File Systems

Access Control

- Typical protection modes
  - *read*, *write*, *append, execute, delete*
  - *privilege* (allows modification of others' rights)
  - *owner* (indicates owner of file
  - *search* (grants permission to search directory).
- Interpretations may depend on type of file/directory/etc.
  - File: execute bit set means file can be executed
  - Directory: execute bit set means directory can be searched

Access Control Lists

- Columns of access control matrix

	file1	*file2*	*file3*
Andy	rx	r	rwo
Betty	rwxo	r	
Charlie	rx	rwo	w

ACLs:

- file1: { (Andy, rx) (Betty, rwxo) (Charlie, rx) }
- file2: { (Andy, r) (Betty, r) (Charlie, rwo) }
- file3: { (Andy, rwo) (Charlie, w) }

Abbreviations

- UNIX shortens list by combining rights
  - 3 classes of users: owner, group, rest
  - rwx rwx rwx

    rest

    group

    owner
  - Ownership assigned based on creating process
    - Most UNIX-like systems: if directory has setgid permission, file group owned by group of directory (Solaris, Linux)

Example: Cisco Router

- Dynamic access control lists

```
access-list 100 permit tcp any host 10.1.1.1 eq telnet
access-list 100 dynamic test timeout 180 permit ip any host 10.1.2.3 time-
  range my-time
time-range my-time
  periodic weekdays 9:00 to 17:00
line vty 0 2
  login local
  autocommand access-enable host timeout 10
```

- Limits external access to 10.1.2.3 to 9AM–5PM
  - Adds temporary entry for connecting host once user supplies name, password to router
  - Connections good for 180 minutes
    - Drops access control entry after that

Conflicts

- Deny access if any entry would deny access
  - AIX: if any entry denies access, *regardless or rights given so far*, access is denied
- Apply first entry matching subject
  - Cisco routers: run packet through access control rules (ACL entries) in order; on a match, stop, and forward the packet; if no matches, deny
    - Note default is deny so honors principle of fail-safe defaults

Capability Lists

- Columns of access control matrix

	file1	*file2*	*file3*
Andy	rx	r	rwo
Betty	rwxo	r	
Charlie	rx	rwo	w

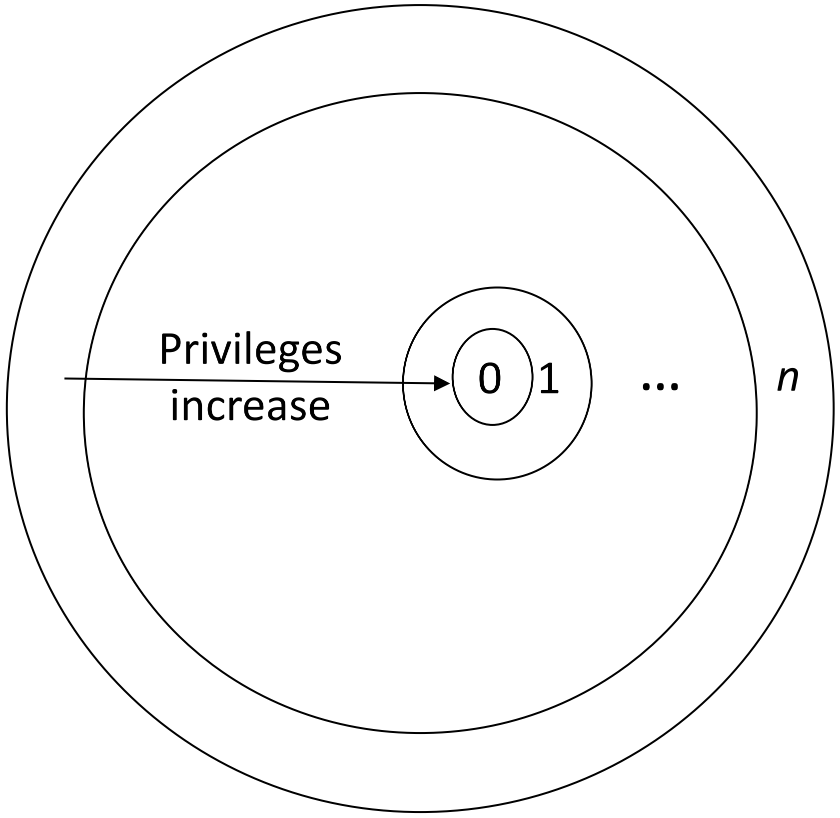
C-Lists:

- Andy: { (file1, rx) (file2, r) (file3, rwo) }
- Betty: { (file1, rwxo) (file2, r) }
- Charlie: { (file1, rx) (file2, rwo) (file3, w) }

Semantics

- Like a bus ticket
  - Mere possession indicates rights that subject has over object
  - Object identified by capability (as part of the token)
    - Name may be a reference, location, or something else
  - Architectural construct in capability-based addressing; this just focuses on protection aspects
- Must prevent process from altering capabilities
  - Otherwise subject could change rights encoded in capability or object to which they refer

Ring-Based Access Control



- Process (segment) accesses another segment
    - read
    - execute
- *Gate* is an entry point for calling segment
- Rights:
    - *r* read
    - *w* write
    - *a* append
    - *e* execute

Reading/Writing/Appending

- Procedure executing in ring r
- Data segment with *access bracket* (a_1, a_2)
- Mandatory access rule
  - $r \le a_1$ allow access
  - $a_1 < r \le a_2$ allow *r* access; not *w*, *a* access
  - $a_2 < r$ deny all access

Executing

- Procedure executing in ring *r*
- Call procedure in segment with *access bracket* (a_1, a_2) and *call bracket* (a_2, a_3)
  - Often written (a_1, a_2, a_3)
- Mandatory access rule
  - $r < a_1$ allow access; ring-crossing fault
  - $a_1 \le r \le a_2$ allow access; no ring-crossing fault
  - $a_2 < r \le a_3$ allow access if through valid gate
  - $a_3 < r$ deny all access

Versions

- Multics
  - 8 rings (from 0 to 7)
- Intel's Itanium chip
  - 4 levels of privilege: 0 the highest, 3 the lowest
- Older systems
  - 2 levels of privilege: user, supervisor

Linux Capabilities

- In Linux, used to override or add access restrictions by adding, masking rights
  - Not capabilities as no particular object associated with the (added or deleted) rights
- 3 sets of privileges
  - Bounding set (all privileges process may assert)
  - Effective set (current privileges process may assert)
  - Saved set (rights saved for future purpose)
- Example: UNIX effective, saved UID

Processes and Files

- Processes operate on files using the following commands:
- *create*: find space for file, allocate it, make an entry in directory
- *open*: begin operations on file
- *close*: end operations on file
- *read*: transfer information from file
- *write*: transfer information to file
- *rewind*: move to the beginning (or a random point) in file
- *delete*: remove file

How Processes Access Files

- Sequential
- Direct, random
- Mapped
- Structured

Sequential Access

- Access one block after the other
- Process keeps track of location using a *read/write pointer* (part of the PCB) indicating where the next action is to be done
- Pointer always advances.

Direct Access, Random Access

- Like sequential, except read/write pointer can move freely

Mapped Access

- Map the file into a virtual segment
- Return the segment number rather than the file descriptor
- Then treat the file as part of the process’ virtual store.
- On closing, just release the storage.
- Examples: TOPS-20, Multics, some versions of Linux and UNIX

Structured Access

- File consists of a sequence of records
  - Sometimes the operating system knows about the file type.
- Example: ISAM (Indexed Sequential Access Method)
  - Small master index points to blocks in secondary index, which in turn point to real file blocks.
  - Takes at most 2 reads to locate any record

Disk Directory

- Like a directory for a disk
- Describes what blocks are in use and which are free.
  - Must keep track of what blocks are not in use; such a list is a *free list*
- Several representations of free list:
  - Bit map, with 1 bit per block
  - Linked list of blocks
  - Like linked list, but in each block of size *n* on free list, store *n*-1 numbers of free blocks; the *n*-th is the address of the next block making up the list
  - Pairs of (block number, number of free blocks from that block on); if there is more than one contiguous block free, this usually saves same space
- Last 3 are sometimes called *file maps*

Allocating Disk Blocks to Files

- Contiguous allocation
- Linked allocation
- Indexed allocation

Contiguous Allocation of Blocks

- Blocks are allocated sequentially (contiguously)
- Advantages:
  - Minimal head motion for sequential reading of file
- Disadvantages:
  - Need to find space for it
    - Use usual algorithms (first fit, best fit, etc.)
    - Can use compaction but this usually requires copying almost everything on disk
  - How much space should be allocated? File may grow beyond its initial allocation (and even if you allocate the maximum space, that’s wasteful)
    - May be room to increase allocation
    - Process may terminate, causing users to ask for more space than needed (wasteful)
    - May move file elsewhere (very slow)

