Cristina Nita-Rotaru

# CS526: Information security

## Operating System Security and UNIX Access Control

# Readings for This Lecture

- Wikipedia
  - CPU modes
  - System call
  - Filesystem Permissions
- Other readings
  - UNIX File and Directory Permissions and Modes
    - http://www.hccfl.edu/pollock/AUnix1/FilePermissions.htm
  - Unix file permissions
    - http://www.unix.com/tips-tutorials/19060-unix-file-permissions.html

# Security Goals for Operating Systems

- Enabling multiple users to securely share a computer
  - Separation and sharing of processes, memory, files, devices, etc.
- Threat model
  - Users may be malicious
  - Users have terminal access to computers
  - Software may be malicious/buggy
- Security mechanisms
  - User authentication – see previous lecture
  - Memory protection
  - Processor modes
  - File access control

# Security Goals of Operating Systems

- Modern OS must ensure secure operation in a networked environment
- What is the threat model?
- Security mechanisms
  - Authentication
  - Access Control
  - Secure Communication (using cryptography)
  - Logging & Auditing
  - Intrusion Prevention and Detection
  - Recovery

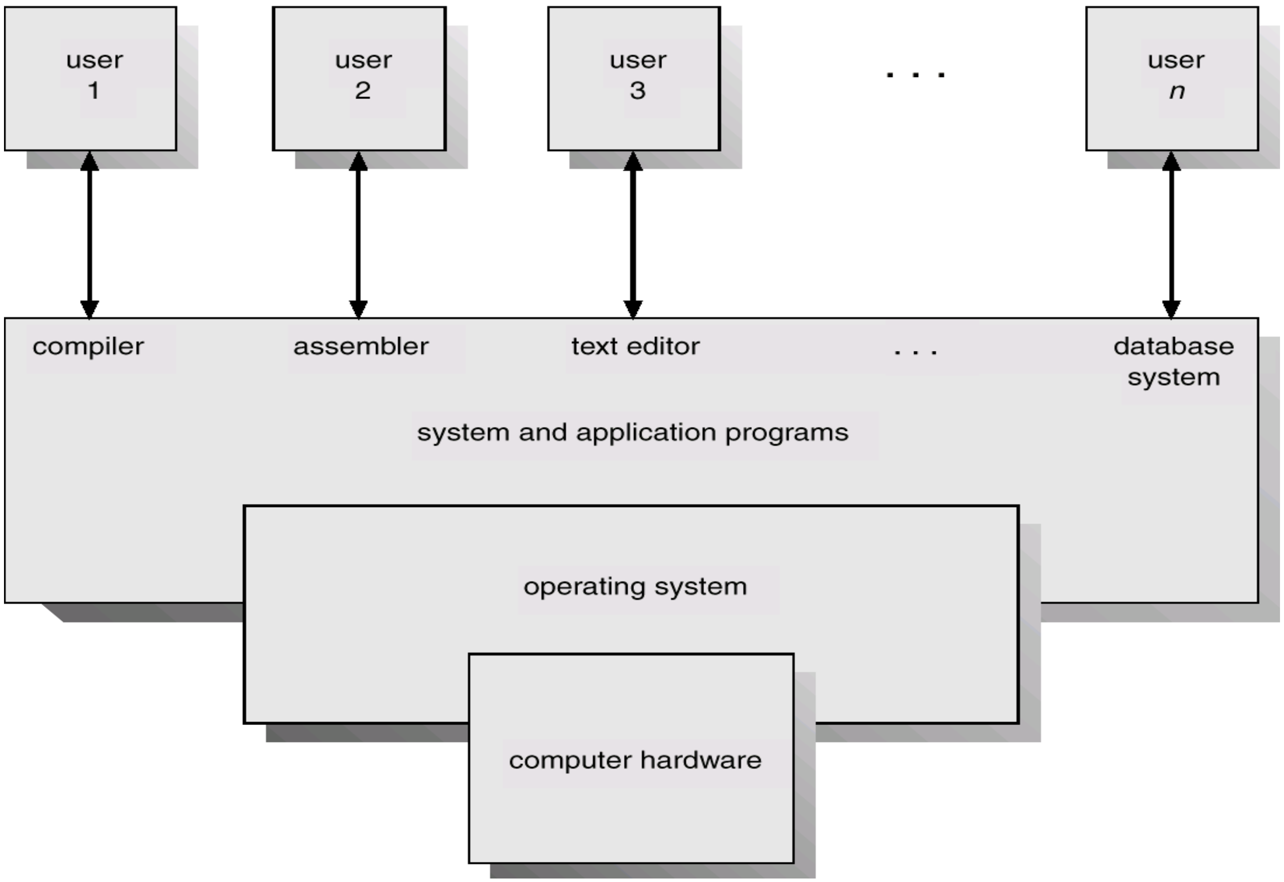
# Reconciling Separation and Sharing

- Ensure separation
  - Physical
  - Temporal
  - Logical
  - Cryptographical
- OS also need to ensure sharing

