OS Security and Software Security

CMSC 23200, Winter 2024, Lecture 2

Grant Ho and Blase Ur

University of Chicago

Today’s Class

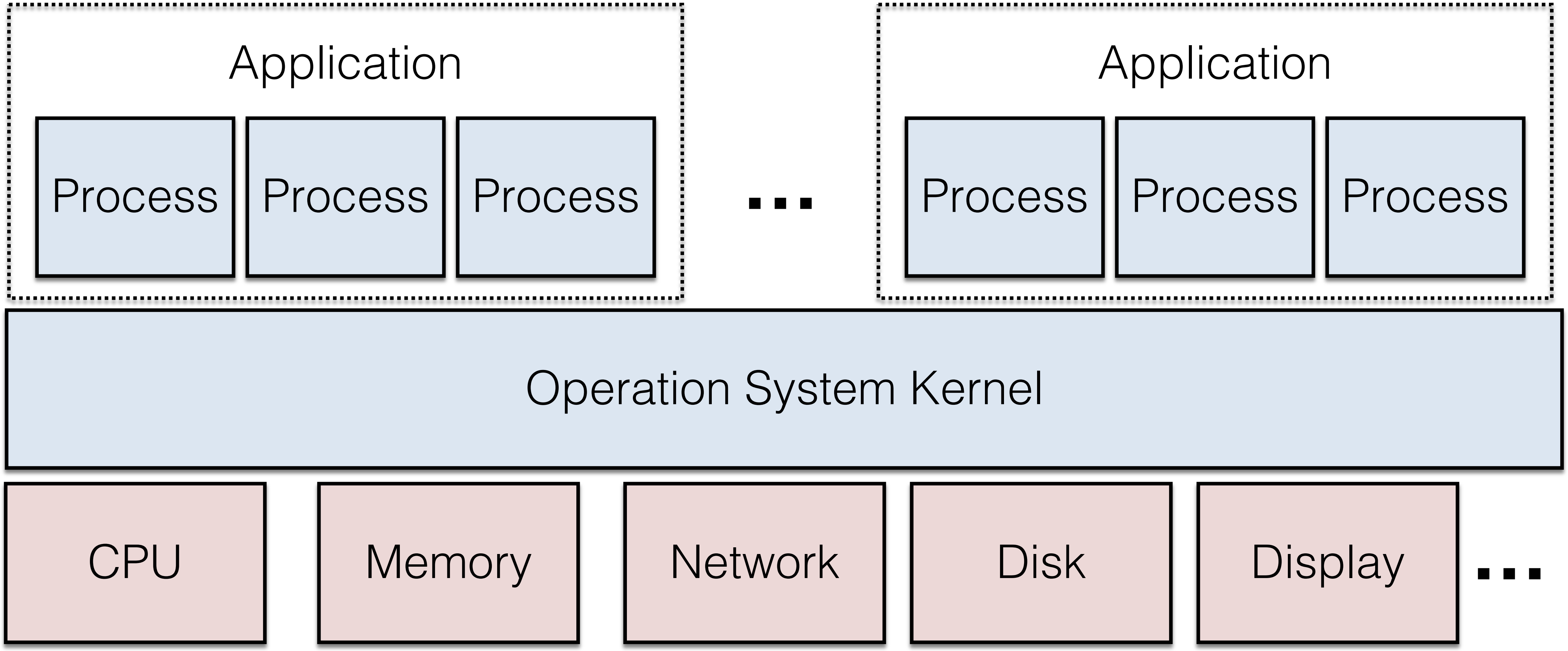
1. **OS Security**:
   How do we ensure that users & programs only access resources they’re allowed to?

2. **Software Security**:
   How can an attacker exploit software bugs to bypass these security restrictions?

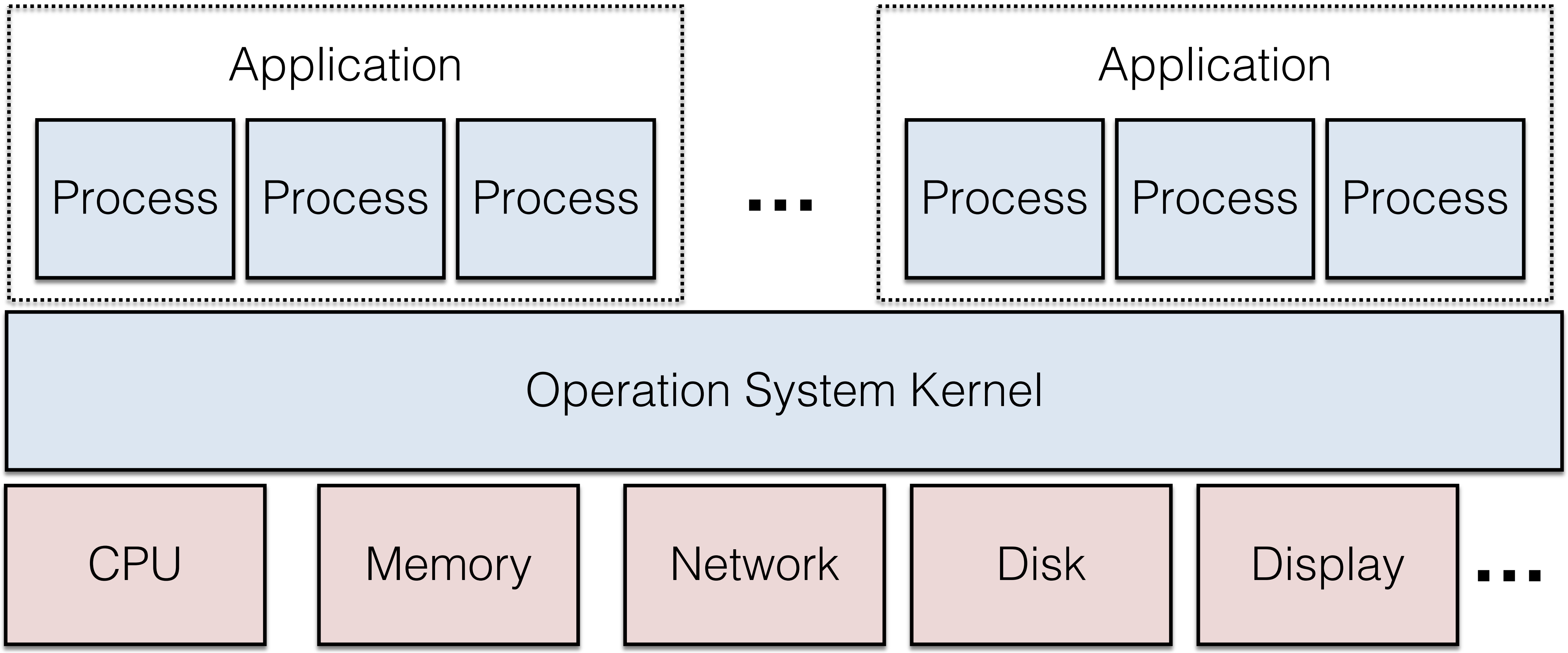
Outline for Lecture 2

1. OS Security: Controlling user & program access
    1. Review of OS Structure
    2. Abstract approaches to access control (5.2)
    3. Concrete Example: The UNIX security model
2. Software Security: Memory Safety & Control Flow Hijacking

Review of OS Structure



Review of OS Structure

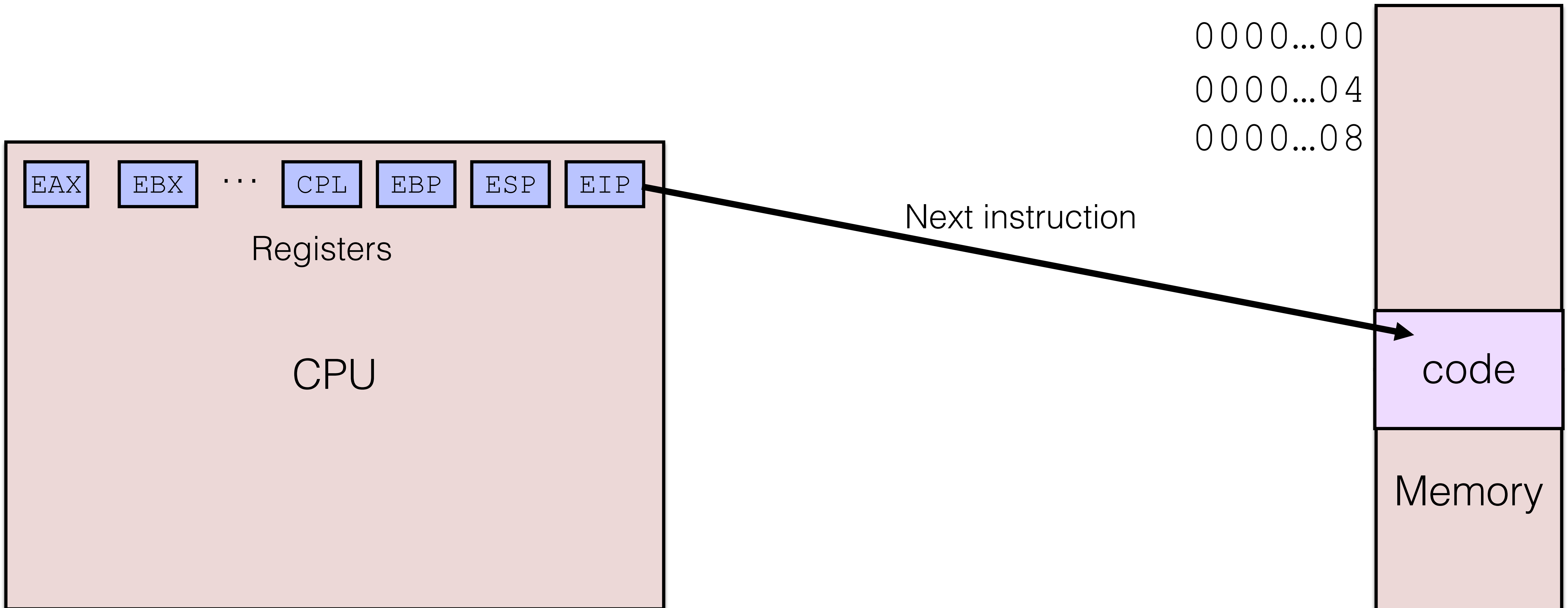


Security/safety: Must protect processes from each other, protect hardware, …

Questions, though:

- What distinguishes the kernel from not-kernel?
- What *is* a process?

How a CPU (x86) Works (extremely high level)

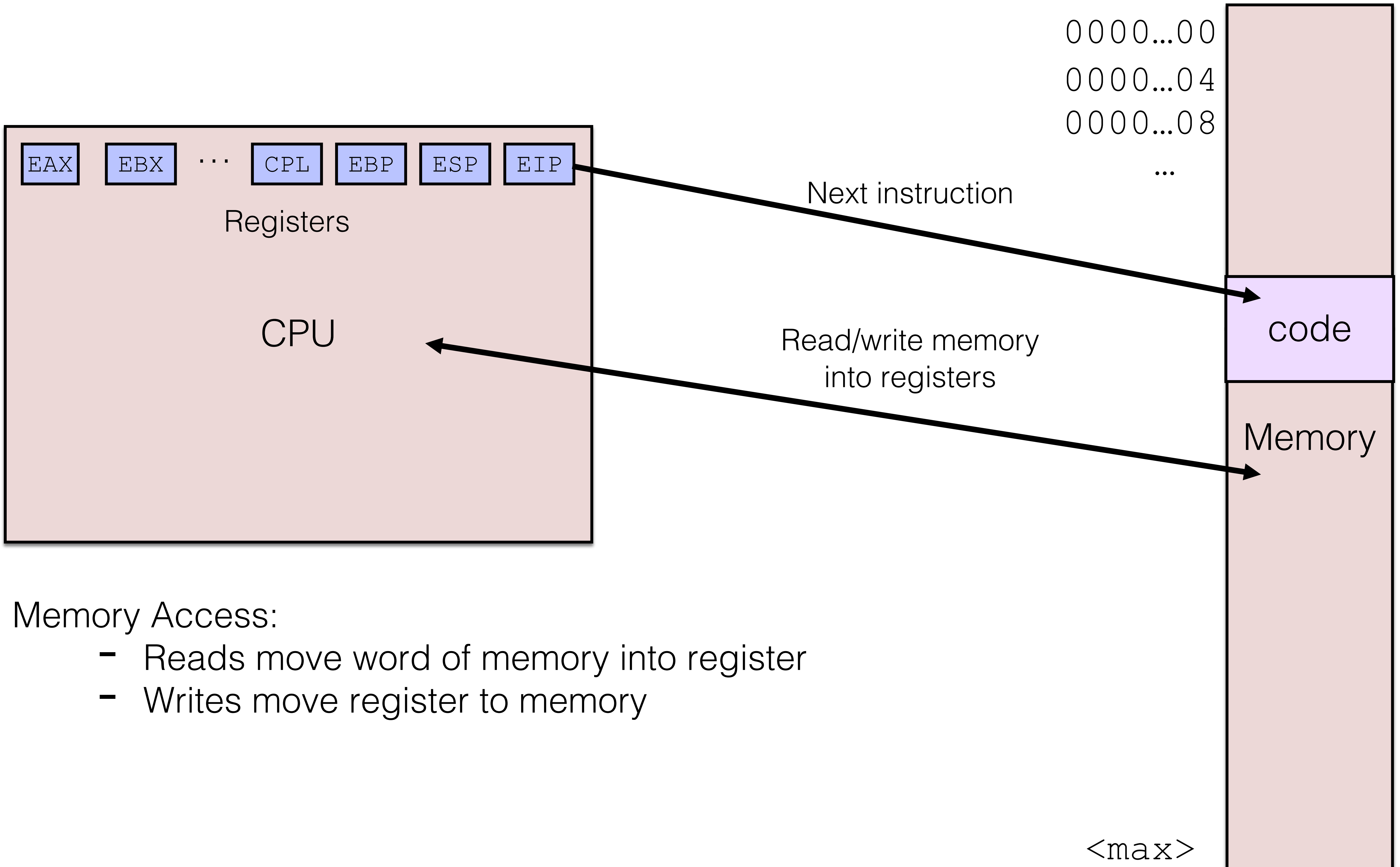


Repeat until HALT:

1. Fetch instruction `inst` pointed to by `EIP`
2. Execute logic of `inst`
3. Increment `EIP` (or update it if `inst=jmp`)

In some cases “interrupts” can occur, which change `EIP` to point at interrupt handler (pointed to by a special reg).

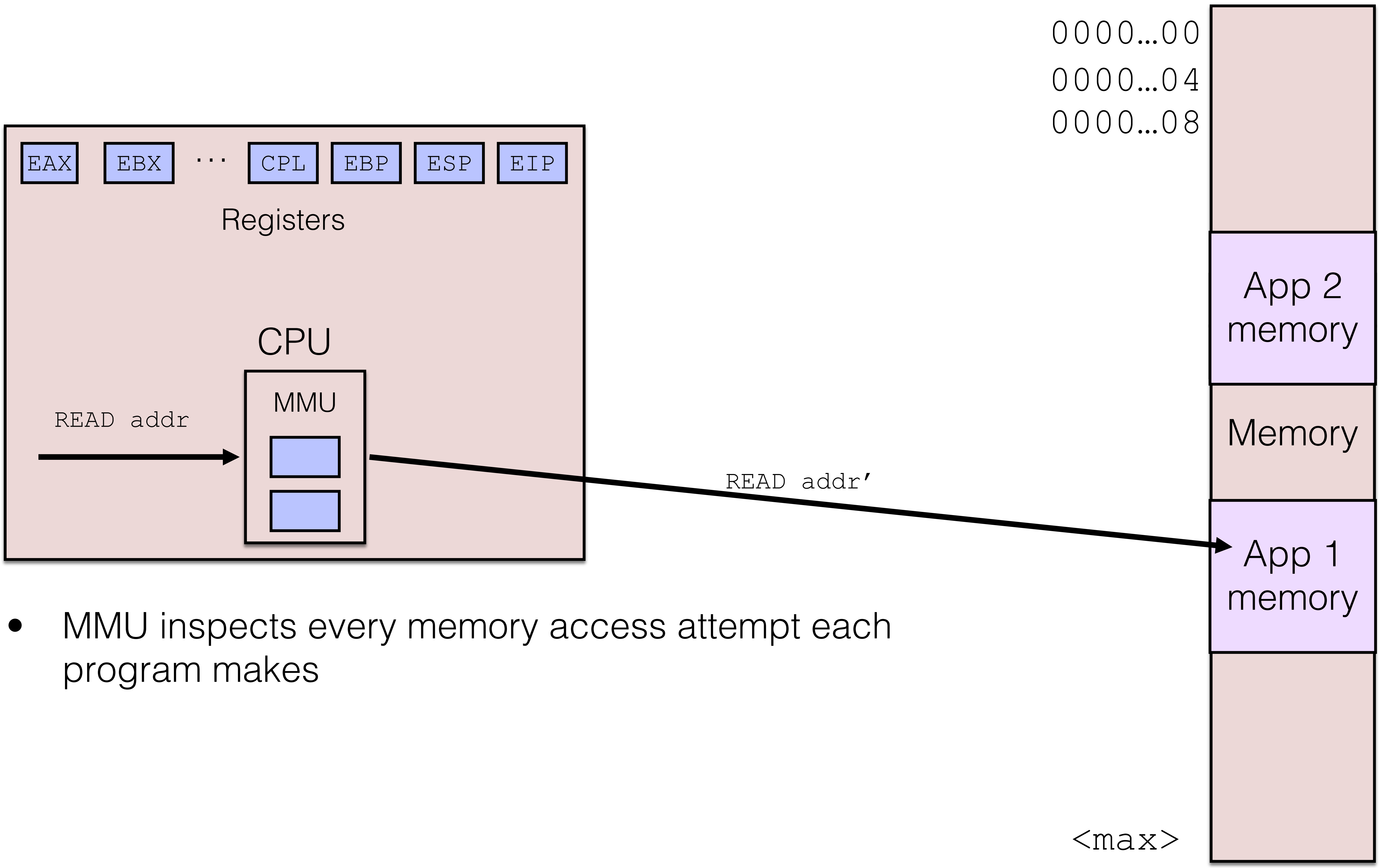
How a CPU (x86) Works (extremely high level)



