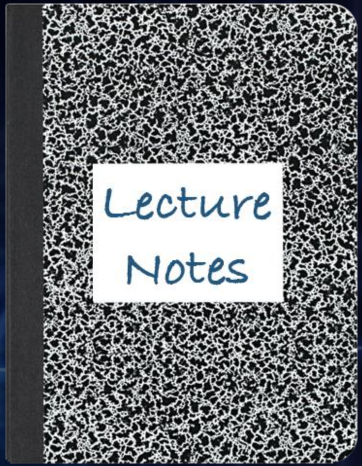


CS 419: Computer Security

Week 6: Access Control

Part 1: Protection

Paul Krzyzanowski

Protection is essential to security

- **Protection =** the mechanism that provides controlled access to resources
  - *A protection mechanism enforces security policies*
- Protection includes
  - **User privileges**: access rights to files, devices, and other system resources
  - **Resource scheduling & allocation**
    - Process scheduling & memory allocation – which processes get priority?
  - **Quotas** (sometimes) – set limits on disk space, CPU, network, memory, …
- And relies on
  - Mechanisms for user accounts & user authentication – identify who we're dealing with
  - Policies – defining who should be allowed do what
  - Auditing: generate audit logs for certain events

Co-located resources

- Earliest computers – *1945+*
  - Single-user batch processing – no shared resources
  - No need for access control – access control was physical
- Then … batch processing … but no shared storage – *1950s*
  - Per-process allocation of tape drives, printers, punched card machines, …
- Later … shared storage & timesharing systems – *1960s-now*
  - Multiple users share the same computer
  - User accounts & access control became important
- Even later … PCs – *1974 to now*
  - Back to single-user systems (mostly), although with a multi-user OS
  - … but software & media became less trusted by the 1990s
- Now: networked PCs + mobile devices + IoT devices + …
  - Shared access: cloud computing, file servers, university systems
  - Even more need to enforce access control

Access control

- Ensure that authorized users can do what they are permitted to do
  … *and no more*

- Real world
  - Keys, badges, guards, policies
- Computer world
  - Hardware
  - Operating systems
  - Web servers, databases & other multi-access software
  - Policies

Goals

- The OS gives us access to resources on a computer:
  - CPU
  - Memory
  - Files & devices
  - Network
- We need to:
  - Protect the operating system from applications
  - Protect applications from each other
  - Allow the OS to stay in control
  - Restrict what users can do

The OS and hardware are the
fundamental parts of the Trusted Computing Base (TCB)

Regaining control: hardware timer

The operating system kernel requests timer interrupts

- One of several timer devices on Intel architectures:
  - High Precision Event Timer (HPET)
  - or Advanced Programmable Interrupt Controller (APIC timer, one per CPU)
- Most current Intel Linux & Windows systems use the APIC timer
  - The kernel sets a periodic interrupt: HZ=250, 300, or 1000 Hz to trigger the scheduler
  - In tickless kernels (`CONFIG_NO_HZ_FULL`), timers fire only when needed
    - The kernel calculates the next relevant event - interrupts are eliminated when the system is idle
    - Microsoft Windows also uses tickless scheduling (since Vista)
    - macOS uses a hybrid scheduler, mostly event-based