# Computer System Components

- Hardware
  - Provides basic computing resources (CPU, memory, I/O devices)
- Operating system
  - Controls and coordinates the use of the hardware among the various application programs
- Applications programs
  - Define the ways in which the system resources are used to solve the computing problems of the users
- Users
  - E.g., people, machines, other computers

# Abstract View of System Components



# Memory Protection: Access Control

- Ensures that one user's process cannot access other's memory
  - fence
  - relocation
  - base/bounds register
  - segmentation
  - paging
  - …
- Operating system and user processes need to have different privileges

# Single-program, no Memory Protection

- One application runs at a time
- Each application runs within a hardwired range of physical memory addresses
- Application can use the same physical addresses every time, across reboots
- Applications typically use the lower memory addresses
- An OS uses the higher memory addresses
- An application can address any physical memory location

# Multiprogramming, no Memory Protection

- When a program is copied into memory, a linker-loader alters the code of the program (e.g., loads, stores, and jumps)
  - To use the address of where the program lands in memory
- Bugs in any program can cause other programs to crash, even the OS

# Multiprogramming with Memory protection

- Memory protection keeps user programs from crashing one another and the OS
  - Main idea: make all memory accesses go through an OS controlled component which provides address translation while checking all addresses
- Two hardware-supported mechanisms
  - Address translation (controlled by the OS)
  - Dual-mode operation for the CPU

# Address Translation

- Each process is associated with an address space, or all the addresses a process can touch
  - Each process believes that it owns the entire memory, starting with the virtual address 0
- A translation table to translate every memory reference from virtual to physical addresses
  - Processes cannot talk about other processes' addresses, nor about the OS addresses
  - OS uses physical addresses directly
    - No translations

# Dual-Mode Operation Revisited

- Translation tables offer protection if they cannot be altered by applications
- An application can only touch its address space under the user mode
- Hardware requires the CPU to be in the kernel/system/ proviledged mode to modify the address translation tables

# CPU Modes (processor modes or privilege)

- System mode (privileged mode, master mode, supervisor mode, kernel mode)
  - Can execute any instruction
  - Can access any memory locations, e.g., accessing hardware devices
  - Can enable and disable interrupts
  - Can change privileged processor state
  - Can access memory management units
  - Can modify registers for various descriptor tables

Reading: http://en.wikipedia.org/wiki/CPU_modes

# Switching from the Kernel to User Mode

- To run a user program, the kernel
  - Creates a process and initialize the address space
  - Loads the program into the memory
  - Initializes translation tables
  - Sets the hardware pointer to the translation table
  - Sets the CPU to user mode
  - Jumps to the entry point of the program

# User Mode

- User mode
  - Access to memory is limited,
  - Cannot execute some  instructions
  - Cannot disable interrupts,
  - Cannot change arbitrary processor state,
  - Cannot access memory management units
- Transition from user mode to system mode can only happen via well defined entry points, i.e., through system calls

