OS Security and Software Security

CMSC 23200, Spring, Lecture 2

Grant Ho & David Cash

University of Chicago
Mar 26, 2026

Logistics

- HW #0 is due TONIGHT (3/26) by 11:59pm
  - Course policy quiz on Gradescope
  - Upload file with SSH public on Canvas
- We do have a mandatory midterm this year
  - Corrected link to this year's website/schedule

Logistics

- Assignment 1: two-parts, both due next Thurs (4/2):
  - Part 1a: Threat modeling released Tomorrow
  - Part 1b: Linux Basics + TOCTOU attacks released by Monday
- Office hours start next week (hours now on website)

Today's Class

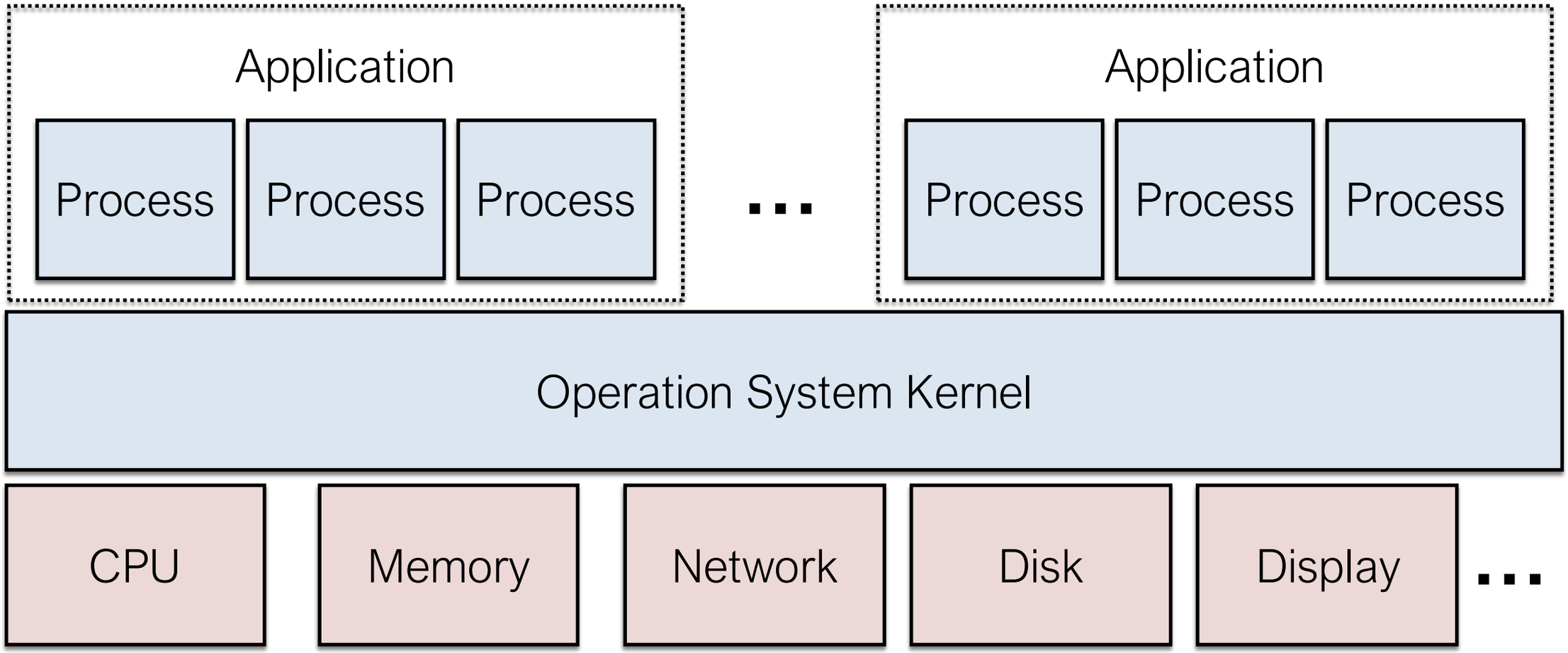
1. **OS Security**:
   How do we ensure that users & programs only access resources they're allowed to?

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

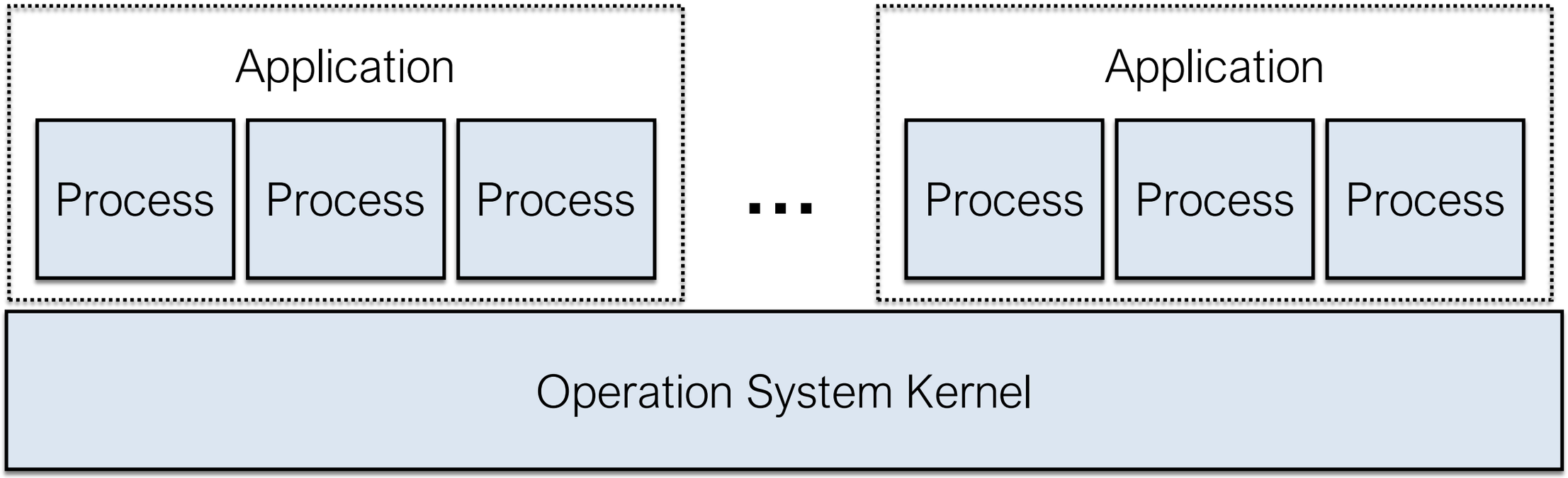
Outline for Lecture 2

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

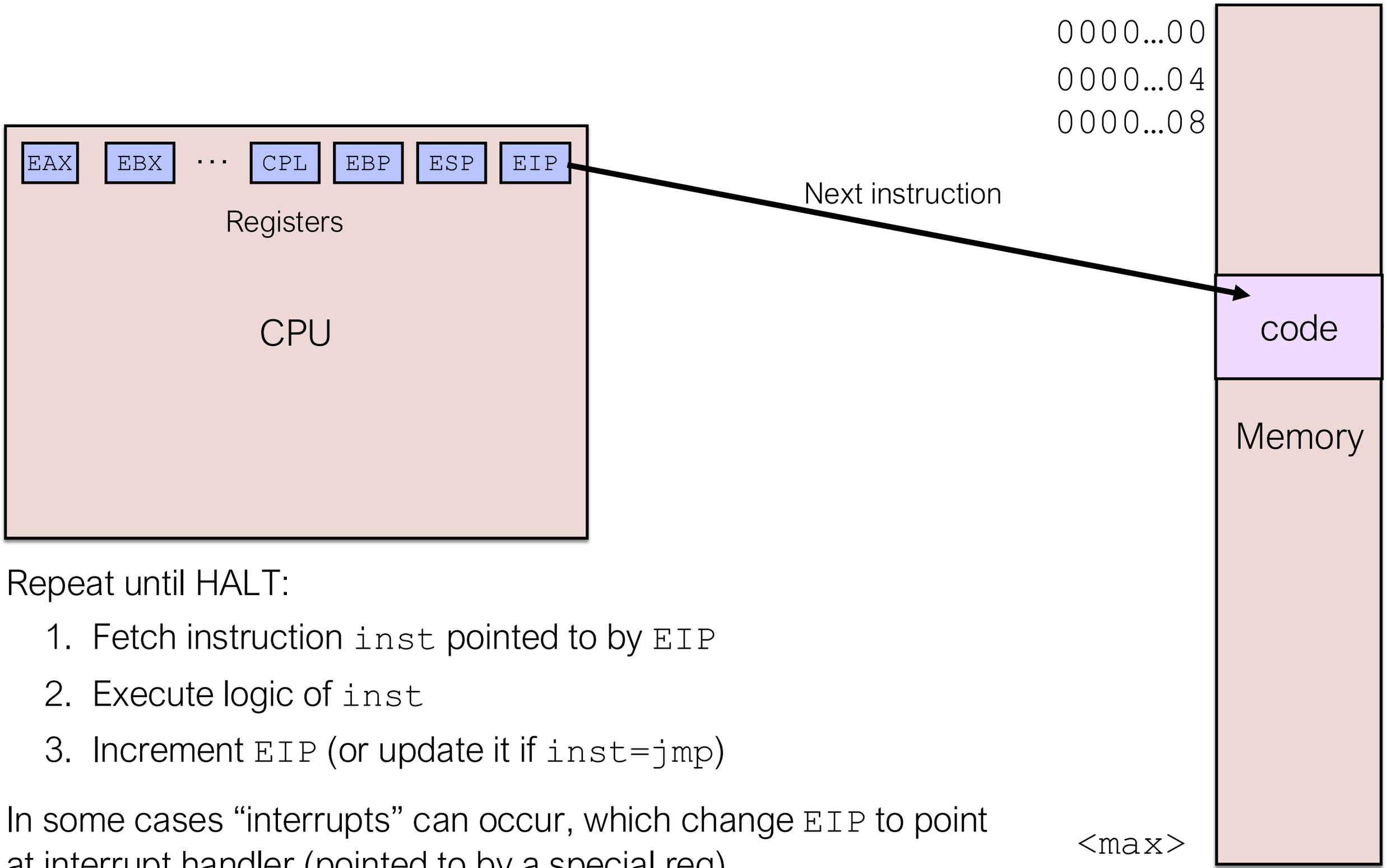


Security/safety: Must protect processes from each other, protect hardware, …

Questions, though:

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

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

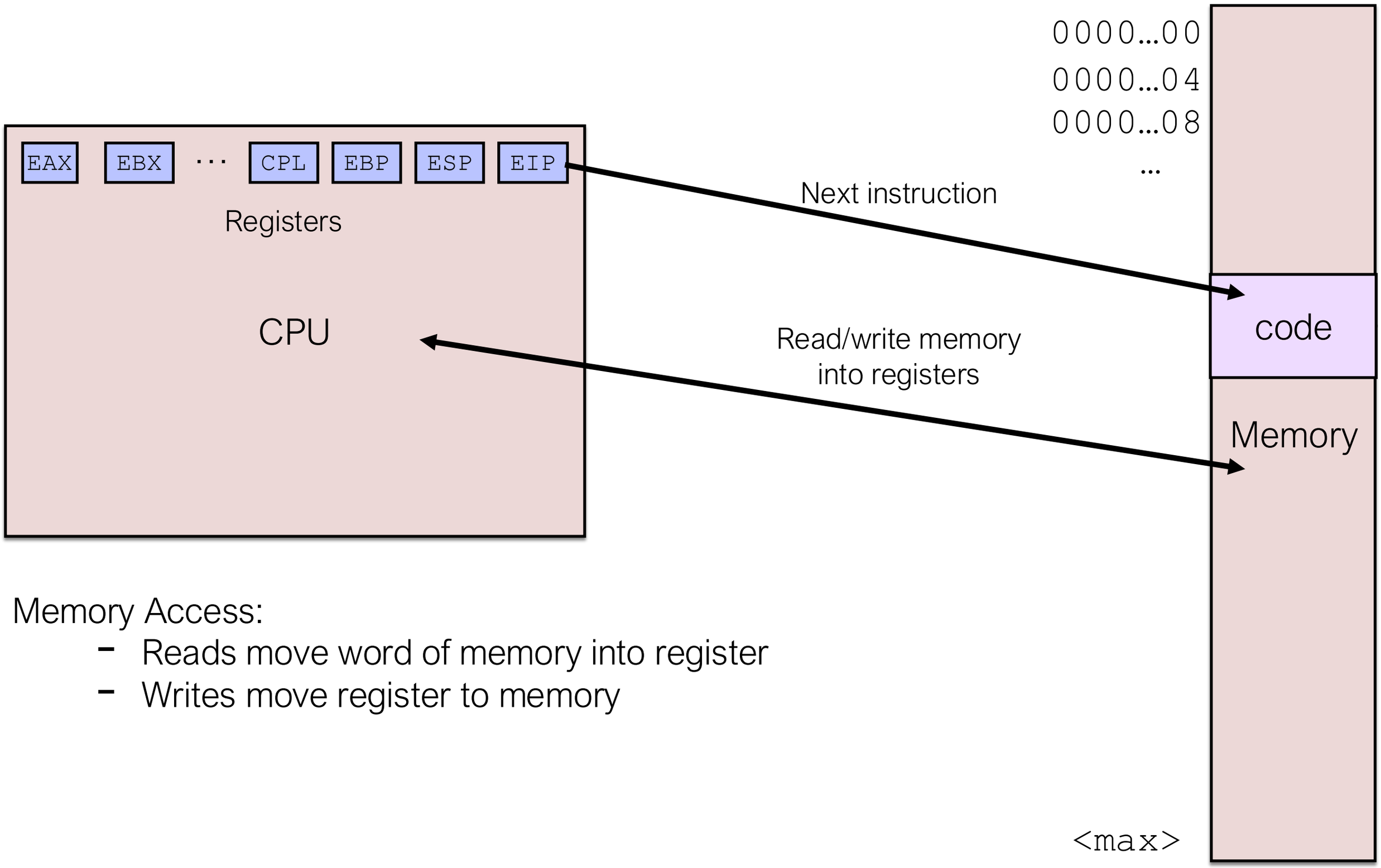


Repeat until HALT:

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)

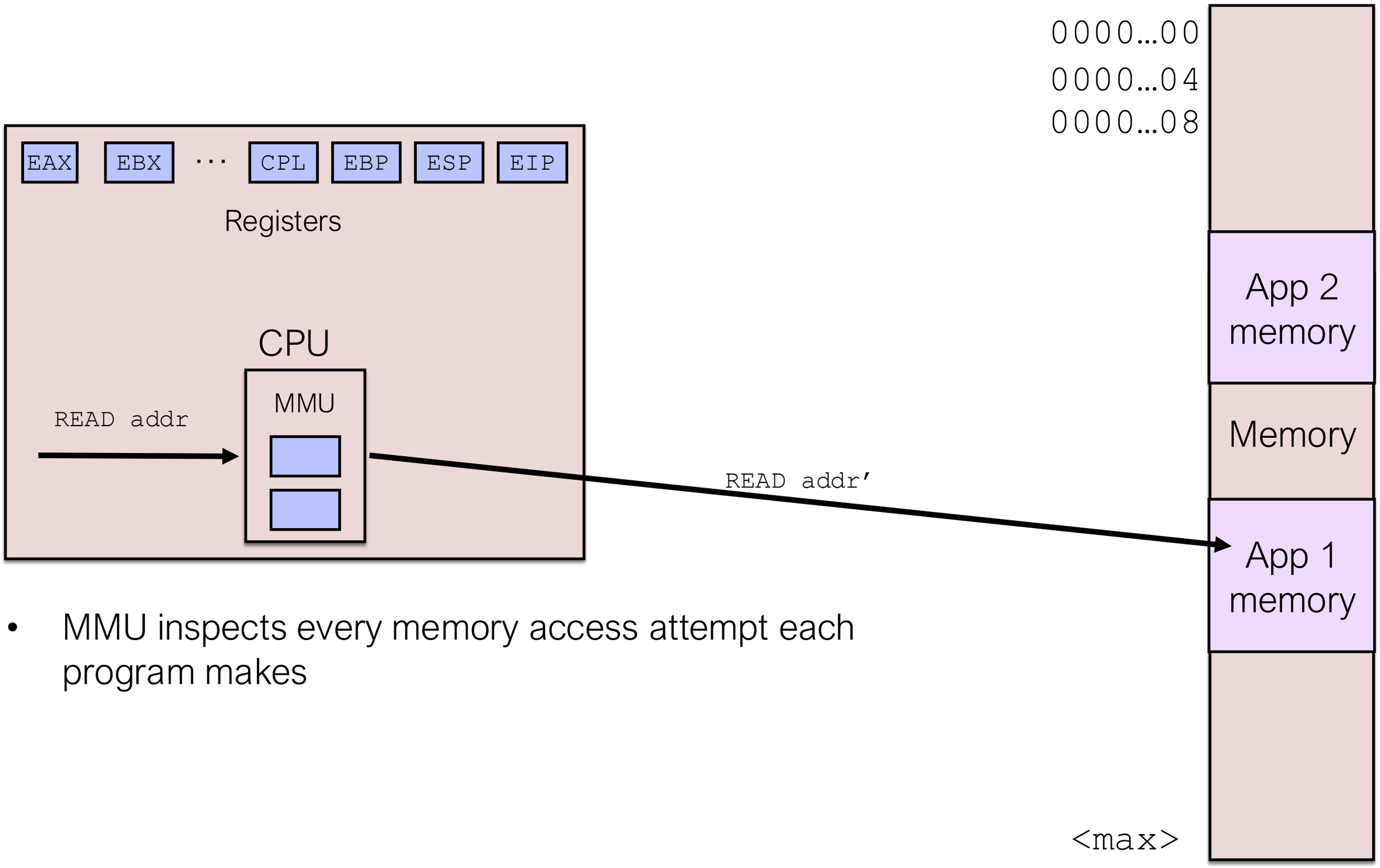


Memory Access:

