OS Security: Access Control and the UNIX Security Model

CMSC 23200/33250, Winter 2022, Lecture 3

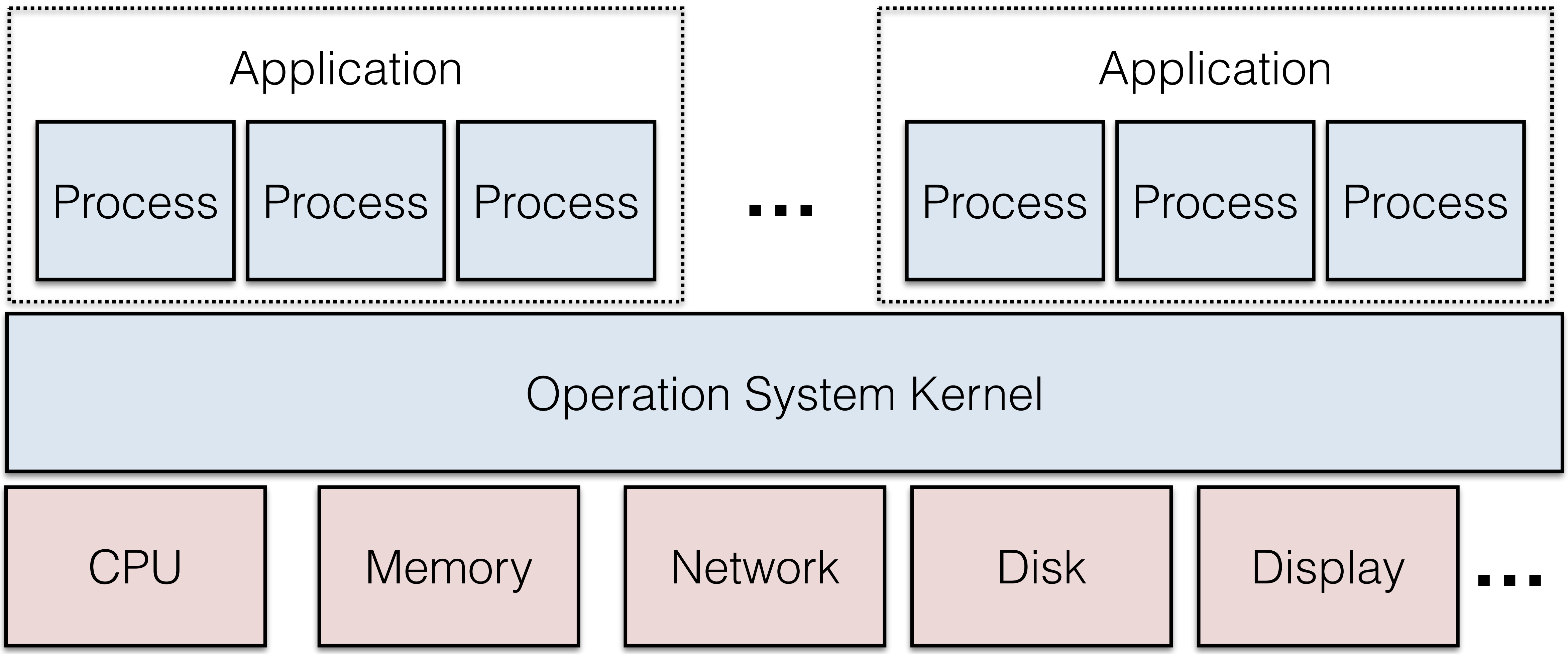
David Cash and Blase Ur

University of Chicago

Outline for Lecture 3

1. Wrap up “What is a process?”
2. Abstract approaches to access control (5.2)
3. UNIX notions of users, ownership, and permissions (5.1,5.3)
4. suid Permissions

Back to our diagram…

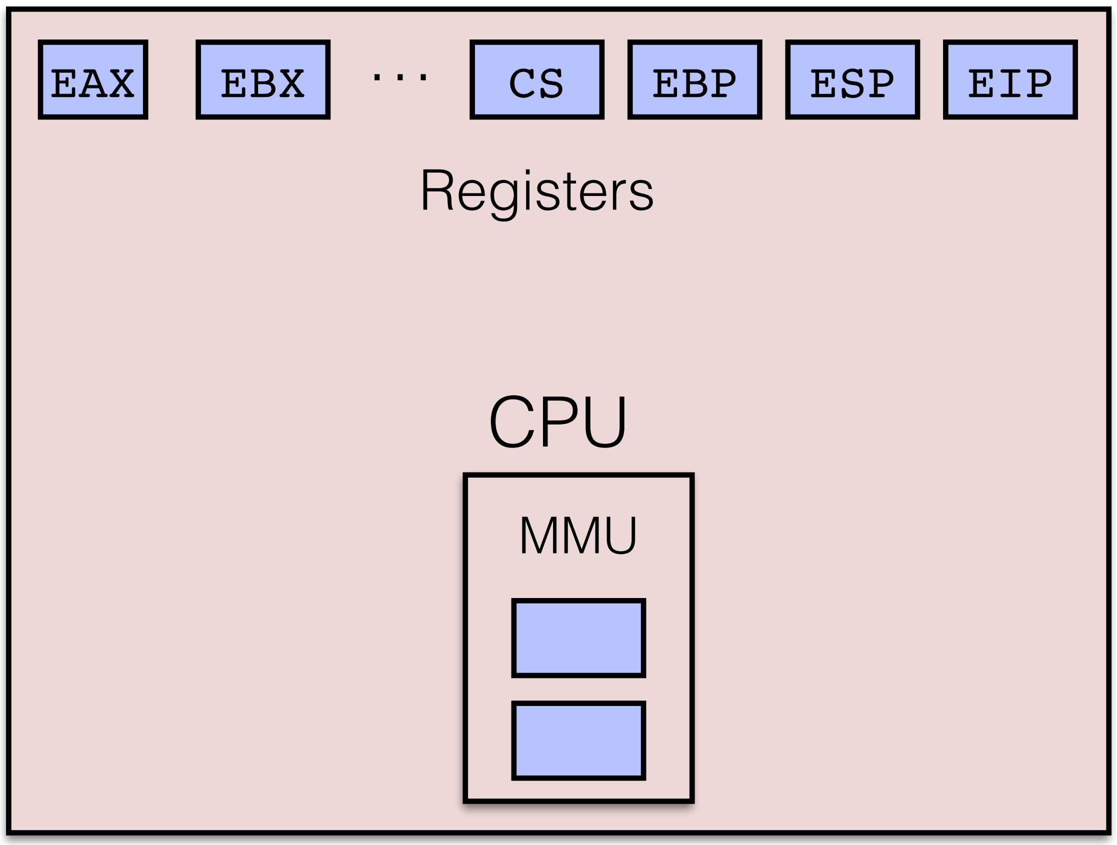


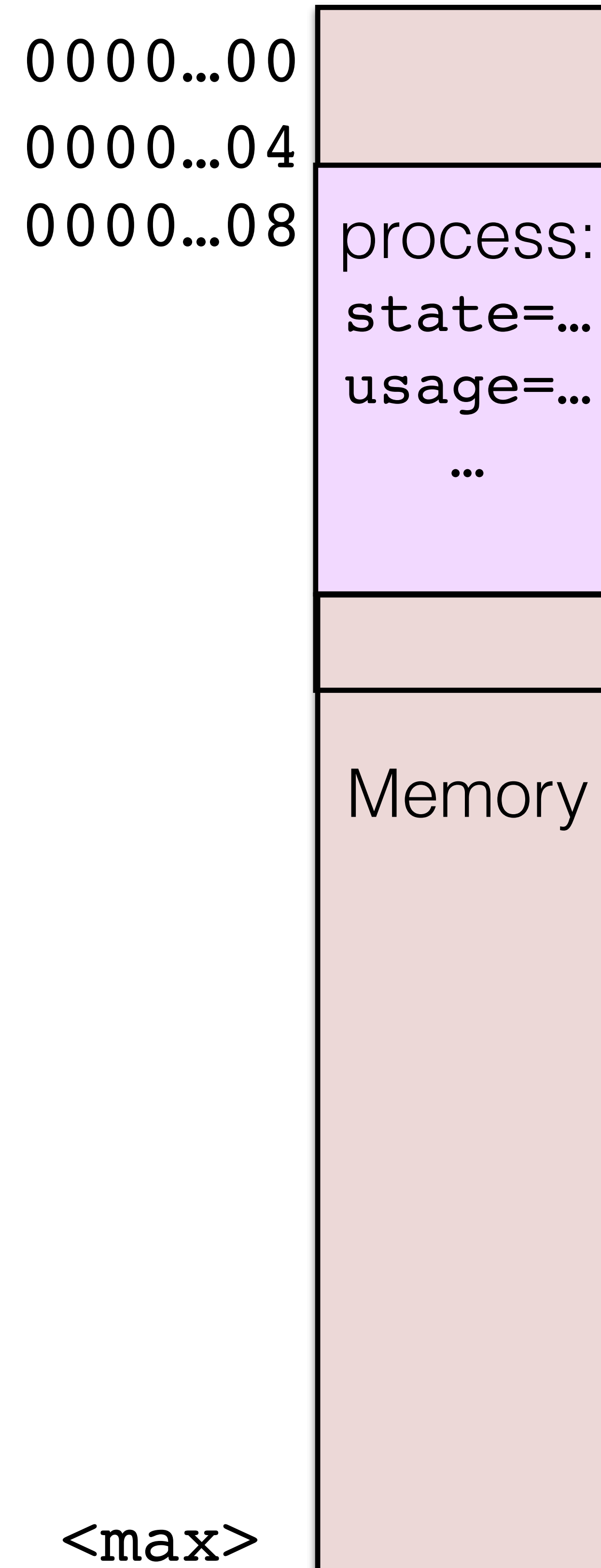
Questions, though:

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

The CPL!