Reading:  http://en.wikipedia.org/wiki/CPU_modes

# System Calls

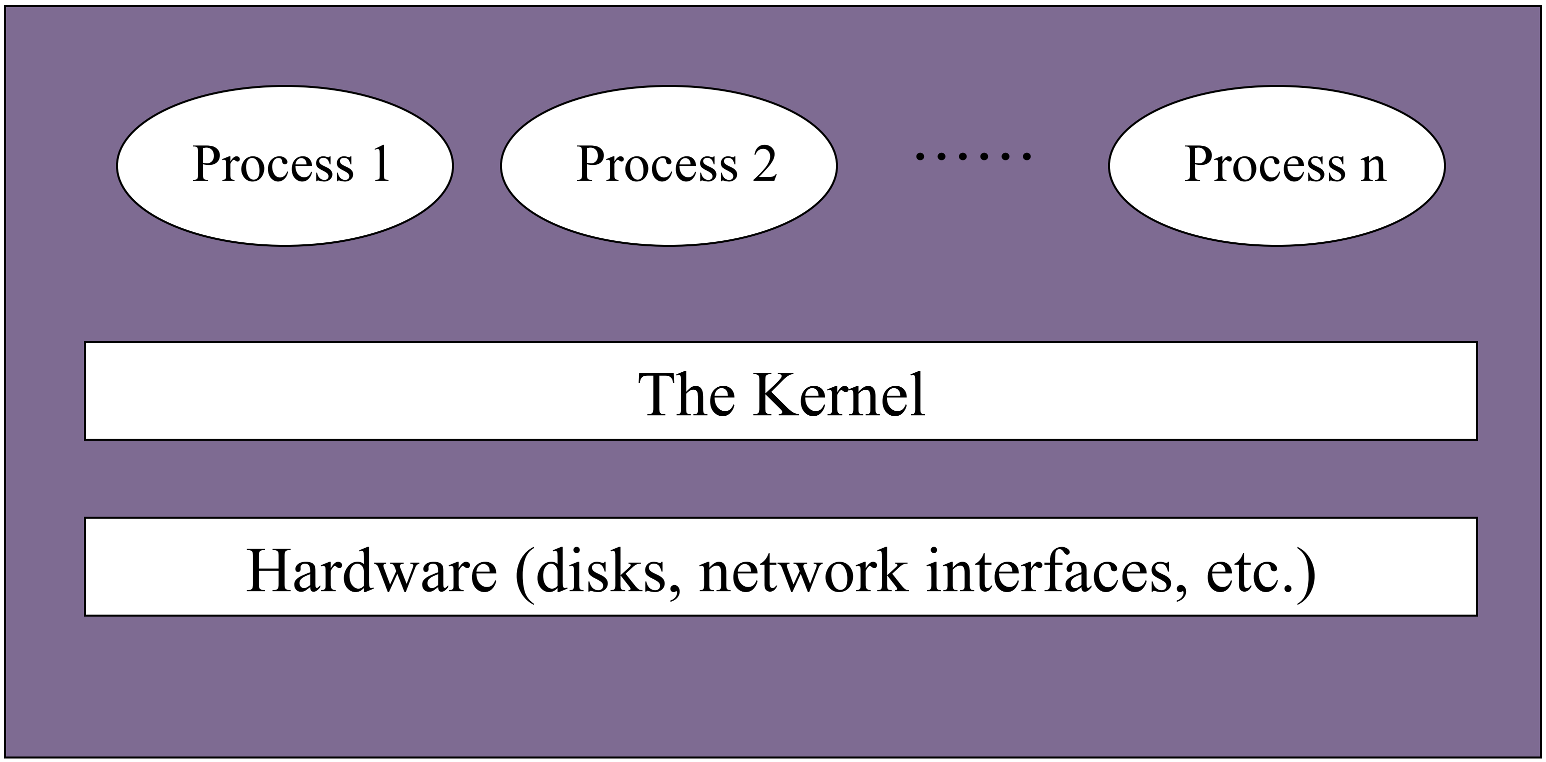
- Guarded gates from user mode (space, land) into kernel mode (space, land)
  - use a special CPU instruction (often an interruption), transfers control to predefined entry point in more privileged code; allows the more privileged code to specify where it will be entered as well as important processor state at the time of entry
  - the higher privileged code, by examining processor state set by the less privileged code and/or its stack, determines what is being requested and whether to allow it

http://en.wikipedia.org/wiki/System_call

# Kernel space vs User space

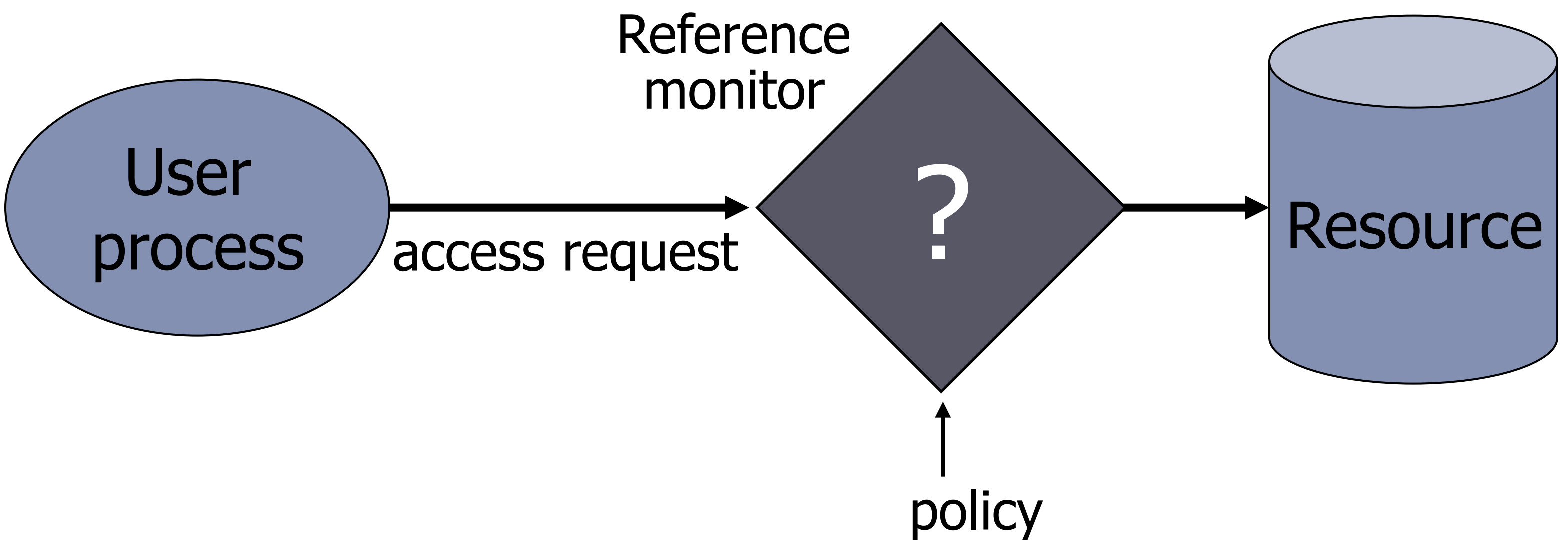
- Part of the OS runs in the kernel model
  - known as the OS kernel
- Other parts of the OS run in the user mode, including service programs (daemon programs), user applications, etc.
  - they run as processes
  - they form the user space (or the user land)
- What is the difference between kernel mode and processes running as root (or superuser, administrator)?