Memory Access:

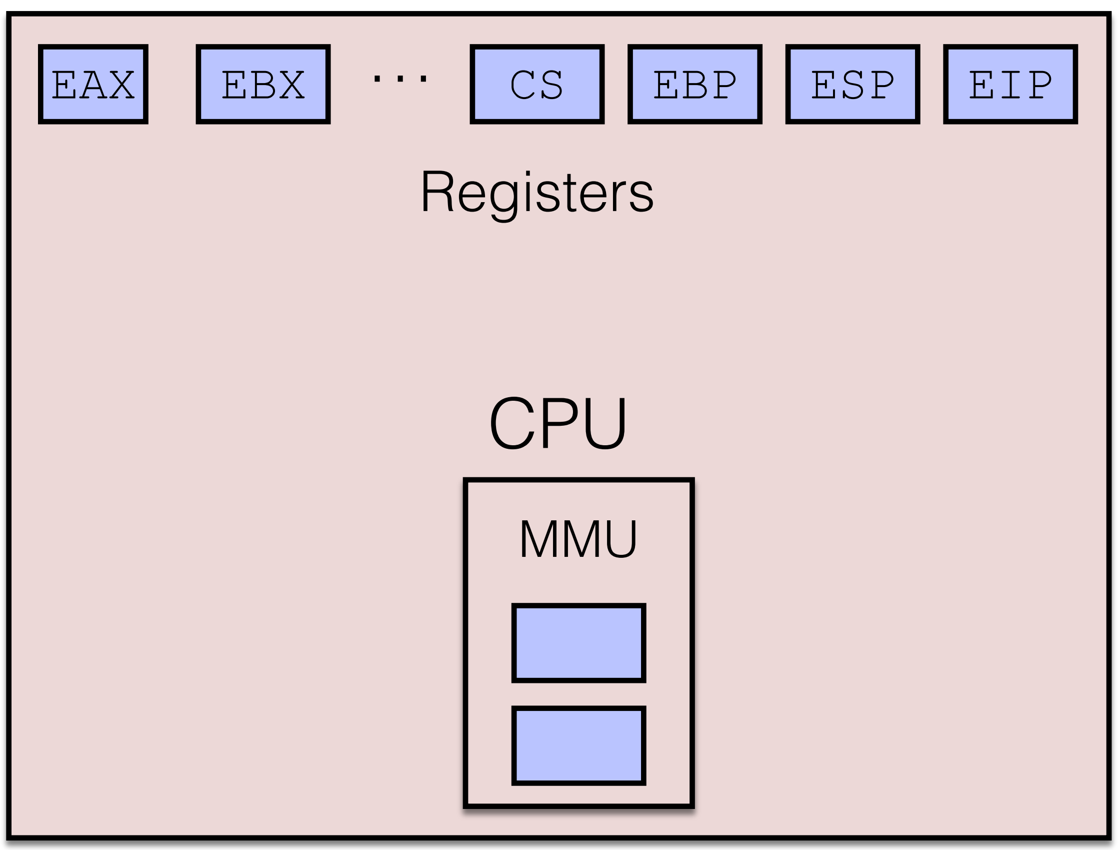
- Reads move word of memory into register
- Writes move register to memory

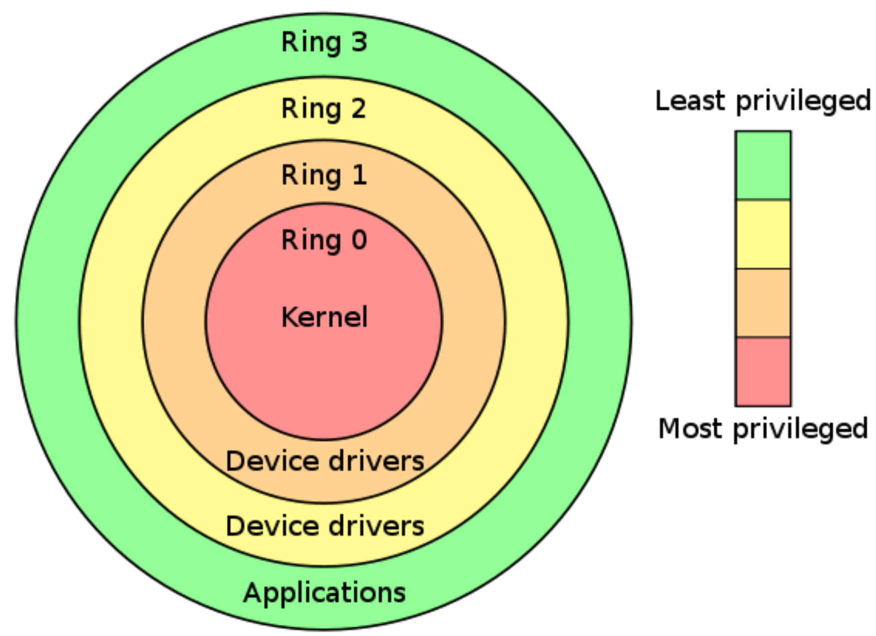
Memory Management Unit (MMU)

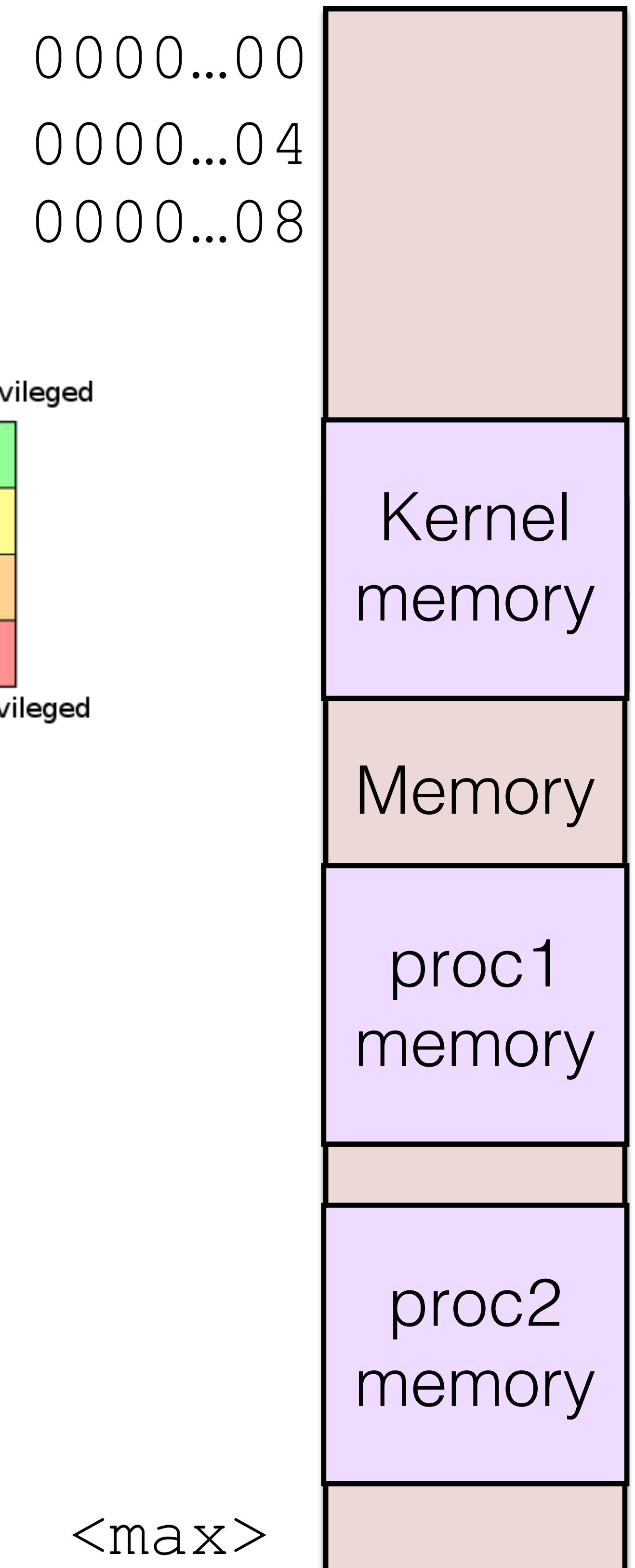


- MMU inspects every memory access attempt each program makes

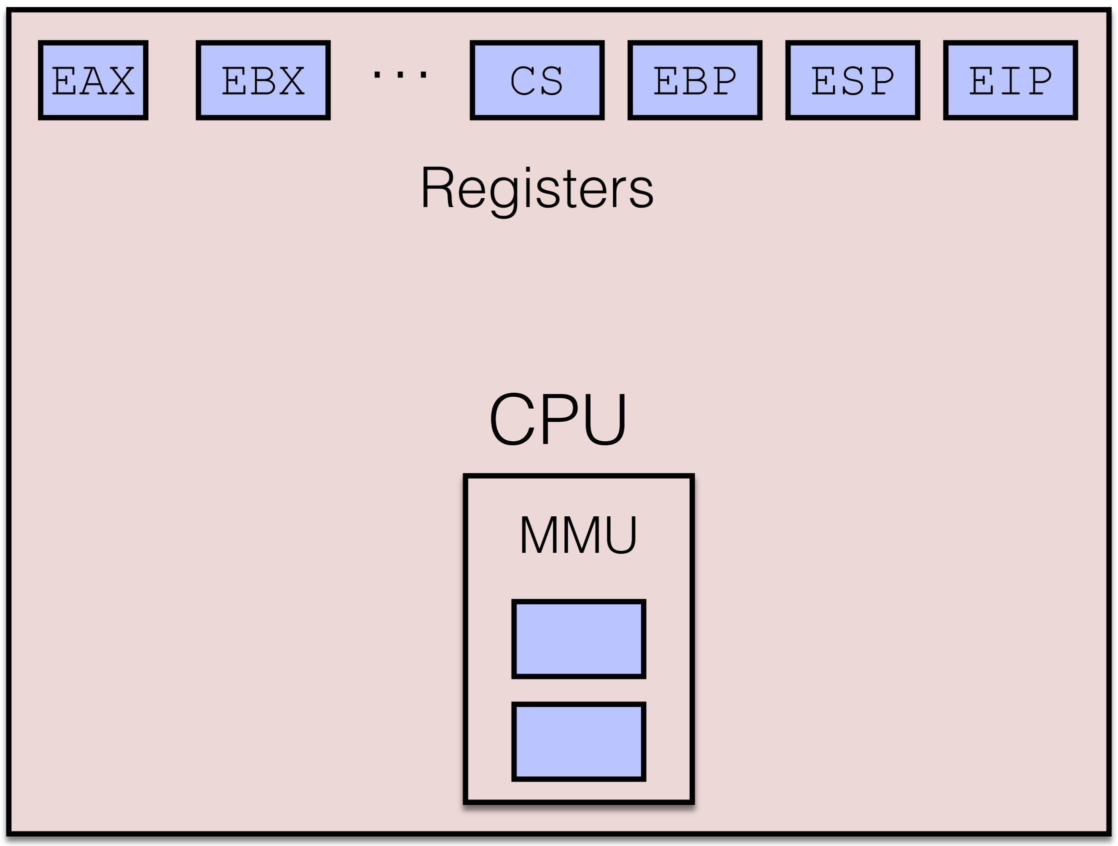
Isolation in x86: It all comes down to CPL

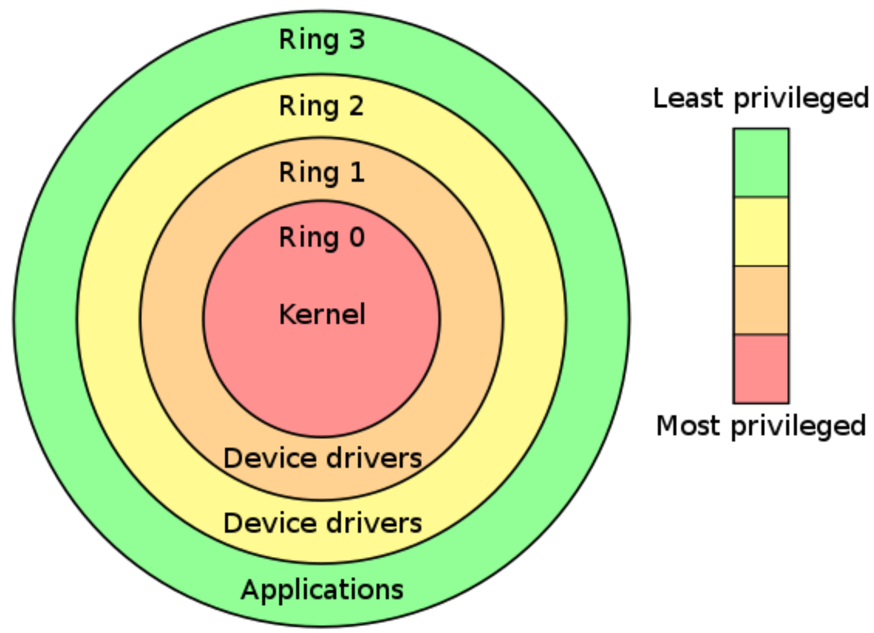


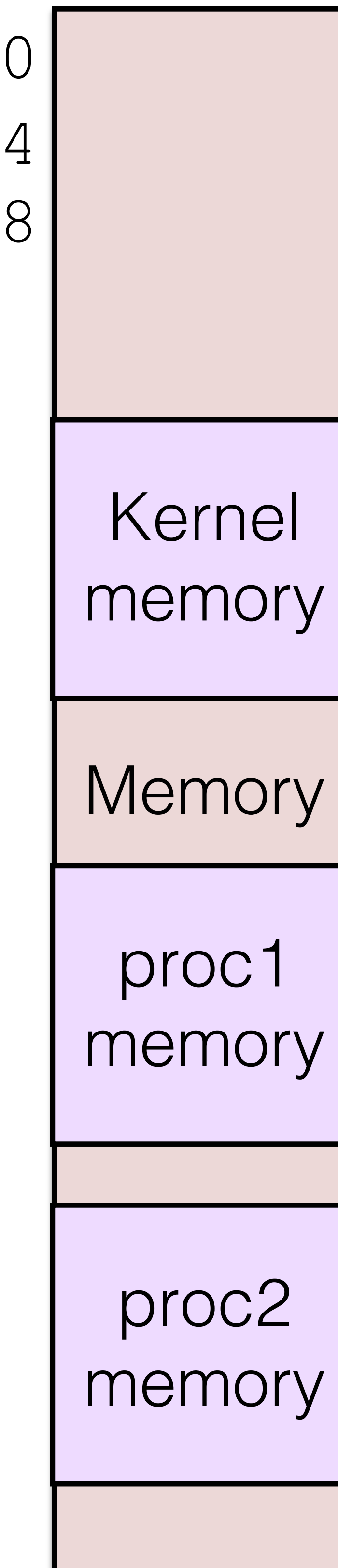




Isolation in x86: It all comes down to CPL

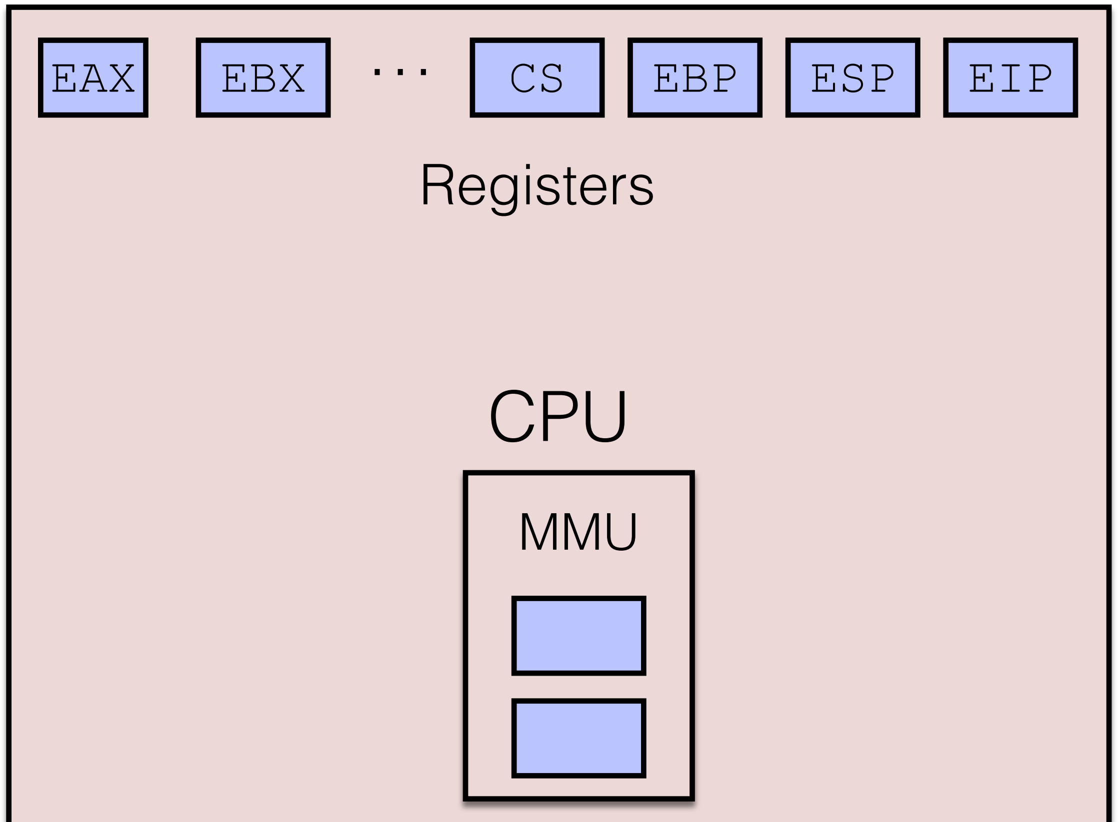






- `CPL` is "current privilege level", two designated bits in `CS` register
- If `CPL = 0`: Then processor will execute any instruction
- If `CPL = 3`: Then processor will only execute subset of instructions