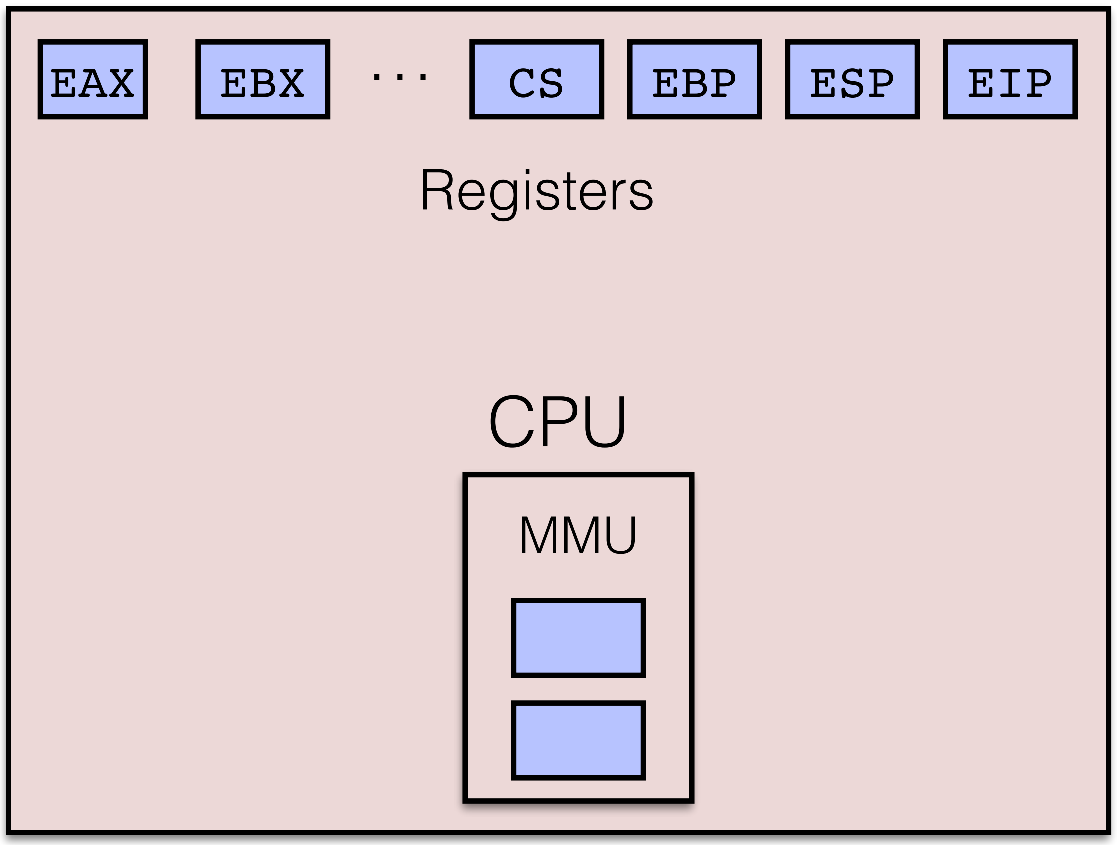
What *is* a process?

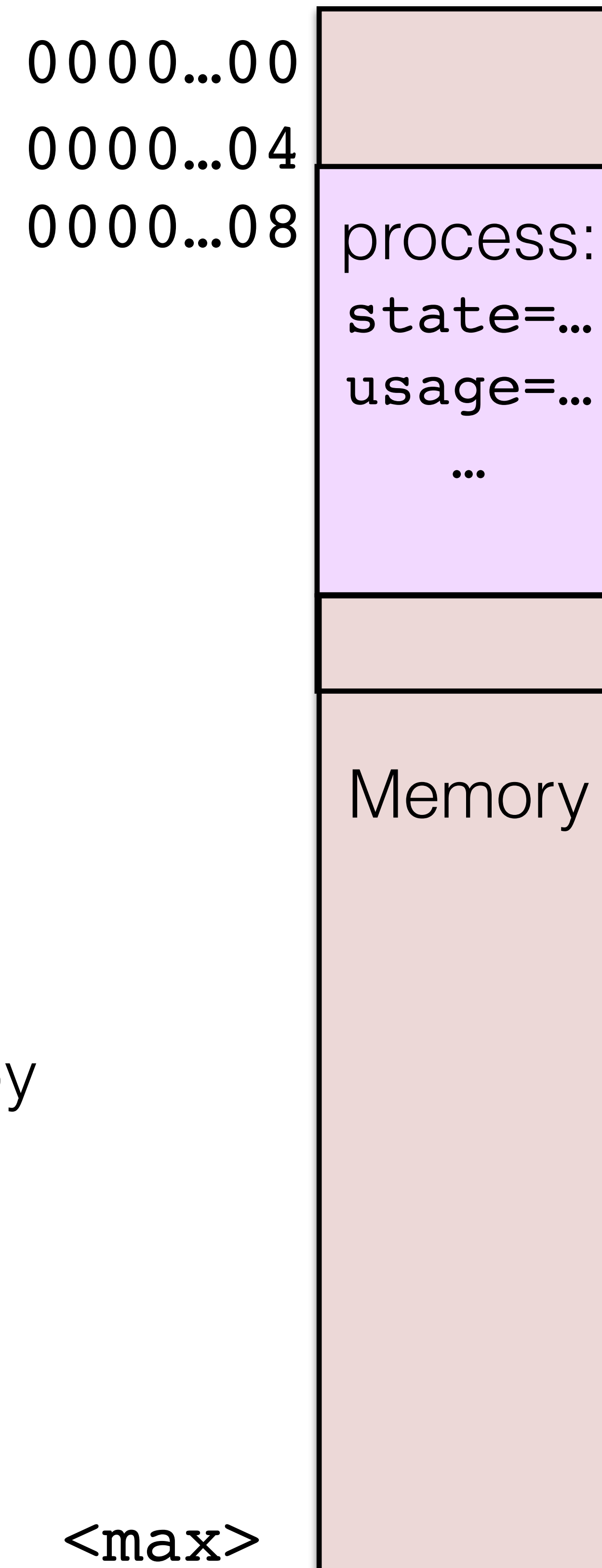




- <u>One Answer</u>: A data structure in “kernel memory”, including
  - MMU configuration
  - Register values
- Kernel can load these values up, set CPL=3, and turn over control “to the process” (i.e. set EIP)
- If kernel regains control, it can save these values to swap process out

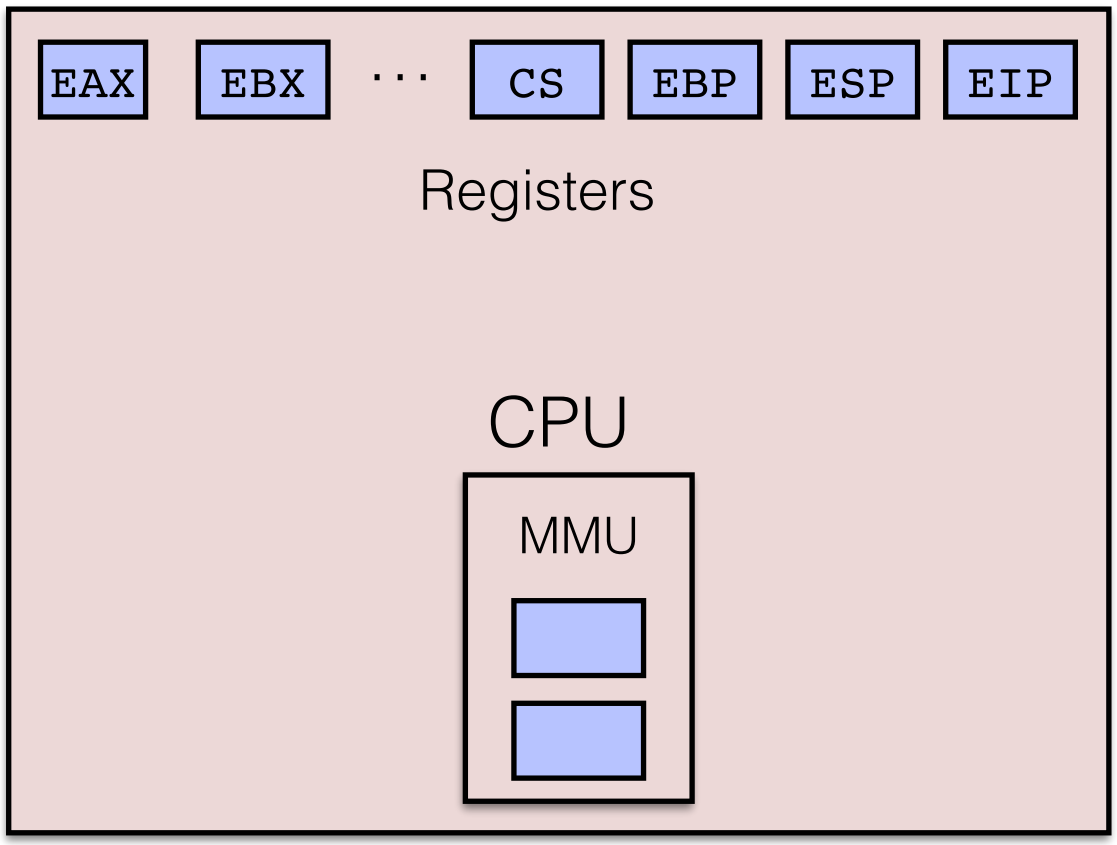
Handling Memory for a Process

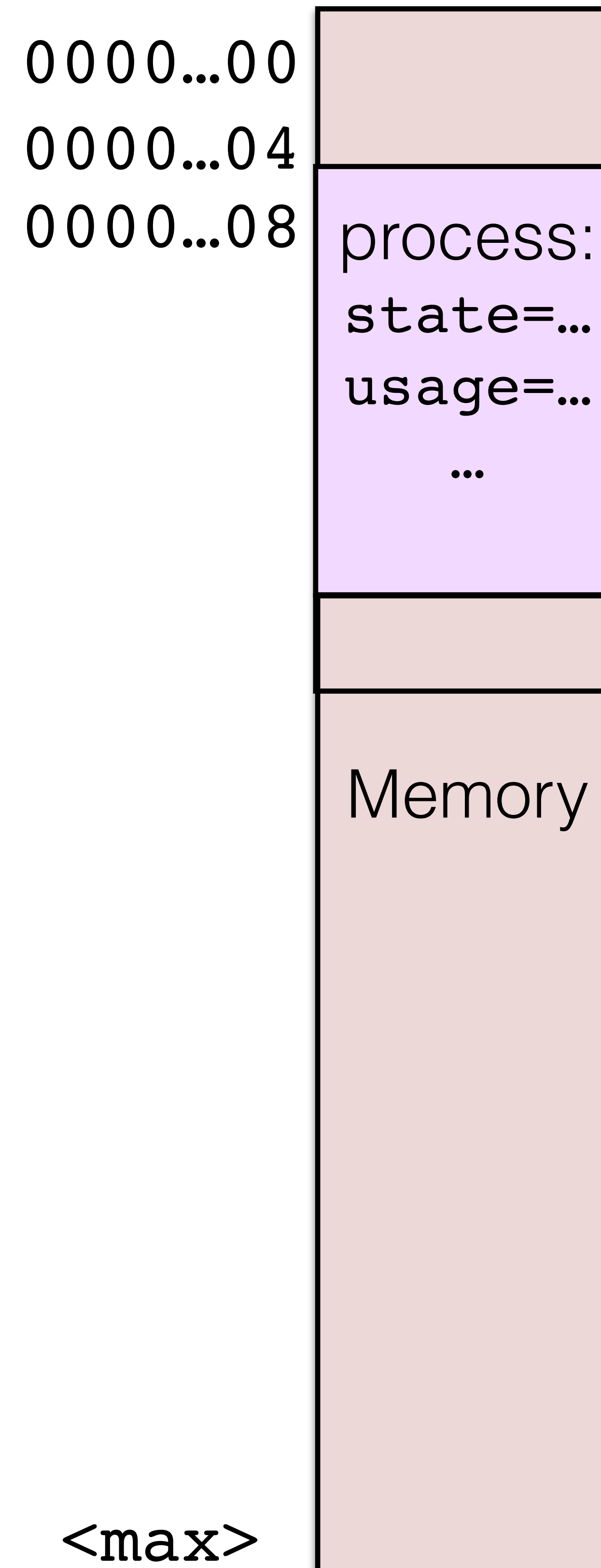




- Kernel creates a "virtual address space" for each process.
- Same virtual addresses (e.g. starting near 0) can be used by every process! They get translated to different physical addresses.
- Kernel can also mark some virtual address ranges (called segments) as "read only" or "do not execute" (**`EIP`**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