# Kernel Space vs. User Space

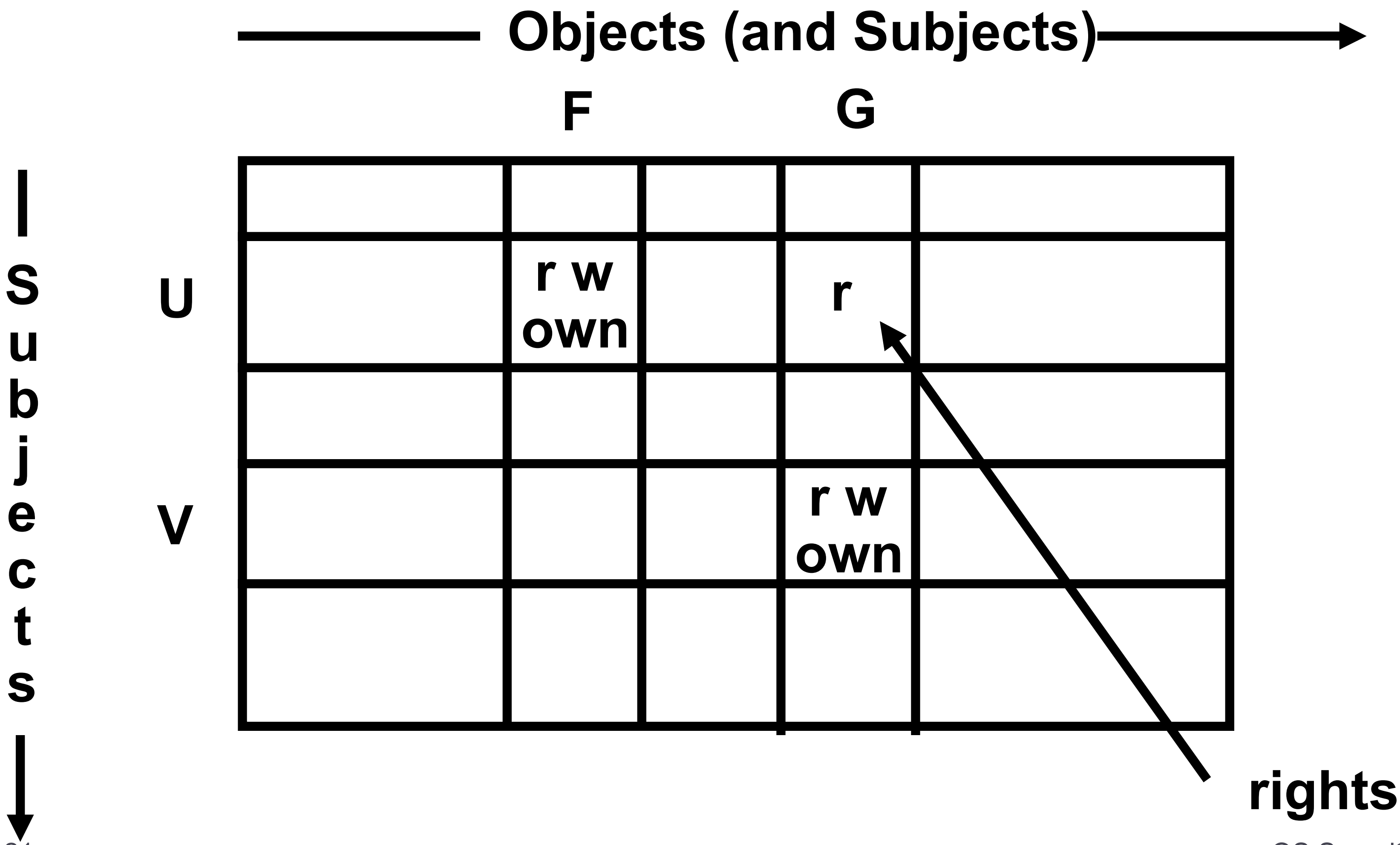


# Access control

- A reference monitor mediates all access to resources
  - Principle: Complete mediation: control **all** accesses to resources