Isolation in x86: It all comes down to CPL



If `CPL`=0, then CPU **will** allow…

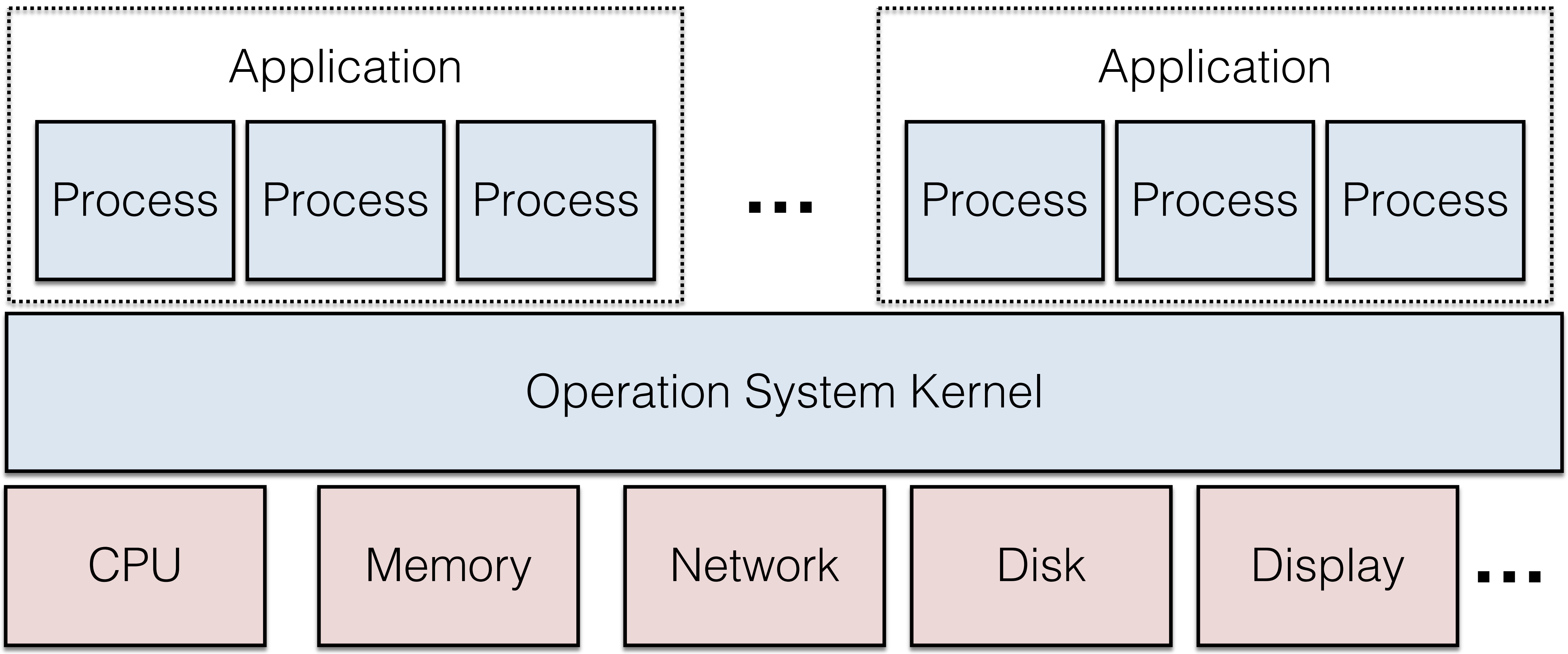
- Direct access to (almost) any addr
- Changes to (almost) any register
- Changes internal state of MMU
- Including setting `CPL`=3!

If CPL=3, then CPU **will not** allow…

- Direct access to memory (only via MMU)
- Changes to several registers
- Changes to internal state of MMU
- Setting `CPL`=0 (!)

Big Idea: Kernel runs with `CPL`=0, and *all* other programs run with `CPL`=3.

Back to our diagram…

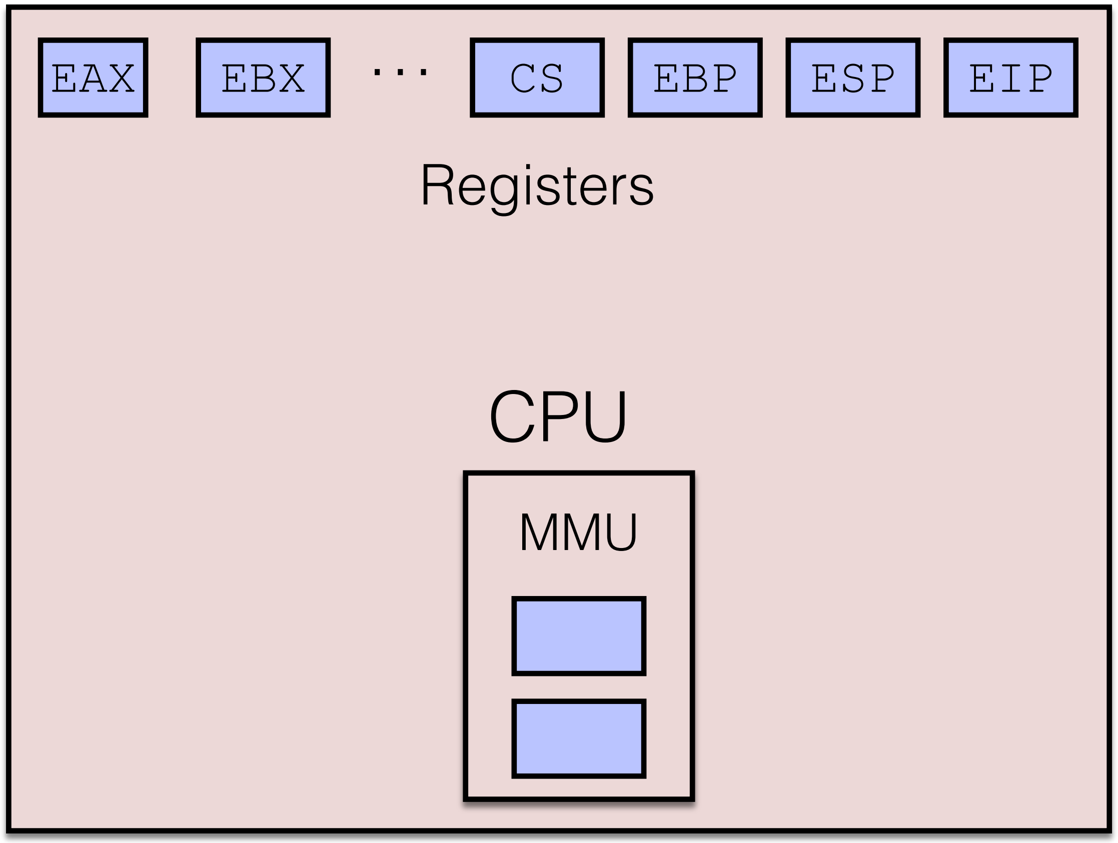


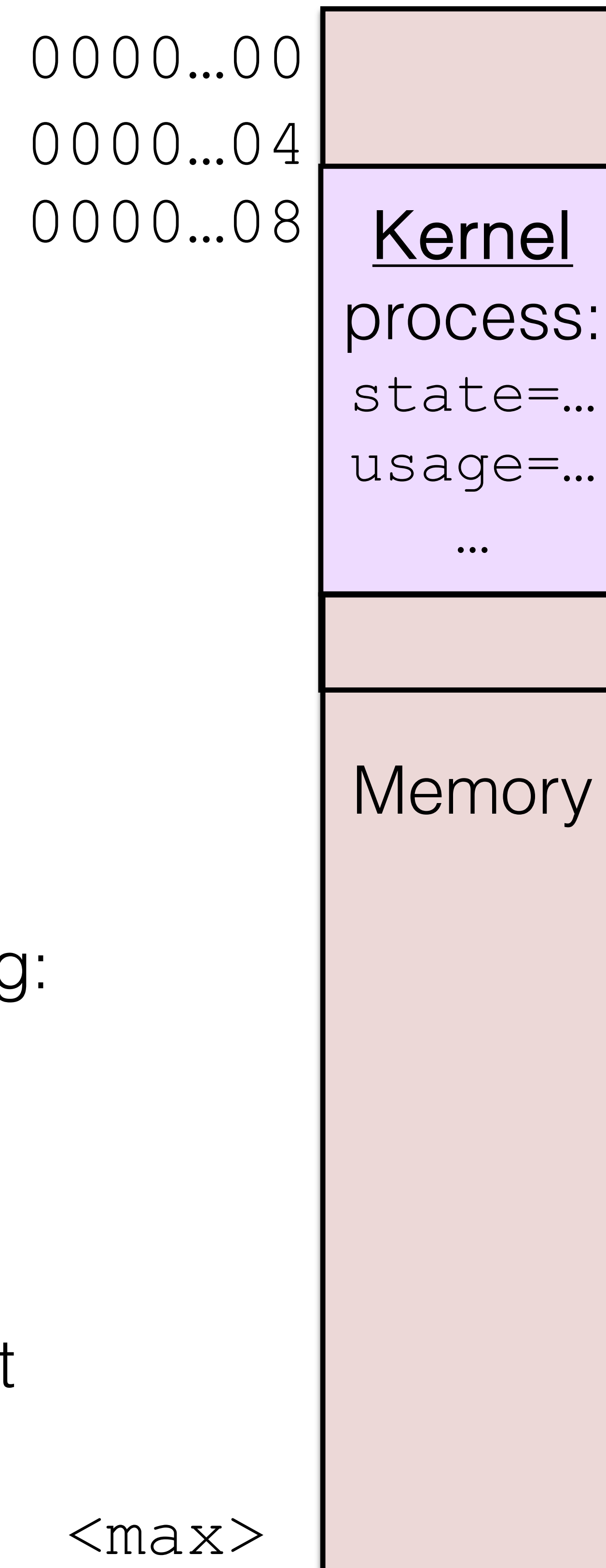
Questions, though:

- What distinguishes the kernel from not-kernel?
- What *is* a process?

The CPL!

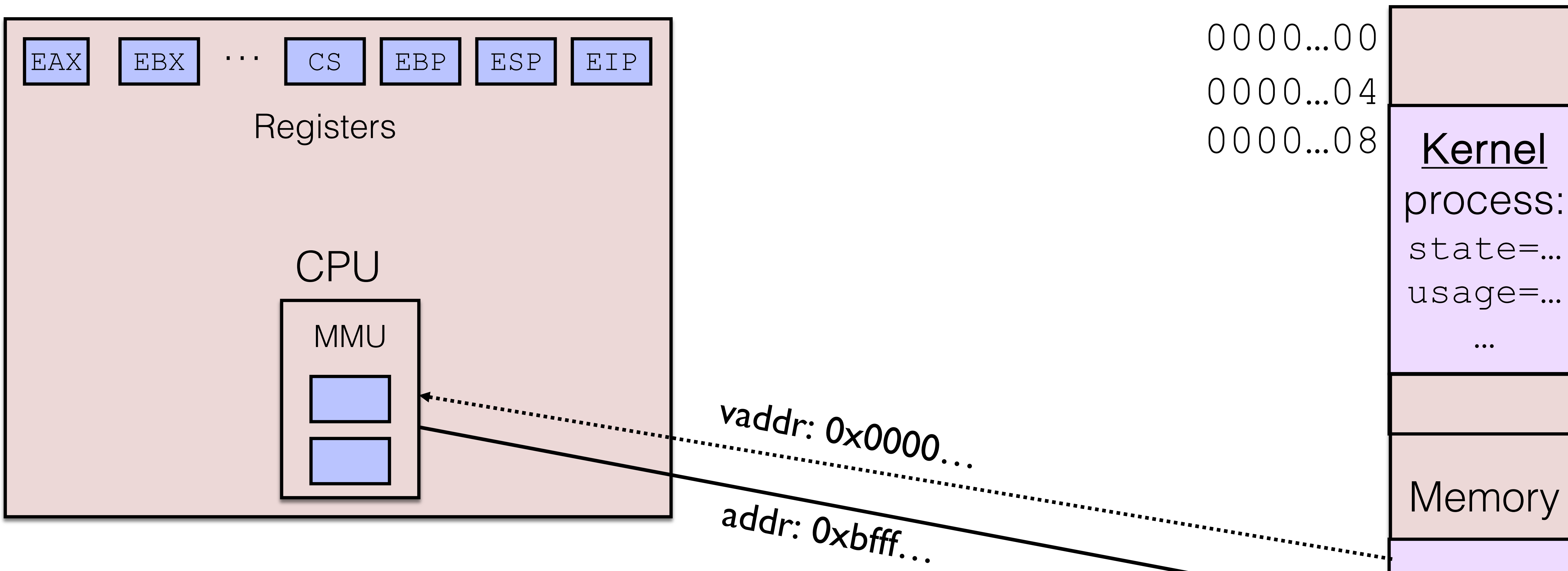
What *is* a process?



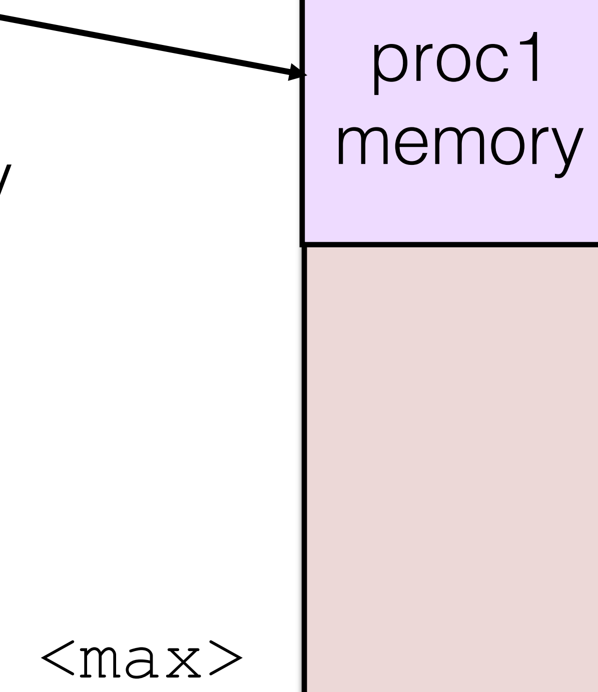


- One Answer: A data structure the kernel manages, including:
    - MMU configuration
    - Register values
- To run application code: Kernel loads these values, sets CPL=3, and turns over CPU control "to the process" (i.e. set EIP)
- If kernel regains control, it can save these values to swap process out

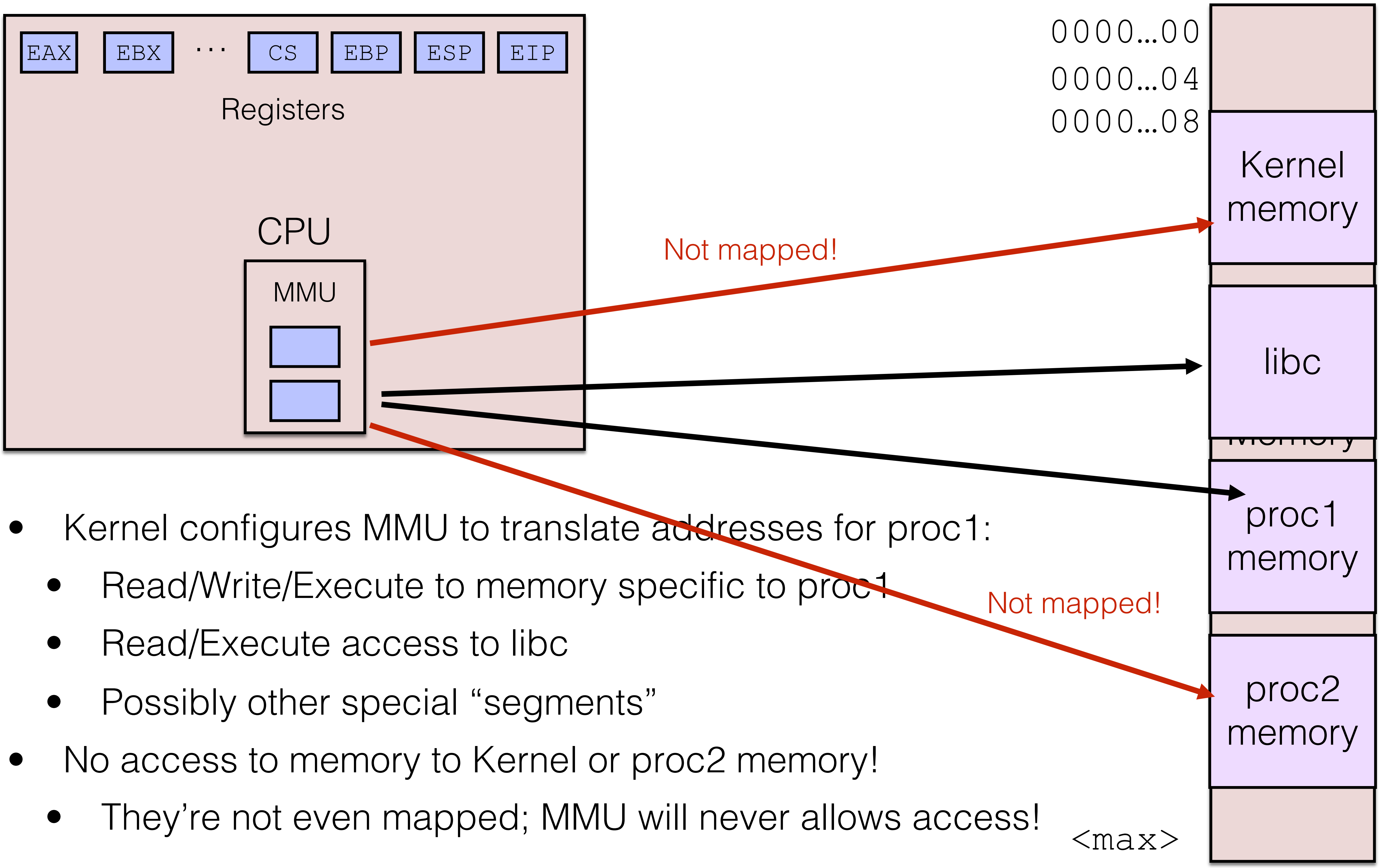
Handling Memory for a Process



- Kernel creates a “virtual address space” for each process.
- Same virtual addresses (e.g. starting near 0) can be used by every process! They get translated to different physical addresses.
- Kernel can also mark some virtual address ranges (called segments) as “read only” or “do not execute” (`EIP` not allowed to point there).
- Violations are `SEGFAULT`s: MMU will take over in this case