Applications cannot disable these interrupts

This ensures that the OS can always regain control

Processes

Timer interrupts ensure the OS can take control periodically

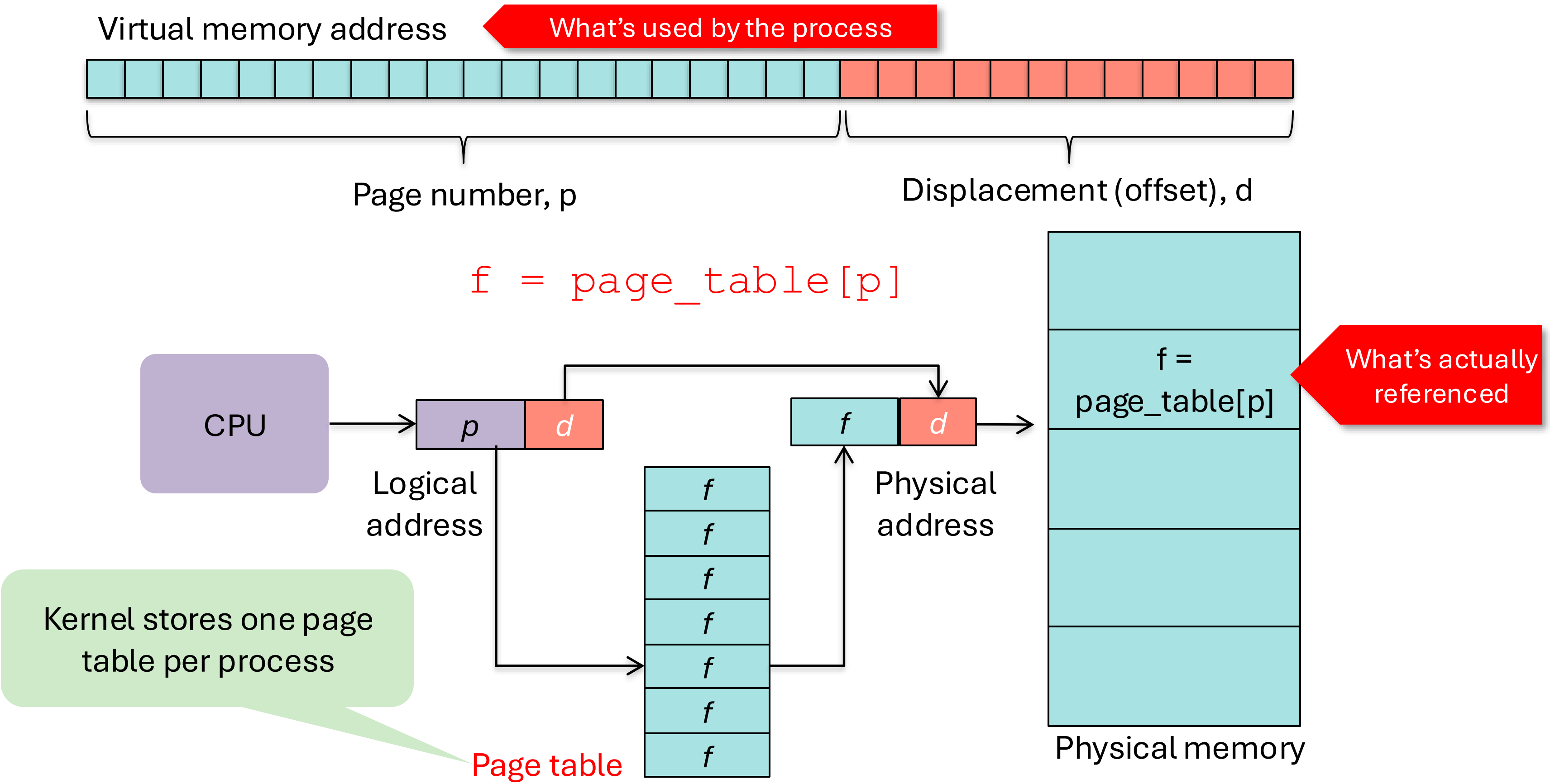
OS Process Scheduler

- Decides whether a process had enough CPU time, and it is time for another process to run
- Prioritizes threads
  - Based on user, user-defined priorities, interactivity, deadlines, “fairness”
  - One process should not adversely affect others
- Avoid **starvation**: ensure all processes will get a chance to run
  - This would be an **availability** attack