# Access Control Matrix



# Access Matrix Model

- Basic Abstractions
  - Subjects
  - Objects
  - Rights
- The rights in a cell specify the access of the subject (row) to the object (column)

# Principals and Subjects

- A subject is a program (application) executing on behalf of some principal(s)
- A principal may at any time be idle, or have one or more subjects executing on its behalf

**What are subjects in UNIX?**

**What are principals in UNIX?**

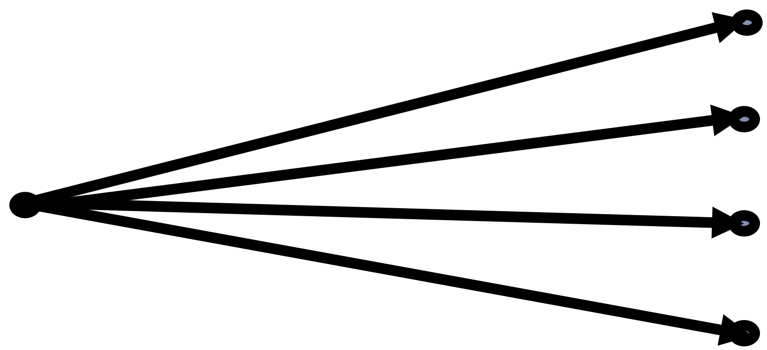
# Objects

- An object is anything on which a subject can perform operations (mediated by rights)
- Usually objects are passive; examples:
  - File
  - Directory (or Folder)
  - Memory segment
- Subjects (i.e. processes) can also be objects, with operations performed on them
  - kill, suspend, resume, send interprocess communication, etc.

# UNIX Access Control

- Users, Groups, Files, Processes
- Each user account has a unique UID
  - The UID 0 means the super user (system admin)
- A user account belongs to multiple groups
- Subjects are processes
  - associated with uid/gid pairs, e.g., (euid, egid), (ruid, rgid), (suid, sgid)
- Objects are files

# Users vs. Principals



**USERS**

**PRINCIPALS**

**Real World User**

**Unit of Access Control and Authorization**

the system authenticates the human user to a particular principal

# Users and Principals

- There should be a one-to-many mapping from users to principals
  - a user may have many principals
  - each principal is associated with an unique user
- This ensures accountability of a user's actions

# Organization of Objects

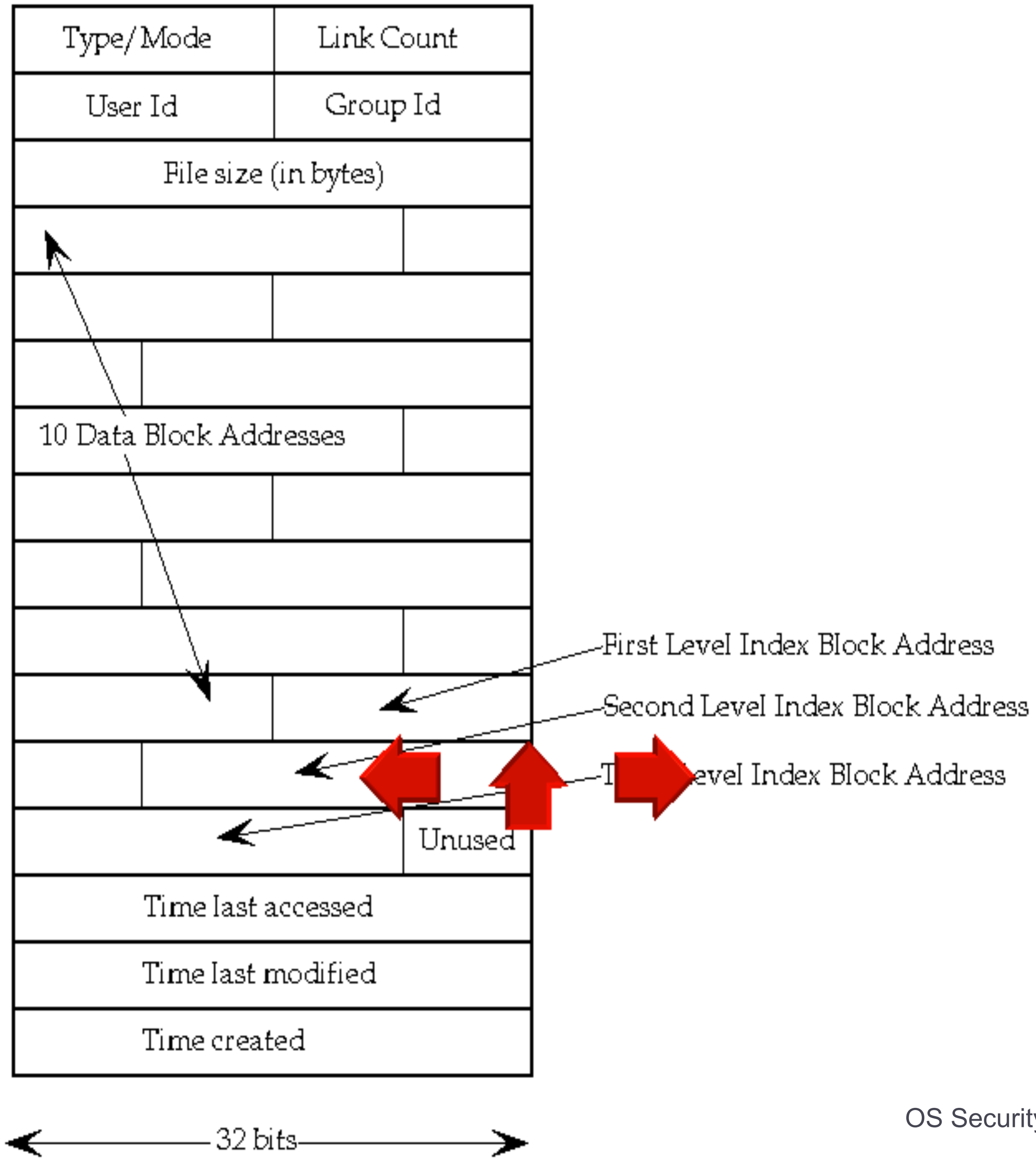
- Almost all objects are modeled as files
  - Files are arranged in a hierarchy
  - Files exist in directories
  - Directories are also one kind of files
- Each object has
  - owner
  - group
  - 12 permission bits
    - rwx for owner, rwx for group, and rwx for others
    - suid, sgid, sticky