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

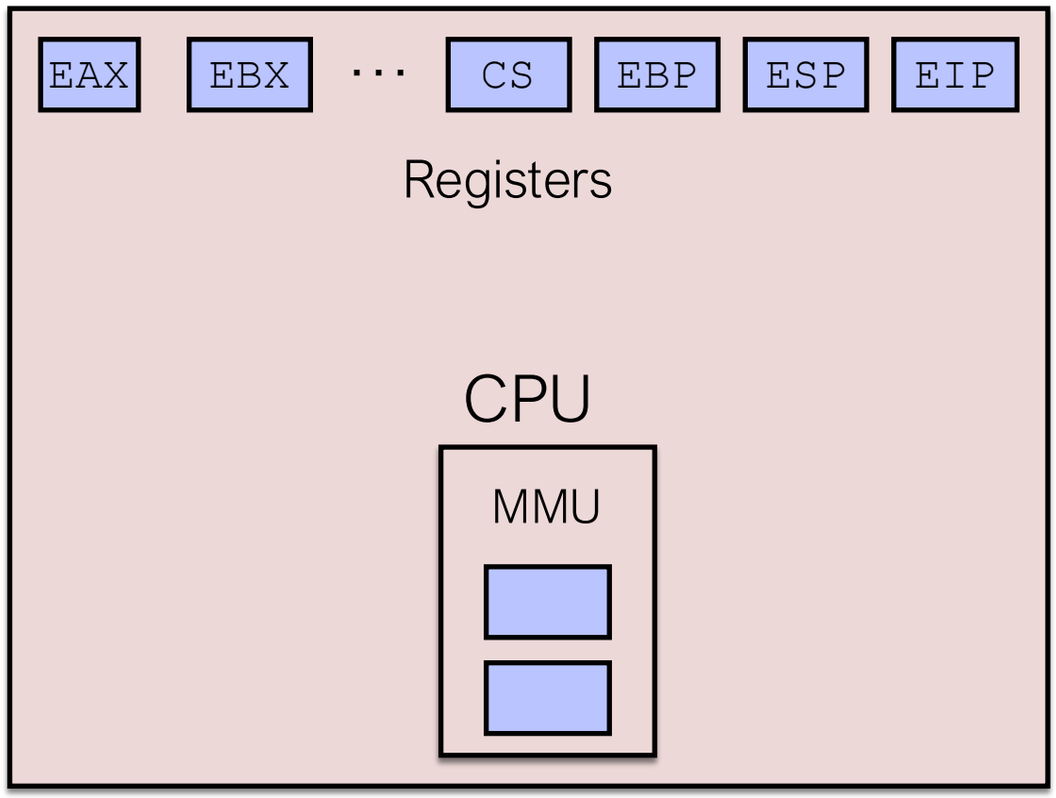
<max>

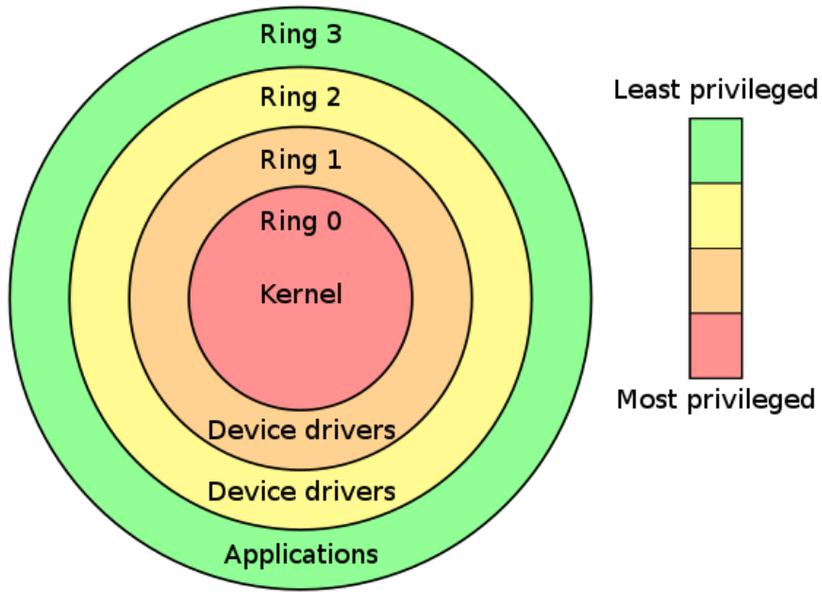
Memory Management Unit (MMU)

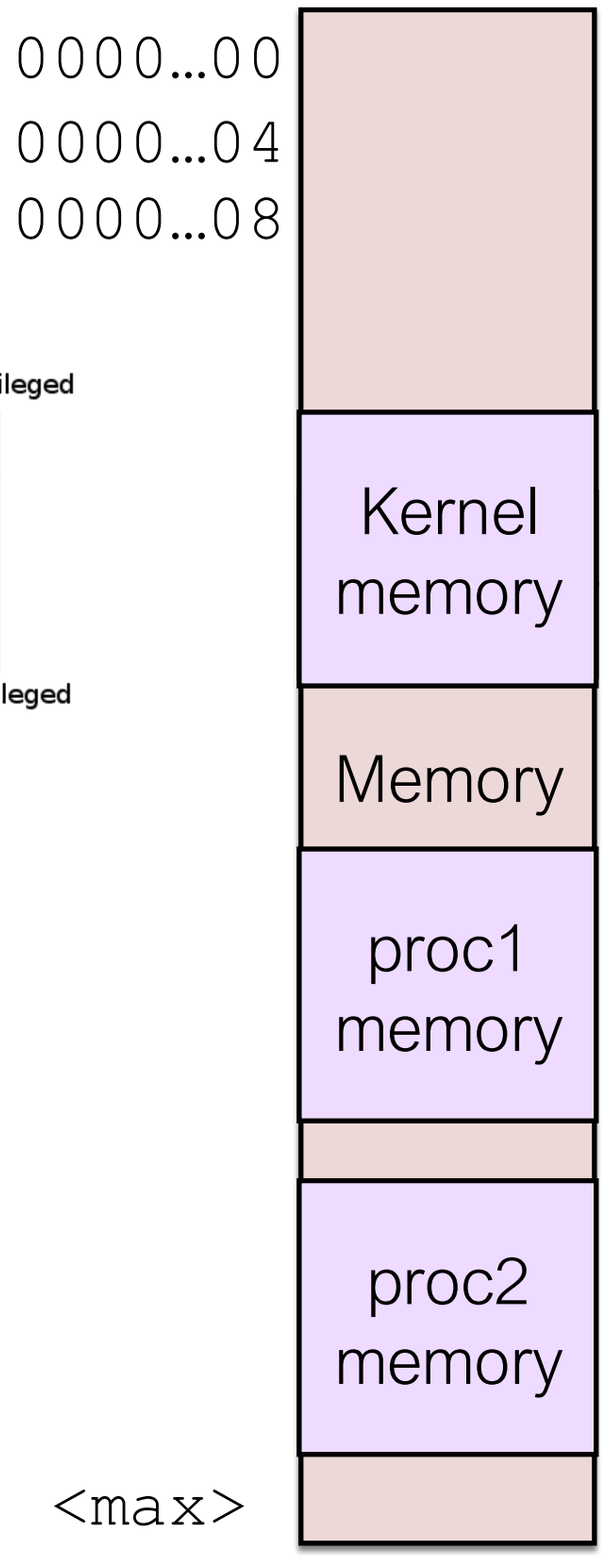


- MMU inspects every memory access attempt each program makes

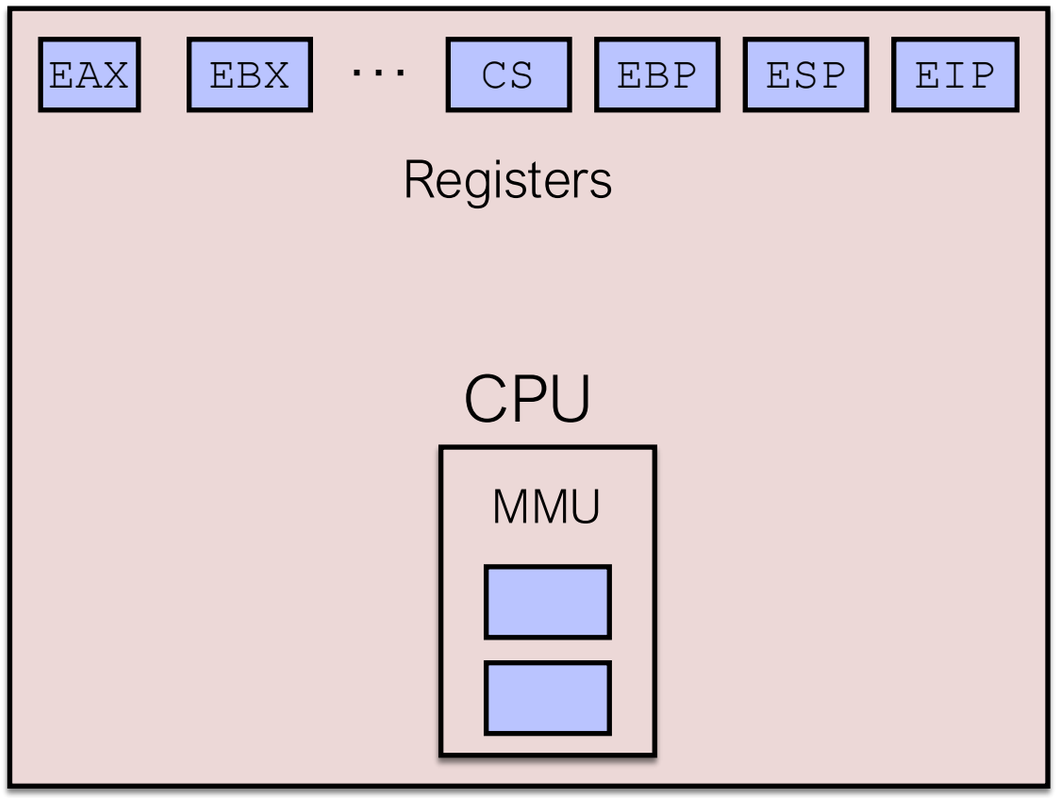
Isolation in x86: It all comes down to CPL

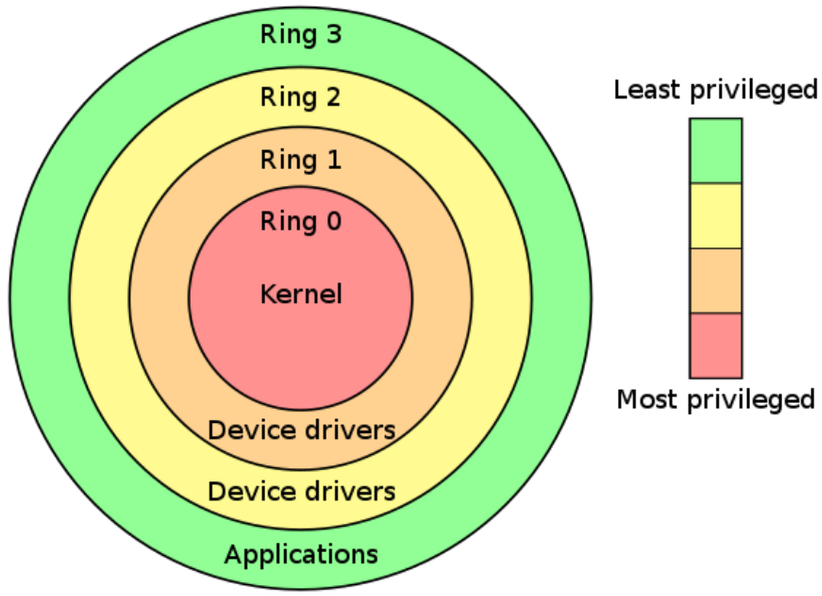


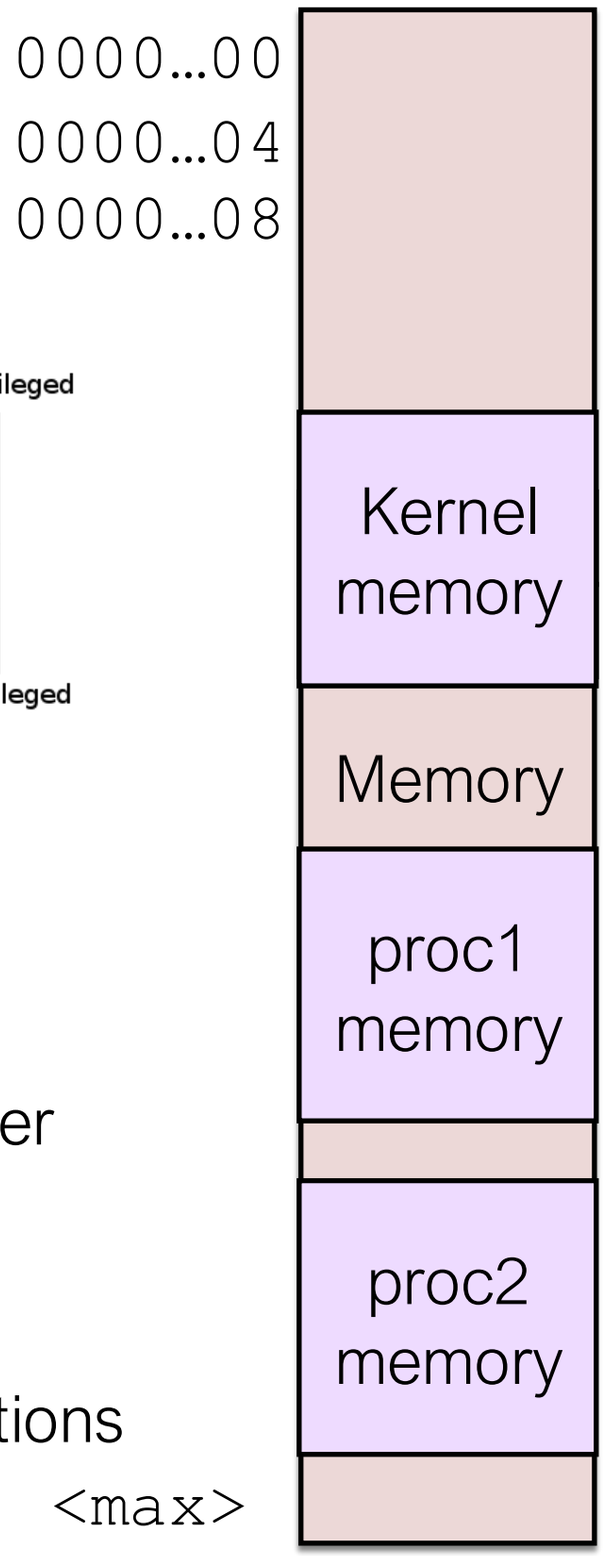




Isolation in x86: It all comes down to CPL

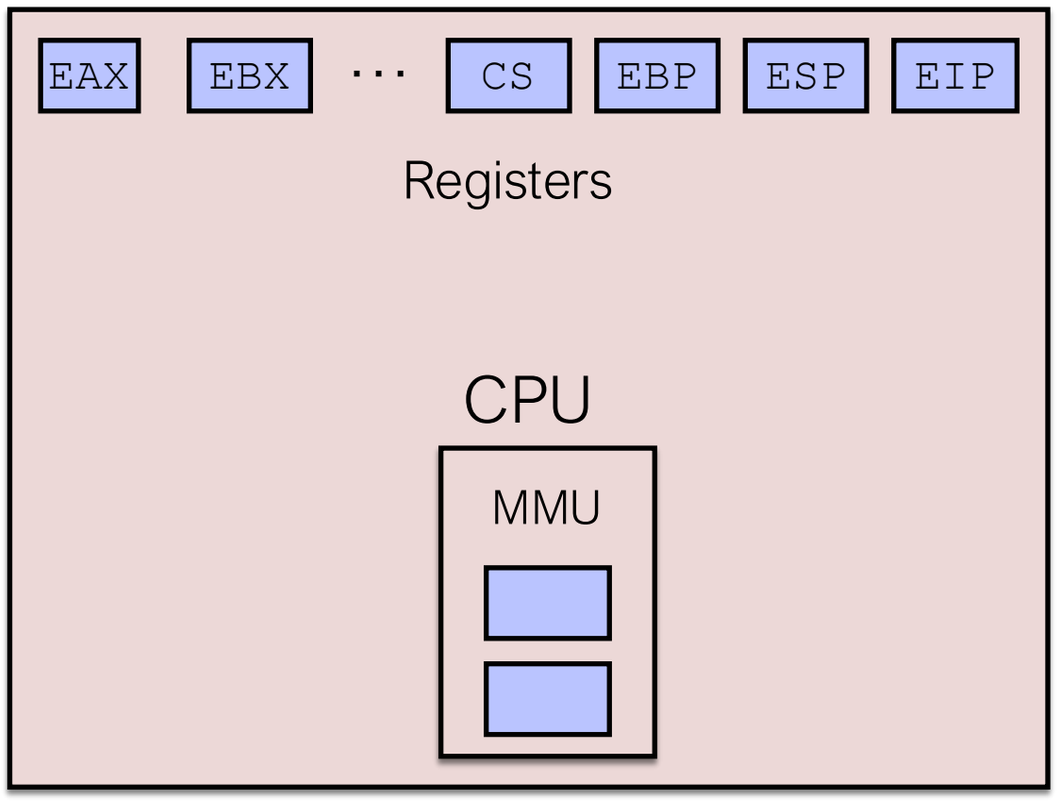






- `CPL` is "current privilege level", two designated bits in `CS` register
- If `CPL = 0`: Then processor will execute any instruction
- If `CPL = 3`: Then processor will only execute subset of instructions

Isolation in x86: It all comes down to CPL



If `CPL`=0, then CPU **will** allow…

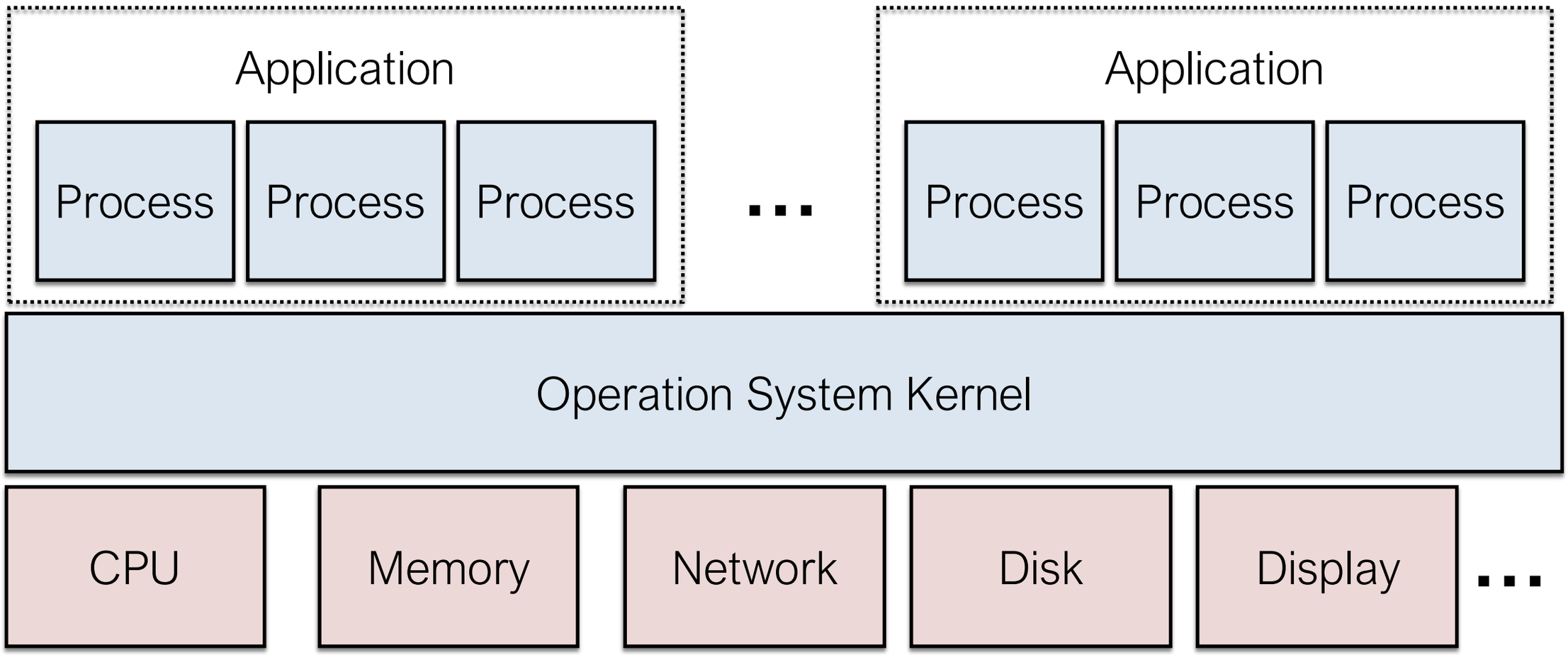
- Direct access to (almost) any addr
- Changes to (almost) any register
- Changes internal state of MMU
- Including setting `CPL`=3!