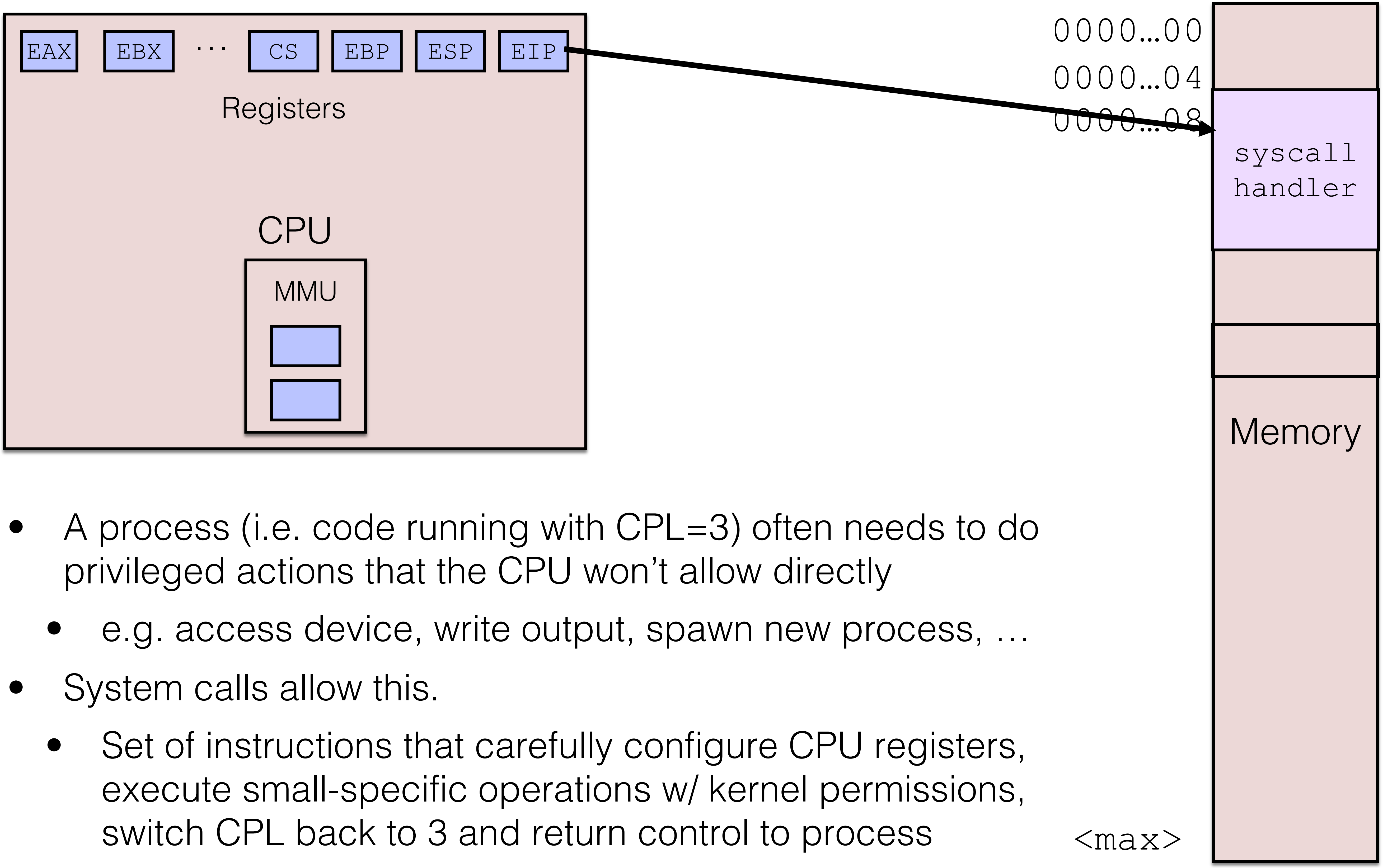


Handling Memory for a Process (cont.)



- Kernel configures MMU to translate addresses for proc1:
  - Read/Write/Execute to memory specific to proc1
  - Read/Execute access to libc
  - Possibly other special “segments”
- No access to memory to Kernel or proc2 memory!
  - They’re not even mapped; MMU will never allows access!

System Calls: How to let processes do privileged ops



- A process (i.e. code running with CPL=3) often needs to do privileged actions that the CPU won't allow directly
  - e.g. access device, write output, spawn new process, …
- System calls allow this.
  - Set of instructions that carefully configure CPU registers, execute small-specific operations w/ kernel permissions, switch CPL back to 3 and return control to process

Outline for Lecture 2

1. OS Security
    1. Review of OS Structure
    2. Abstract approaches to access control (5.2)
    3. Concrete Example: The UNIX security model
2. Software Security: Memory Safety & Control Flow Hijacking
    - Overview of software exploits
    - Memory layout and function calls in a process
    - Stack-based buffer overflow attacks

So we have a secure kernel… What now?

1. Maybe all processes should not be "created equal"?

   - e.g. Should one process be able to kill another?

2. Enable different people to use same machine?

   - e.g. Need to enable confidential storage of files, sharing network, …

3. System calls allow for safe entry into kernel, but only make sense for low-level stuff.

   - We need a higher level to "do privileged stuff" like "change my password".

All of this will be supported by an "access control" system.

Fundamentals of Access Control: Policies

<u>Guiding philosophy:</u> Utter simplicity.

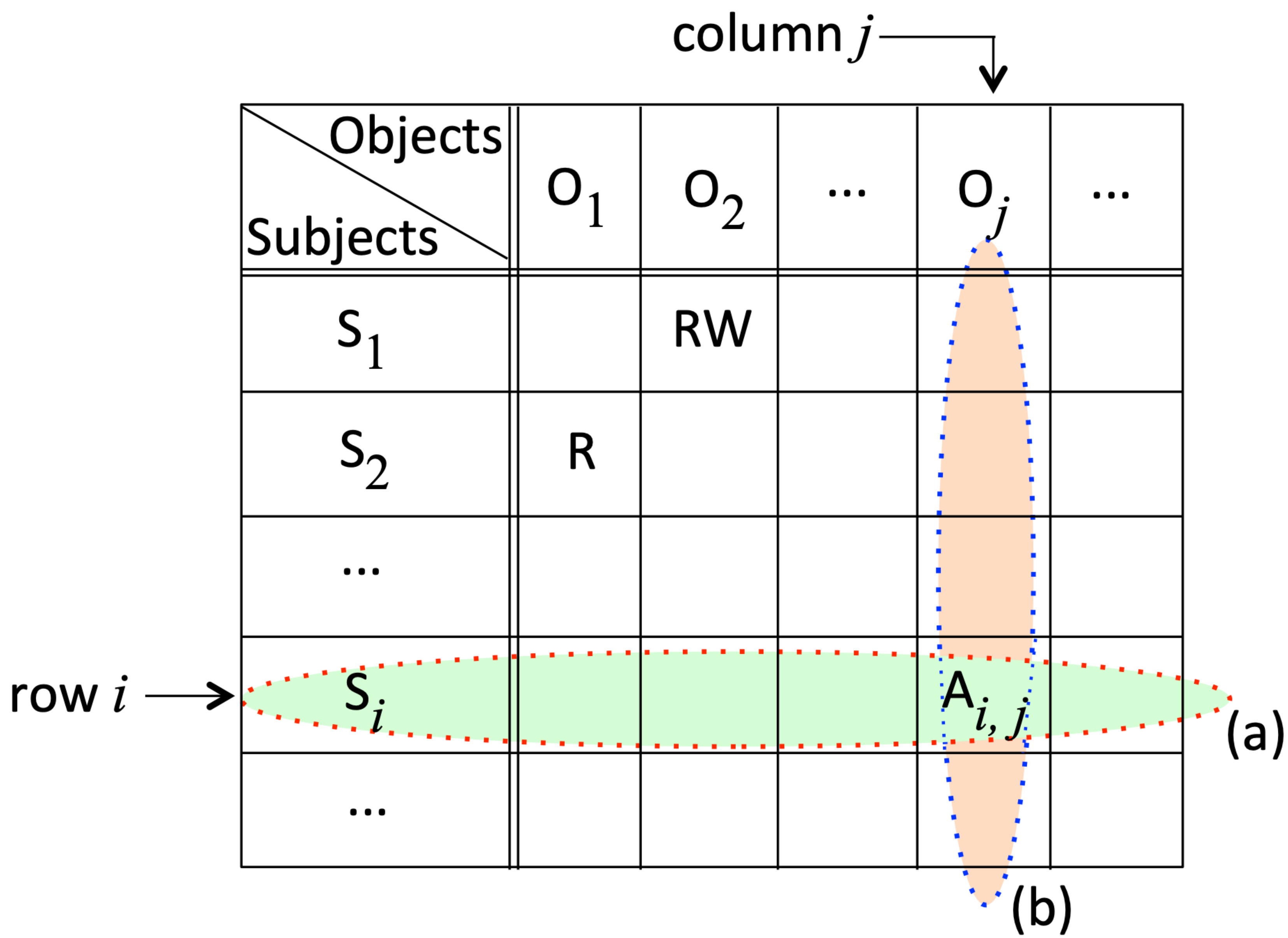
Step 1: Give a crisp definition of a **policy** to be enforced.

1. Define a sets of **subjects**, **objects**, and **verbs**.
2. A **policy** consists of a yes/no answer for every combination of subject/object/verb.

<u>Example</u>

- Subjects: Grant, Blasé, Student
- Objects: HW1, Exam
- Verbs: Create, Submit, Grade
- Policy: {Grant, Blasé -> Create, Submit, Grade -> HW1, Exam}
  {Student -> Submit -> HW1, Exam}

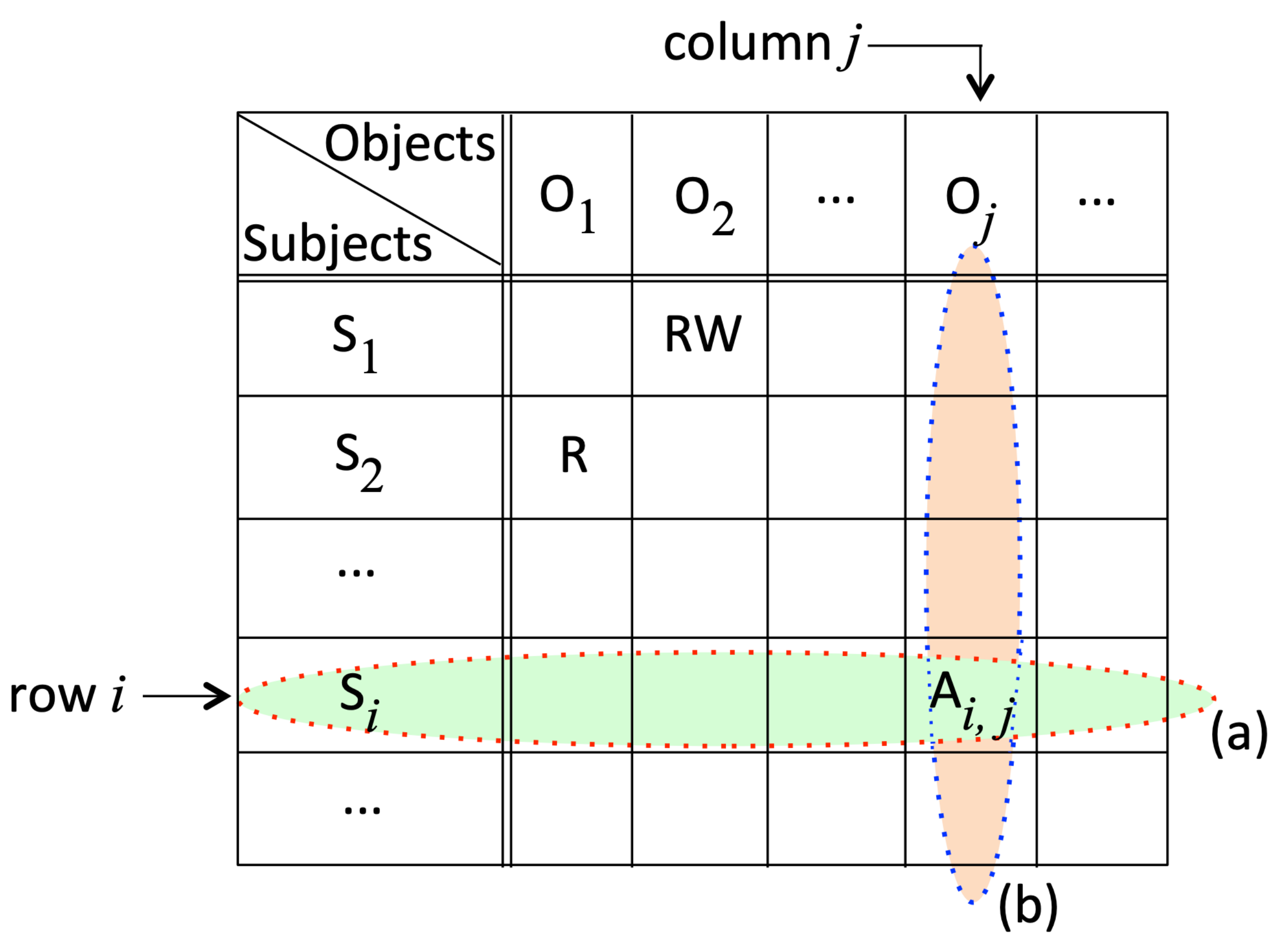
The Access Control Matrix



- Entry in matrix is list of allowed verbs
- The matrix is not usually actually stored; It is an abstract idea.

Implementing Access Policies: ACLs

- ACL = "access control list"
- Logically, ACL is just a column of matrix
- Usually stored with object
- Can quickly answer question: "Who can access this object?"

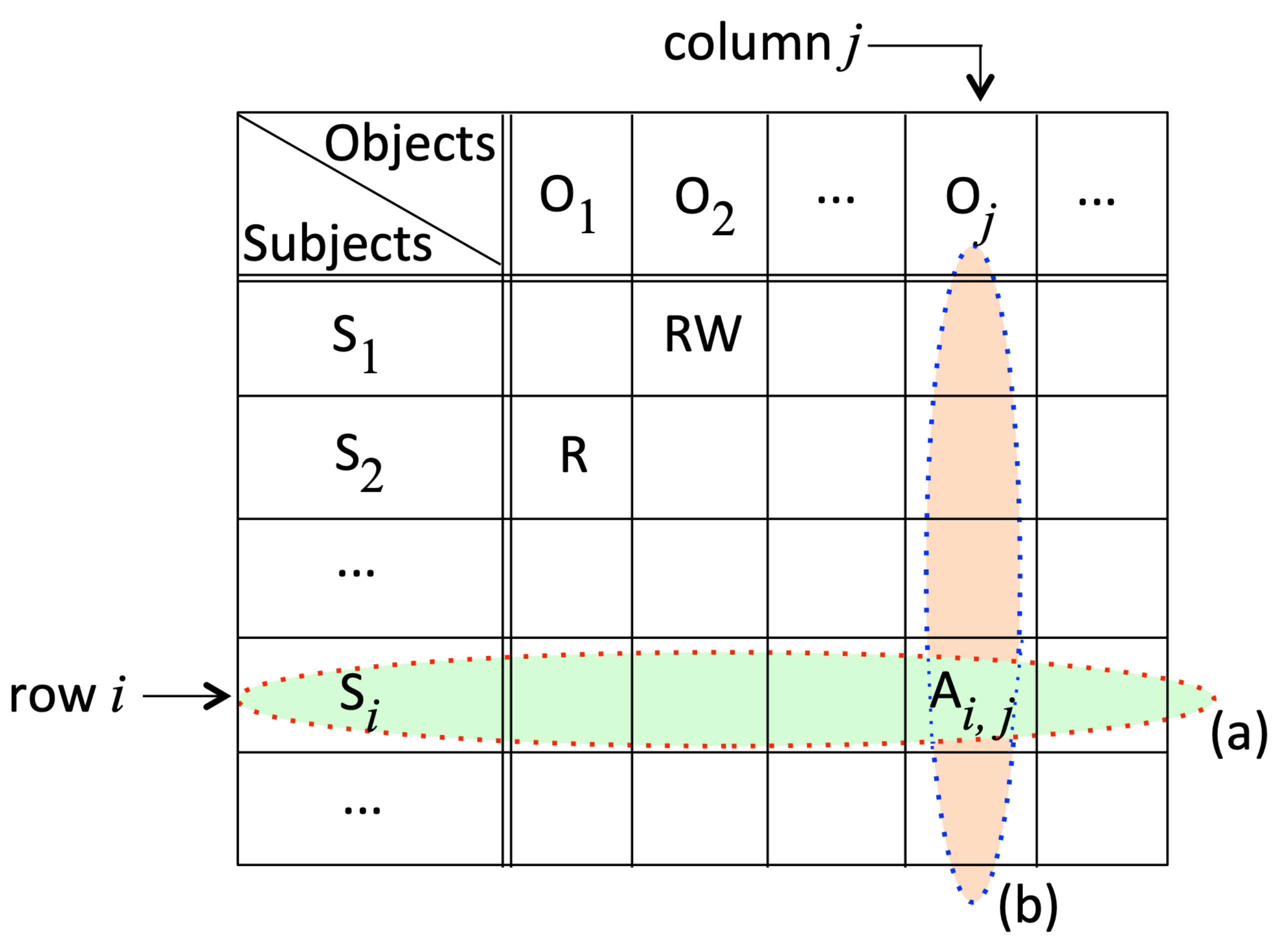


Examples:

1. VIP list at event
2. This class on Canvas

Implementing Access Policies: Capabilities

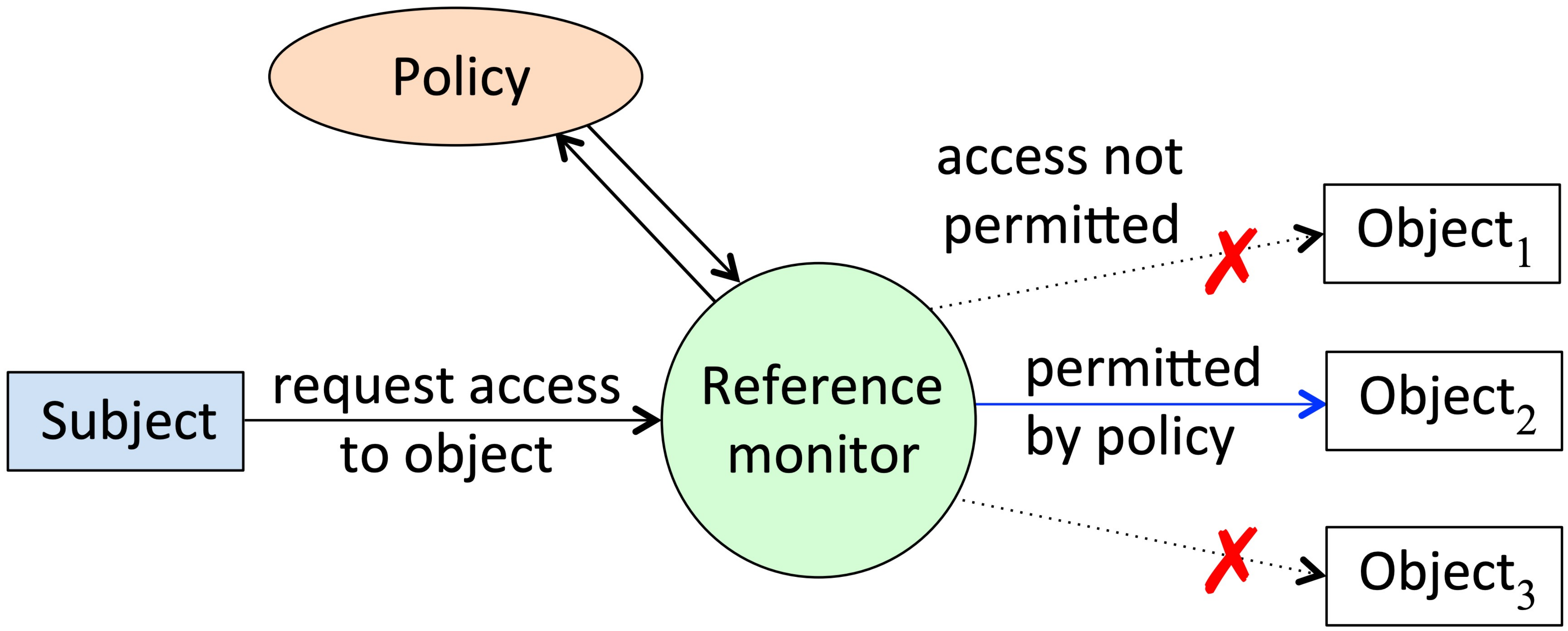
- "Capability" (of a subject) is a row of matrix
- Usually stored with subject
- Can quickly answer question: "What can this subject access?"



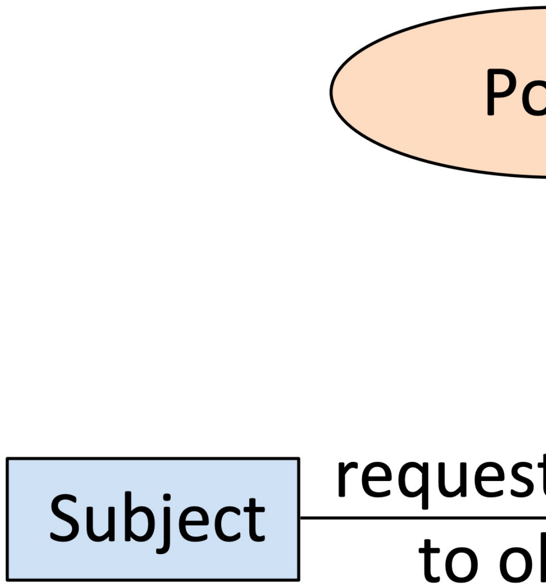
Examples:

1. Movie ticket
2. Physical key to door lock

Enforcing Policy: Reference Monitors



Enforcing Policy: Reference Monitors



Requirements:

1. Always invoked

2. Tamper-proof.

3. Verifiable; Simp

4. (Usually) Logs a

Outline for Lecture 2

1. OS Security
    1. Review of OS Structure
    2. Abstract approaches to access control (5.2)
    3. Concrete Example: The UNIX security model
2. Software Security: Memory Safety & Control Flow Hijacking
    - Overview of software exploits
    - Memory layout and function calls in a process
    - Stack-based buffer overflow attacks

What is “UNIX”? Why should we study it?

- Initially an OS developed in the 1970s by AT&T Bell Labs.
- A riff on “Multics”. UNIX was meant to be simpler and leaner.
  - Philosophy of small programs with simple communication mechanisms
- Licensed to vendors who developed their own versions. “BSD” = “Berkeley Software Distribution” may be most famous of those.
- Linux also later derived from UNIX. MacOS based on UNIX since 2000.

Why study UNIX?

1. Simple, even beautiful security design.
2. You will almost certainly use it.
3. Looking at something concrete is enlightening.

Ken Thompson and Dennis Ritchie, 1971

Subjects, Objects, and Verbs in UNIX (incomplete lists)

Subjects:

1. Users, identified by numbers called UIDs
2. Processes, identified by numbers called PIDs

Objects:

1. Files
2. Directories
3. Memory segments
4. Access control information (!)
5. Processes (!)
6. Users (!)

Verbs (listed by object):

1. For files and memory: Read, Write, Execute
2. For processes: Kill, debug
3. For users: Delete user, Change groups

File Permissions: Users and Groups

- A "user" is a sort of avatar that may or may not correspond to a person.
- Each user is identified by a number called UID that is fixed and unique.
- Each user may belong to 1 or more "groups", each identified by number called GID.

All files are owned by one user and one group.

<u>inode:</u>

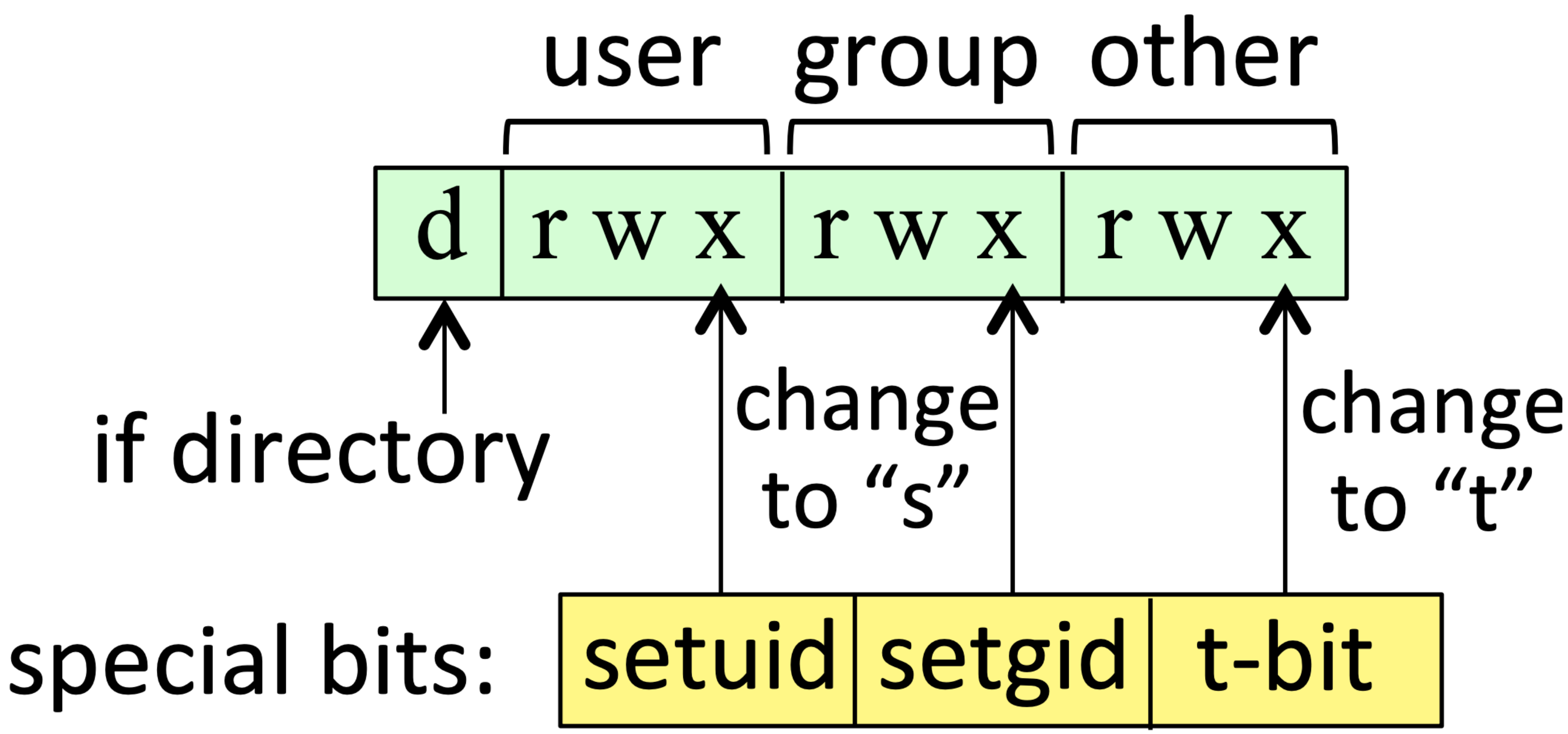
```
mode=1010100…
uid=davidcash
gid=cs232
ctime=…
```

- Changed with commands `chown` and `chgrp`.

File Permissions: UGO Model

- Three bits for each of user, group, and other/all.
- Indicate read/write/execute permission respectively.

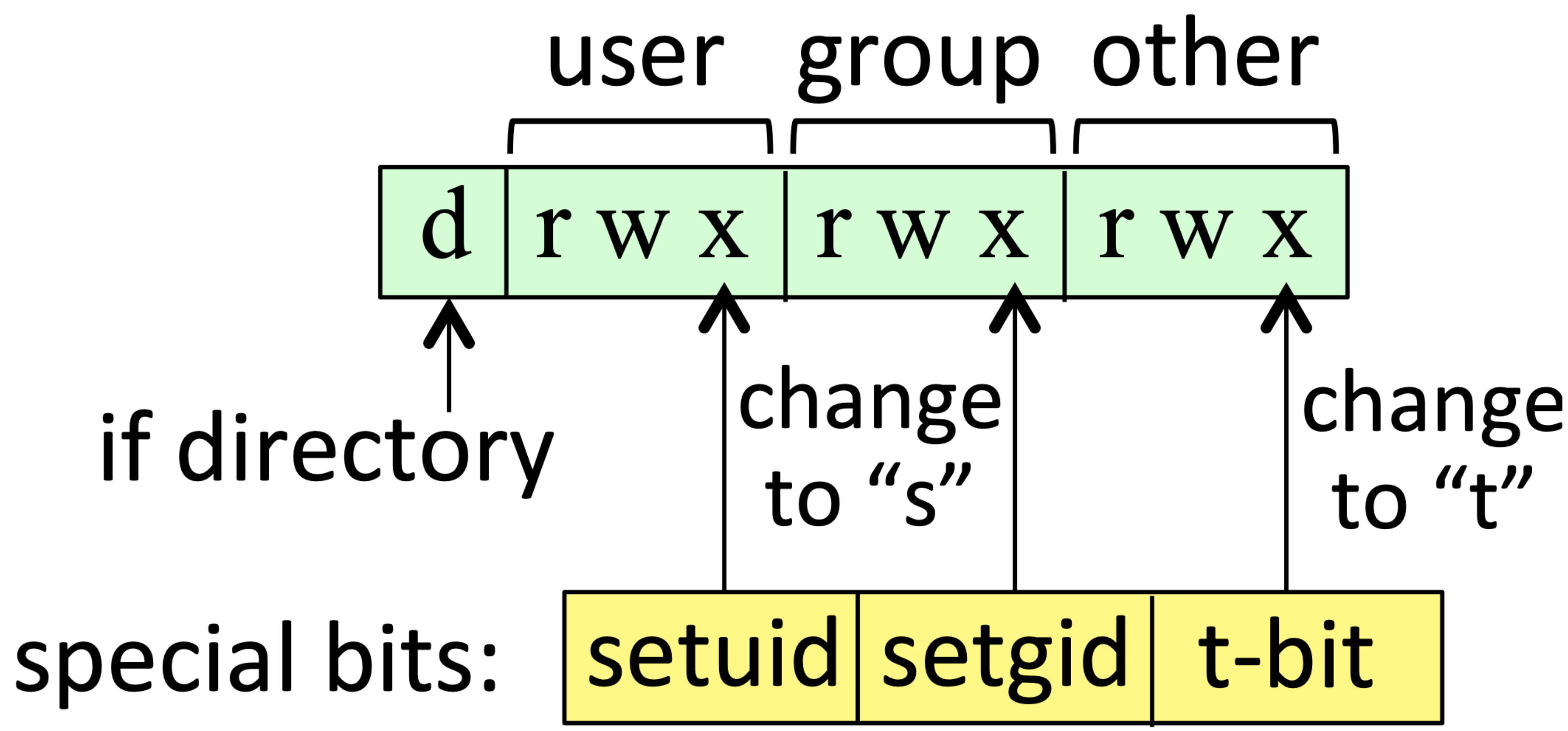
inode:
mode=1010100…
uid=davidcash
gid=cs232
ctime=…



File Permissions: UGO Model

inode:
mode=1010100…
uid=davidcash
gid=cs232
ctime=…

- Three bits for each of user, group, and other/all.
- Indicate read/write/execute permission respectively.



To check access:
1. If user is owner, then use owner perms.
2. If user is not owner but in group, user group perms.
3. Otherwise use “other” perms.

ACL or Capability?

The Root User

- “root” is the name for the administrator account
- UID = 0
- Can open/modify any file, kill any process, etc
- Rarely used as a log-in; Root’s powers are typically accessed via `sudo`
    - Why not? (Which design principle(s) does this follow?)

Process Ownership and Permissions

- Every process has an owner; That process runs with permissions of the owner.

  Actually…. a process has three UIDs associated with it:

  1. Real UID
  2. Effective UID
  3. Saved UID

- Why? To allow for fine-grained control over privileges via `setuid()` syscall.
- Implement *least-privilege* (P6) and *isolated compartments* (P5) in applications

Brief Recap of OS Security

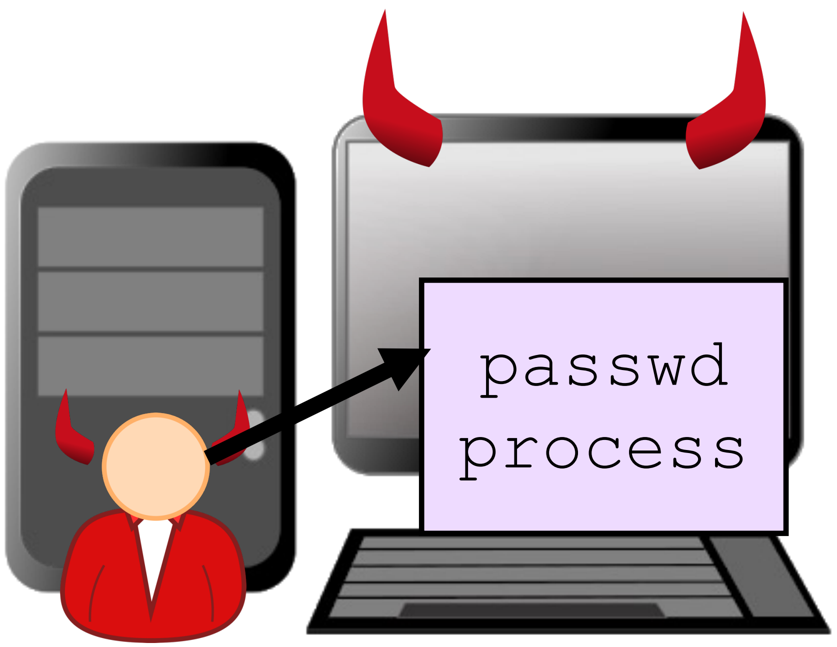
- The OS Kernel ensures that multiple programs can securely run together at the same time
  - The CPU has a dedicated CS register that tracks the privilege (CPL) of the currently running code
  - The OS Kernel & MMU use virtual addressing to help isolate the memory of different processes
- To control what data (e.g., files) users can access and what operations (e.g., programs and code) users can run:
  - The OS implements an access control system, where an administrator specifies policies (e.g., ACLs) about what actions each subject can perform on different objects

2 MINUTE BREAK

Outline for Lecture 2

1. OS Security
    1. Review of OS Structure
    2. Abstract approaches to access control (5.2)
    3. Concrete Example: The UNIX security model
2. Software Security: Memory Safety & Control Flow Hijacking
    - Overview of software exploits
    - Memory layout and function calls in a process
    - Stack-based buffer overflow attacks