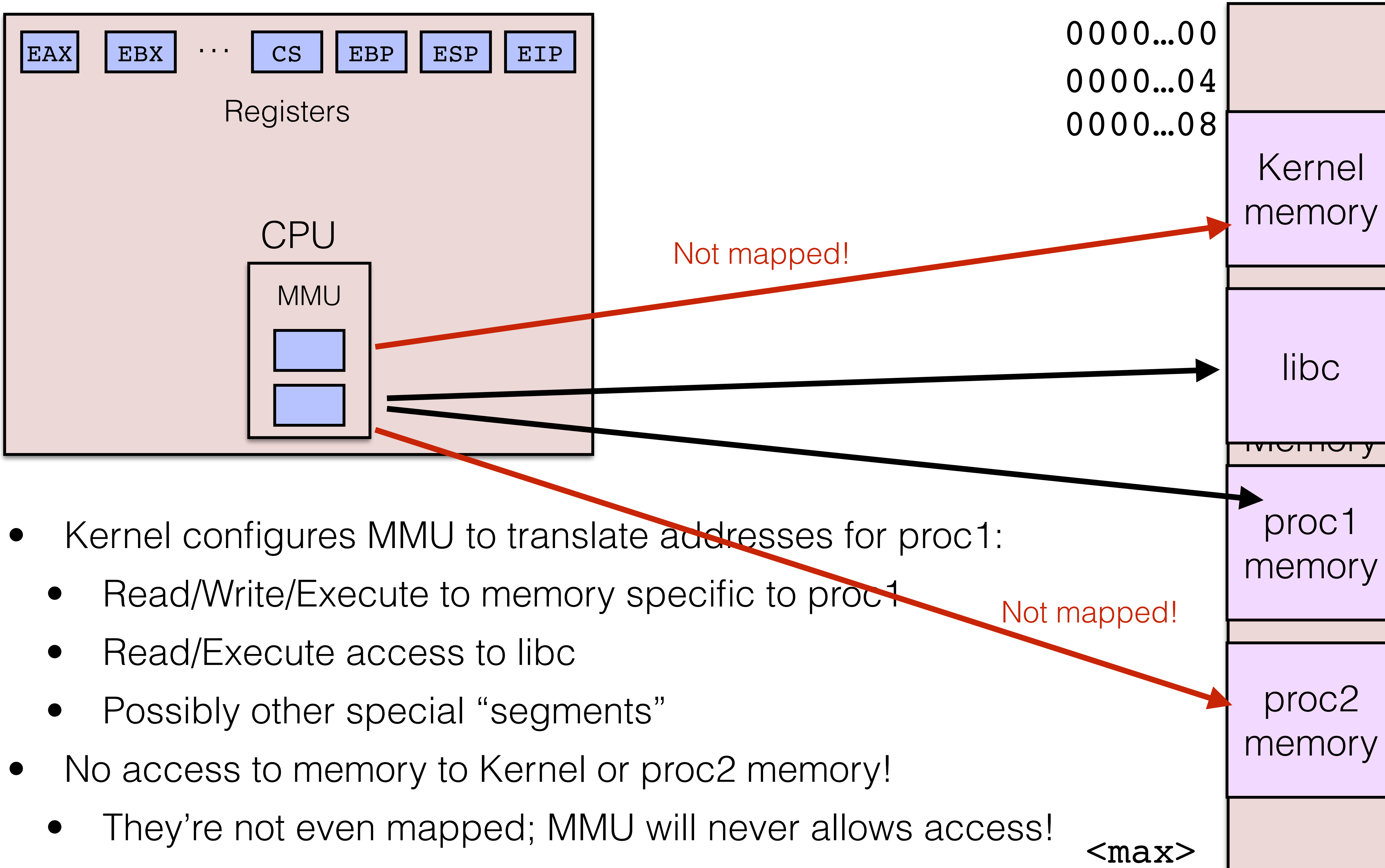
Handling Memory for a Process (cont.)





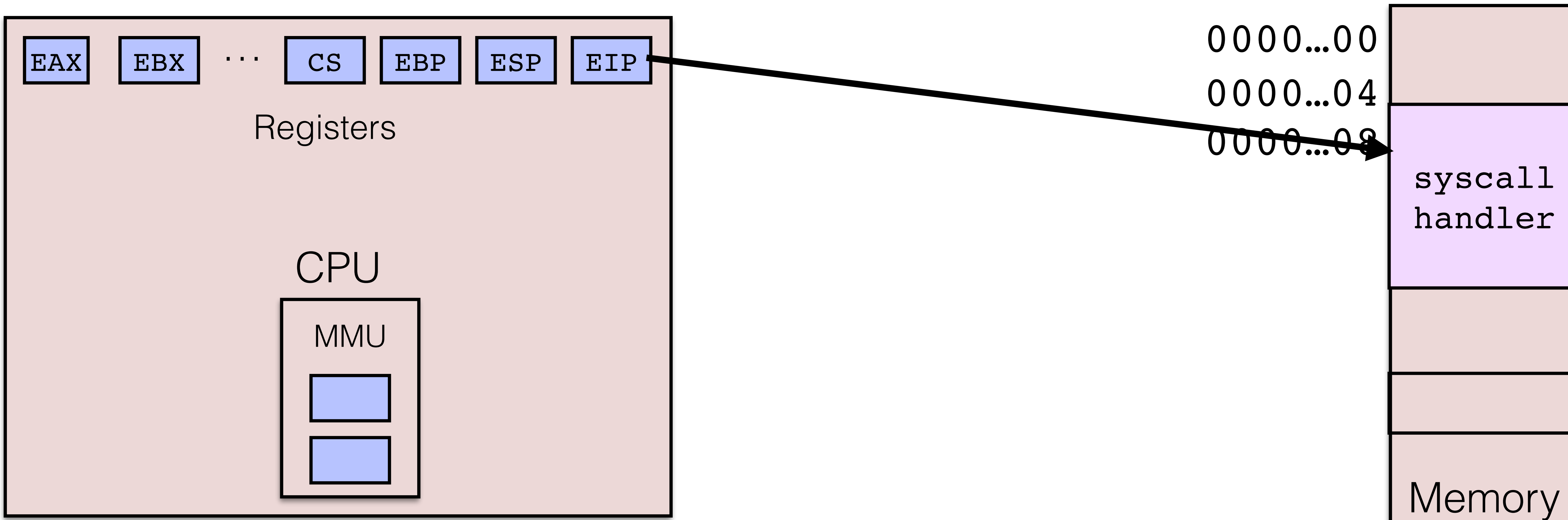
- Kernel can also map same memory into several processes' virtual address space
- Ex: Code for `malloc` is not copied for every process.

Handling Memory for a Process (cont.)

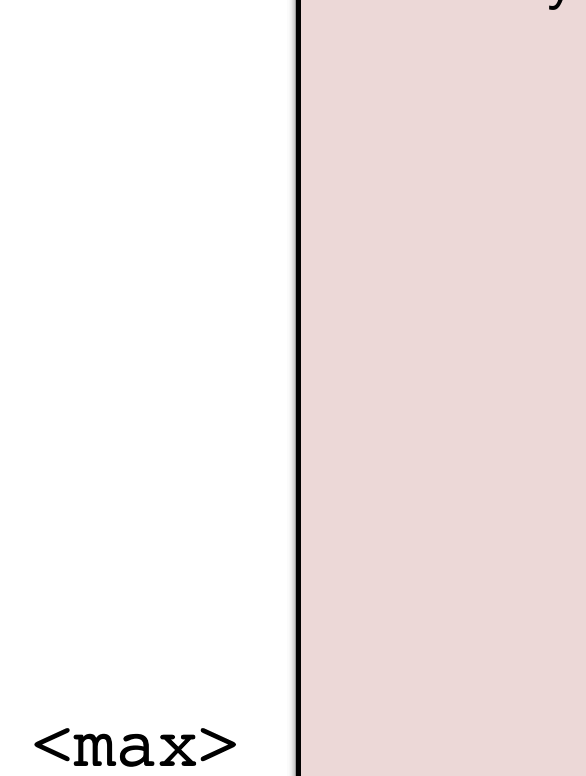


- Kernel configures MMU to translate addresses for proc1:
  - Read/Write/Execute to memory specific to proc1
  - Read/Execute access to libc
  - Possibly other special "segments"
- No access to memory to Kernel or proc2 memory!
  - They're not even mapped; MMU will never allows access!

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. They work roughly as follows:
  - Process sets up arguments in pre-defined registers
  - Then process executes instruction `int 0x80`
  - CPU will set `CPL`=0 and jump to kernel handler



Outline for Lecture 3

1. Wrap up “What is a process?”

2. Abstract approaches to access control (5.2)

3. UNIX notions of users, ownership, and permissions (5.1,5.3)

4. suid Permissions

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.