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…

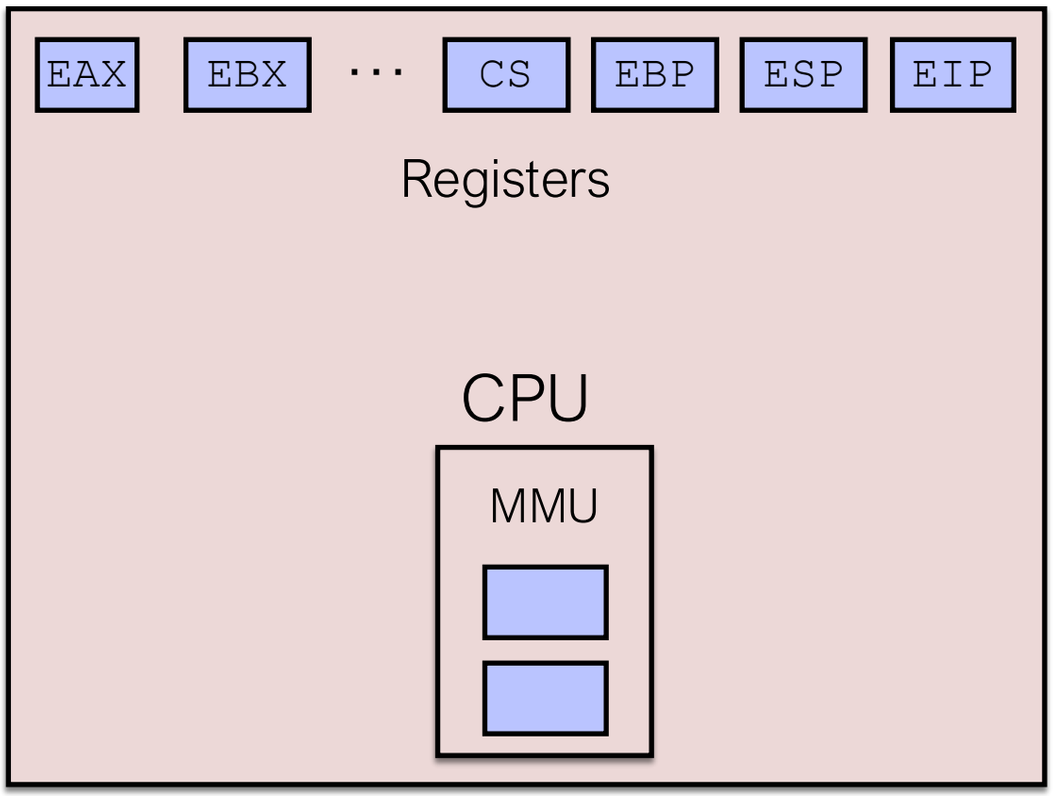


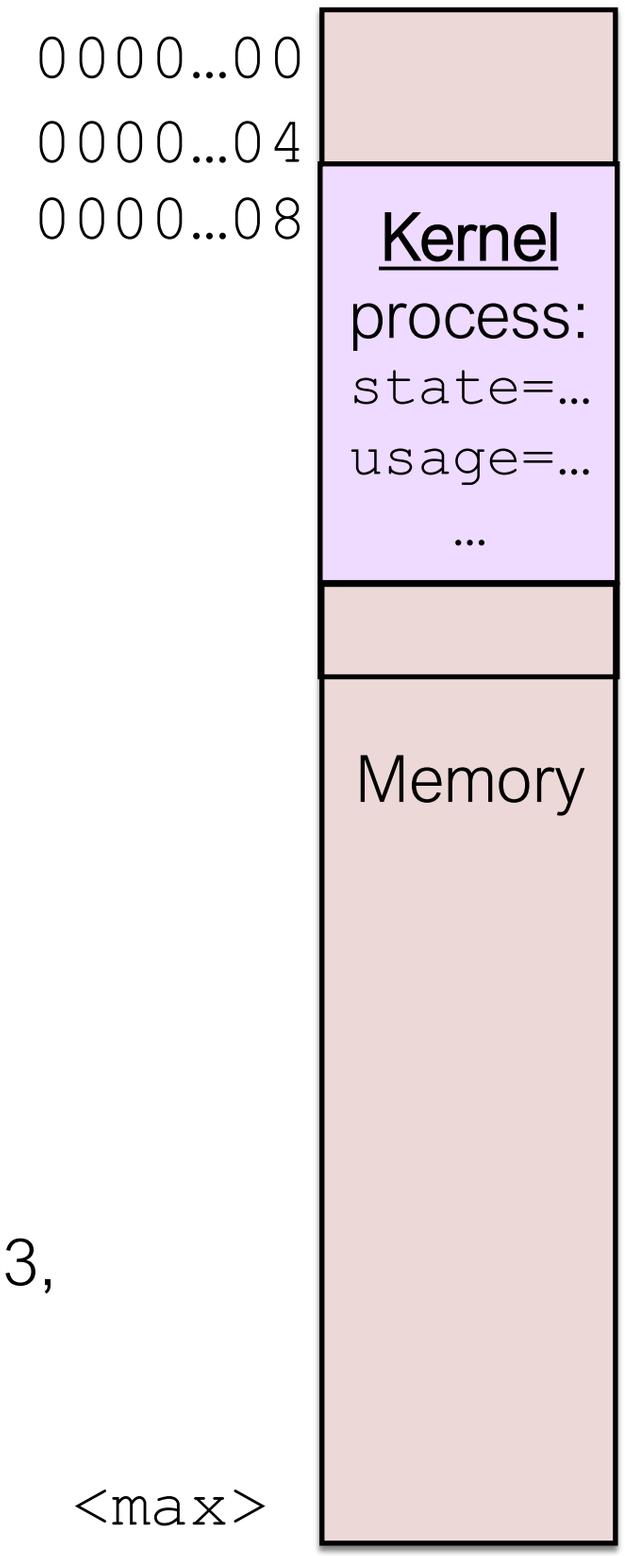
Questions, though:

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

The CPL!

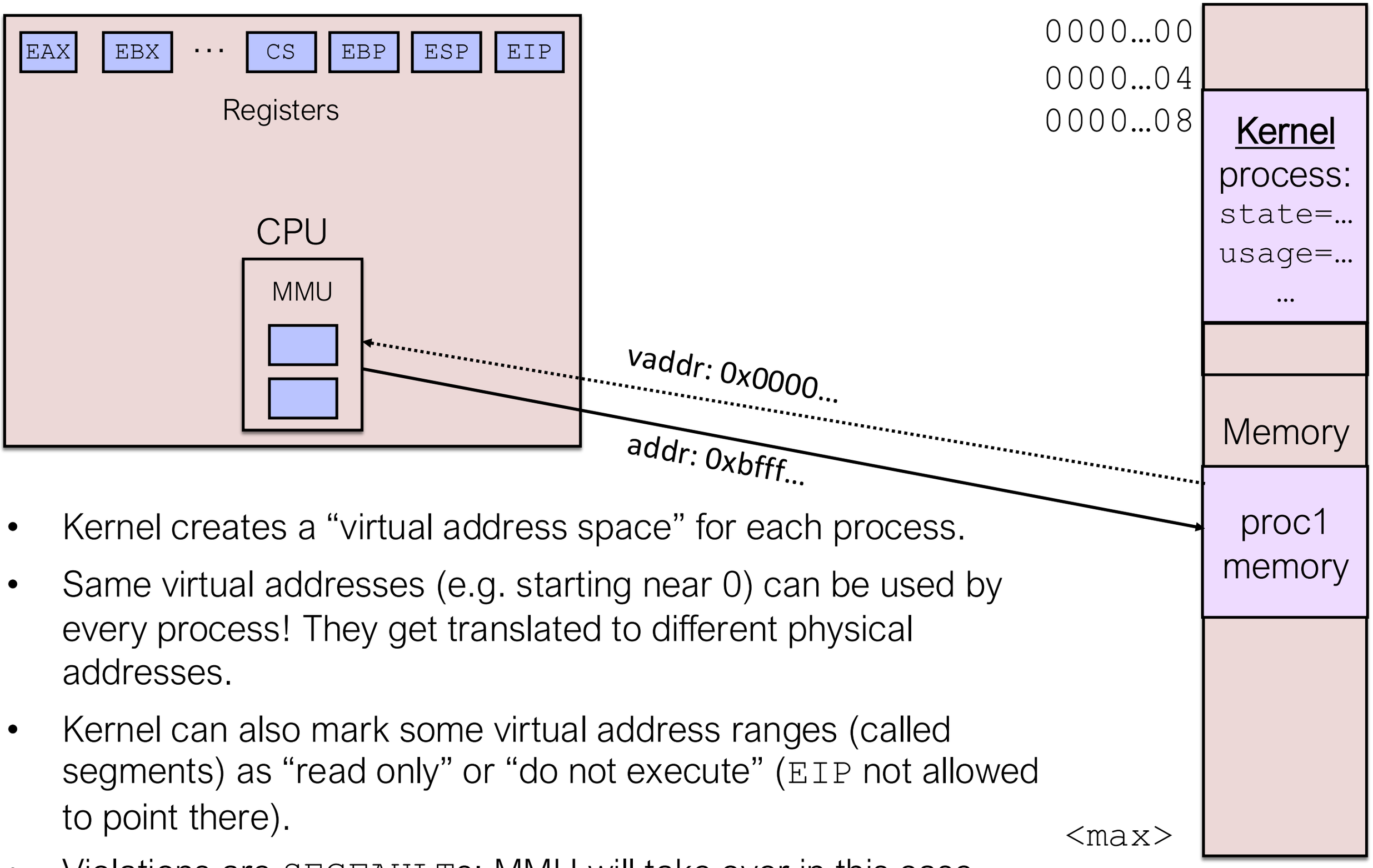
What *is* a process?





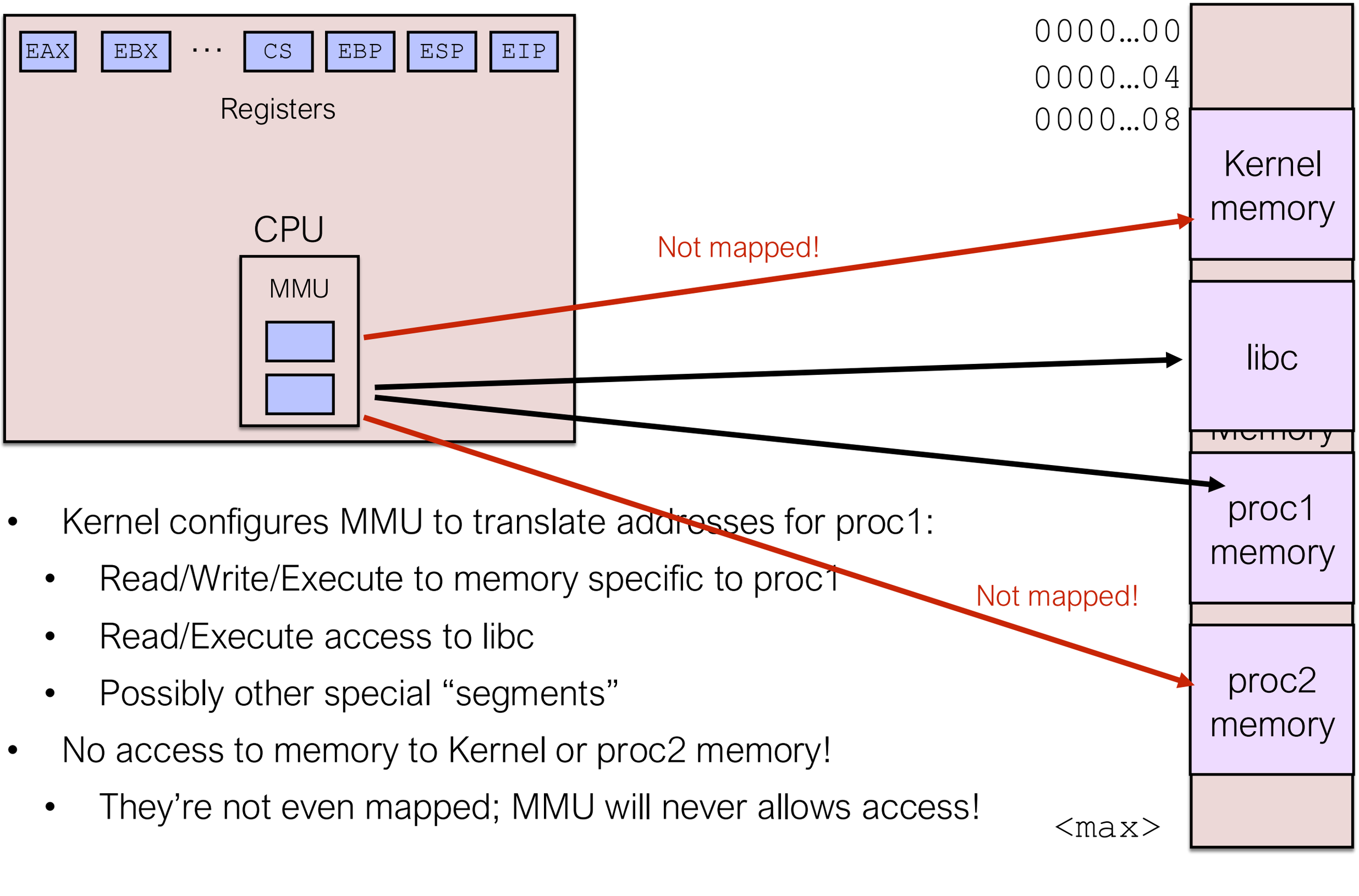
- One Answer: A data structure the kernel manages, including:
  - MMU configuration
  - Register values
- To run application code: Kernel loads these values, sets CPL=3, and turns over CPU control "to the process" (i.e. set EIP)
- If kernel regains control, it can save these values to swap process out

Handling Memory for a Process



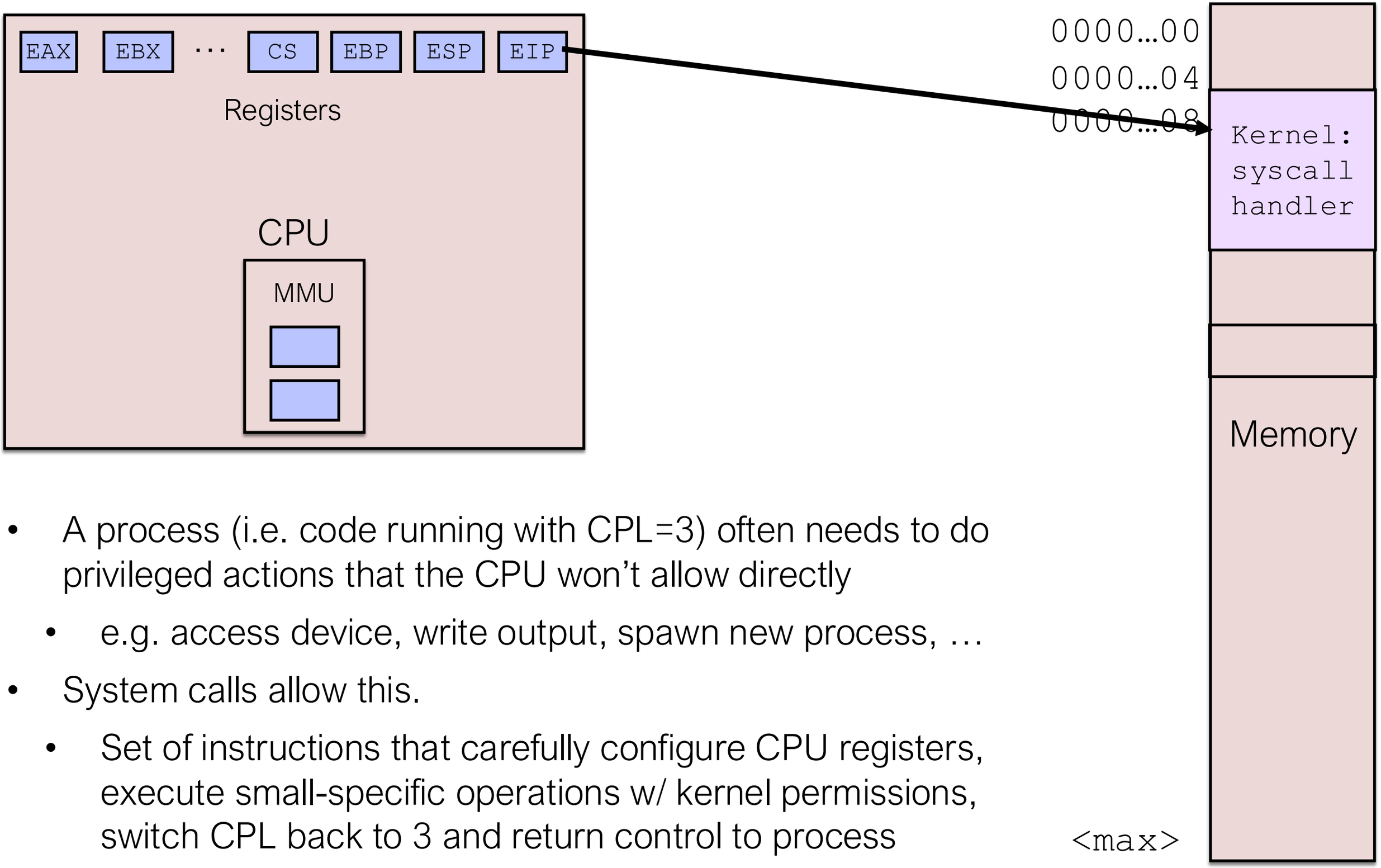
- Kernel creates a “virtual address space” for each process.
- Same virtual addresses (e.g. starting near 0) can be used by every process! They get translated to different physical addresses.
- Kernel can also mark some virtual address ranges (called segments) as “read only” or “do not execute” (`EIP` 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

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.