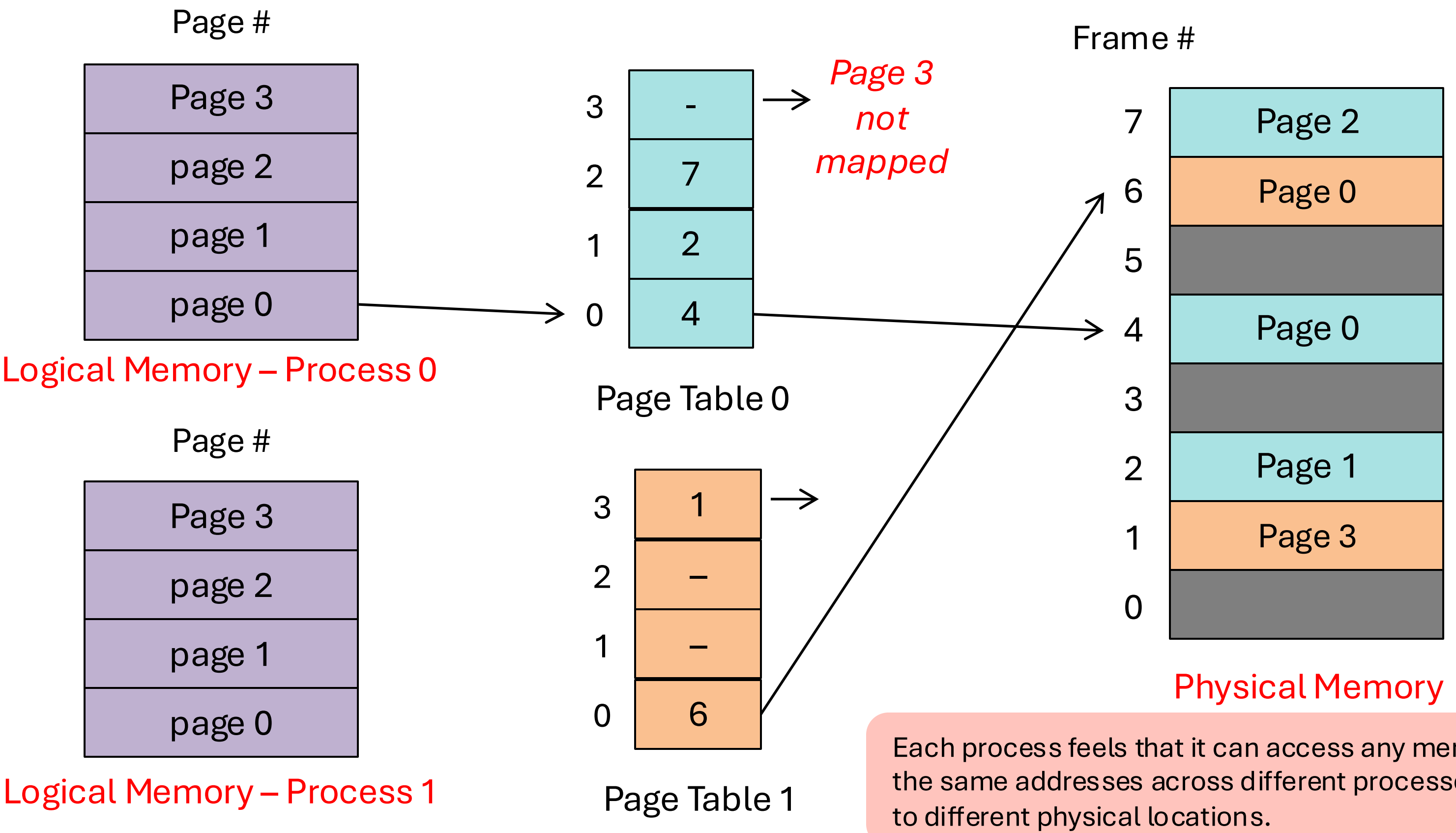
Memory Protection: Memory Management Unit

- All modern CPUs have a Memory Management Unit (MMU)
- OS provides each process with virtual memory
- Gives each process the illusion that it has the entire address space
- One process cannot see another process' address space
- Enforce memory access rights
  - Read-only (code)
  - Read-write (program's data)
  - Execute (code)
  - Unmapped

Page translation



Logical vs. physical views of memory



Each process feels that it can access any memory but the same addresses across different processes will map to different physical locations.

User & kernel mode

Kernel mode = privileged, system, or supervisor mode

- Access restricted regions of memory
- Modify the memory management unit by changing the page table register and memory map (page tables)
- Set hardware timers
- Define interrupt vectors
- Halt the processor
- Etc.