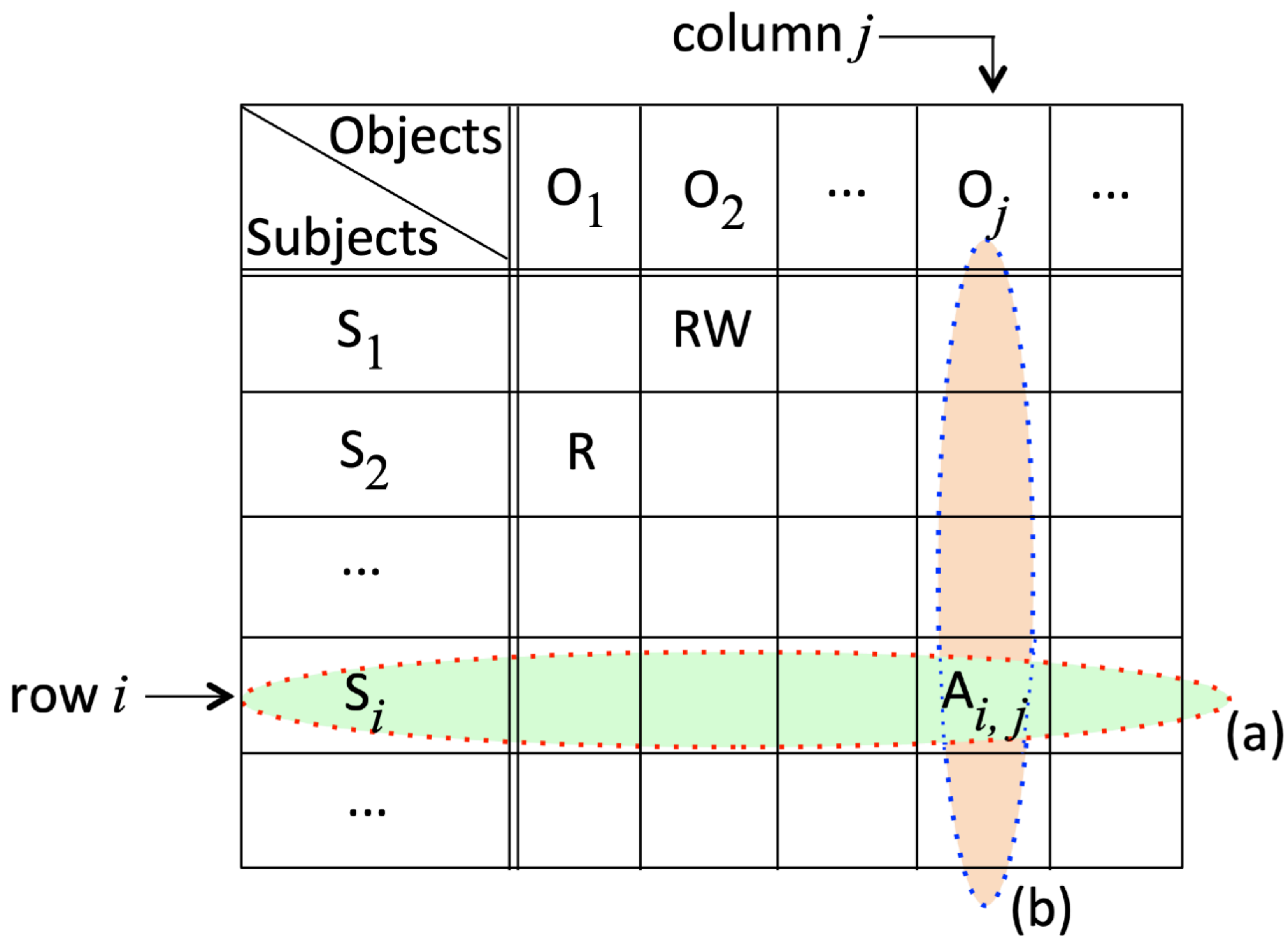
Fundamentals of Access Control: Policies

Guiding philosophy: Utter simplicity.

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

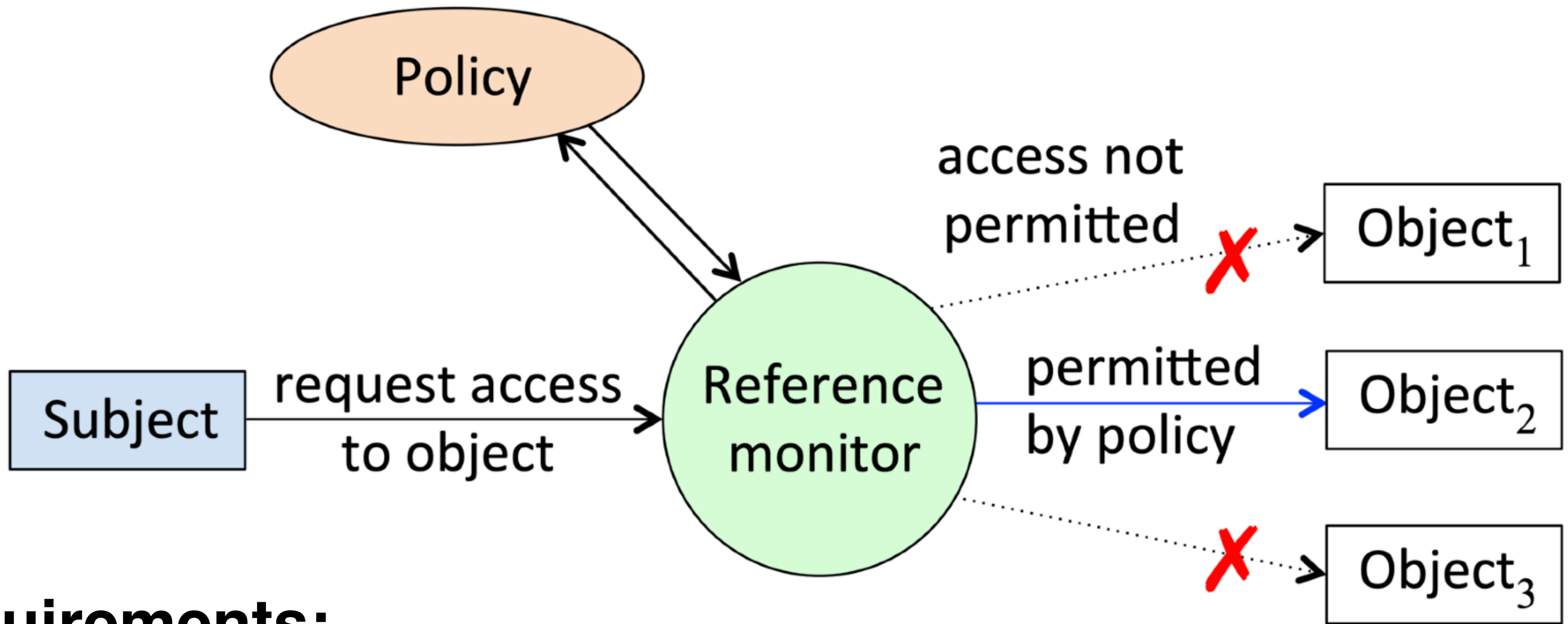
1. Define a sets of **subjects**, **objects**, and **verbs**.
2. A **policy** consists of a yes/no answer for every combination of subject/object/verb.

The Access Control Matrix



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

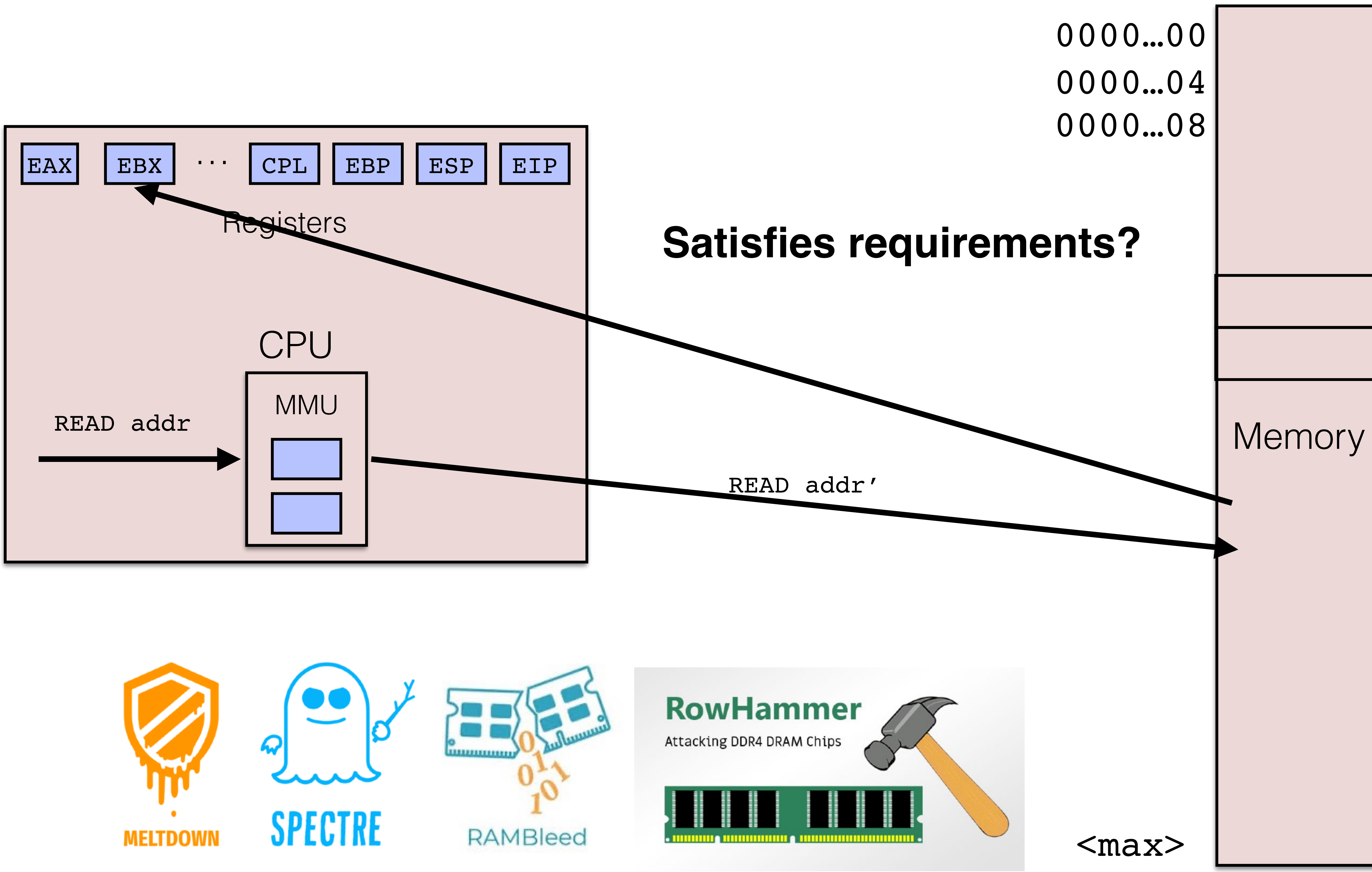
Enforcing Policy: Reference Monitors



Requirements:

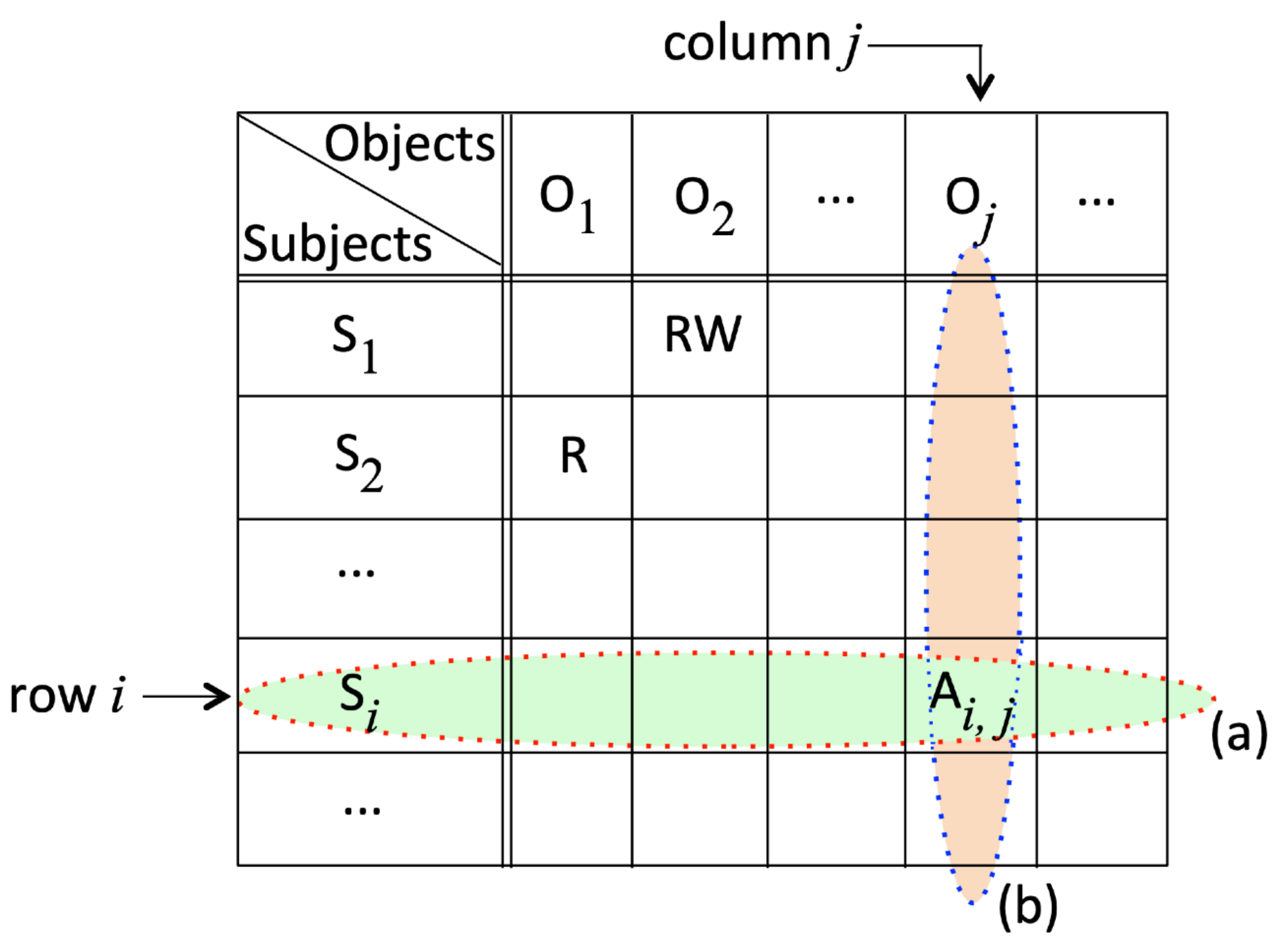
1. Tamper-proof.
2. Always invoked (not circumventable).
3. Verifiable; Simple enough to test thoroughly.
4. (Usually) Logs all requests.

Example Reference Monitor: The MMU



Implementing Reference Monitors: 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?”



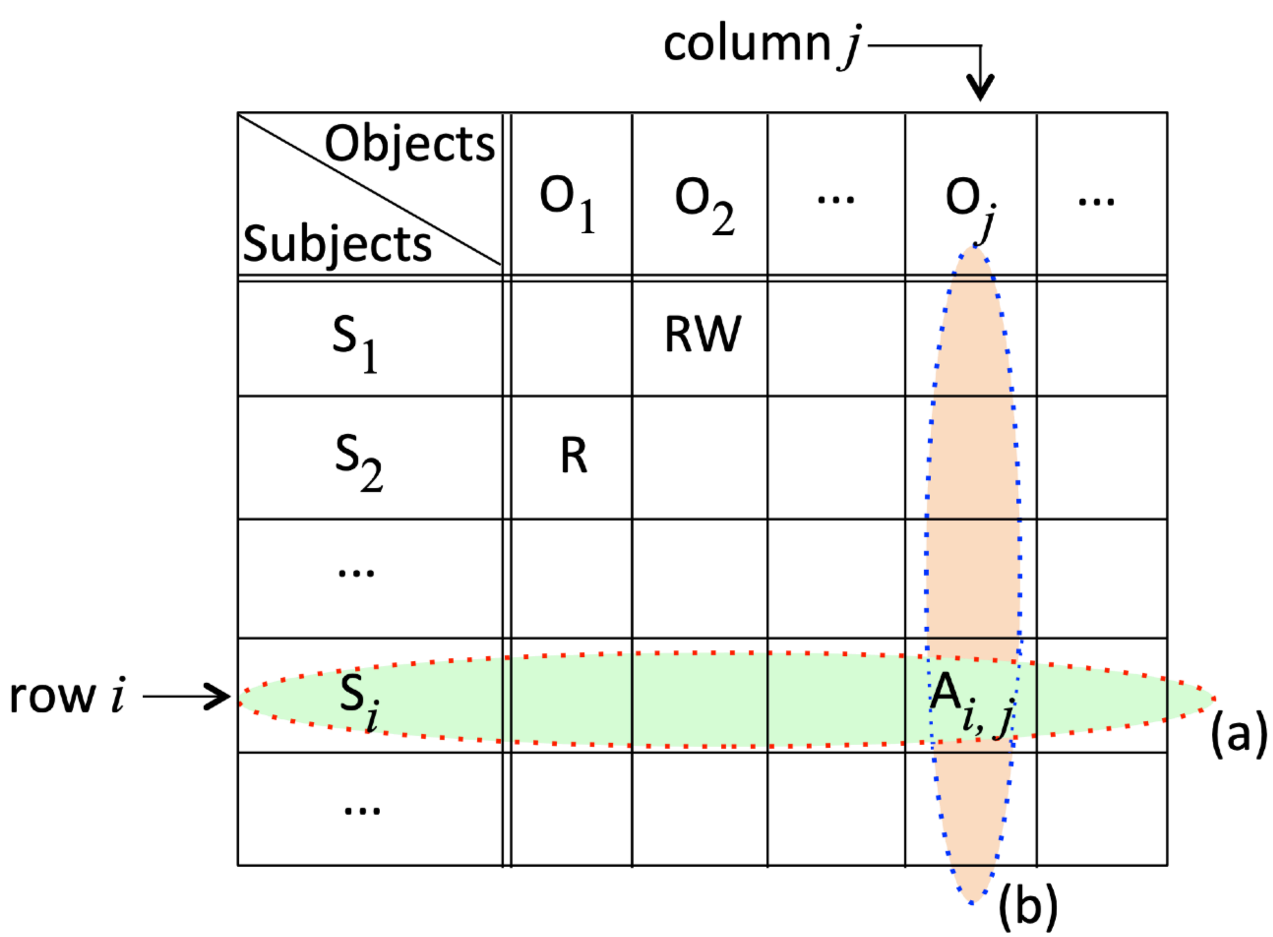
Examples:

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

More?

Implementing Reference Monitors: Capabilities

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

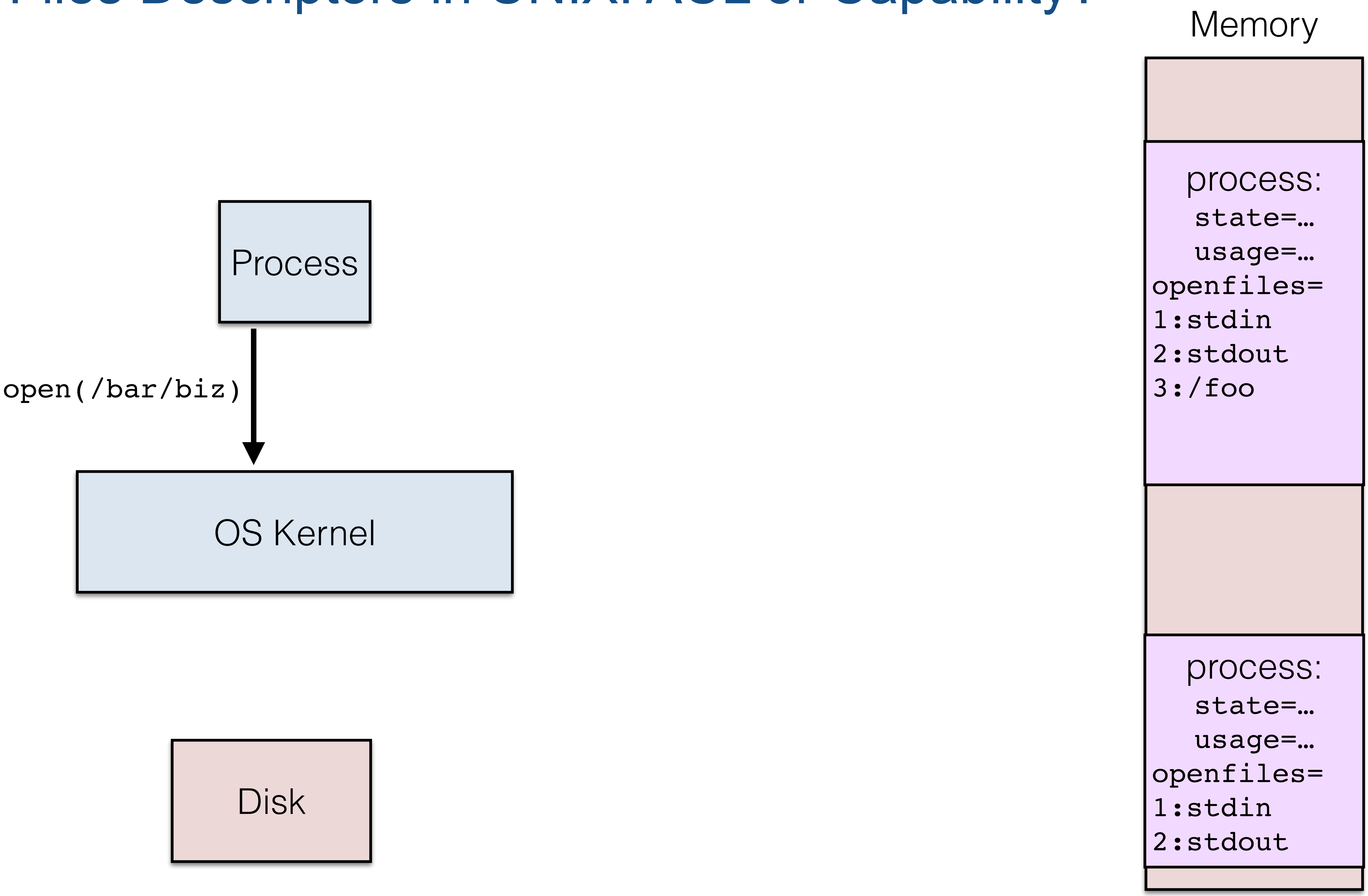


Examples:

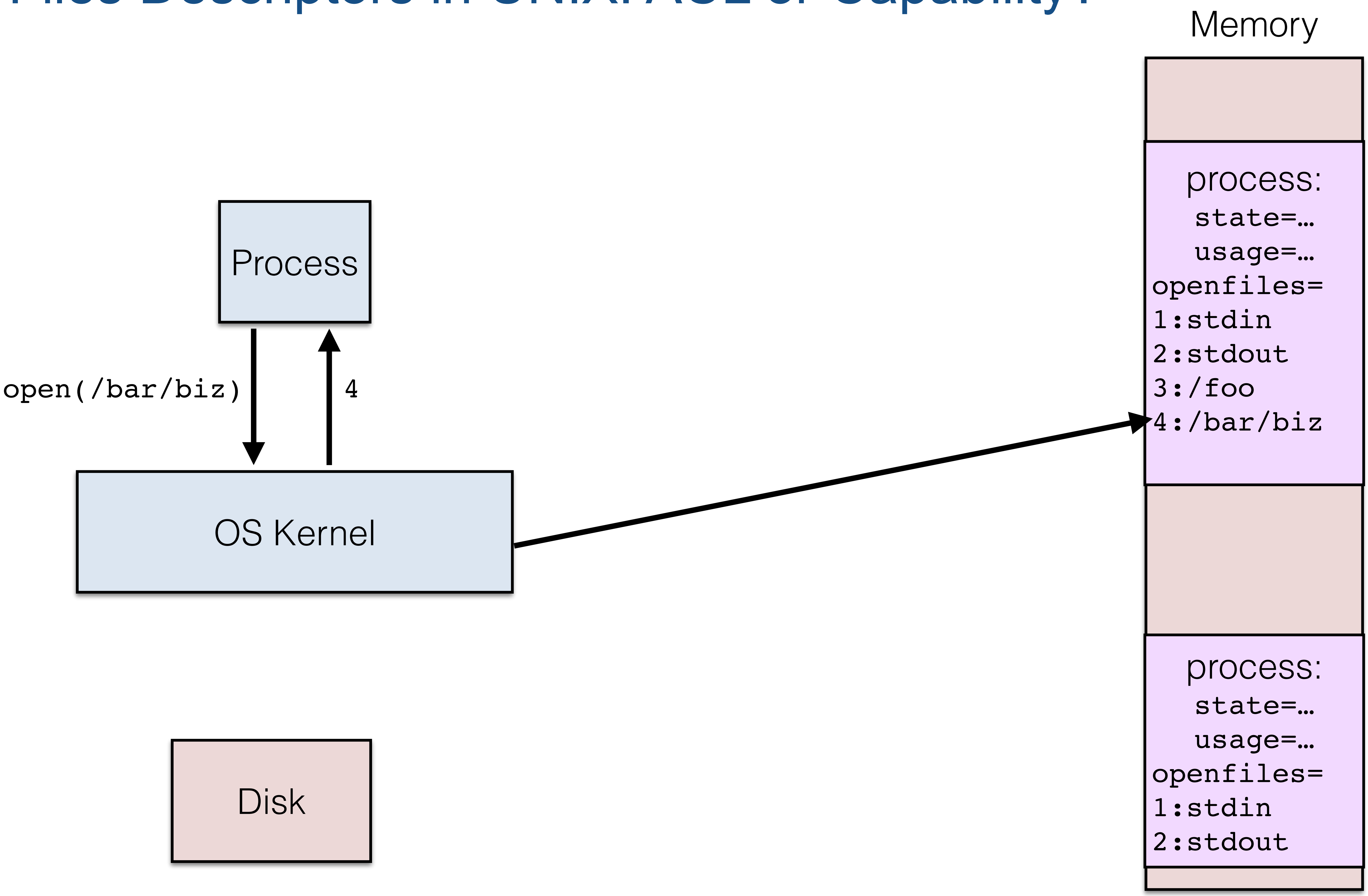
1. Movie ticket
2. Physical key to door lock

More?

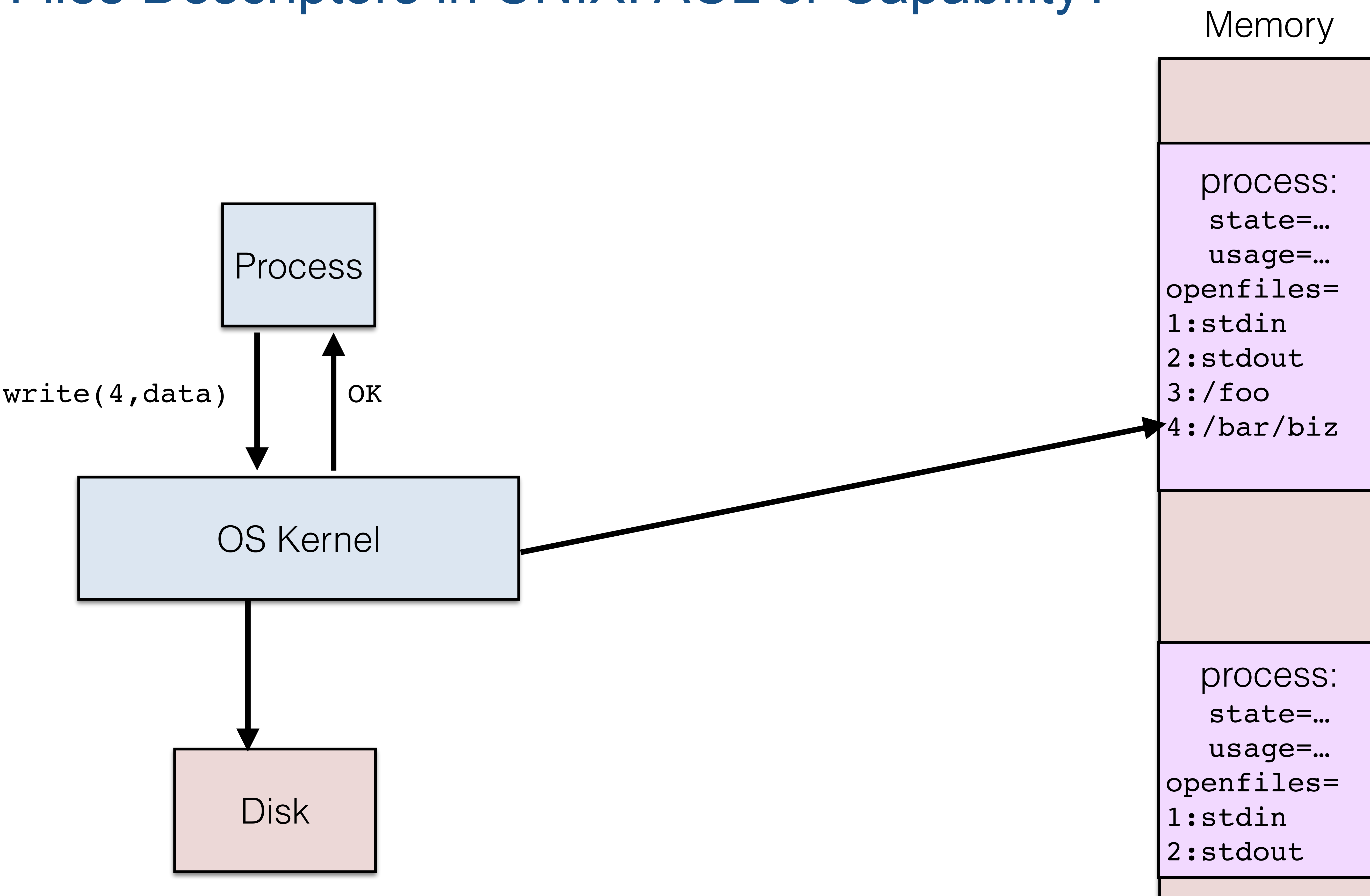
Files Descriptors in UNIX: ACL or Capability?



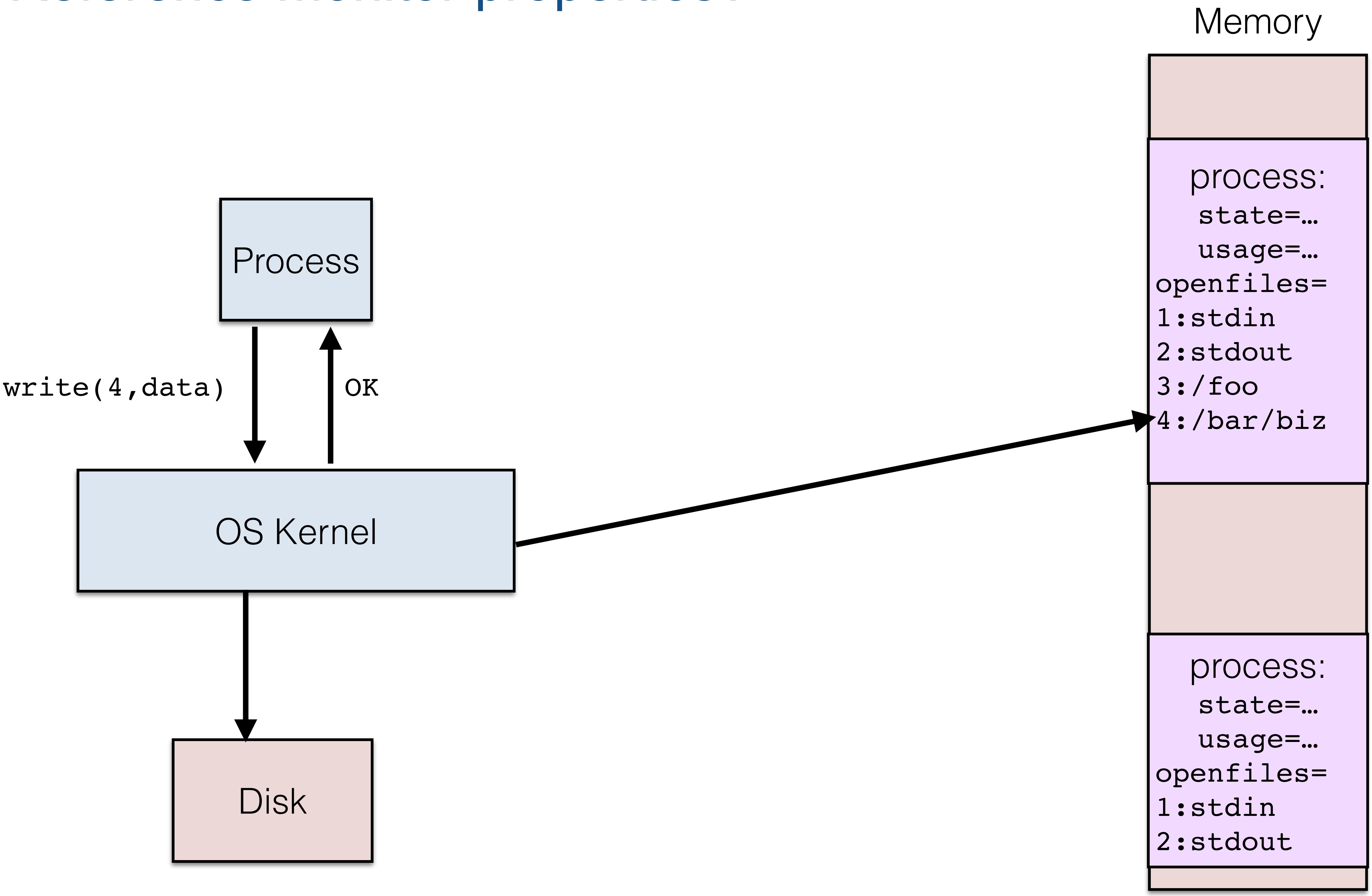
Files Descriptors in UNIX: ACL or Capability?