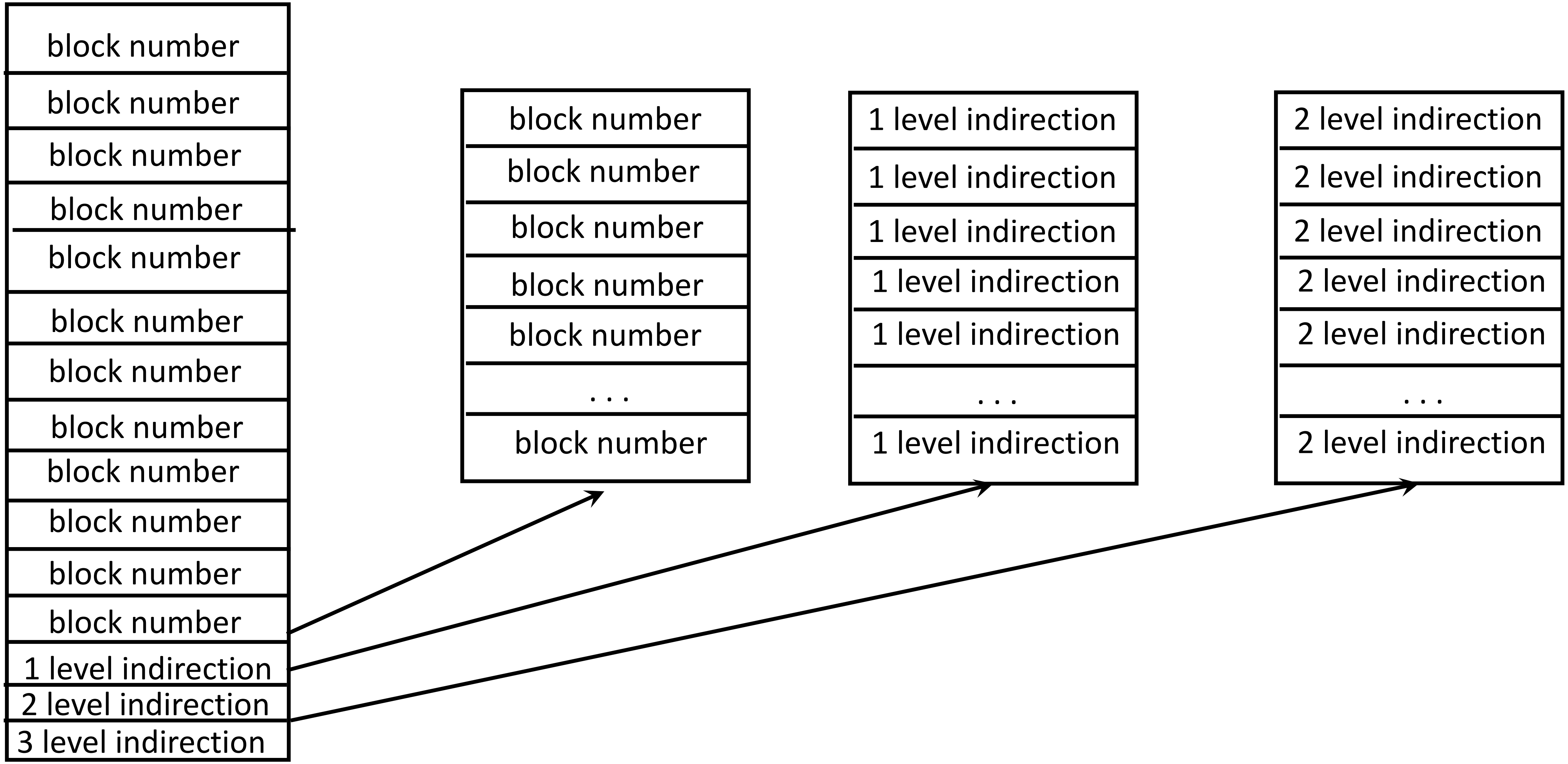
Linked Allocation

- Directory contains pointers to first, last blocks of file
- Last *n* bytes of each block point to next block
- Advantages
  - No need to know size of file in advance
  - Good for files accessed sequentially
- Disadvantages
  - Poor for random access as operating system must follow links to get to desired block
  - Wastes *n* bytes per disk block
  - Unreliable; if 1 pointer gets scrambled or deleted, file is garbled or lost
    - Doubly linked list might help but uses more memory