# inode

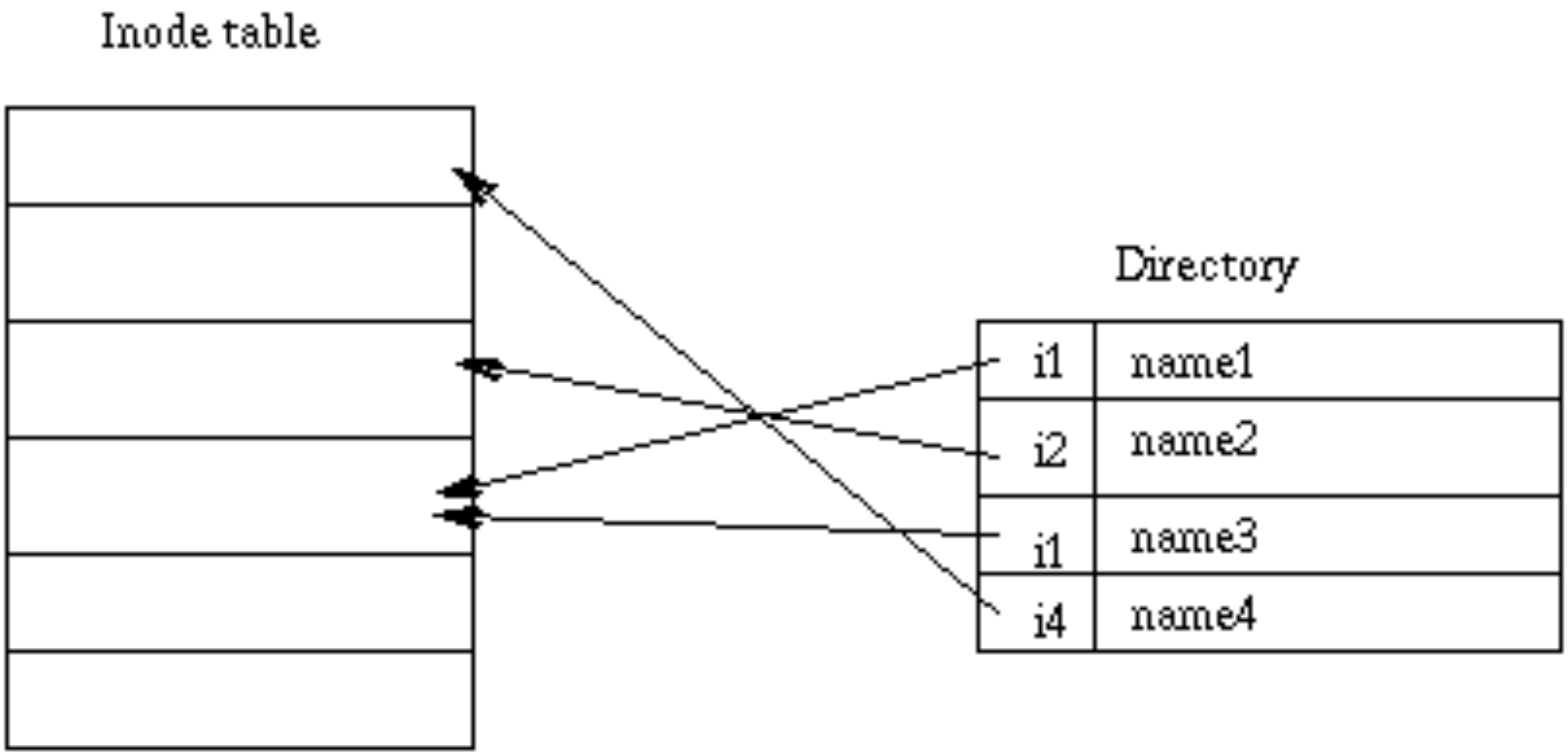
- Data structure:
  - represents a filesystem object (file or a directory)
- Stores:
  - Filesystem attributes:
    - manipulation metadata (e.g. creation, access, modify time), as well as <u>owner</u>
    - <u>permission data (e.g. group-id, user-id, permissions)</u>
  - Disk block location(s) of the filesystem object's data
  - Filesystem object attributes may include

