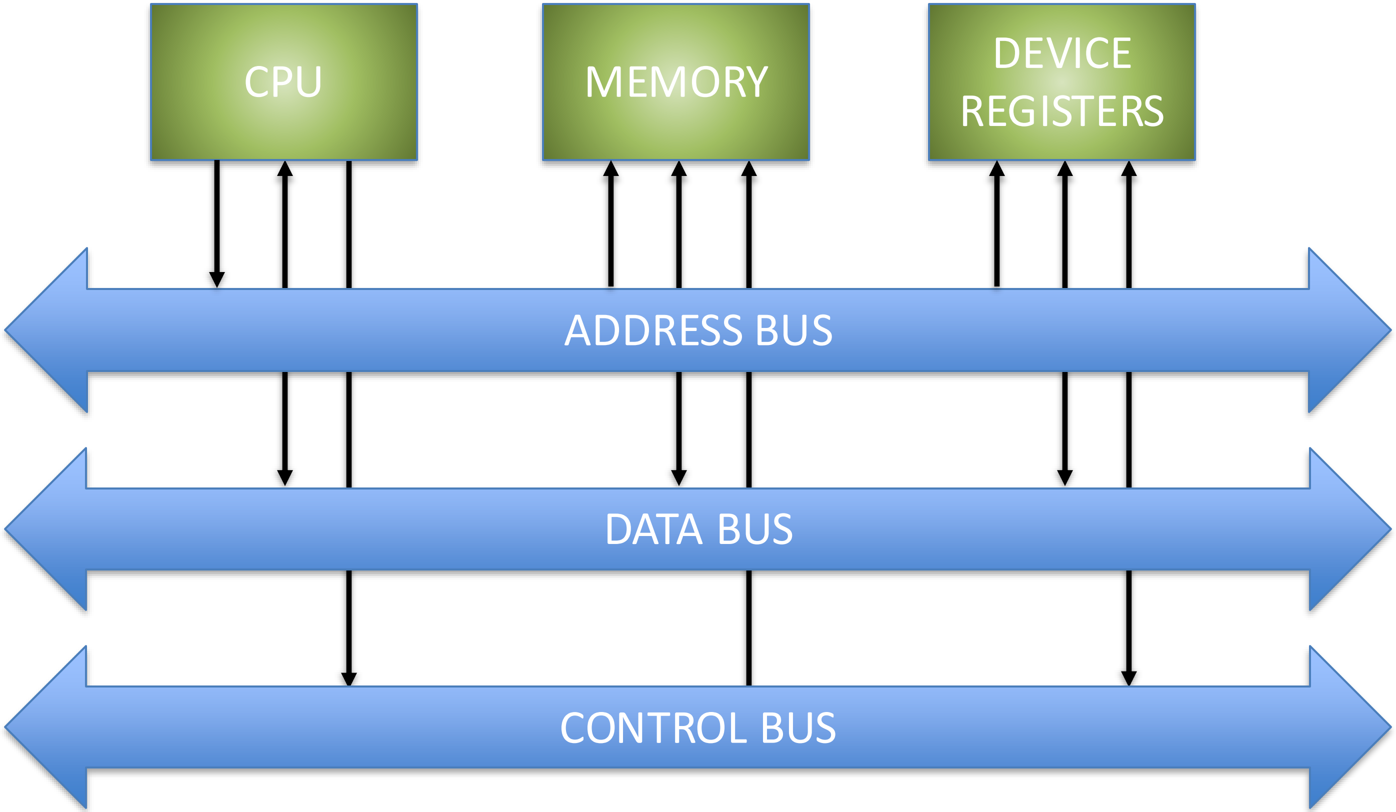
- Allocate a PCB (with a kernel stack)
- Allocate and initialize page table
- Point SP to user stack
- Push process arguments
- Clear other registers
- *Simulate an interrupt:*
  - *set pre-interrupt mode in mstatus register to "user"*
  - *store initial PC in mepc register*
- mret instruction

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

Architecture Diagram



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
  - do not confuse with CPU registers!

Example: Disk Device (simplified)

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

Example: Network Device (simplified)

Device registers:

1. **receive memory address**: for incoming packets
2. **send memory address**: for outgoing packets
3. **command**: to send/receive packet
   - device interrupts CPU upon completion
4. **interrupt ack**: to tell device interrupt received
5. **status**: 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 (bootstrapping) an O.S.

- “pull oneself up 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