Indexed Allocation

- Put all pointers into one block
- Advantages:
  - Compact; easy to reference blocks
- Disadvantages:
  - Waste space as an entire block is pointers rather than just 1 word per block
  - So a 2 block file and a 511 block file use a single block to store pointers
- Implementation issue: if more than 1 block needed for pointers, link them together or use indirection
  - If 1024 pointers/block, then 2 levels of indirection allows $1024^2 = 1048576$ blocks

Example: UNIX/Linux



Example: UNIX/Linux

- Room for:
  - 12 (main block)
  - 1024 (first indirect block)
  - $1024^2 = 1048576$ (second indirect block)
  - $1024^3 = 1073741824$ (triply indirect block)
- So total space this can cover:
  - $12 + 1024 + 1024^2 + 1024^3 = 1,074,791,436$ blocks

Networked File Server

- System must know where file is kept and be able to communicate with file server
- *Centralized file server*: system determines location using a table showing where it is
  - Network File System (NFS) works this way
- *Distributed file data*: system accesses file containing information about location, and uses that to get contents of file
  - BitTorrent works this way

Example: NFS Protocol

- NFS: Network File System
  - Developed by Sun Microsystems in late 1980s; RFC 1094 (March 1989)
  - Current version is NFS v4.2, RFC 7862 (Nov. 2016)
- Kernel sees it as just another file until you reach the *mount point*
  - At that point, kernel acts as client to (remote) NFS server

Mounting Remote File System

- Kernel. server exchange messages to make file system available to client (kernel)
- Access modes controlled by various configuration files
- Common mounting options:
  - *soft*: file system calls that fail after a certain number of retries return failure rather than continuing to try
  - *rdonly*: mount file system read-only
  - *nodev*: ignore any device files on NFS file system
  - *nosuid*: ignore any setuid bits

Opening a File

- Given file name, handle it as usual until you reach the mount point of the NFS file system
- System then uses *file handles* identifying remote files to find right file
  - File handles are all that is needed for access
  - Fine handles include generation number to detect conflicts
  - *Every* file access uses this handle

Security

Security Basic Components

- Confidentiality
  - Keeping data and resources hidden
- Integrity
  - Data integrity (integrity)
  - Origin integrity (authentication)
- Availability
  - Allowing access to data and resources

Policies and Mechanisms

- Policy says what is, and is not, allowed
  - This defines "security" for the site/system/*etc*.
- Mechanisms enforce policies
- Composition of policies
  - If policies conflict, discrepancies may create security vulnerabilities

Goals of Security

- Prevention
  - Prevent attackers from violating security policy
- Detection
  - Detect attackers violating security policy
- Recovery
  - Stop attack, assess and repair damage
  - Continue to function correctly even if attack succeeds

Assumptions and Trust

- Underlie *all* aspects of security
- Policies
  - Unambiguously partition system states
  - Correctly capture security requirements
- Mechanisms
  - Assumed to enforce policy
  - Support mechanisms work correctly

Requirements

- Trusted Computer Security Evaluation Criteria (TCSEC)
  - And its derivatives, the “Rainbow Series”
- FIPS 140
  - For cryptographic implementations
- Common Criteria
  - For systems that match protection profiles
- System Security Engineering Capability Maturity Model (SSE-CMM)
  - For processes used to develop systems
- GDPR and CCPA
  - Laws in the EU and California that govern privacy

Design Principles

- Least privilege
  - Process should be given only those privileges necessary to complete its task
- Fail-safe defaults
  - Default is to deny permission
  - If action fails, system stays as secure as when action began
- Economy of mechanism
  - Keep things as simple as possible (KISS principle)
- Complete mediation
  - Check permissions on every access

Design Principles

- Open design
  - Security should not depend on secrecy of design or implementation
- Separation of privilege
  - Require multiple conditions to hold in order to grant privilege
- Least common mechanism
  - Minimize sharing of resources
- Least astonishment
  - Security mechanisms should be designed so users understand why the mechanism works the way it does, and using mechanism is simple
  - Earlier version: principle of psychological acceptability, which says security mechanisms should not add to difficulty of accessing resource