Principle of Least Privilege

Subjects (system entities like users) should:

- Only have access to the data and resources needed to perform their authorized tasks AND
- **Nothing more than this necessary access**

Real World Examples:

- Faculty can only change grades for classes they teach
- Doctors should only see medical records for their patients
- Apps should only have access to their program’s data

Fundamentals of Access Control: Policies

Guiding philosophy: Utter simplicity.

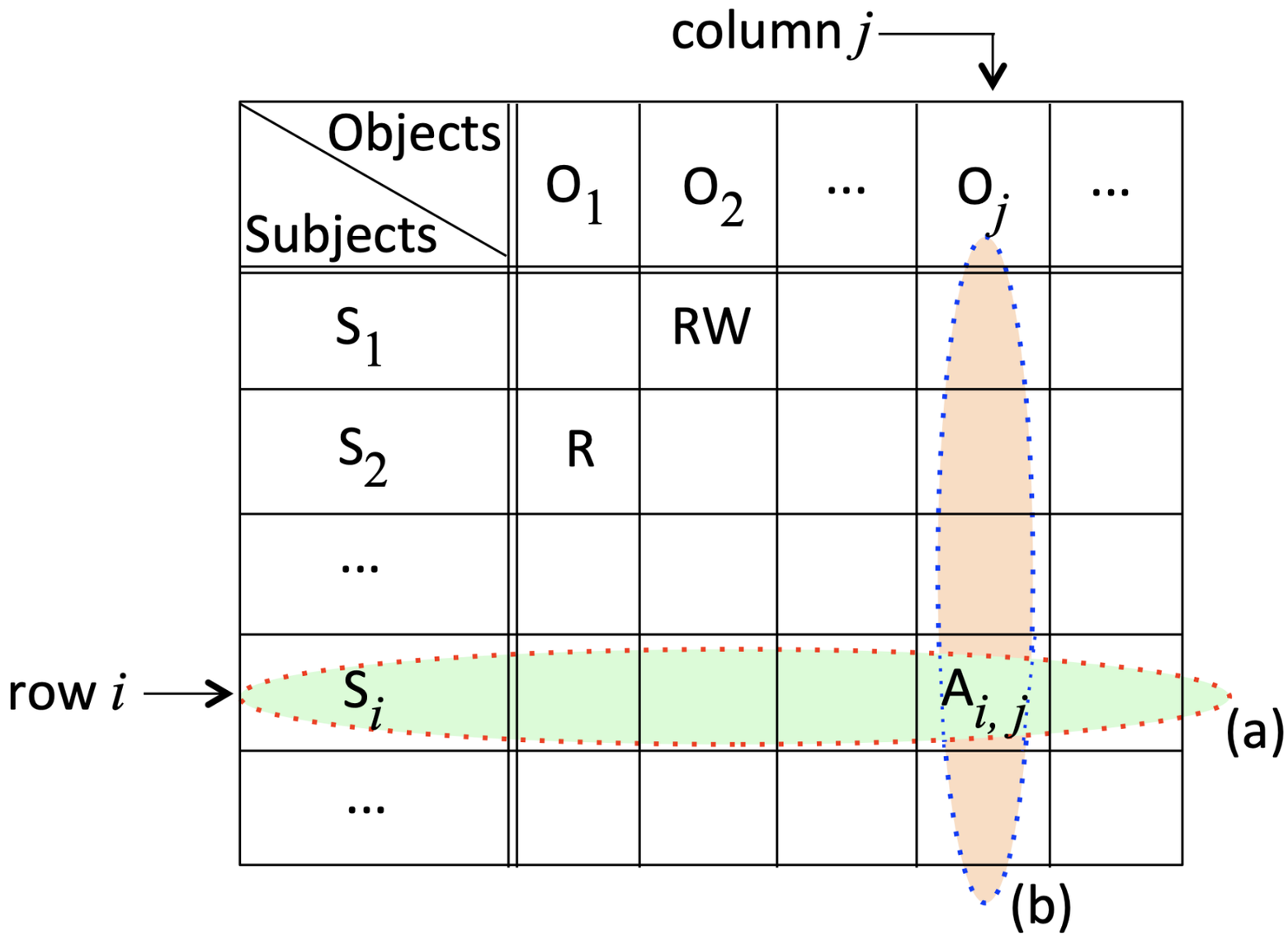
Step 1: Give a crisp definition of a **policy** to be enforced.

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.

Example

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

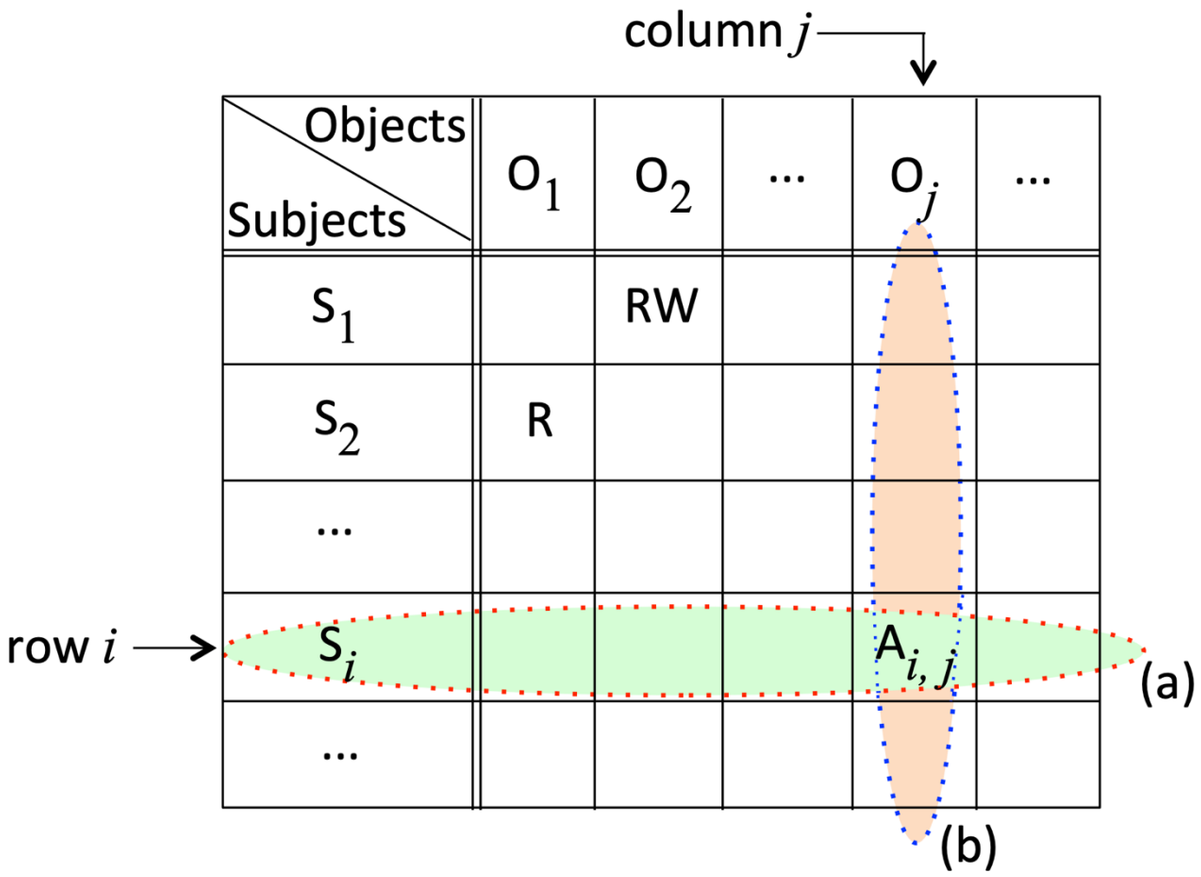
The Access Control Matrix



- 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?”

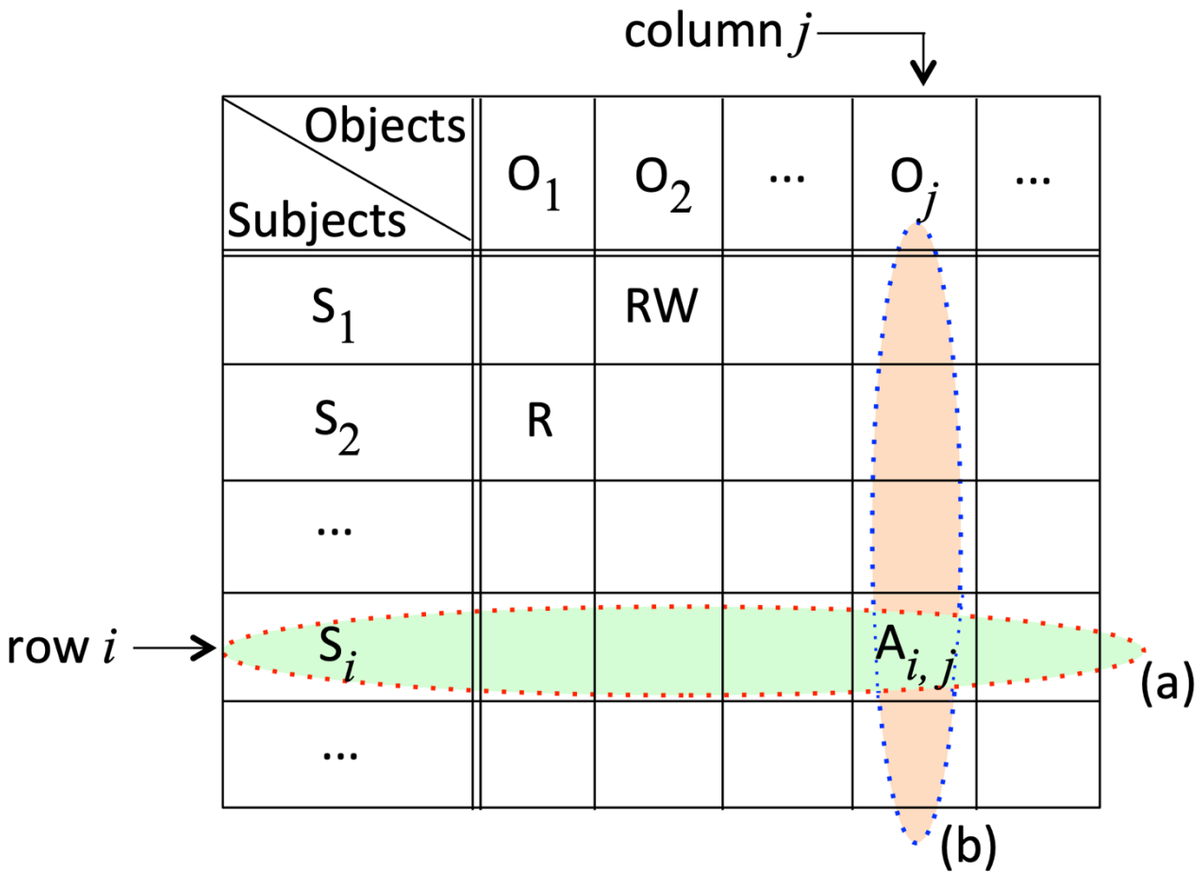


Examples:

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

Implementing Access Policies: Capabilities

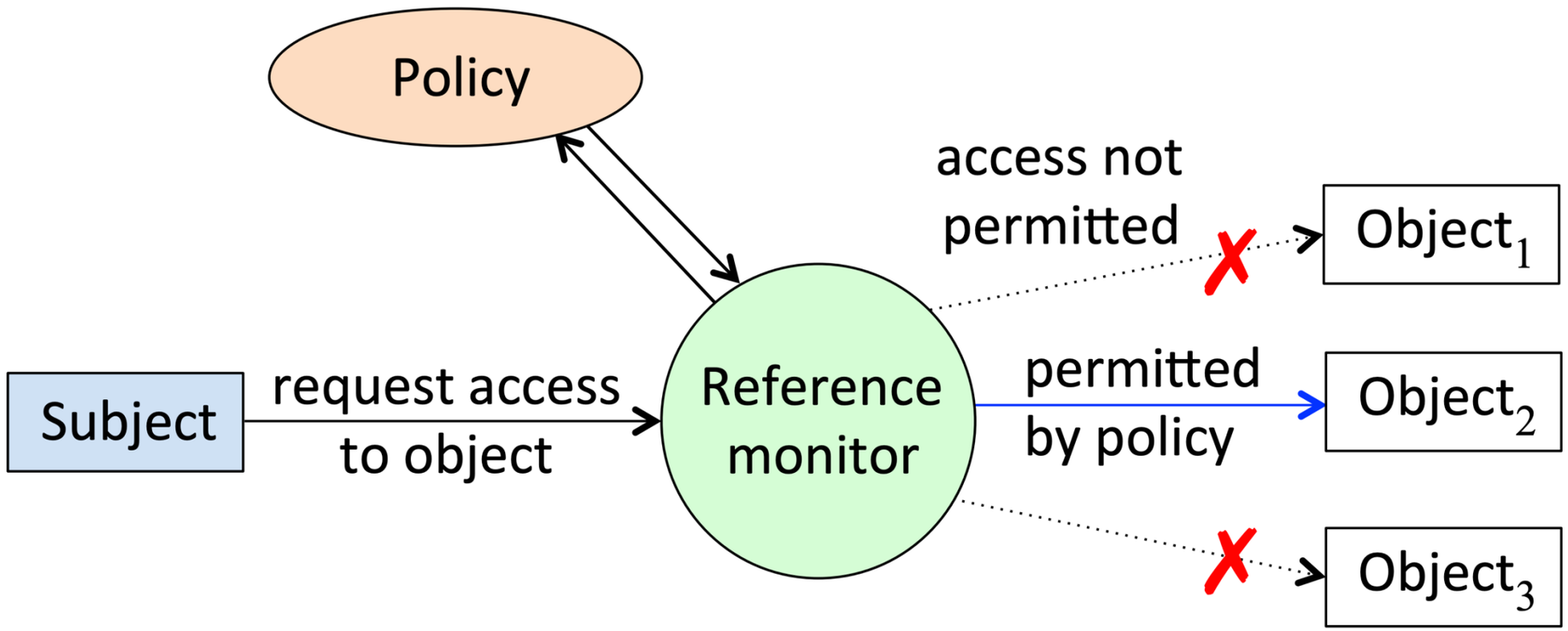
- "Capability" (of a subject) is a row of matrix
- Usually stored with subject
- Can quickly answer question: "What can this subject access?"



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; Simpl
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

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

- Initially an OS developed in the 1970s by AT&T Bell Labs.
  - Philosophy of small programs with simple communication mechanisms
- Linux and MacOS based on Unix.

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.