Files Descriptors in UNIX: ACL or Capability?



Reference monitor properties?



Outline for Lecture 3

1. Wrap up “What is a process?”
2. Abstract approaches to access control (5.2)
3. **UNIX notions of users, ownership, and permissions (5.1)**
4. suid Permissions

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. Looking at something concrete is enlightening.
3. You will almost certainly use it.

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

Users, Groups, UIDs/GIDs and File Ownership

- 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.

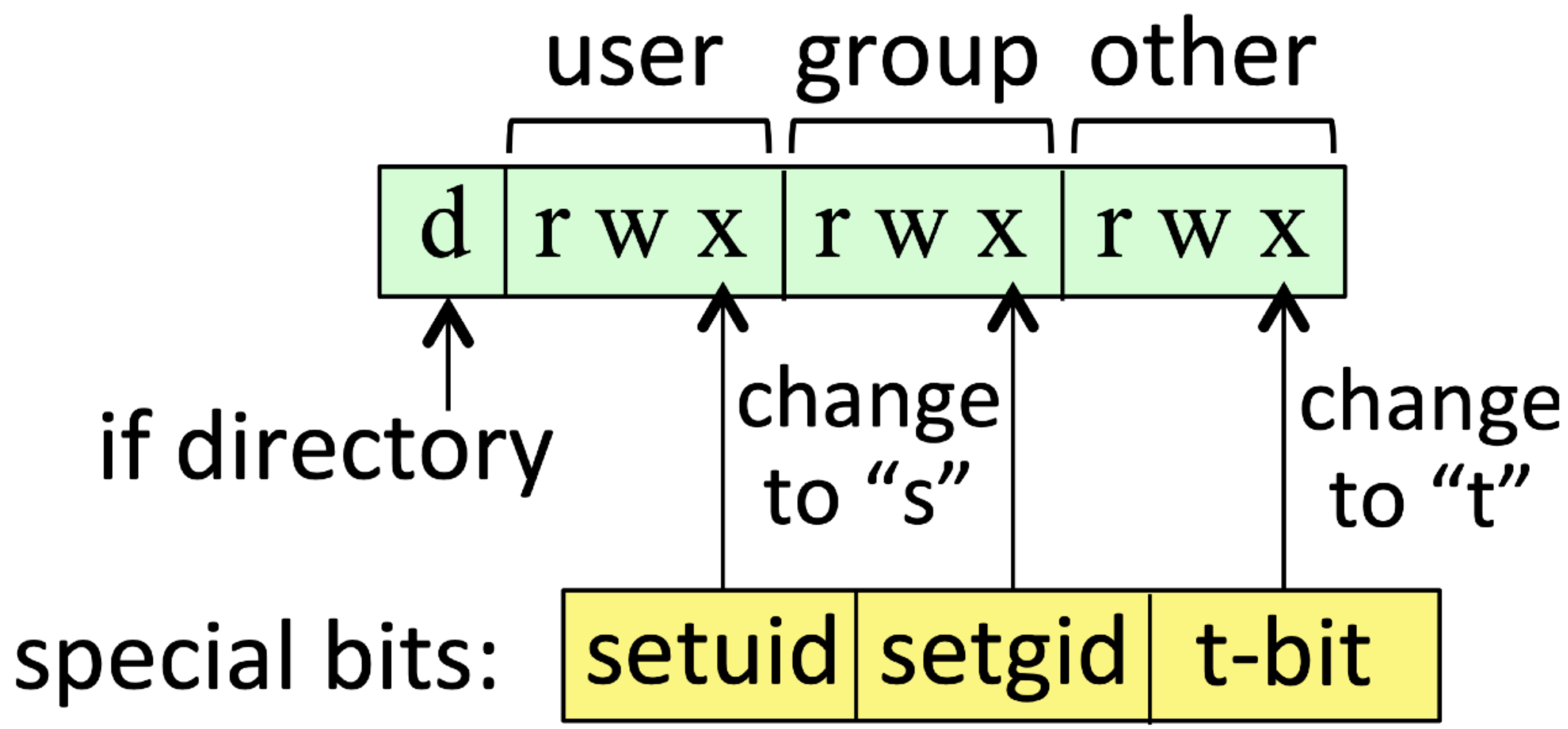
inode:
```
mode=1010100…
uid=davidcash
gid=cs232
ctime=…
```

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

File Permissions

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?

- Exception: Superuser (“root”) with UID=0 may bypass permissions.

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.
- **`fork()`** creates child process with same 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

Example: Web Servers

- Due to design of Linux, a web server must be run as **`root`** (!)
- Apache/NGINX written in C, a language in which vulnerabilities are common (next week!)

Apache » Http Server : Vulnerability Statistics

Vulnerabilities (**232**) CVSS Scores Report Browse all versions Possible matches for this product Related Metasploit Modules

Related OVAL Definitions : Vulnerabilities (288) Patches (241) Inventory Definitions (3) Compliance Definitions (0)

Vulnerability Feeds & Widgets

Vulnerability Trends Over Time

Year	# of Vulnerabilities	DoS	Code Execution	Overflow	Memory Corruption	Sql Injection	XSS	Directory Traversal	Http Response Splitting	Bypass something	Gain Information	Gain Privileges	CSRF	File Inclusion	# of exploits
1999	8	3	2	1											
2000	7		1				1								
2001	12	1								5	1				
2002	20	6	5	3			2	1			2				
2003	16	9	3	1							1				
2004	20	8	2	4				1		3	1	1			
2005	10	5	2	3			3			2					
2006	4	1	2				1			1					
2007	17	5	3				4	2		1	2	1			
2008	12	2			1		6		1			1	1		
2009	8	5								1		1			
2010	9	3	2	1			1				3				1
2011	12	8		1								1			2
2012	8	4		1			1				2	1			
2013	5	1	1				2								
2014	11	9	1	2						2	1				1
2015	4	2								1					
2016	4	2								1					
2017	11	1		1					1	1	1				
2018	13	3		1					1						
2019	14	1	1	2			1			2					
Total	225	79	25	21	1		22	4	3	20	14	6	1		4
% Of All		35.1	11.1	9.3	0.4	0.0	9.8	1.8	1.3	8.9	6.2	2.7	0.4	0.0	

Example: Web Servers

- Due to design of Linux, a web server must be run as `root` (!)
- Apache/NGINX written in C, a language in which vulnerabilities are common (next week!)

Vulnerability Details : CVE-2004-0492

Heap-based buffer overflow in proxy_util.c for mod_proxy in Apache 1.3.25 to 1.3.31 allows remote attackers to cause a denial of service (process crash) and possibly execute arbitrary code via a negative Content-Length HTTP header field, which causes a large amount of data to be copied.

Publish Date : 2004-08-06 Last Update Date : 2017-10-10

Collapse All Expand All Select Select&Copy Scroll To Comments External Links

Search Twitter Search YouTube Search Google

– CVSS Scores & Vulnerability Types

CVSS Score	10.0
Confidentiality Impact	Complete (There is total information disclosure, resulting in all system files being revealed.)
Integrity Impact	Complete (There is a total compromise of system integrity. There is a complete loss of system protection, resulting in the entire system being compromised.)
Availability Impact	Complete (There is a total shutdown of the affected resource. The attacker can render the resource completely unavailable.)
Access Complexity	Low (Specialized access conditions or extenuating circumstances do not exist. Very little knowledge or skill is required to exploit.)
Authentication	Not required (Authentication is not required to exploit the vulnerability.)
Gained Access	Admin
Vulnerability Type(s)	Denial Of Service Execute Code Overflow
CWE ID	CWE id is not defined for this vulnerability

– Vendor Statements

Fixed in Apache HTTP Server 1.3.32: http://httpd.apache.org/security/vulnerabilities_13.html

Source: Apache

Example: Web Servers

- Due to design of Linux, a web server must be run as `root` (!)
- Apache/NGINX written in C, a language in which vulnerabilities are common (next week!)

Nginx » Nginx : Vulnerability Statistics

Vulnerabilities (**26**) CVSS Scores Report Browse all versions Possible matches for this product Related Metasploit Modules