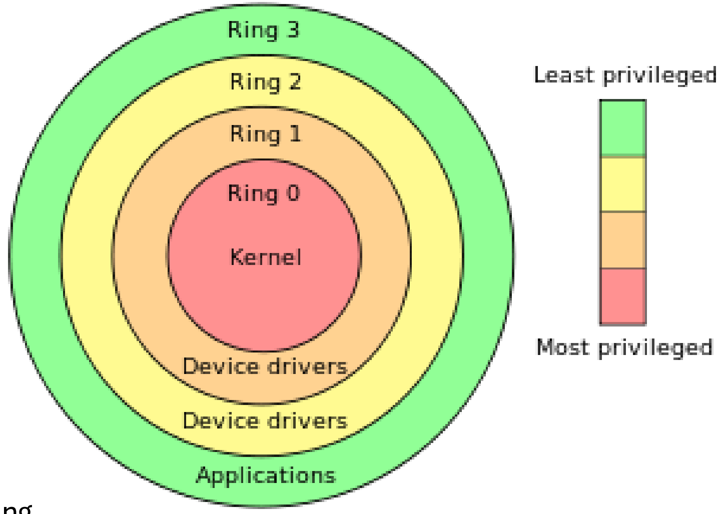
Getting into kernel mode

- **Trap**: explicit instruction
  - Intel architecture: **INT** instruction (interrupt)
  - ARM architecture: **SWI** instruction (software interrupt)
  - System call instructions (*SYSCALL*)
- **Violation** (e.g., access unmapped memory, illegal instruction)
- Hardware **interrupt** (e.g., receipt of network data or timer)

Protection Rings

All modern operating systems support two modes of operation: **user** & **kernel**

- Multics (an early OS) defined a ring structure with 6 different privilege levels
  Intel inherited this:
  - Each ring is protected from higher-numbered rings
  - Special call instruction (**call gates**) to cross rings: jump to predefined locations
  - Most of the OS did not run in ring 0
- Intel x86, IA-32 and IA-64 support 4 rings
- Today's OSes only use
  - **Ring 0**: kernel
  - **Ring 3**: user
- Additional protection levels
  - **Ring -1**: Hypervisor (virtual machine monitor)
  - **Ring -2**: System Management Mode (SMM)
    - Low-level, high-priority tasks like power management, thermal monitoring



https://en.wikipedia.org/wiki/Protection_ring

Subjects, Principals, and Objects

Subject: the process carrying out actions on behalf of the principal

Principal: a specific identity that can be authenticated – user or service

User: a human who interacts with the system, a subset of principals

Object: the resource the subject may access

- Typically, files and devices – they do not perform operations

***Subjects** access **objects**: they perform actions on objects*

Access control

- Define what operations subjects can perform on objects

Most of today's operating systems control who can do what to each object
(access permissions are associated with each object)

User authentication

Must be performed before we can do access control

Establish user identity – determine the ***principal***

- Operating system privileges are granted based on user identity

Common steps

1. **Get user credentials** (e.g., name, password)
2. **Authenticate user** by validating the credentials
   - Get user ID(s), group ID(s)
3. **Control access**: grant access to resources based on user/group IDs & policies

Modeling Protection

The Access Control Matrix

Domains of protection

Subjects interact with objects

- A process runs with the authority of a principal – and is the *subject* in an access control decision
- Objects include:
  hardware (CPU, memory, I/O devices)
  software: files, processes, semaphores, messages, signals

A process should be allowed to access only objects that it is authorized to access

- A process operates in a **protection domain**
- It's part of the **context of the process**
- The protection domain **defines the objects the process may access** and how it may access them

Modeling Protection: Access Control Matrix

Rows: domains
(subjects or groups of subjects)

Columns: objects

Each entry in the matrix represents an access right of a domain on an object

Objects

Subjects
domains of protection

	F_0	F_1	Printer
D_0	read	read-write	print
D_1	read-write-execute	read	
D_2	read-execute		
D_3		read	print
D_4			print

An **Access Control Matrix** is the primary abstraction for protection in computer security

We may need some more controls

- Domain transfers
  - Allow a process to run under another domain's permissions
- Copy rights (delegation of access)
  - Allow a user to grant certain access rights for an object
- Owner rights
  - Identify a subject as the owner of an object
  - Can change access rights on that object for any domain
- Domain control
  - A process running in one domain can change any access rights for another domain

Access Control Matrix: Domain Transfers

Switching from one domain to another is a configurable policy

Domain transfers

Allow a process to run under another domain's permissions

Why? Log a user in – how would you run the first user's process?

objects

Subjects

domains of protection

	F_0	F_1	Printer	D_0	D_1	D_2	D_3	D_4
D_0	read	read-write	print	–	switch	switch		
D_1	read-write-execute	read			–			
D_2	read-execute				switch			
D_3		read	print					
D_4			print					

A process in D_0 can switch to running in domain D_1

Access Control Matrix: Delegation of Access

Copy rights: allow a user to grant certain rights to others

- Copy a specific access right on an object from one domain to another

objects

Subjects
domains of protection

	F_0	F_1	Printer	D_0	D_1	D_2	D_3	D_4
D_0	read	read-write	print	–	switch	sw		
D_1	read-write-execute	read*						
D_2	read-execute				swtich	–		
D_3		read	print					
D_4			print					