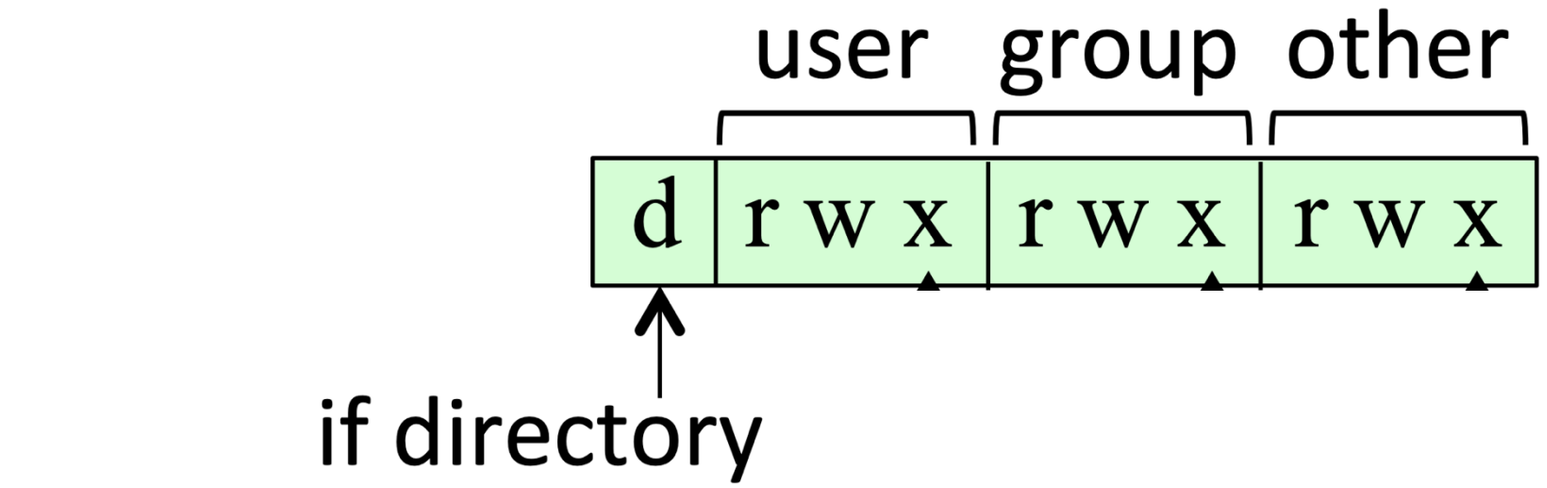
All files are owned by one user and one group.

```
grantho@focal7:/stage/classes/archive/2026/spring/23200-1$ ls -la
total 5417
drwxrwxr-x  2 davidcash faculty      18 Mar 24 13:11 .
drwxr-xr-x 62 root      root         62 Mar 23 08:38 ..
-rwxrwxr-x  1 grantho   faculty 3162706 Mar 24 17:52 01.pdf
-rwxrwxr-x  1 grantho   grantho     404 Mar 23 18:07 custom.css
-rwxrwxr-x  1 grantho   grantho  915789 Mar 23 18:07 david.jpg
```

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

File Permissions: UGO Model

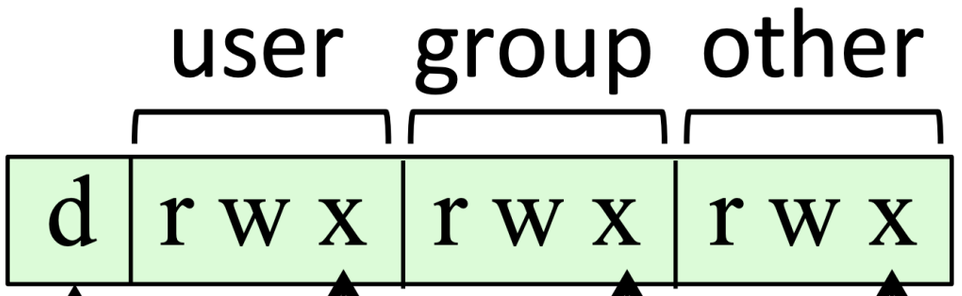
- Three bits for each of user, group, and other/all.
- Indicate read/write/execute permission respectively.




```
grantho@focal7:/stage/classes/archive/2026/spring/23200-1$ ls -la
total 5417
drwxrwxr-x  2 davidcash faculty      18 Mar 24 13:11 .
drwxr-xr-x 62 root      root         62 Mar 23 08:38 ..
-rwxrwxr-x  1 grantho   faculty 3162706 Mar 24 17:52 01.pdf
-rwxrwxr-x  1 grantho   grantho     404 Mar 23 18:07 custom.css
-rwxrwxr-x  1 grantho   grantho  915789 Mar 23 18:07 david.jpg
```

File Permissions: UGO Model

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




```
grantho@focal7:/stage/classes/archive/2026/spring/23200-1$ ls -la
total 5417
drwxrwxr-x  2 davidcash faculty       18 Mar 24 13:11 .
drwxr-xr-x 62 root      root          62 Mar 23 08:38 ..
-rwxrwxr-x  1 grantho   faculty 3162706 Mar 24 17:52 01.pdf
-rwxrwxr-x  1 grantho   grantho     404 Mar 23 18:07 custom.css
-rwxrwxr-x  1 grantho   grantho  915789 Mar 23 18:07 david.jpg
```

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.

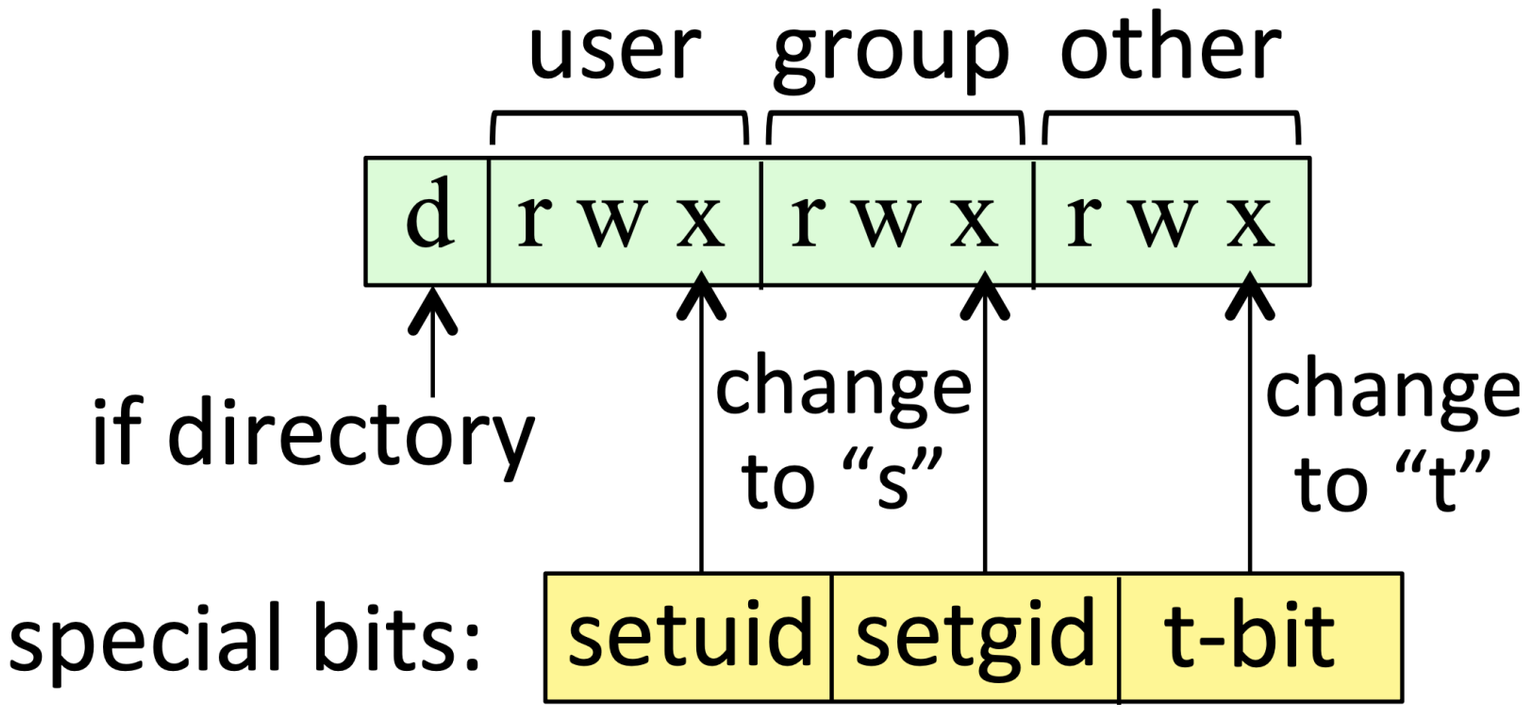
ACL or Capability?

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?)

File Permissions: UGO Model

- Three bits indicating read/write/exec for each of user, group, and other/all.
- For executable files, there are also 3 special bits that can be set.




```
grantho@focal5:~$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 68208 Feb  6  2024 /usr/bin/passwd
```

```
grantho@focal5:~$ ls -l /usr/bin/su*
-rwsr-xr-x 1 root root  67816 Apr  9  2024 /usr/bin/su
-rwsr-xr-x 1 root root 166056 Apr  4  2023 /usr/bin/sudo
```

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

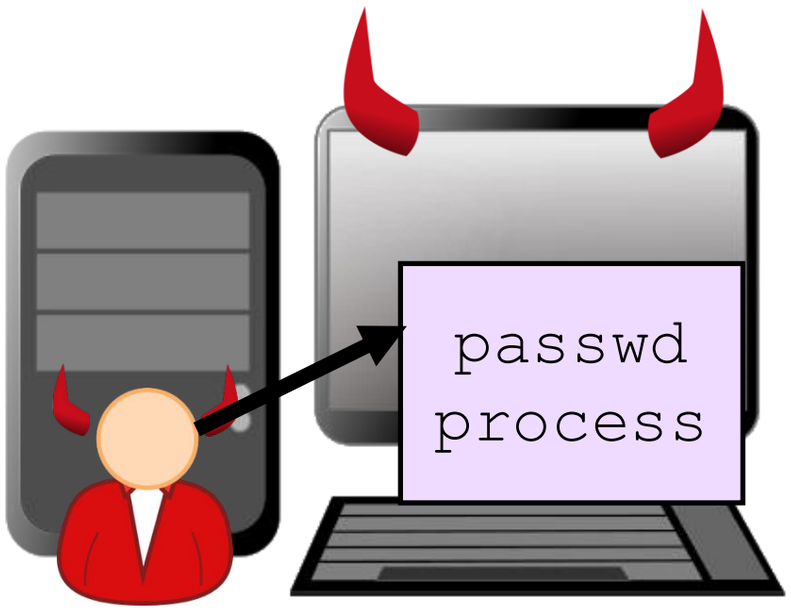
- The OS Kernel ensures that multiple programs can securely run together at the same time
    - The CPU has a dedicated CS register that tracks the privilege (CPL) of the currently running code
    - The OS Kernel & MMU use virtual addressing to help isolate the memory of different processes
- To control 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

Software Attacks: One Common Setting



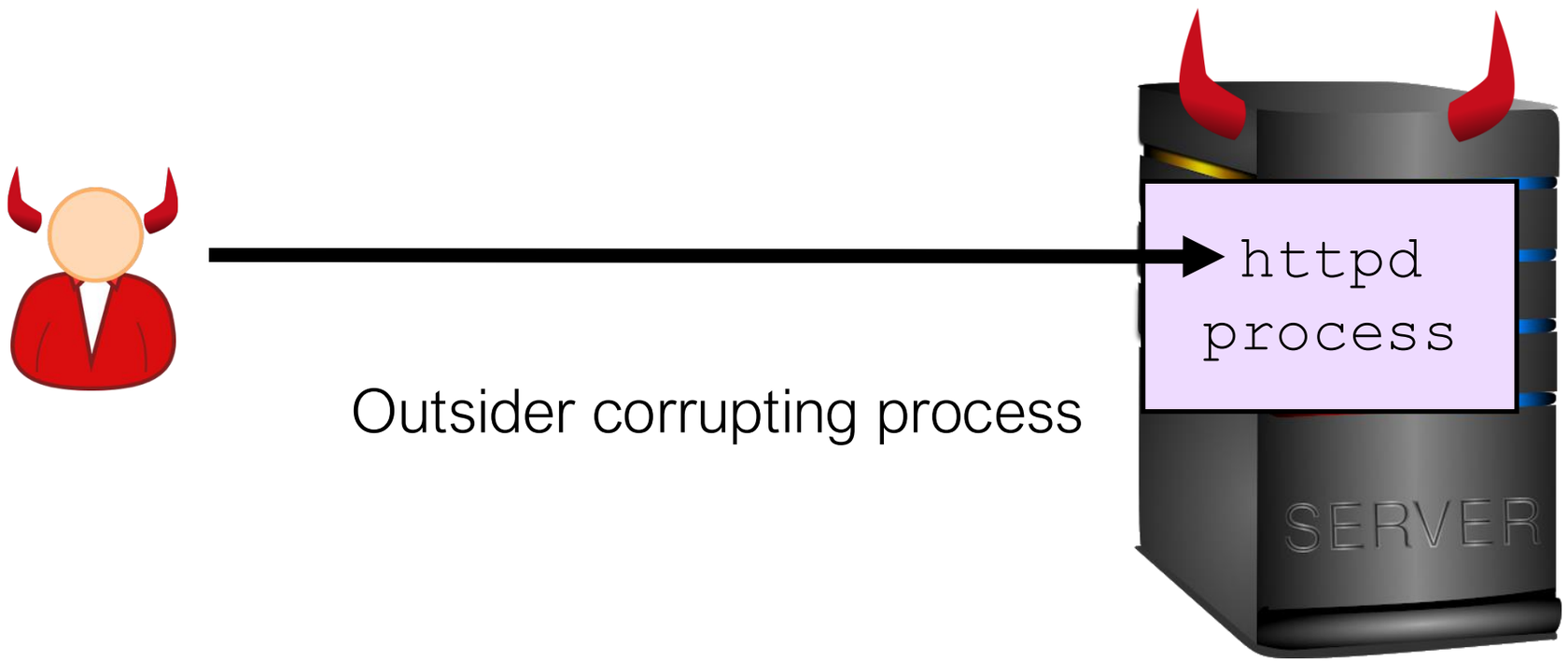
Insider escalating privilege

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

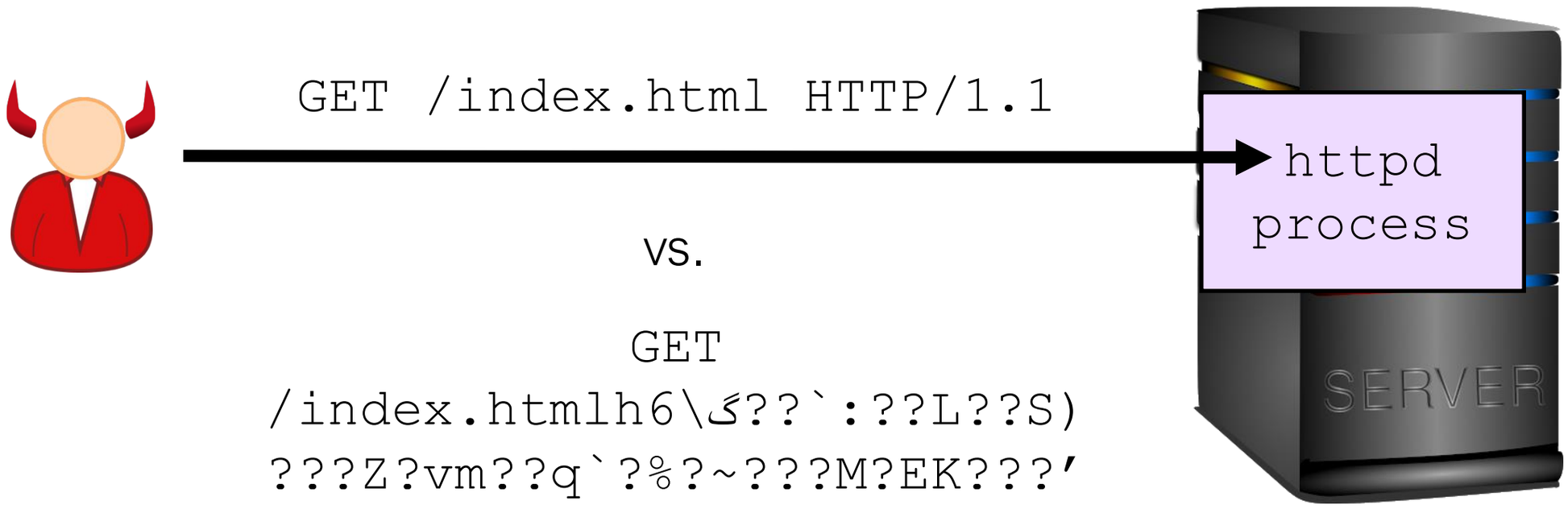
According to vulnerability researcher and author Dave Aitel:

- 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.*

Recent research suggests AI methods accelerate this process significantly.

Two Basic Principles of Most Attacks

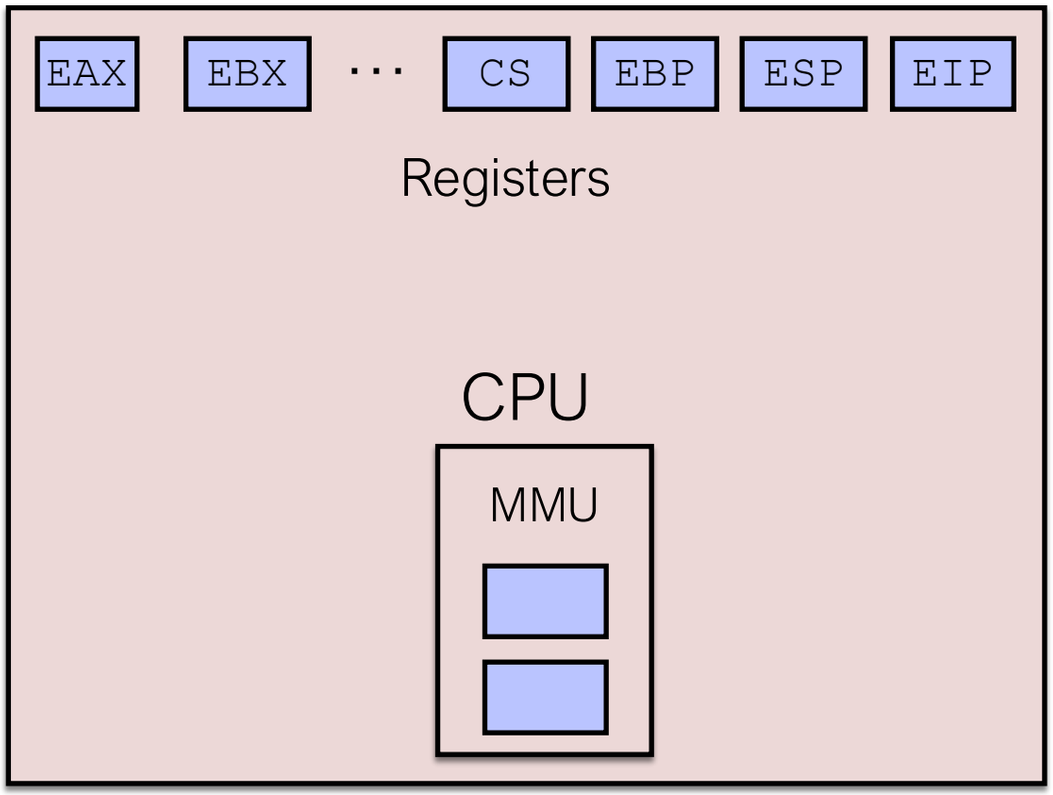
- Adversaries get to inject ***their*** bytes into ***your*** machine / program
- "Data" and "Code" are interchangeable; They are fundamentally the same "thing".