# UNIX inodes:

Each file corresponds to an inode



# Unix Directories



# Basic Permissions Bits on Files

- Read controls reading the content of a file
  - i.e., the read system call
- Write controls changing the content of a file
  - i.e., the write system call
- Execute controls loading the file in memory and execute
  - i.e., the execve system call

# Execution of a file: binary vs script

- Having execute but not read, can one run a binary file?
- Having execute but not read, can one run a script file?
- Having read but not execute, can one run a script file?

# Permission Bits on Directories

- Read bit allows one to show file names in a directory
- The execution bit controls traversing a directory
  - does a lookup, allows one to find inode # from file name
  - chdir to a directory requires execution
- Write + execution control creating/deleting files in the directory
  - Deleting a file under a directory requires no permission on the file
- Accessing a file identified by a path name requires execution to all directories along the path

# Ruid and rgid

- **Real user ID (ruid)** and **real GID (rgid)** identify the real owner of the process
  - Affect the permissions for sending signals. A process without superuser privilege can signal another process only if the sender's ruid or euid matches the ruid or suid of the receiver. Since child processes inherit the credentials from the parent, they can signal each other.

# Euid and egid

- OS kernel uses the euid and egid of the process to determine if it can access the file
- **Effective user ID (euid)** of a process is the ownership assigned to files created by that process
- **Effective GID (egid)** of a process may also affect file creation, depending on the semantics of the specific kernel implementation being used and possibly also by the mount options used
  - In Unix newly created files will get assigned the group ownership of the egid of the process that creates them

# Suid

- Saved user ID (suid) of a process  is used when a program running with elevated privileges needs to temporarily do some unprivileged work
- It changes its euid from a privileged value (typically root) to some unprivileged one, and this triggers a copy of the privileged user ID to the suid.
- Later, it can set its euid back to the suid (an unprivileged process can only set its euid to three values: its ruid , its suid, and its euid—i.e., unchanged) to resume its privileges.

# The suid, sgid, sticky bits

| | suid | sgid | sticky bit |
|---|---|---|---|
| non-executable files | no effect | affect locking (unimportant for us) | not used anymore |
| executable files | change euid when executing the file | change egid when executing the file | not used anymore |
| directories | no effect | new files inherit group of the directory | only the owner of a file can delete |

# Some Examples

- What permissions are needed to access a file/ directory?
  - read a file: /d1/d2/f3
  - write a file: /d1/d2/f3
  - delete a file: /d1/d2/f3
  - rename a file: from /d1/d2/f3 to /d1/d2/f4
  - …
- File/Directory Access Control is by System Calls
  - e.g., open(2), stat(2), read(2), write(2), chmod(2), opendir(2), readdir(2), readlink(2), chdir(2), …

# The Three Sets of Permission Bits

- Intuition:
  - if the user is the owner of a file, then the r/w/x bits for owner apply
  - otherwise, if the user belongs to the group the file belongs to, then the r/w/x bits for group apply
  - otherwise, the r/w/x bits for others apply

- Can one implement negative authorization, i.e., only members of a particular group are not allowed to access a file?