A process executing in *D_1* can give a *read* right on F_1 to another domain

Access Control Matrix: Object Owner

Owner: allow new rights to be added or removed

Identify a subject as the owner of an object
Can change access rights on that object for any domain (column)

objects

Subjects
domains of protection

	F_0	F_1	Printer	D_0	D_1	D_2	D_3	D_4
D_0	read owner	read-write	print	–	switch	sw		
D_1	read-write-execute	read*						
D_2	read-execute				swtich			
D_3		read	print					
D_4			print					

A process executing in D_0 owns F_0, so it can give a *read* right on F_0 to domain D_3 and remove the *execute* right from D_1

Access Matrix: Domain Control

- A process running in one domain can change any access rights for another domain
- Change entries in a row (all objects)

objects

Subjects
domains of protection

	F_0	F_1	Printer	D_0	D_1	D_2	D_3	D_4
D_0	read owner	read-write	print	–	switch	switch		
D_1	read-write-execute	read*			–			control
D_2	read-execute				switch			
D_3		read	print					
D_4			print					

A process executing in D_1 can modify any rights in domain D_4

This gets messy!

- An access control matrix does not address everything we may want
- Processes execute with the rights of the user (domain)
  - But sometimes they need extra privileges
    - Read configuration files
    - Read/write from/to a restricted device
    - Append to a queue
- We don't want the user to be able to access these objects
  - Adding domains to the row of objects is not sufficient
  - We may need a 3-D access control matrix: (subjects, objects, processes)
- This gets messy!
  - One solution is to give an executable file a **temporary domain transfer**
    - Assumption is this is a trusted application that can access these resources
  - When run, it assumes the privileges of another domain

Implementing an access matrix

A single table to store an access matrix is impractical

- Big size: # domains (users) × # objects (files)
- Objects may come and go frequently
- Lookup needs to be efficient

Implementing an access matrix

Access Control List

– Associate a column of the table with each object

objects

Subjects

domains of protection

	F_0	F_1	F_2	F_3	F_3	Printer
D_0	read owner	read-write	read-execute	read		print
D_1	read-write-execute	read	read-execute	read	write	
D_2	read-execute		read-execute		write	
D_3		read	read-execute			print
D_4			read-execute		write	print

ACL for file F_0

Access Control Lists

Access control list =
For each object:

$$\{\ \text{subject}_1\text{:permitted\_operations},\ \underbrace{\text{subject}_2\text{:permitted\_operations}}_{=\ \text{ACE: access control entry}},\ \ldots\ \}$$

Possessing the capability means that access is allowed

A capability is a protected object

Capability List

– Associate a row of the table with each domain

objects

Subjects / *domains of protection*

	F_0	F_1	F_2	F_3	F_3	Printer
D_0	read owner	read-write	read-execute	read		print
D_1	read-write-execute	read	read-execute	read	write	
D_2	read-execute		read-execute		write	
D_3		read	read-execute			print
D_4			read-execute			

Capability list for domain D_1

Capability Lists

Capability list = list of objects together with the operations a specific subject can perform on the objects

- Each item in the list is a *capability*: the operations allowed on a specific object
  - Also known as a **ticket** or **access token**
- A process presents the capability to the OS along with a request
  - Possessing the capability means that access is allowed
- The capability is a protected object
  - A process cannot modify its capability list

Capability Lists

Capability list =
For each subject:

$$\{\ \text{object}_1\text{:permitted\_operations},\ \underbrace{\text{object}_2\text{:permitted\_operations},}_{\text{capability}}\ \ldots\ \}$$

capability

= ticket = access token

Possessing the capability means that access is allowed

Capability Lists

- Advantages
  - Run-time checking is more efficient
  - Delegating rights is easy
- Disadvantages
  - Creating or deleting files means updating all capability lists
  - Changing a file's permissions is hard
  - Hard to find all users that have access to a resource
  - Lists can be huge – the system might have millions of objects
- Not used in mainstream systems in place of ACLs
  - Limited implementations: Cambridge CAP, IBM AS/400, Google Fuchsia OS
- Capability lists are more commonly used for network services
  - Used in single sign-on services and other authorization services such as OAuth and Kerberos (sort of)
  - **Access Tokens**
    - Identifies a user's identity and the access rights permitted on the requested service (not objects!)

The End