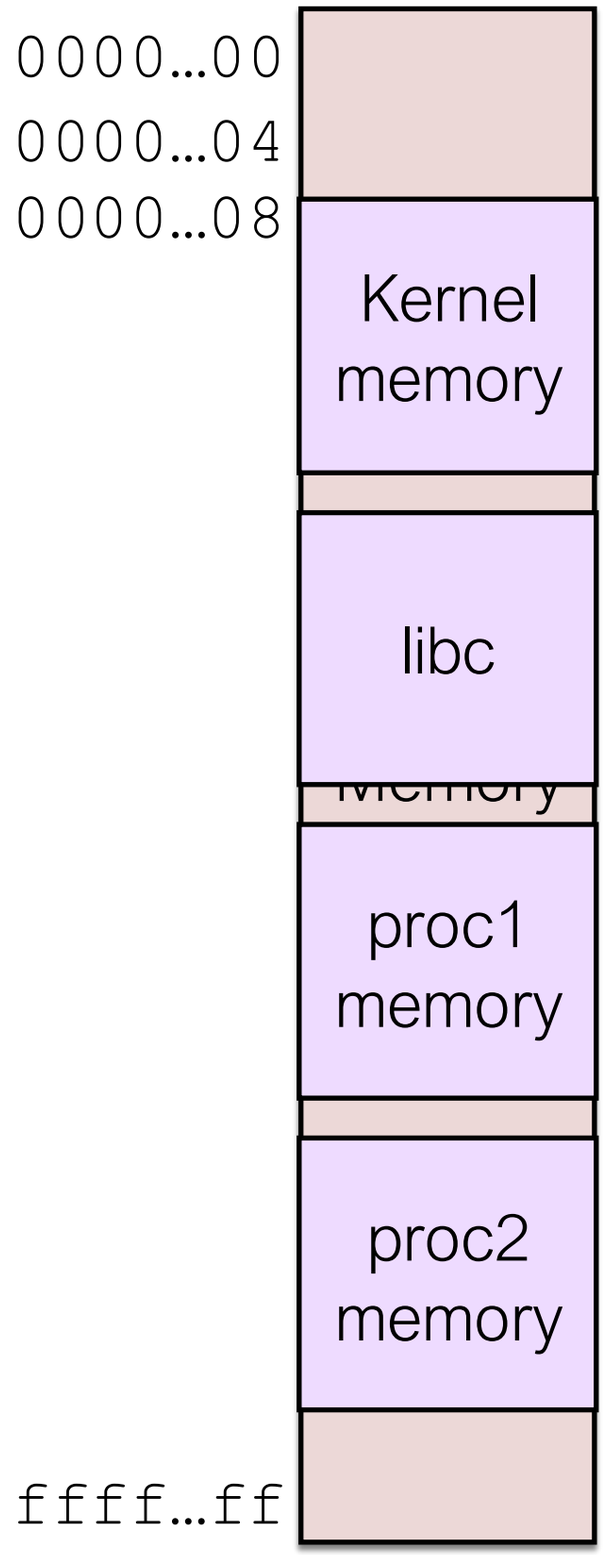


Outline for Lecture 2

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

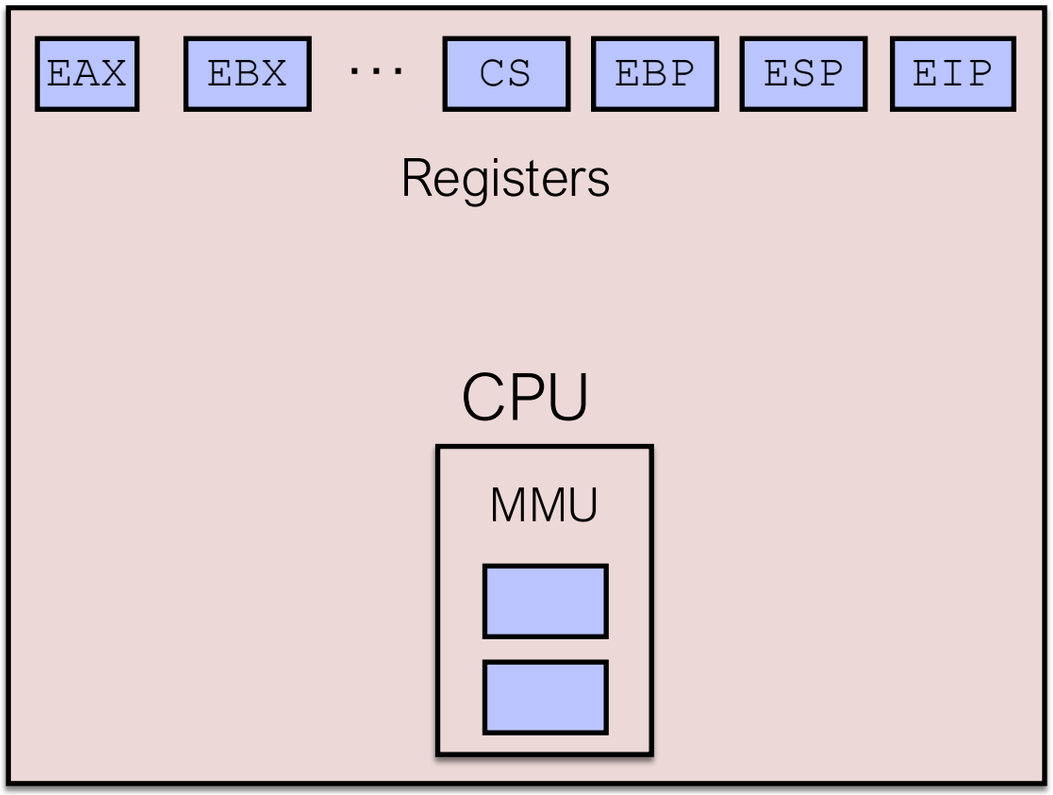
Program Execution: CPU & Memory

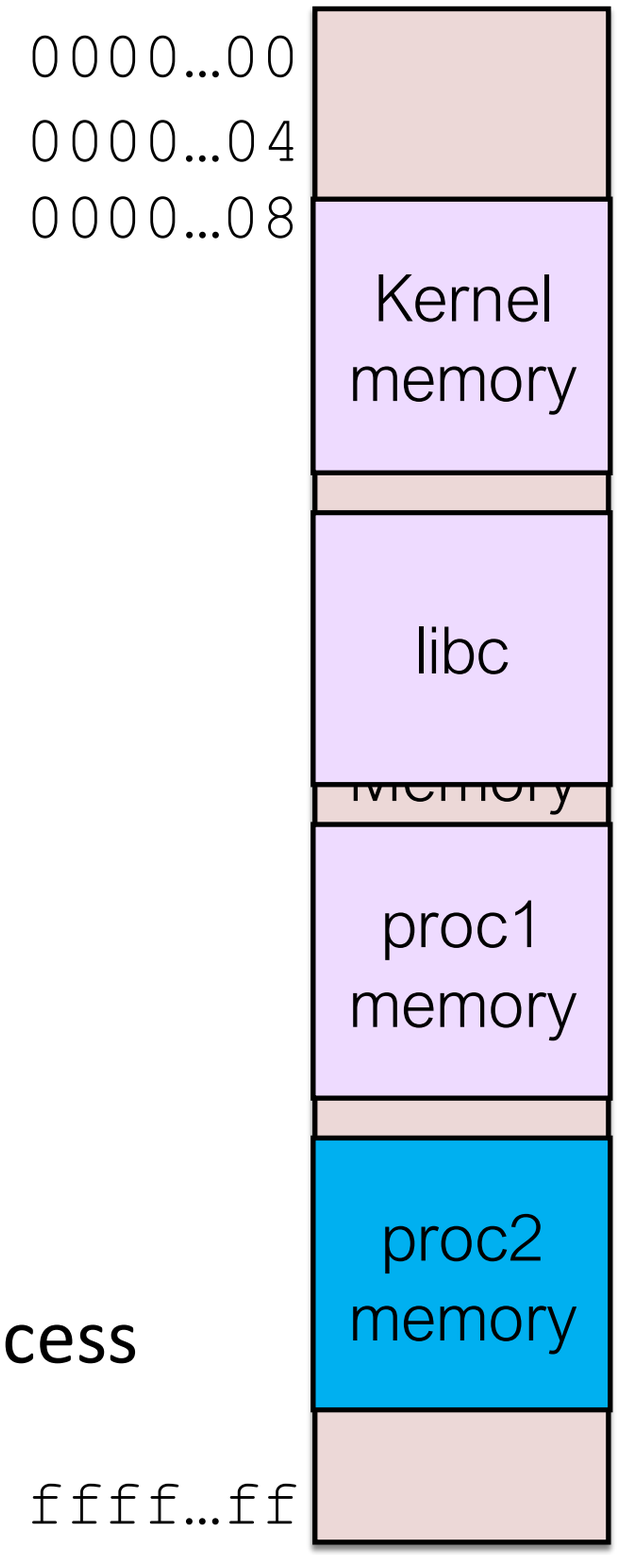




- **CPU**: executes the instructions (code) of processes
- **Memory**: stores code & runtime state of each process

Program Execution: CPU & Memory





- **CPU**: executes the instructions (code) of processes
- **Memory**: stores code & runtime state of each process
- **Virtual Memory** abstraction: the illusion that each process owns the entire address space

Memory Layout of a Process (in Linux)

.text: Machine executable code

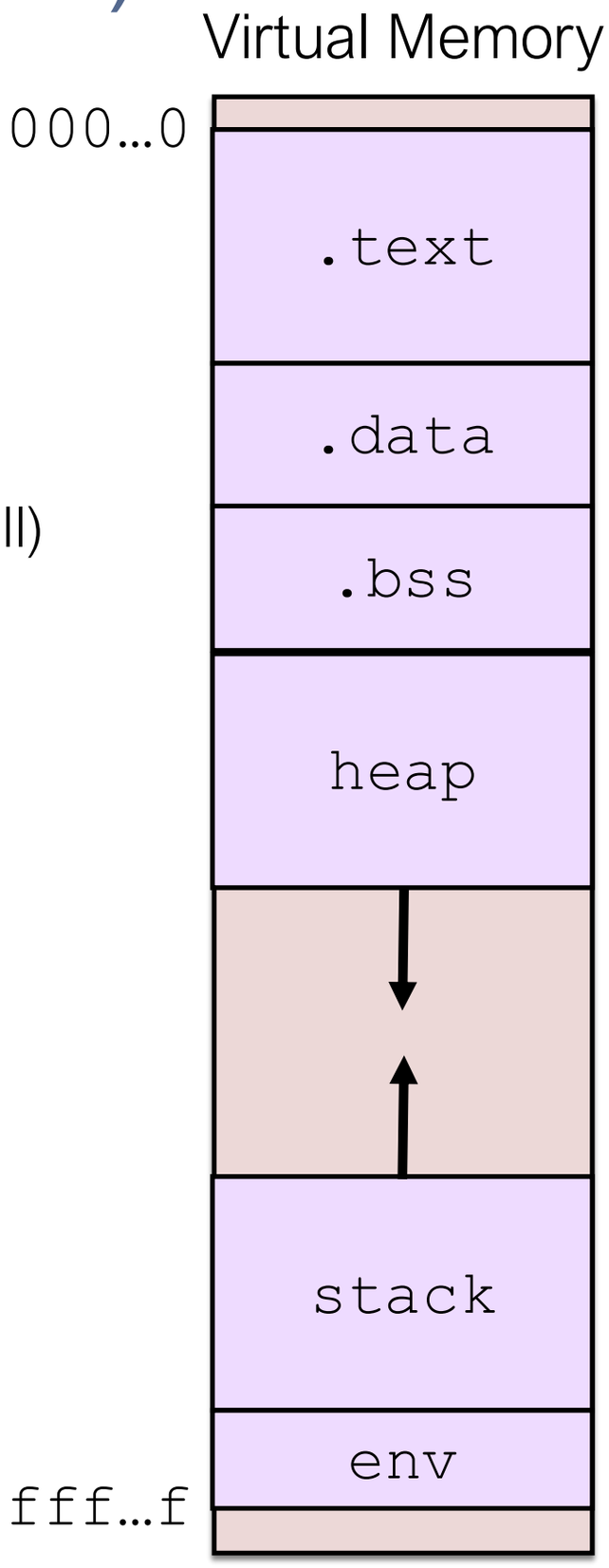
.data: Global initialized static variables

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

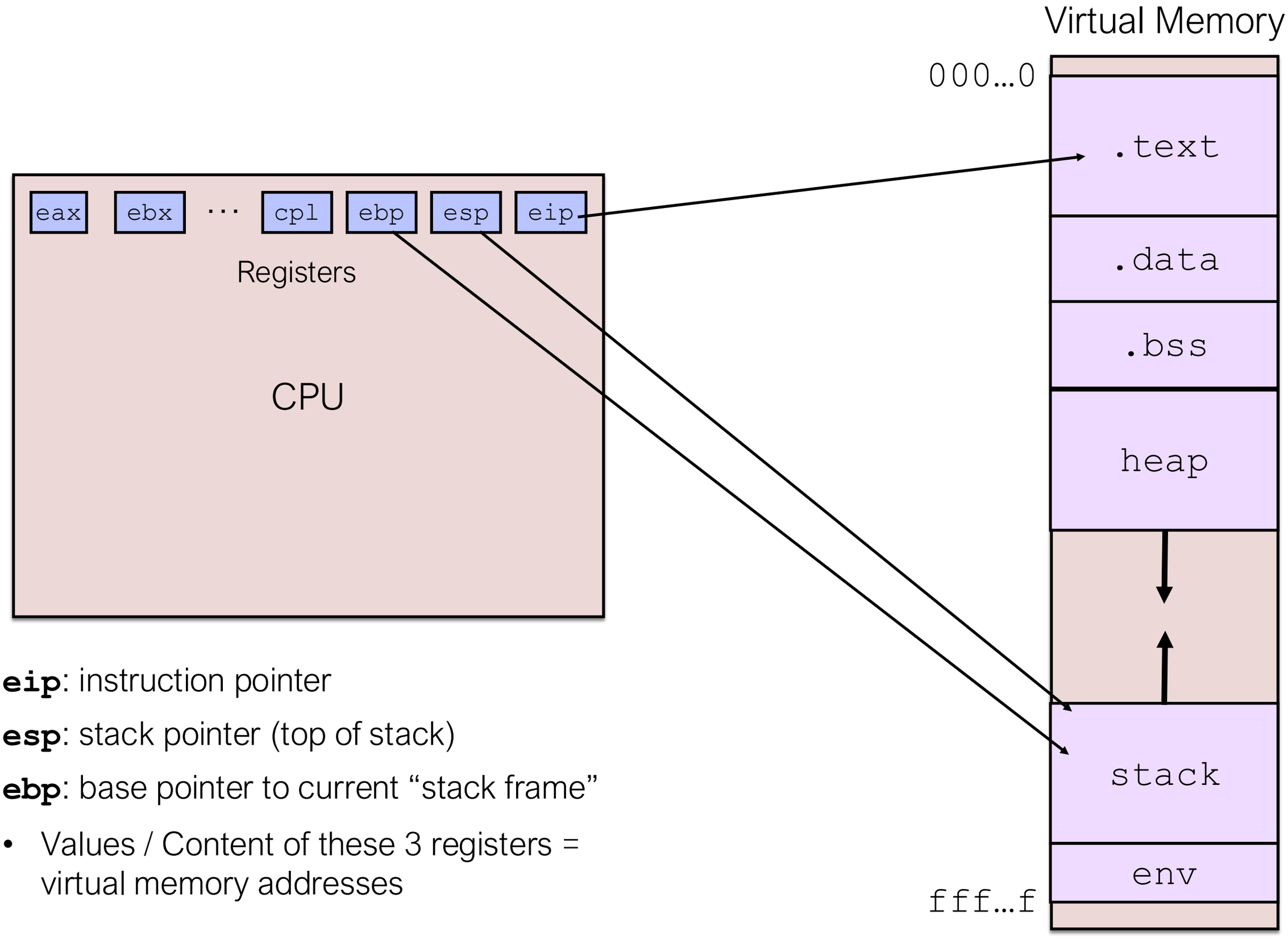
heap: Dynamically allocated memory (via `brk`/`sbrk`/`mmap` syscall)

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"

- Values / Content of these 3 registers = virtual memory addresses

Stack Frames

- When a function is called, the CPU allocates space on the stack to:
  - Track this function's local state: data like arg's and local var's
  - Store control flow info: who called this function & what code to execute after the function