# Other Issues On Objects in UNIX

- Accesses other than read/write/execute
  - Who can change the permission bits?
    - The owner can
  - Who can change the owner?
    - Only the superuser
- Rights not related to a file
  - Affecting another process
  - Operations such as shutting down the system, mounting a new file system, listening on a low port
    - traditionally reserved for the root user

# Subjects vs. Principals

- Access rights are specified for users (accounts)
- Accesses are performed by processes (subjects)
- The OS needs to know on which users' behalf a process is executing

# Process User ID Model in Modern UNIX Systems

- Each process has three user IDs
  - real user ID (ruid) — owner of the process
  - effective user ID (euid) — used in most access control decisions
  - saved user ID (suid)
- and three group IDs
  - real group ID
  - effective group ID
  - saved group ID

# Process User ID Model in Modern UNIX Systems

- When a process is created by *fork*
  - it inherits all three users IDs from its parent process
- When a process executes a file by *exec*
  - it keeps its three user IDs unless the set-user-ID bit of the file is set, in which case the effective uid and saved uid are assigned the user ID of the owner of the file
- A process may change the user ids via system calls

# The Need for suid/sgid Bits

- Some operations are not modeled as files and require user id = 0
  - halting the system
  - bind/listen on "privileged ports" (TCP/UDP ports below 1024)
  - non-root users need these privileges
- File level access control is not fine-grained enough
- System integrity requires more than controlling who can write, but also how it is written

# Security Problems of Programs with suid/sgid

- These programs are typically setuid root
- Violates the least privilege principle
  - every program and every user should operate using the least privilege necessary to complete the job
- Why violating least privilege is bad?
- How would an attacker exploit this problem?
- How to solve this problem?

# Changing Effective user IDs

- A process that executes a set-uid program can drop its privilege; it can
  - drop privilege permanently
    - removes the privileged user id from all three user IDs
  - drop privilege temporarily
    - removes the privileged user ID from its effective uid but stores it in its saved uid, later the process may restore privilege by restoring privileged user ID in its effective uid

# What Happens during Logging in

| login | |
|---|---|
| pid | 2235 |
| euid | 0 |
| ruid | 0 |
| suid | 0 |

setuid(500) →

| login | |
|---|---|
| pid | 2235 |
| euid | 500 |
| ruid | 500 |
| suid | 500 |

exec(“bash”) →

| bash | |
|---|---|
| pid | 2235 |
| euid | 500 |
| ruid | 500 |
| suid | 500 |

fork() →

After the login process verifies that the entered password is correct, it issues a setuid system call.

The login process then loads the shell, giving the user a login shell.

The user types in the passwd command to change his password.

| bash | |
|---|---|
| pid | 2235 |
| euid | 500 |
| ruid | 500 |
| suid | 500 |

| bash | |
|---|---|
| pid | 2297 |
| euid | 500 |
| ruid | 500 |
| suid | 500 |

exec("passwd")

| passwd | |
|---|---|
| pid | 2297 |
| euid | 0 |
| ruid | 500 |
| suid | 0 |

Drop privilege permanently

| passwd | |
|---|---|
| pid | 2297 |
| euid | 500 |
| ruid | 500 |
| suid | 500 |

Drop privilege temporarily

| passwd | |
|---|---|
| pid | 2297 |
| euid | 500 |
| ruid | 500 |
| suid | 0 |

The fork call creates a new process, which loads "passwd", which is owned by root user, and has setuid bit set.

# Access Control in Early UNIX

- A process has two user IDs: real uid and effective uid and one system call setuid
- The system call setuid(id)
  - when euid is 0, setuid set both the ruid and the euid to the parameter
  - otherwise, the setuid could only set effective uid to real uid
    - Permanently drops privileges
- A process cannot temporarily drop privilege

Setuid Demystified, In USENIX Security ' 02

# System V

- To enable temporarily drop privilege, added saved uid & a new system call
- The system call seteuid
  - if euid is 0, seteuid could set euid to any user ID
  - otherwise, could set euid to ruid or suid
    - Setting euid to ruid temp. drops privilege
- The system call setuid is also changed
  - if euid is 0, setuid functions as seteuid
  - otherwise, setuid sets all three user IDs to real uid

# BSD

- Uses ruid & euid, change the system call from setuid to setreuid
  - if euid is 0, then the ruid and euid could be set to any user ID
  - otherwise, either the ruid or the euid could be set to value of the other one
    - enables a process to swap ruid & euid

# Modern UNIX

- System V & BSD affect each other, both implemented setuid, seteuid, setreuid, with different semantics
  - some modern UNIX introduced setresuid

- Things get messy, complicated, inconsistent, and buggy
  - POSIX standard, Solaris, FreeBSD, Linux

# Suggested Improved API

- Three method calls
  - drop_priv_temp
  - drop_priv_perm
  - restore_priv
- Lessons from this?
  - “Mechanism, not policy” not necessarily a good idea for security (flexibility not always a good thing)
  - Psychological acceptability principle
    - “human interface should be designed for ease of use”
    - the user’s mental image of his protection goals should match the mechanism

# Take home lessons

- Access control matrix model used to specify what right do Subjects have on Objects (or other files)
- CPU Mode vs User mode defines different access rights
- Unix access control is based on the concepts of users, files, groups and processes