Software Attacks: One Common Setting



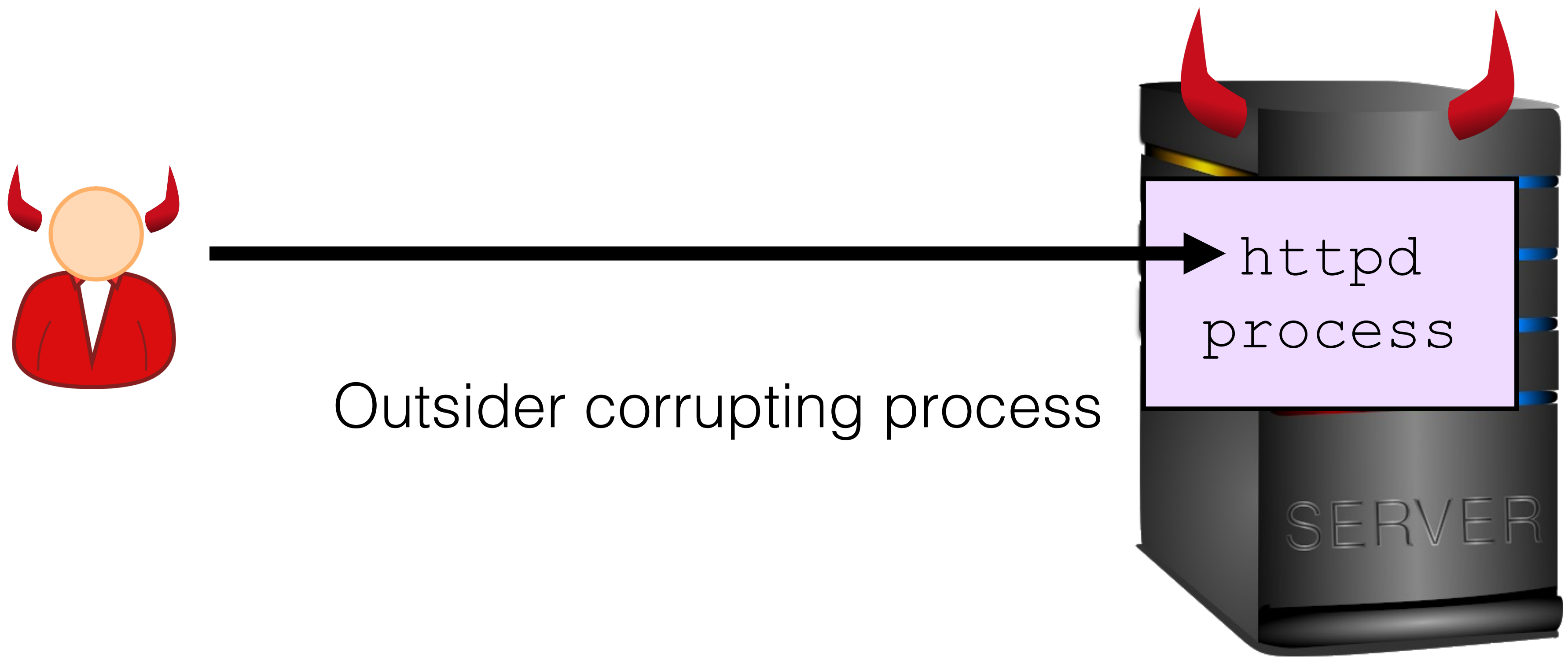
Insider escalating privilege

Example: Attacker has account “bob” on a machine and wants to access sensitive files, but:

- “bob” is not listed in ACLs of sensitive files
- “bob” also lacks sudo/root permissions

Goal: Exploit a bug in a privileged process (e.g., passwd) that lets “bob” run code with that privileged process’s permissions

Software Attacks: Another Common Setting



- Attacker wants to run code or access data on a server, but is on a remote machine

- **Goal:** Exploit a bug in a program running on the server that cause the program to run code that you send it.
  - Attacker causes Gmail server to run code that returns other users' email
  - Attacker sends a Slack msg to Bob that causes Bob's Slack app to run Attacker's code

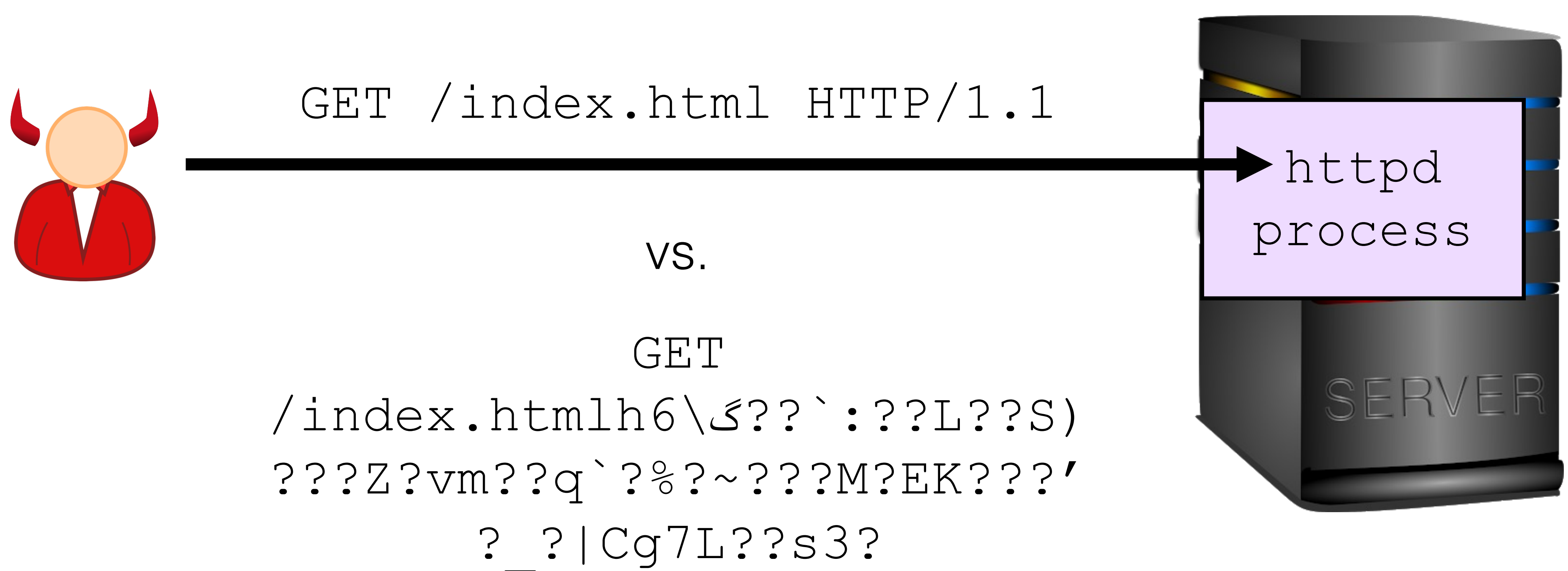
Software Vulnerabilities are Very Common

According to vulnerability researcher and author Dave Aitel:

- In **one hour** of analysis of a binary, one can find *potential* vulnerabilities
- In **one week** of analysis of a binary, one can find *at least one good vulnerability*
- In **one month** of analysis of a binary, one can find *a vulnerability that no one else will ever find.*

Two Basic Principles of Most Attacks

- Adversaries get to inject *their* bytes into *your* machine
- “Data” and “Code” are interchangeable; They are fundamentally the same “thing”.



Outline for Lecture 2

1. OS Security
    1. Review of OS Structure
    2. Abstract approaches to access control (5.2)
    3. Concrete Example: The UNIX security model
2. Software Security: Memory Safety & Control Flow Hijacking
    - Overview of software exploits
    - Memory layout and function calls in a process
    - Stack-based buffer overflow attacks

Memory Layout of a Process (in Linux)

.text: Machine executable code

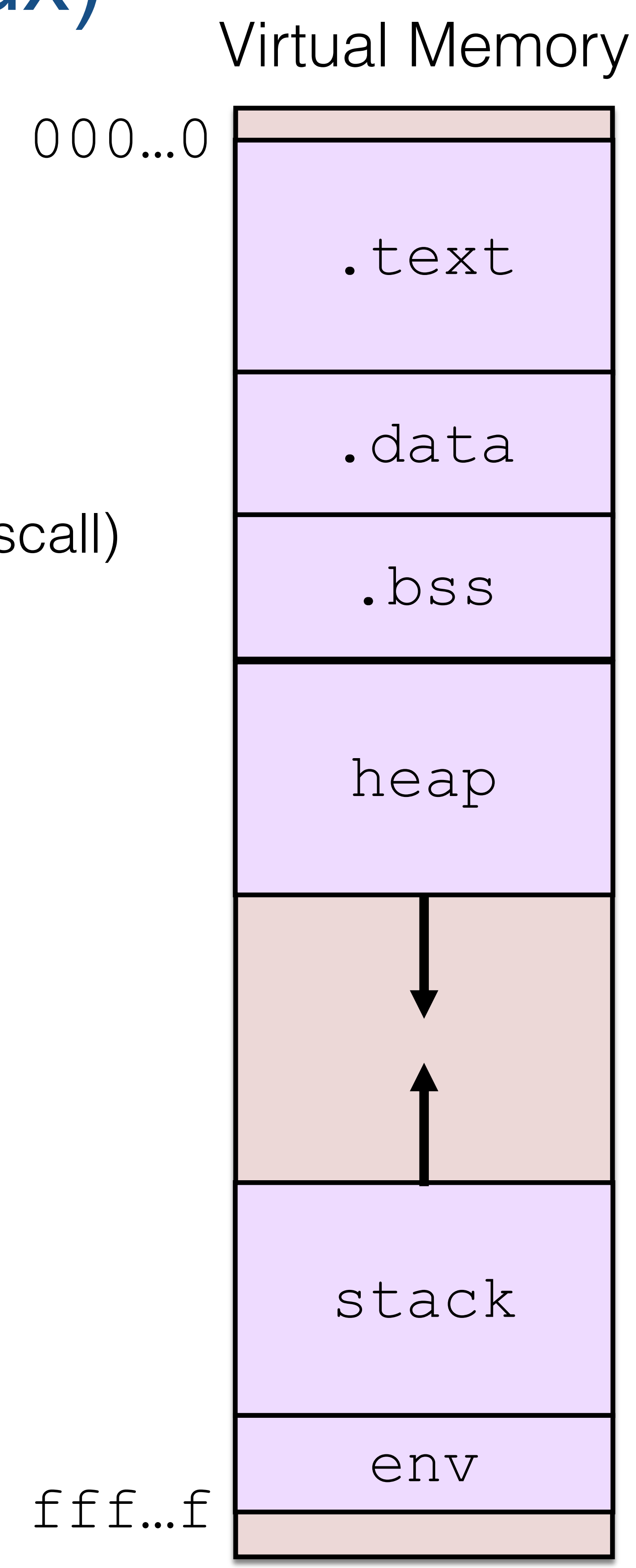
.data: Global initialized static variables

.bss: Global uninitialized variables (“block starting symbol”)

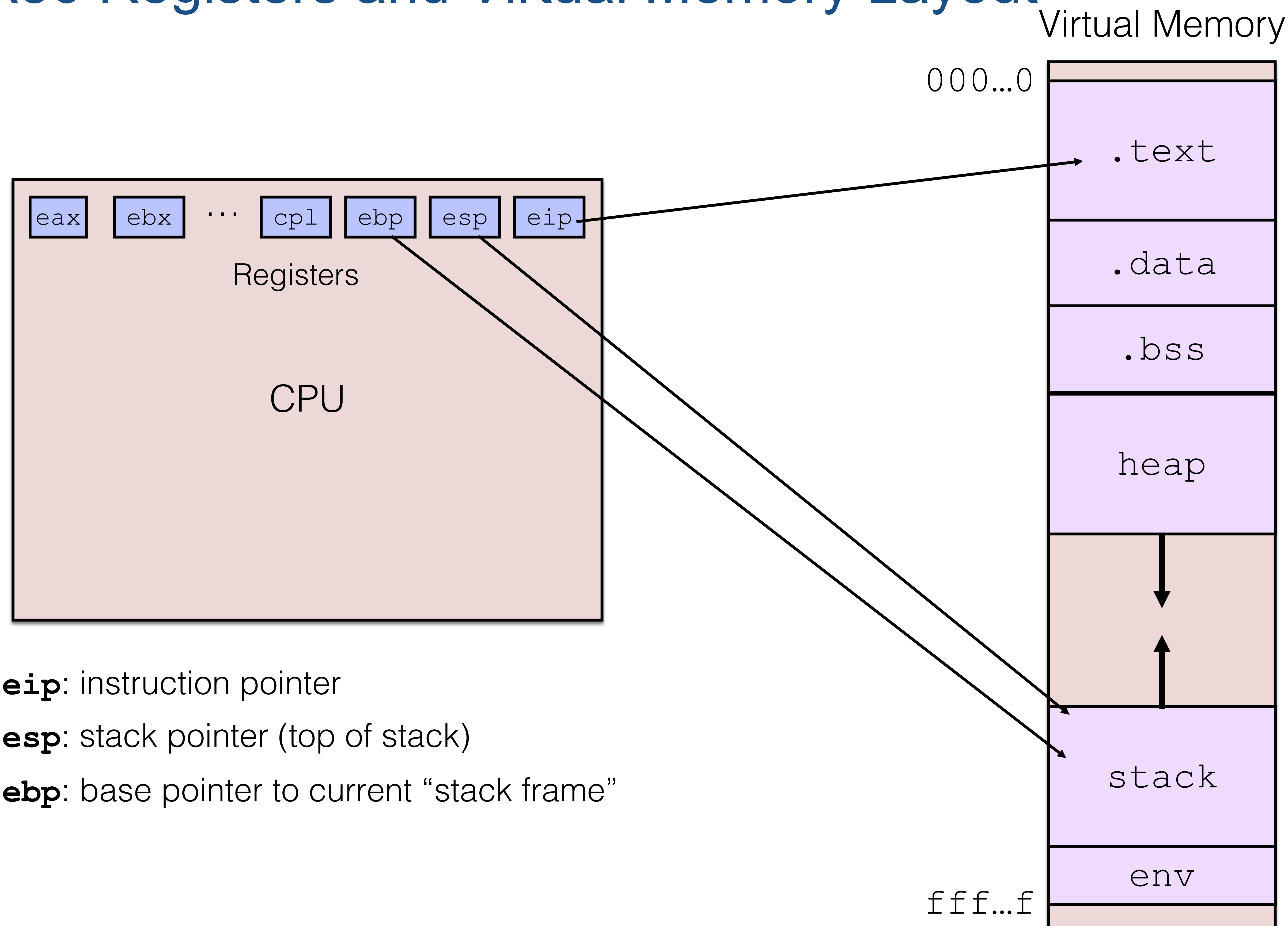
heap: Dynamically allocated memory (via `brk`/`sbrk`/`mmap` syscall)

stack: Local variables and functional call info

env: Environment variables (PATH etc)



x86 Registers and Virtual Memory Layout



eip: instruction pointer

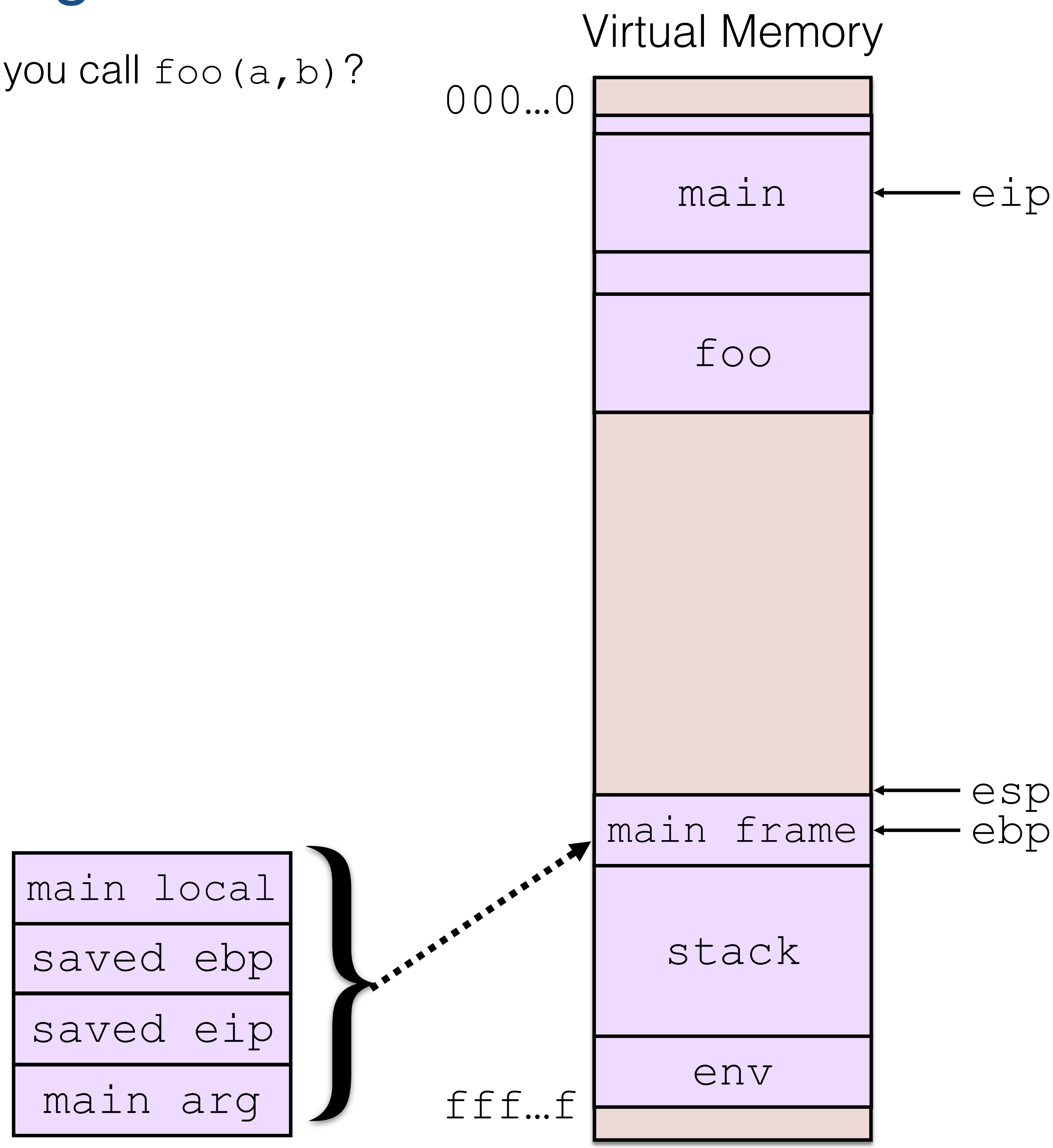
esp: stack pointer (top of stack)

ebp: base pointer to current “stack frame”

The Stack and Calling a Function in C

What happens to memory when you call `foo(a,b)`?

```
int foo(int a, int b) {
  int d = 1;
  return a+b+d;
}
```