- This space on the stack is known as the function's ***stack frame***

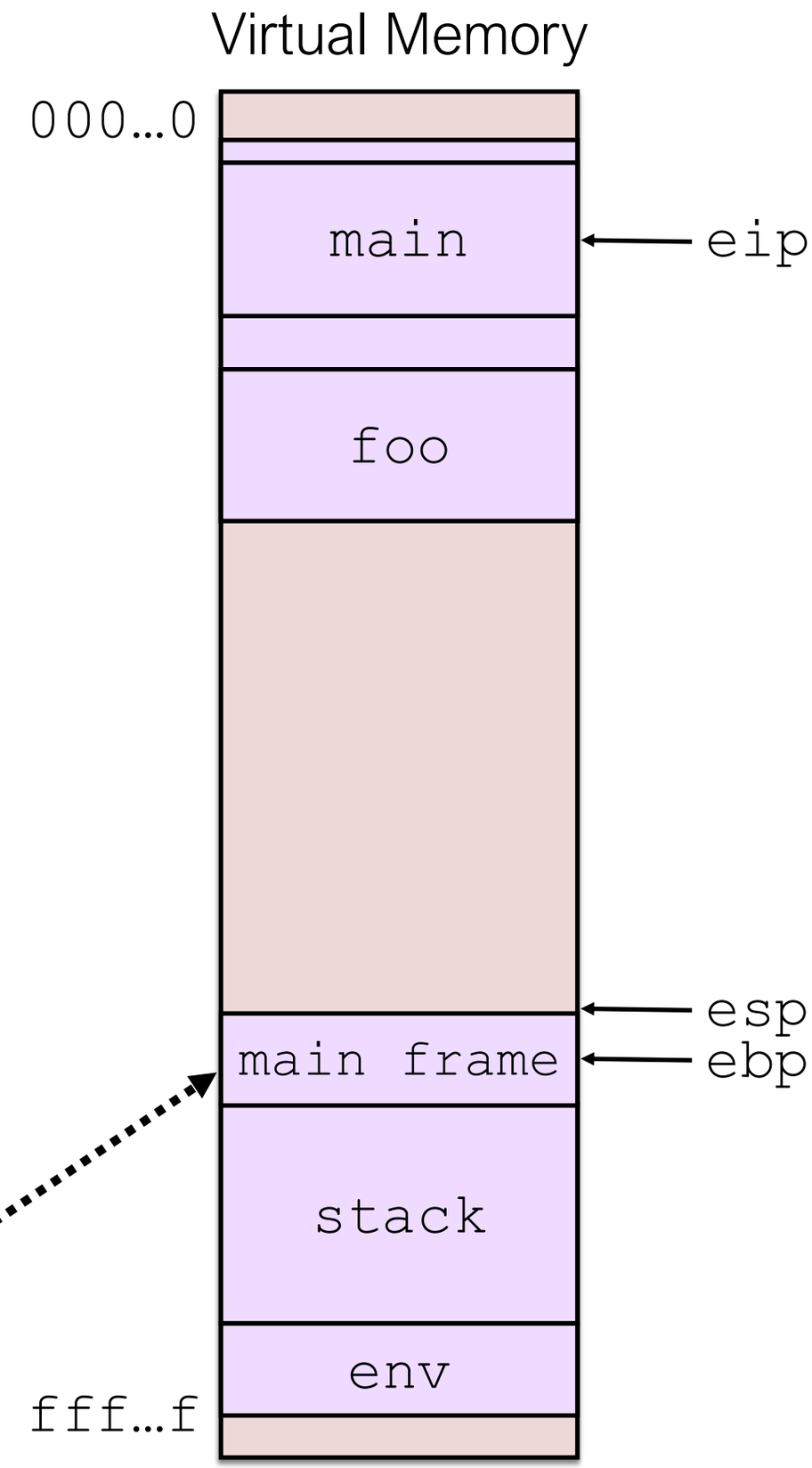
- The EBP and ESP registers track where the current function's stack frame lives in memory (starting & ending memory addr's)

- A function's stack frame is created & destroyed in part by both:
  - The calling function: ***Caller***
  - The function that was called: ***Callee***

The Stack and Calling a Function in C

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

int main(…) {
  …
  int x = foo(5, 6);
  …
}
```



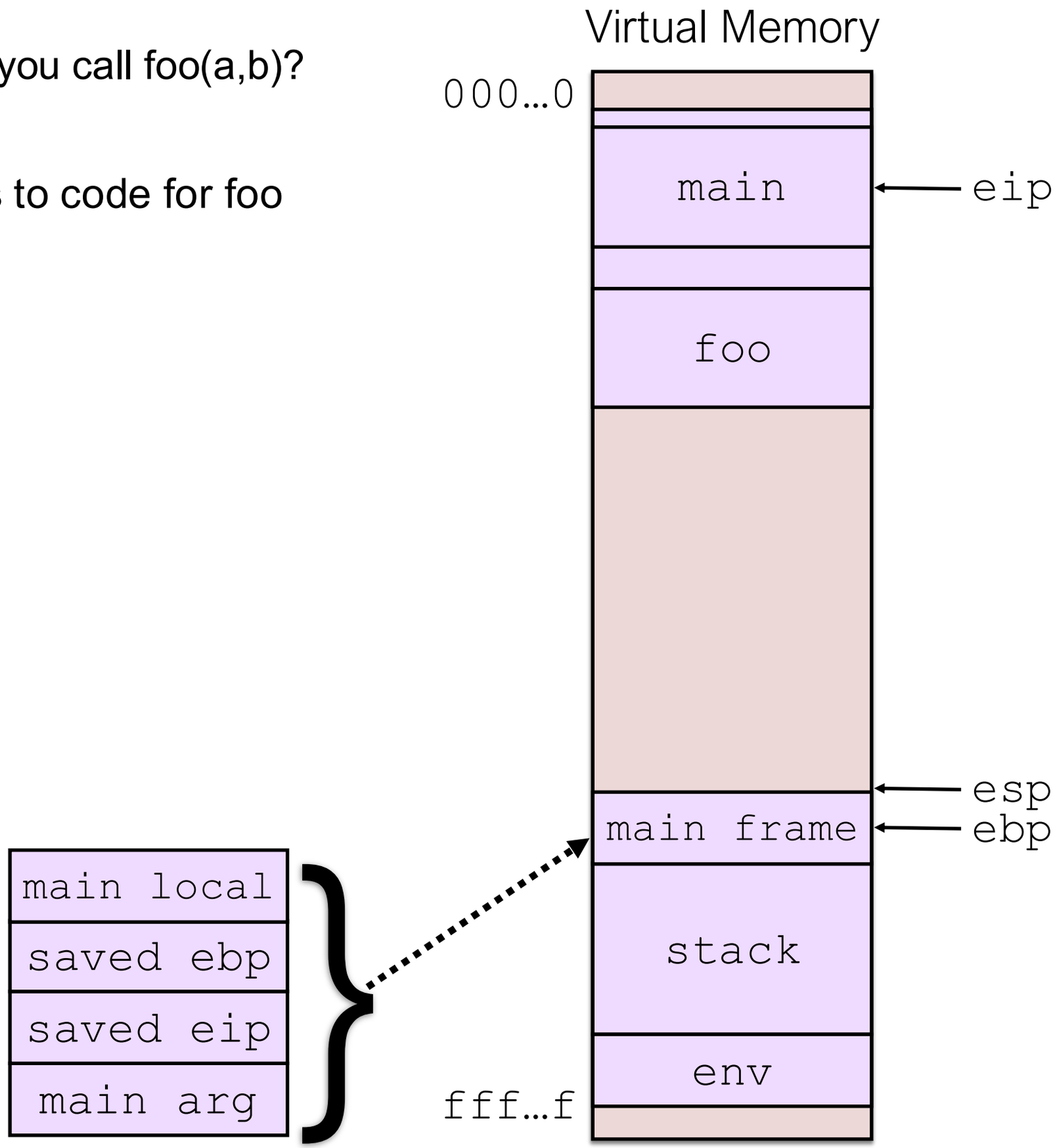
The Stack and Calling a Function in C

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

- A “stack frame” is added
- Instruction pointer eip moves to code for foo

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

int main(…) {
  …
  int x = foo(5, 6);
  …
}
```



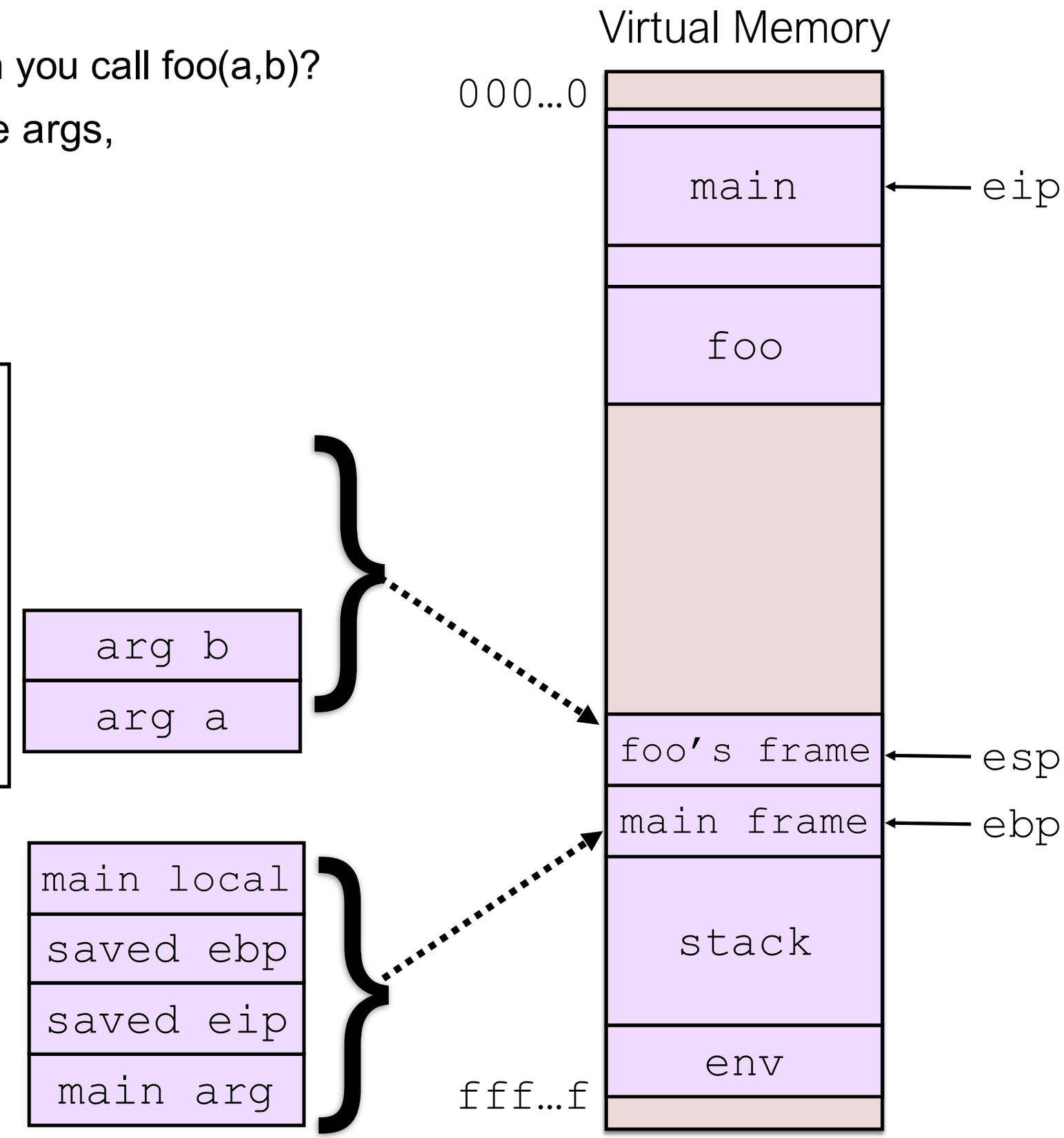
The Stack and Calling a Function in C

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

- Caller (main): (i) push callee args,

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

int main(…) {
  …
  int x = foo(5, 6);
  …
}
```



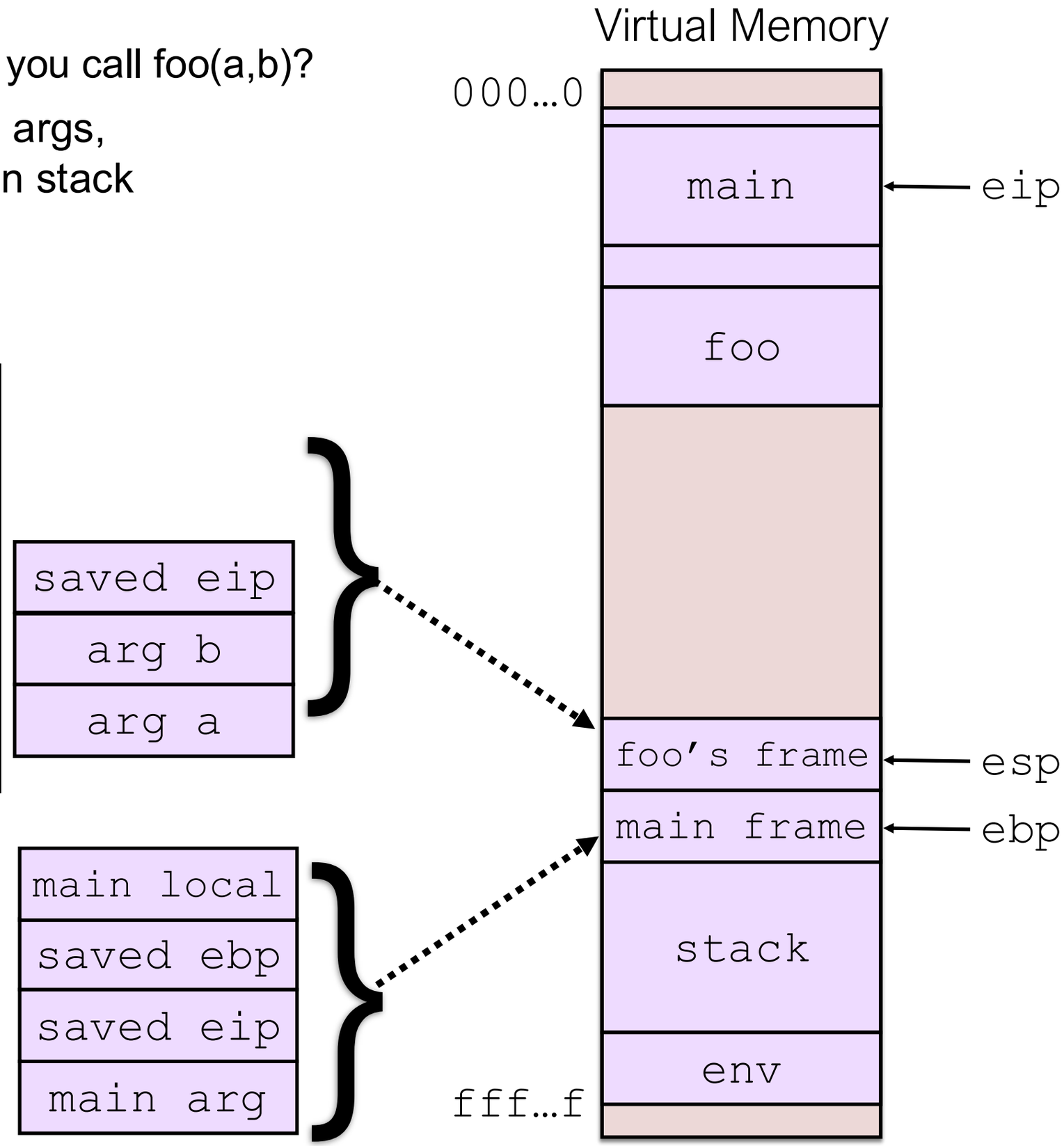
The Stack and Calling a Function in C

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

- Caller (main): (i) push callee args,
  (ii) save EIP register value on stack

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

int main(…) {
  …
  int x = foo(5, 6);
  …
}
```



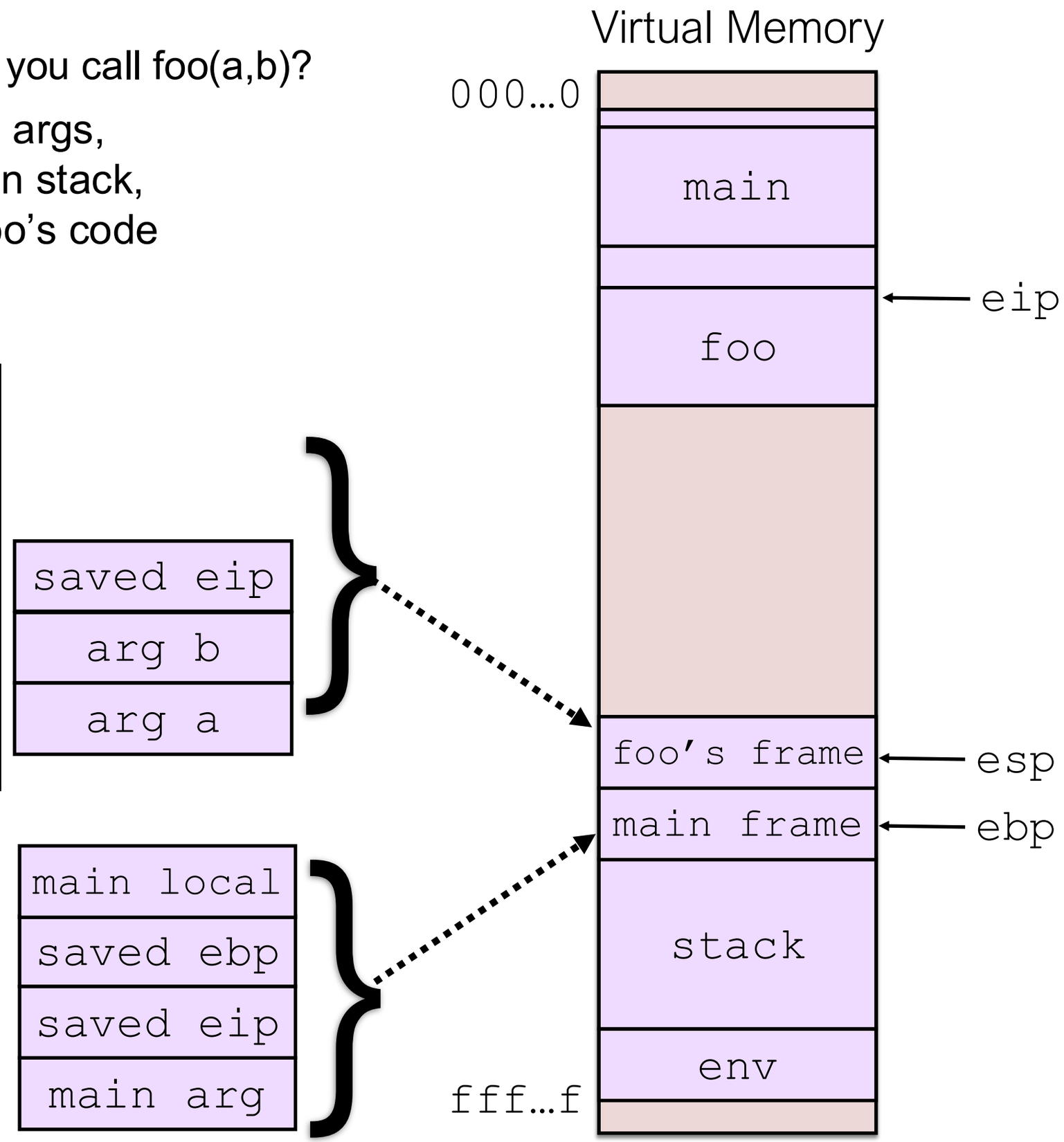
The Stack and Calling a Function in C

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

- Caller (main): (i) push callee args,
  (ii) save EIP register value on stack,
  (iii) move EIP register into foo's code