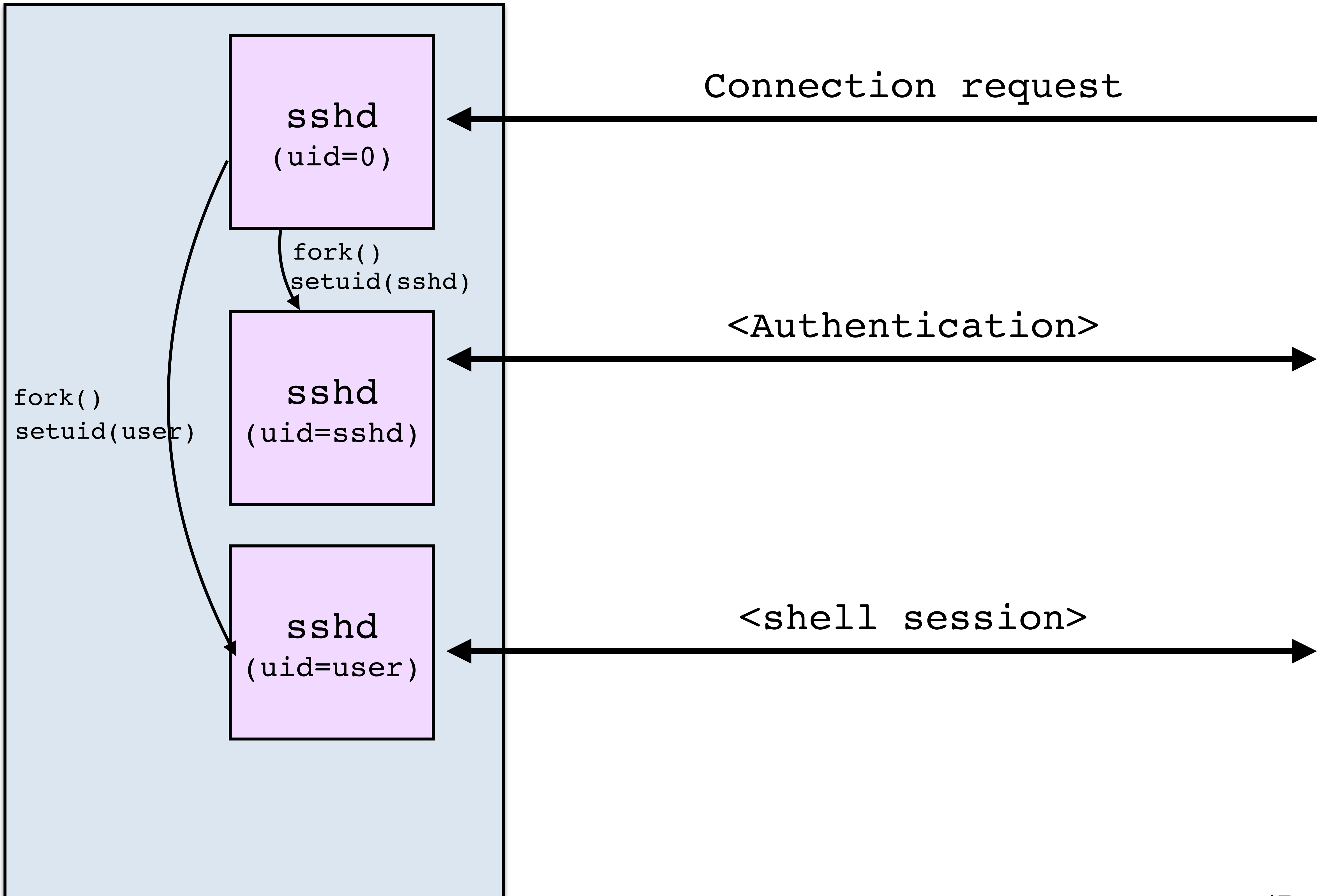
Related OVAL Definitions : Vulnerabilities (1) Patches (2) Inventory Definitions (0) Compliance Definitions (0)

Vulnerability Feeds & Widgets

Vulnerability Trends Over Time

Year	# of Vulnerabilities	DoS	Code Execution	Overflow	Memory Corruption	Sql Injection	XSS	Directory Traversal	Http Response Splitting	Bypass something	Gain Information	Gain Privileges	CSRF	File Inclusion	# of exploits
2009	3	1	1	2				1							
2010	2	1			1			1			1				3
2011	1	1		1											
2012	3	1	1	1						1	1				
2013	4	2	1	1						1	2				
2014	4		2	2											
2016	5	4										1			
2017	1			1							1				
2018	3														
Total	26	10	5	8	1			2		2	5	1			3
% Of All		38.5	19.2	30.8	3.8	0.0	0.0	7.7	0.0	7.7	19.2	3.8	0.0	0.0	

Example: Dropping Privileges in OpenSSH Server



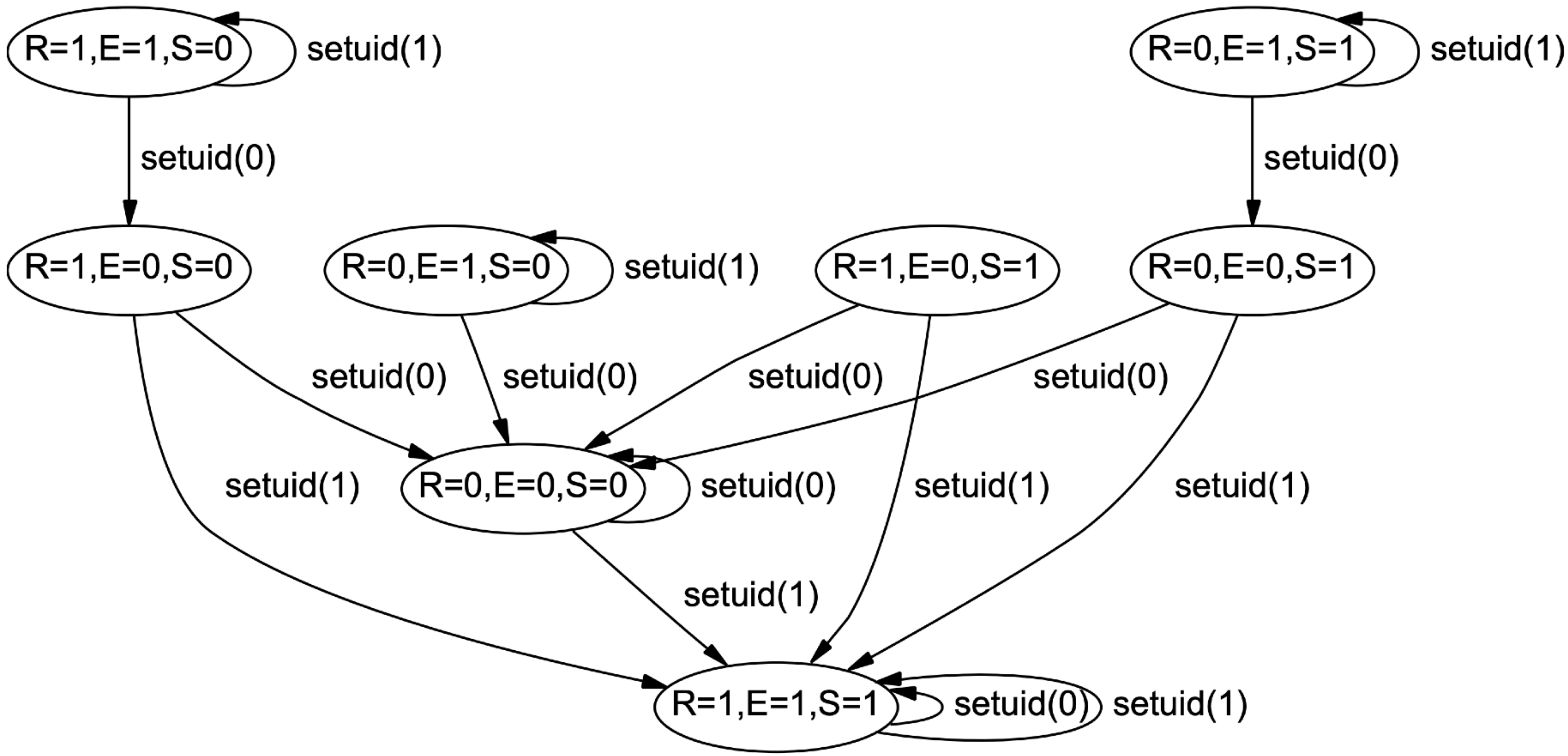
(Demo)

`setuid()` details are complicated

Setuid Demystified*

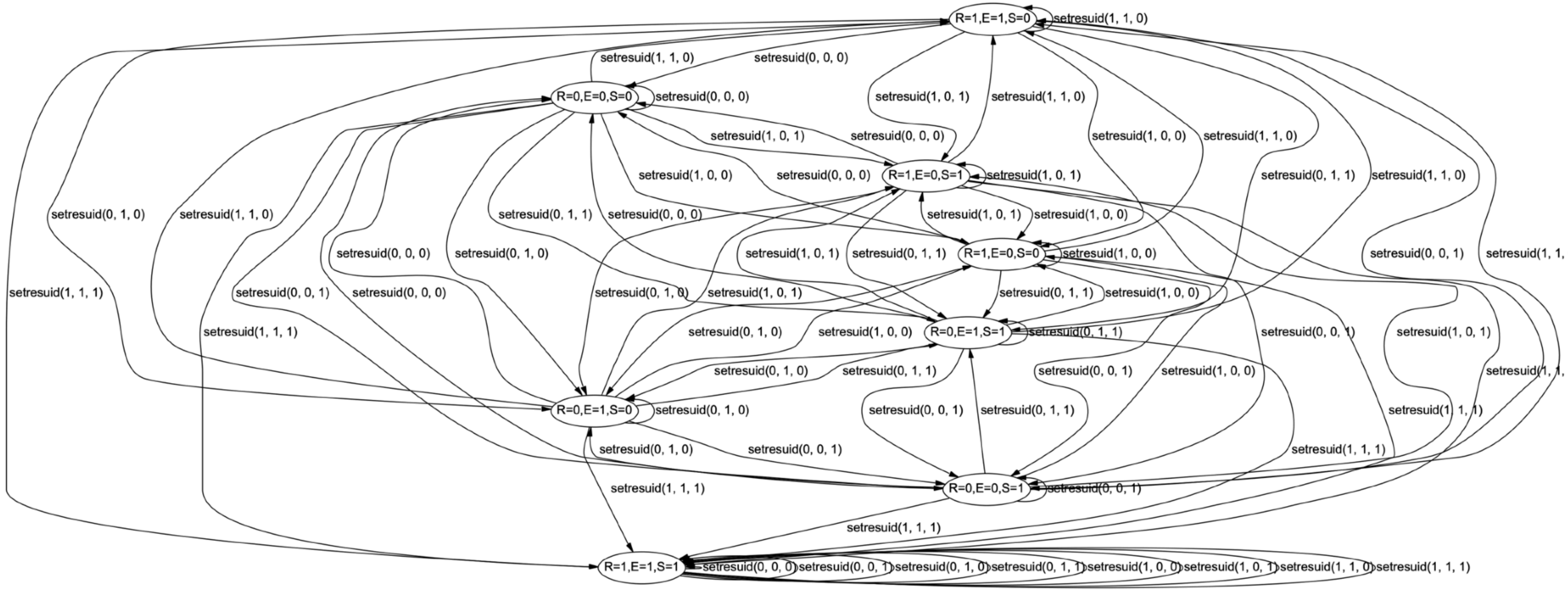
Hao Chen David Wagner
University of California at Berkeley
{hchen,daw}@cs.berkeley.edu

Drew Dean
SRI International
ddean@csl.sri.com



(a) An FSA describing *setuid* in Linux 2.4.18

… really complicated



(c) An FSA describing *setresuid* in Linux

Outline for Lecture 3

1. Wrap up “What is a process?”
2. Abstract approaches to access control (5.2)
3. UNIX notions of users, ownership, and permissions (5.1)
4. **suid Permissions**

`suid` Permission: Necessity and Danger

- Passwords stored in `/etc/shadow`, which is owned by `root`
- To change my password, I need to edit that file!
- Maybe add a syscall to kernel?
  - We'd have to add a ton of syscalls… violating P8: Small Trusted Base

Solution: Special permission on a program that allows anyone to "run it as root."

(Actually, anyone can run file with owner as uid.)

(Demo)

The End