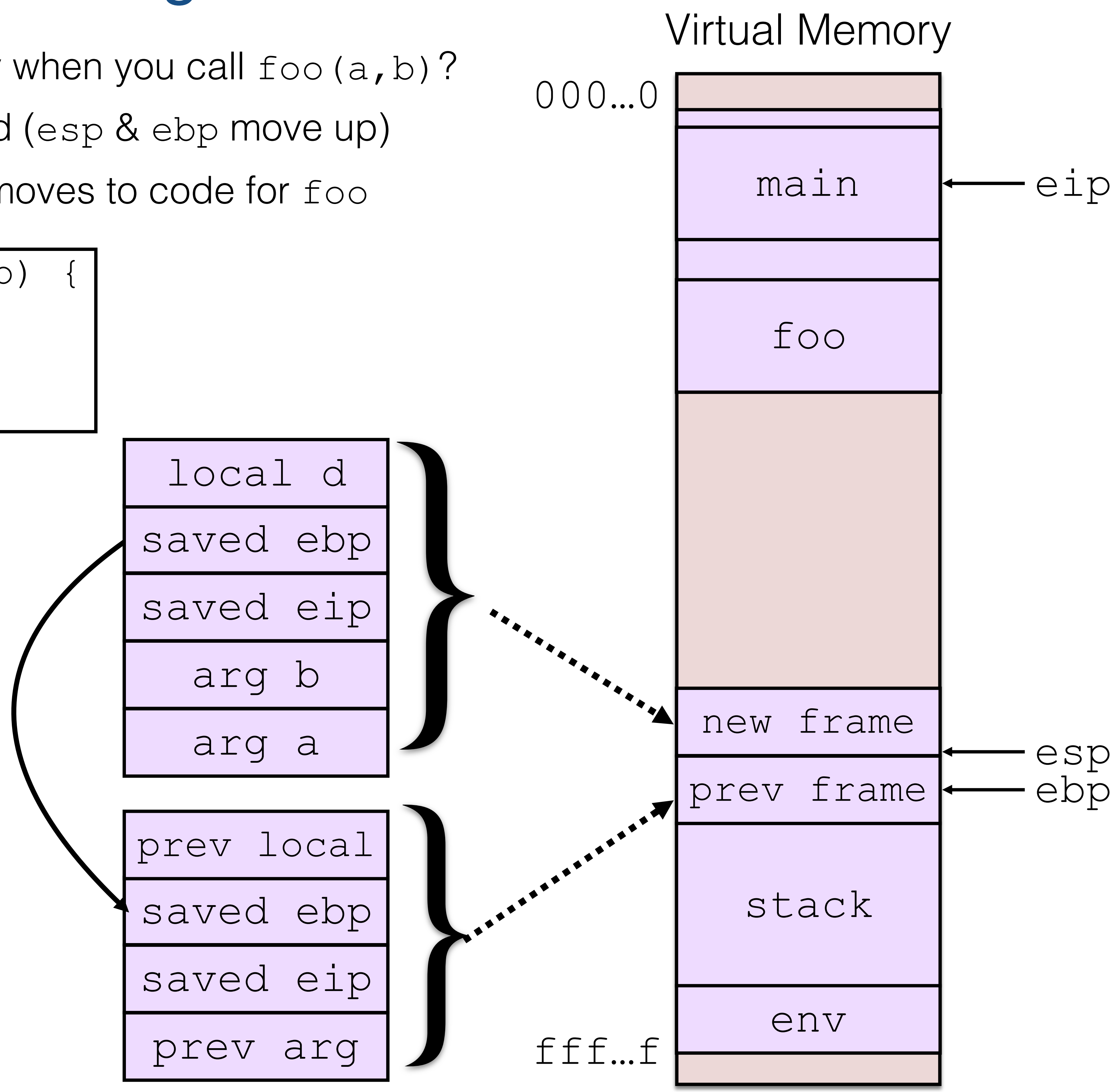
The Stack and Calling a Function in C

What happens to memory when you call `foo(a,b)`?

- A "stack frame" is added (`esp` & `ebp` move up)

- Instruction pointer `eip` moves to code for `foo`

```
int foo(int a, int b) {
  int d = 1;
  return a+b+d;
}
```

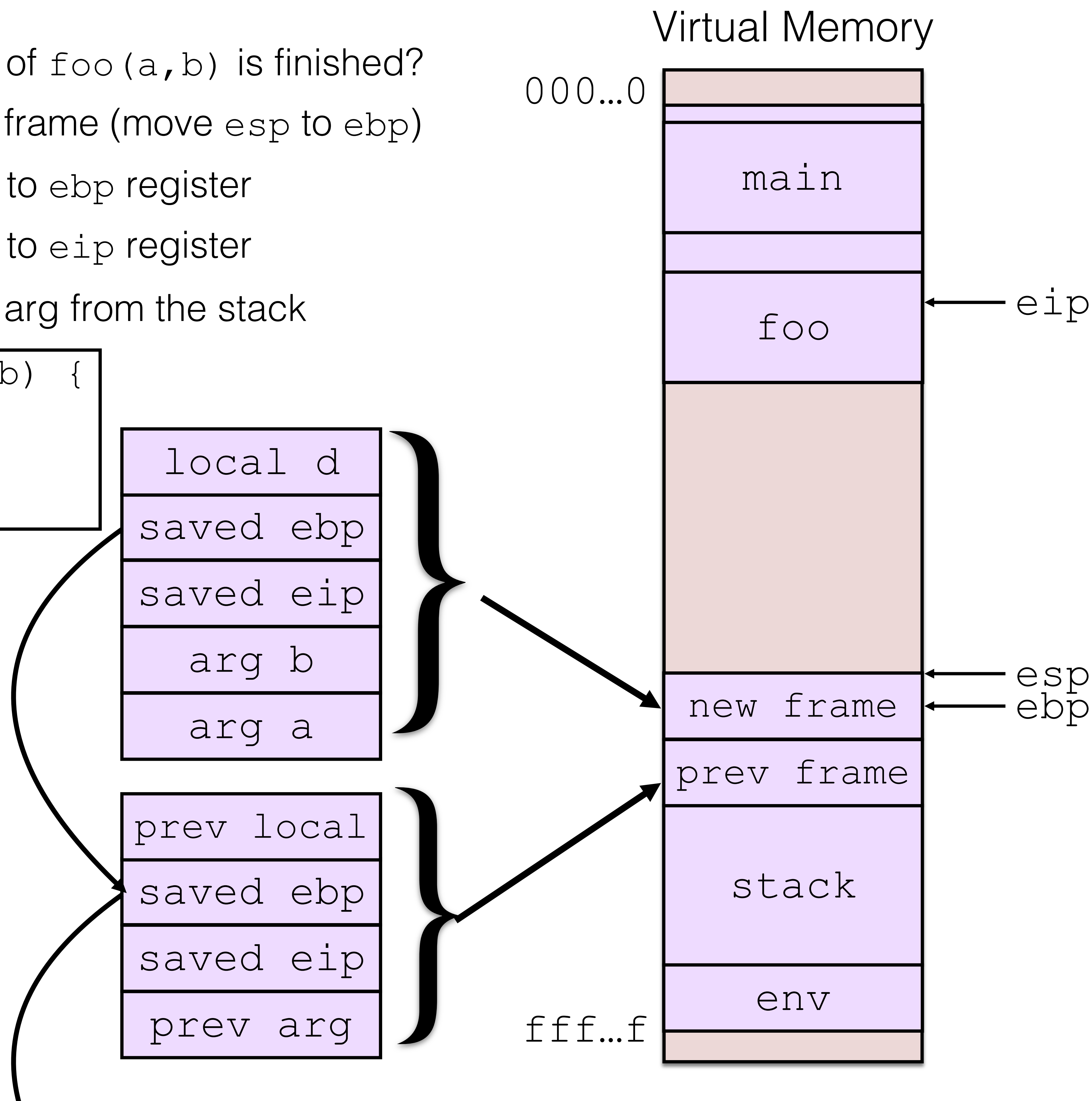


Returning from a function

What happens after code of `foo(a,b)` is finished?

- Pop the function's stack frame (move `esp` to `ebp`)
- Pop (moves) saved `ebp` to `ebp` register
- Pop (moves) saved `eip` to `eip` register
- Caller (main) pops foo's arg from the stack

```
int foo(int a, int b) {
  int d = 1;
  return a+b+d;
}
```



Outline for Lecture 2

1. OS Security
    1. Review of OS Structure
    2. Abstract approaches to access control (5.2)
    3. Concrete Example: The UNIX security model
2. Software Security: Memory Safety & Control Flow Hijacking
    - Overview of software exploits
    - Memory layout and function calls in a process
    - Stack-based buffer overflow attacks

Classic Attack: Overflowing a buffer on the stack

Function `bad` copies a string into a 64 character buffer.

— strcpy continues copying until it hits NULL character!

— If s points to longer string, this overwrites rest of stack frame.

— Most importantly saved `eip` is changed, altering control flow.

```
void bad(char *s) {
  char buf[64];
  strcpy(buf, s);
}
```

Classic Attack: Overflowing a buffer on the stack

Function `bad` copies a string into a 64 character buffer.

— strcpy continues copying until it hits NULL character!

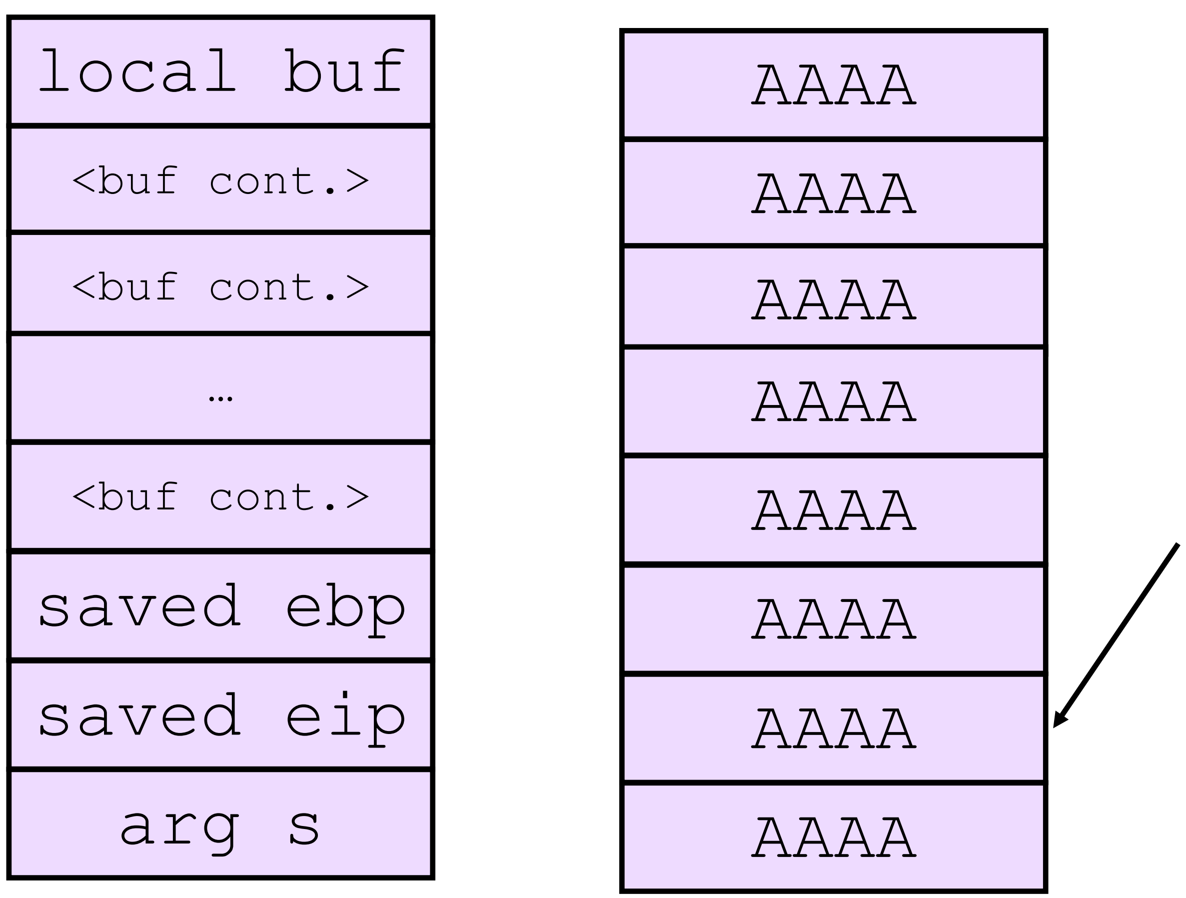
— If s points to longer string, this overwrites rest of stack frame.

— Most importantly saved `eip` is changed, altering control flow.

```
void bad(char *s) {
  char buf[64];
  strcpy(buf, s);
}
```

`s="AAAA...AAAA"` (70 or more characters)

Frame before `strcpy`   Frame after `strcpy`



saved `eip` should be here!
`AAAA=0x41414141` will be used
as return address

What will happen?   SEGFAULT!

How to exploit a stack buffer overflow

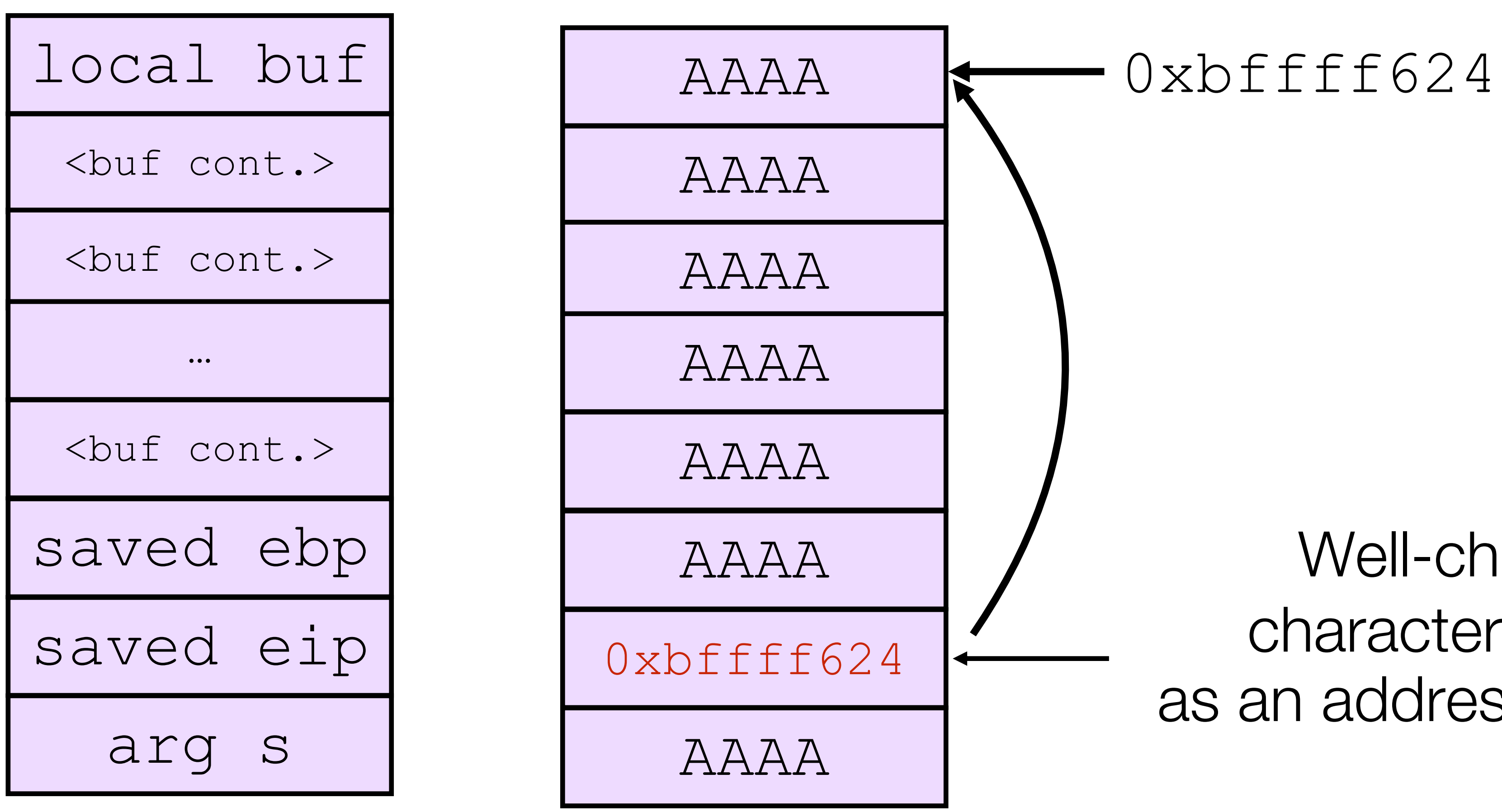
Suppose attacker can cause bad to run with an `s` it chooses.

- Step 1: Set correct bytes to *point back to input*(!)

```
void bad(char *s) {
  char buf[64];
  strcpy(buf, s);
}
```

`s="AAAAA…AAAA\x24\xf6\xff\xbfAAA…"`

Frame before strcpy   Frame after strcpy



Well-chosen characters used as an address for `eip`!

What will happen? Illegal instruction!