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

int main(…) {
  …
  int x = foo(5, 6);
  …
}
```



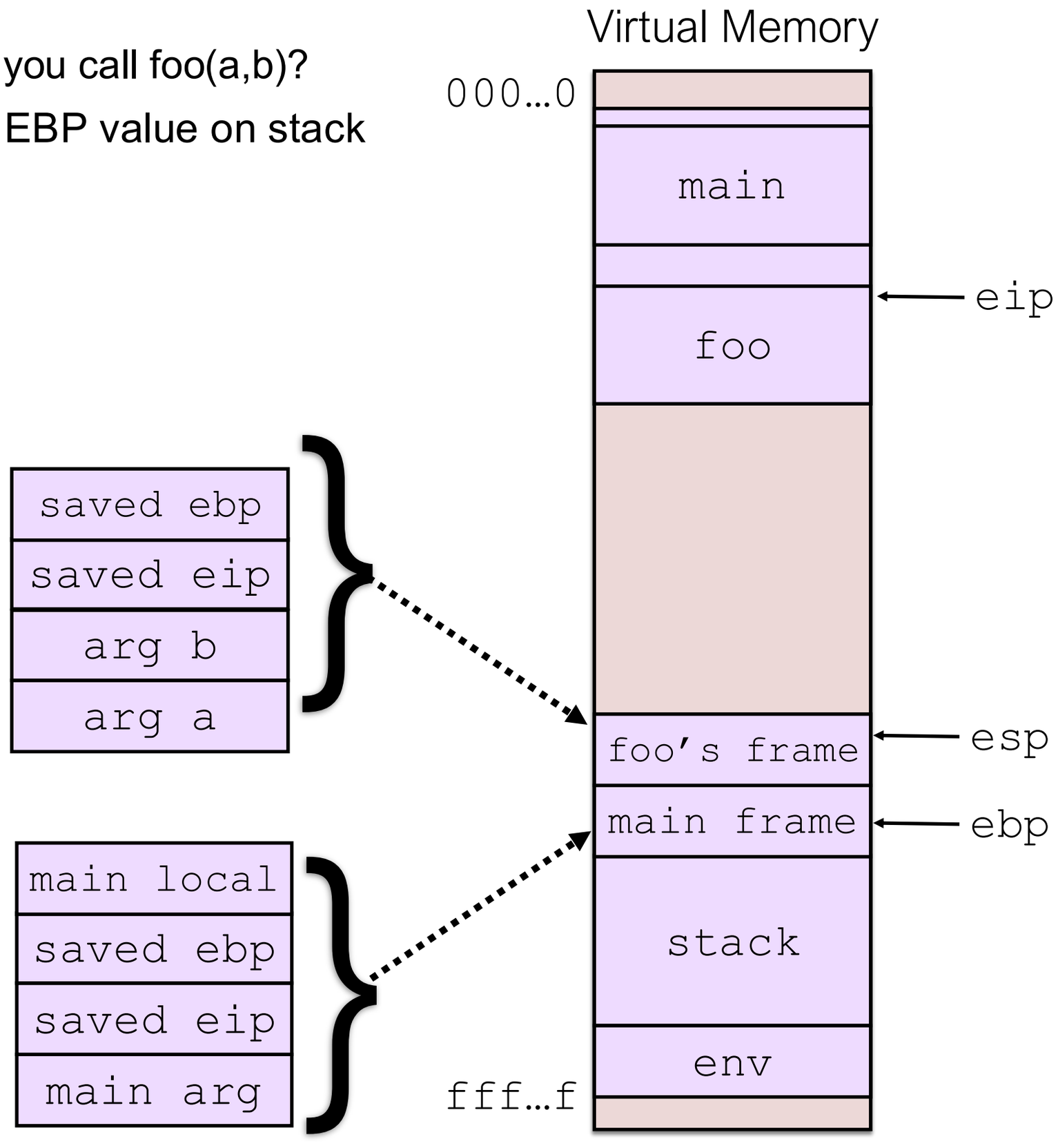
The Stack and Calling a Function in C

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

- Callee (foo): (i) save caller's EBP value on stack

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

int main(…) {
  …
  int x = foo(5, 6);
  …
}
```



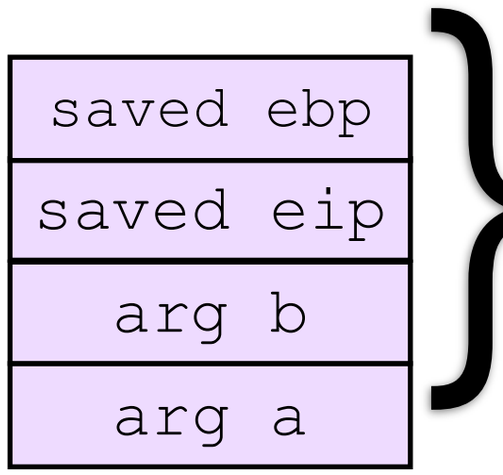
The Stack and Calling a Function in C

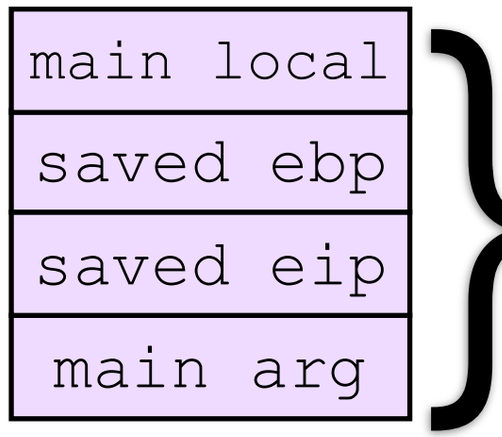
What happens to memory when you call foo(a,b)?

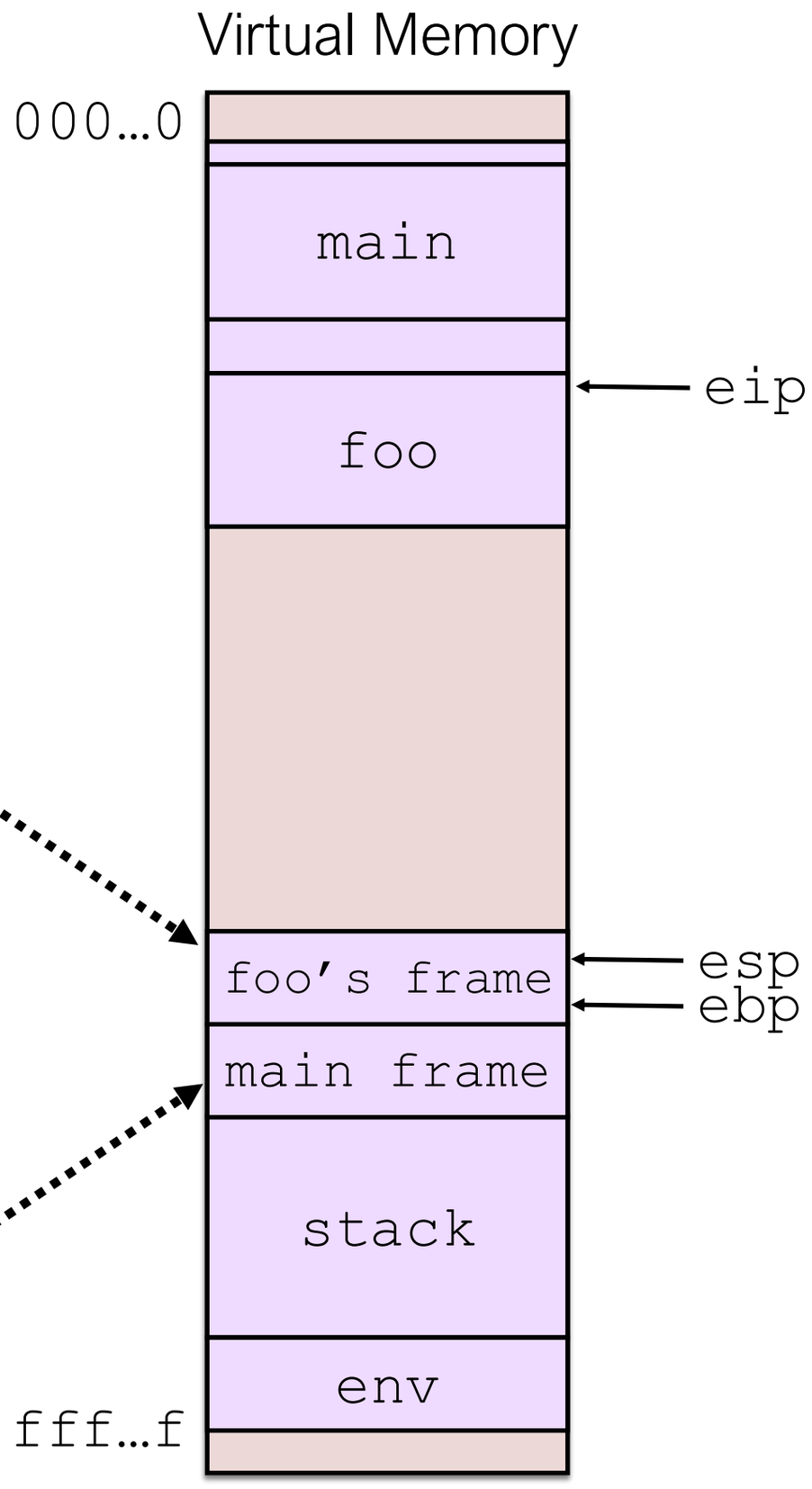
- Callee (foo): (i) save caller's EBP value on stack, (ii) move EBP register to point to its stack frame,

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

int main(…) {
  …
  int x = foo(5, 6);
  …
}
```







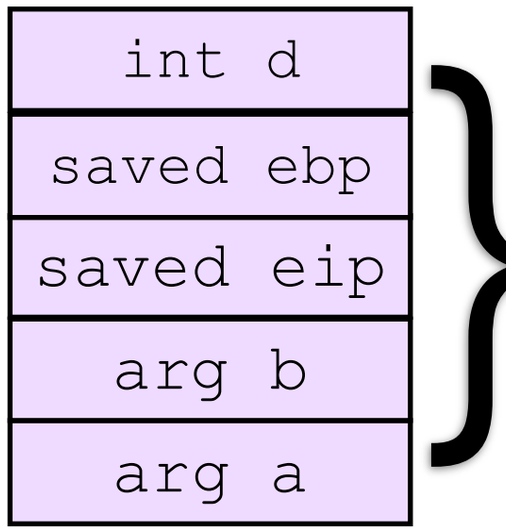
The Stack and Calling a Function in C

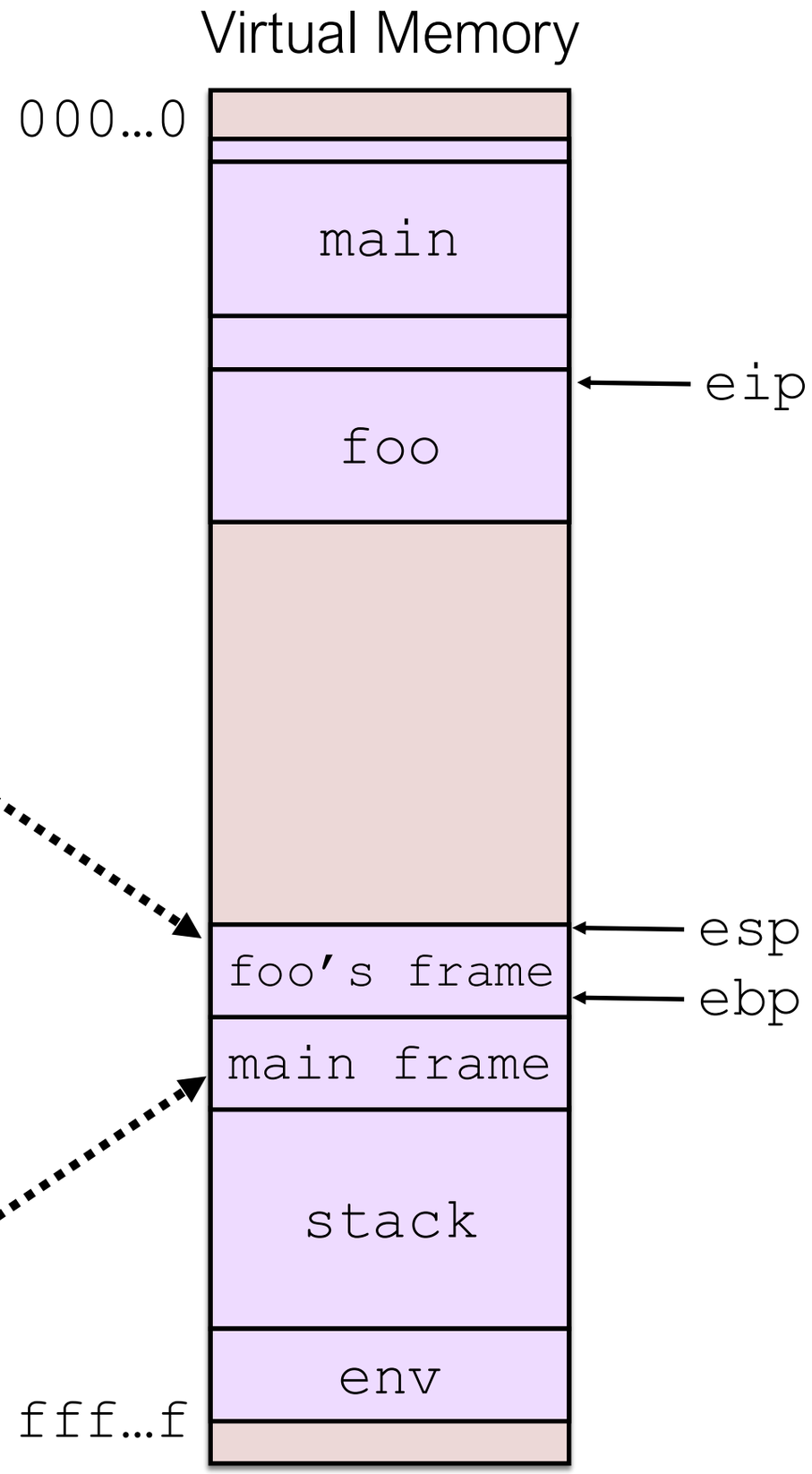
What happens to memory when you call foo(a,b)?

- Callee (foo): (i) save caller's EBP value on stack, (ii) move EBP register to point to its stack frame, (iii) allocate local var's

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

int main(…) {
  …
  int x = foo(5, 6);
  …
}
```





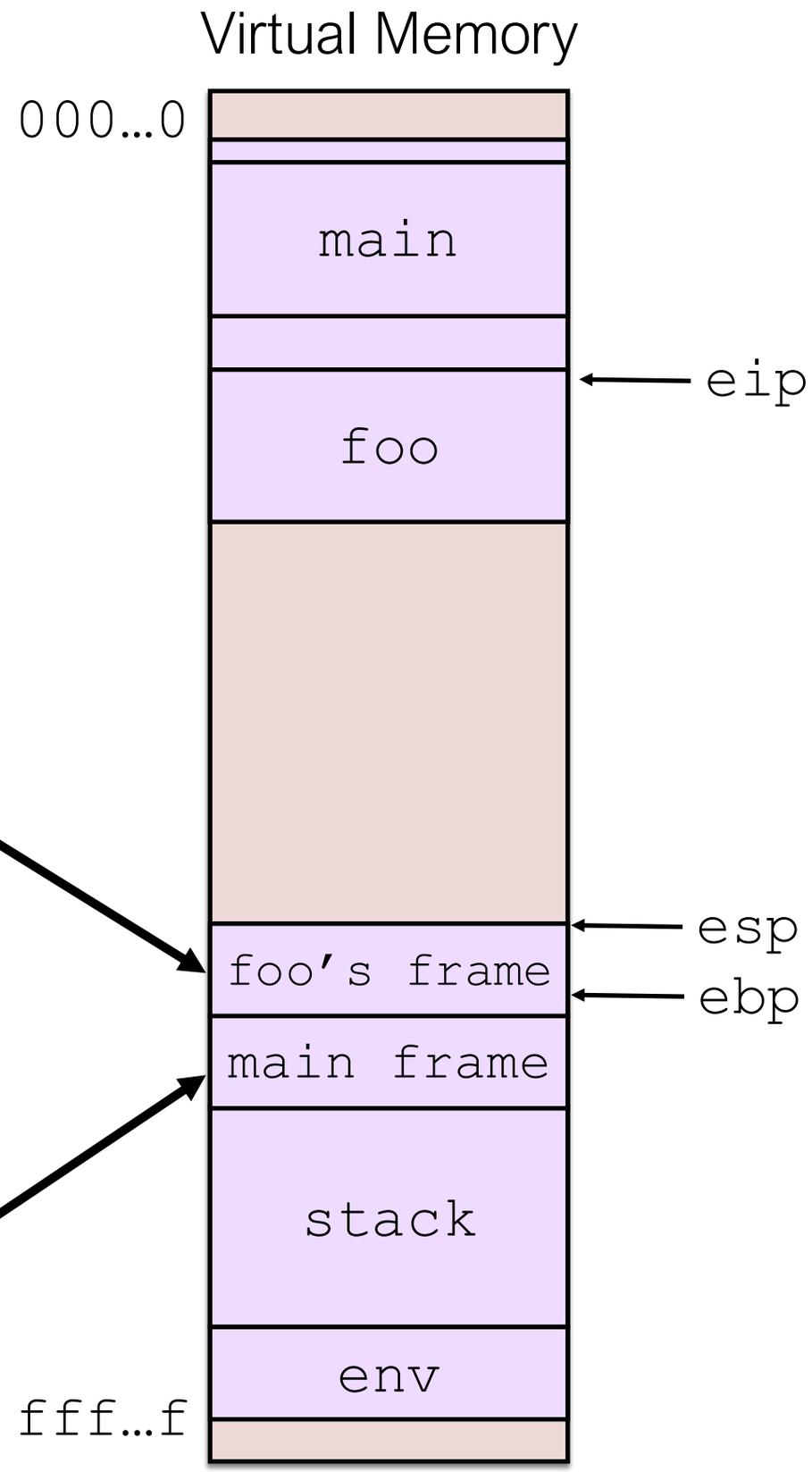
Returning from a function

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;
}

int main(…) {
  …
  int x = foo(5, 6);
  …
}
```



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

Key Point:

The CPU determines what code & data to execute next, based ***entirely*** on values ***stored on the stack***