How to exploit a stack buffer overflow

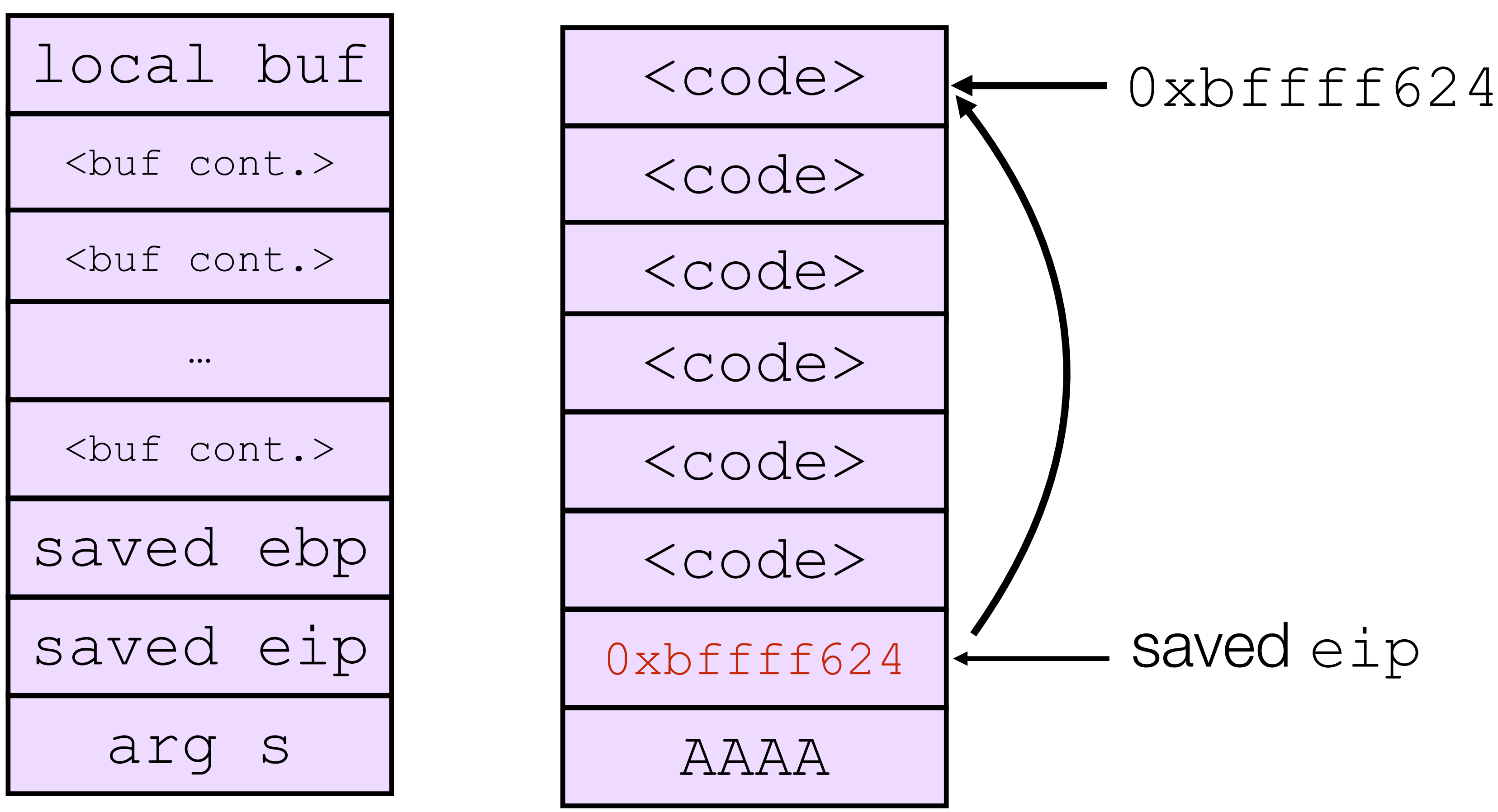
Suppose attacker can cause bad to run with an `s` it chooses.

- Step 1: Set correct bytes to *point back to input(!)*
- Step 2: Make input *executable machine code(!)*

```
void bad(char *s) {
  char buf[64];
  strcpy(buf, s);
}
```

```
s="<machine code>\x24\xf6\xff\xbfAAA…"
```

Frame before strcpy   Frame after strcpy



What will happen?   Success!

What to put in for <code>?

The possibilities are endless!

— Spawn a shell

— Spawn a new service listening to network

— Change files

— ...

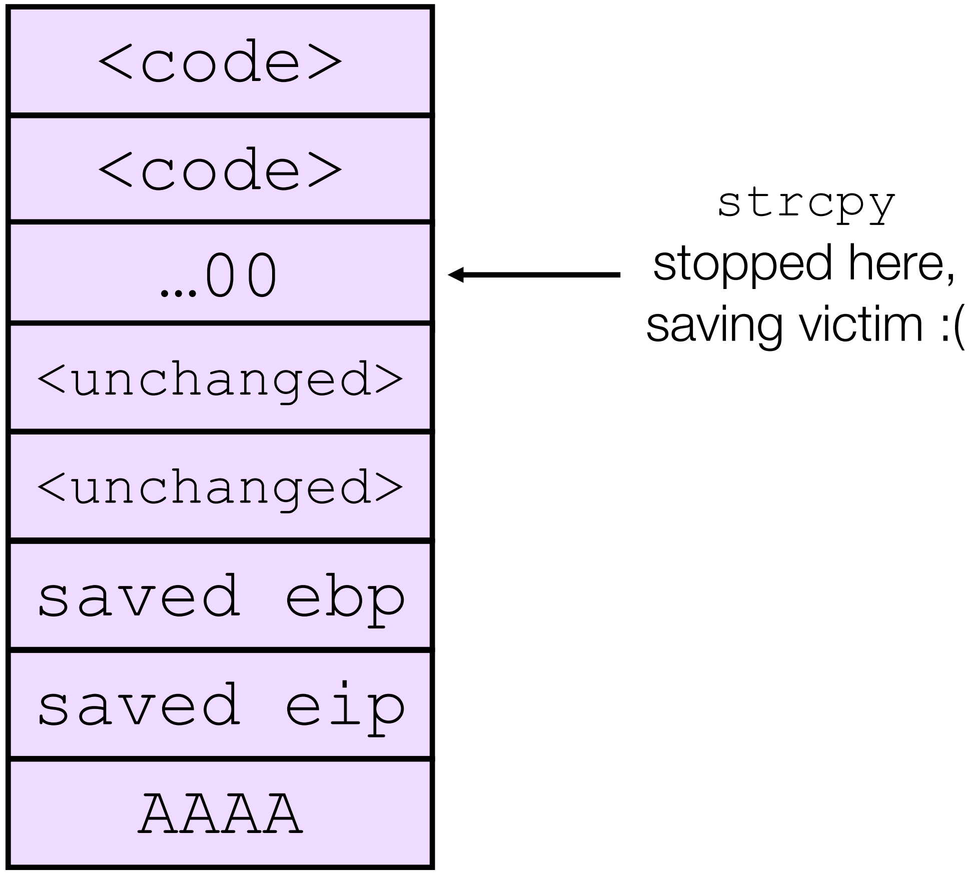
`s="<machine code>\x24\xf6\xff\xbfAAA…"`

But wait… what about NULL bytes?

Solution: Find machine instructions with no NULLs!

— Can even find machine code with all alpha bytes.

Frame after strcpy



Example Shellcode

```
char shellcode[] =
“\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
“\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

Basically equivalent to:

```
#include <stdio.h>
void main() {
  char *name[2];
  name[0] = “/bin/sh";
  name[1] = NULL;
  execve(name[0], name, NULL);
}
```

Finally, where did that magic address come from?

Assignment: GDB is your friend ☺

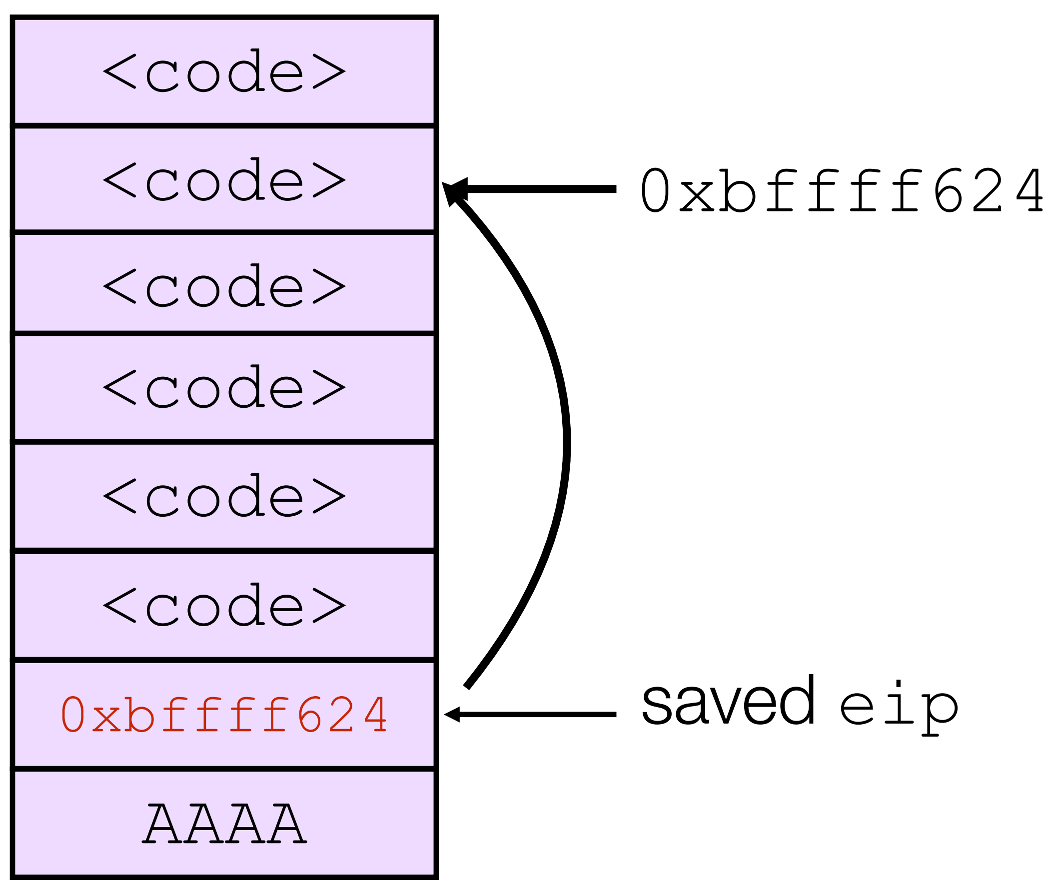
```
void bad(char *s) {
  char buf[64];
  strcpy(buf, s);
}
```

Two issues:

— Need address to jump to beginning of shellcode

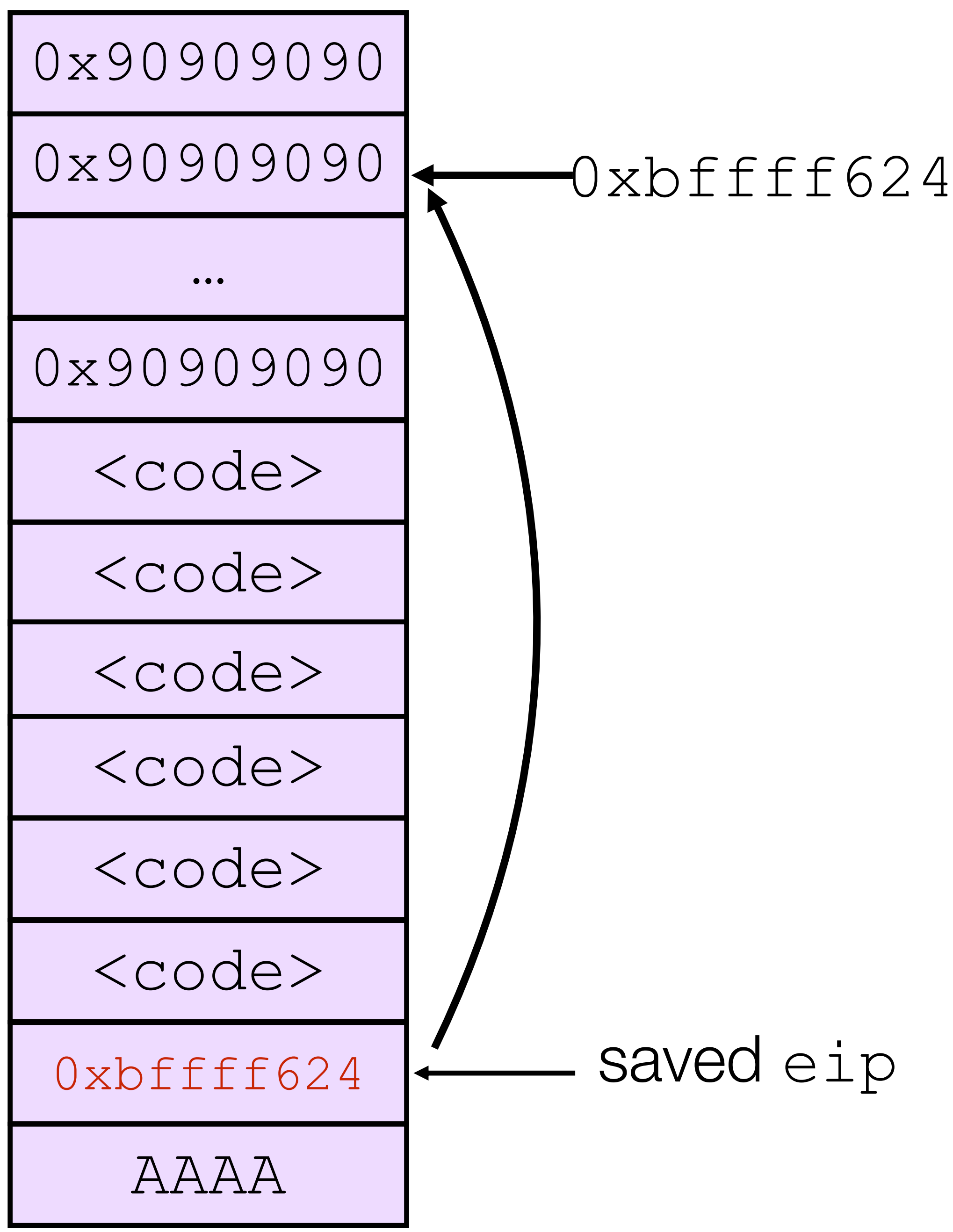
— Need to know where to overwrite saved EIP

s="<code>\x24\xf6\xff\xbfAAA…"



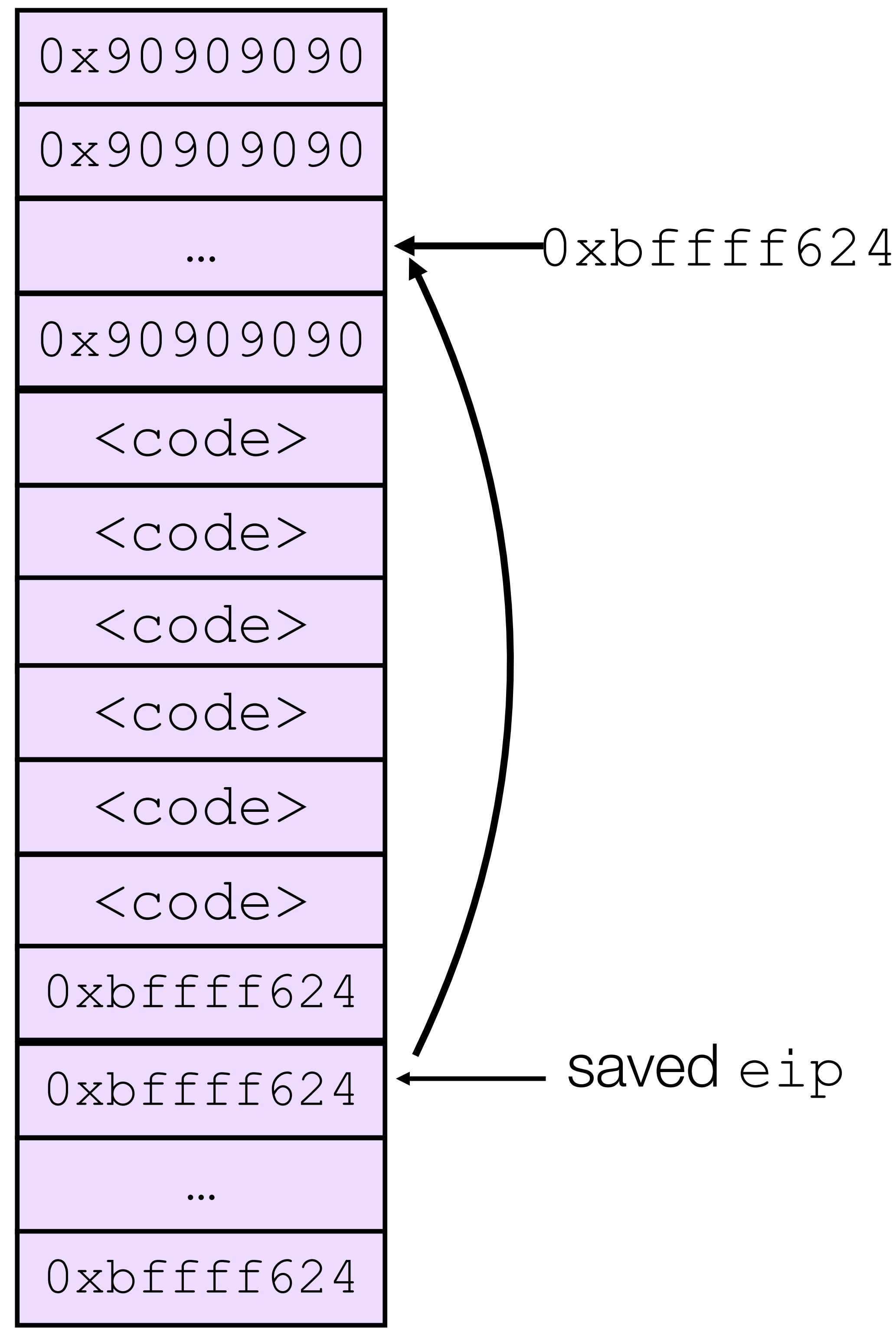
Technique #1: NOP Sleds

- Instruction `0x90` is “`xchg eax, eax`”, i.e. does not thing. This is a “No Op” or “NOP”.
- Just add a ton of NOPs (as many as you can, even many MB) and hope pointer lands there



Technique #2: Placing malicious EIP

— Simple: Just copy it many times



Brief Recap of Software Attacks

- Bugs in code can allow attackers to bypass OS security and access control policies

- The CPU stores critical “control flow” information on the stack
  - Saved EIP & Saved EBP: controls what the CPU does after a function returns
  - Buffer overflow attack: vulnerable program doesn’t check if a (stack) buffer has enough space to hold copied data
  - Attacker can provide input of {malicious code} + {new return address, that points to the malicious code}
  - CPU will run the attacker’s code, instead of the program’s actual code

The End