V22.0202-001
Computer Systems Organization II (Honors)
(Introductory Operating Systems)

Lecture 21
Secondary Storage (cont’d), Protection

April 26, 2004
Outline

• Announcements
  – Send updates on Lab 5 (and earlier labs) to TA as soon as possible
  – Lab 6 demos on May 3rd, 4th
  – Questions?

• Secondary storage structure (cont’d)
  – disk scheduling algorithms (cont’d)
  – RAID structure

• Protection
  – Domain of protection
  – Access matrix
  – Access rights and capabilities
  – Language-based protection

[Silberschatz/Galvin/Gagne: Sections 14.2—14.5, Chapter 18]
(Review) Disk Scheduling: Scan Scheduling

- The “elevator” algorithm
  - Starting from one end, process requests in order of increasing track number
  - when no more requests are available, reverse direction
- Performance improvement over SSTF: no starvation
  - favors middle tracks (encountered twice) over end tracks

Request sequence:
98, 183, 37, 122, 14, 124, 65, 67

Head at: cylinder 53

Head movement of 208 cylinders
Disk Scheduling: Circular Scan

- SCAN avoids starvation, but provides non-uniform wait times
  - at each direction reversal, looks at the tracks recently visited
- C-SCAN scheduling (circular scan) scans in only one direction
  - at end of scan, return to other end immediately

Request sequence:
98, 183, 37, 122, 14, 124, 65, 67

Head at: cylinder 53
Disk Scheduling: LOOK and C-LOOK

- Variants of SCAN and C-SCAN where the arm goes only as far as the final request in each direction
  - Saves on redundant movement

Request sequence:
98, 183, 37, 122, 14, 124, 65, 67

Head at: cylinder 53
Choosing between Disk Scheduling Schemes

Issues

- The usual tradeoff between
  - the amount of work done by the scheme
  - and the quality of the scheduling
- File allocation strategy
  - e.g. FCFS is OK with contiguous, but not linked allocation
- Relative locations of files and directories
  - some schemes attempt to put these close together

- Disk scheduling typically implemented in the disk controller
  - very sensitive to the implementation (of the disk)
  - different tradeoffs between seek times and rotational latency
  - SSTF and LOOK are typical choices
Using Multiple Disks

• Store data across multiple disks, each with its own physical channel
  – mirroring improves reliability
  – striping, where contiguous "sub-blocks" are stored on different disks, improves transfer time
    • To access a block, all of its sub-blocks are accessed in parallel
    • Does not change seek time or latency

RAID: Redundant Array of Inexpensive Disks
• The elementary approach (only mirroring): RAID Level 1
  – maintains a copy of each disk: wasteful and expensive

• A somewhat less expensive approach RAID Level 3
  – given k disks in the array …
  – use the first (k-1) for storage: data striped across disks at byte level
  – use the last disk for parity
  – if any one disk fails, its contents can be recomputed
Block-Interleaved Parity (RAID Level 4)

- Parity for a set of bits is 1 if number of 1 bits is even

- Basic scheme
  - for n blocks b[1] ... b[n]
  - store parity block (per-bit) on parity disk
Rotating Block-Interleaved Parity (RAID Level 5)

- RAID Level 4 takes a performance hit on small, random requests
  - All requests access the parity drive, which becomes a bottleneck

- Solution: Distribute the parity information
  - No loss in reliability
  - Good performance properties
Outline

• Announcements
  – Send updates on Lab 5 (and earlier labs) to TA as soon as possible
  – Lab 6 demos on May 3rd, 4th
  – Questions?

• Secondary storage structure (cont’d)
  – disk scheduling algorithms (cont’d)
  – RAID structure

• Protection
  – Domain of protection
  – Access matrix
  – Access rights and capabilities
  – Language-based protection

[ Silberschatz/Galvin/Gagne: Sections 14.2—14.5, Chapter 18 ]
Protection

• The overall need
  – initially driven by the explosion in multiprogramming
    • recently by the explosion in networking
  – needed multiple co-existent processes not to encroach/interfere
    • accidentally or maliciously ...
    • into unwanted regions

Solution:
• Define specific domains that specify the privileges of a
  user/process/procedure to access system resources
  – For each resource, what are the operations that the process can do
    • Resources: files, memory segments/pages, printer, …
• Provide mechanisms for enforcing these privileges
  – Hardware: Protection faults, privileged instructions, exceptions
  – Software: System calls
Domain of Protection

• Components
  – Logical processes
  – Classes of logical objects that processes can access in various ways
    • e.g. files, directories, devices
  – Each object class has a set of rights
    • e.g. read, write, delete

• A process at any given time
  – is in a particular domain
  – inherits all of its privileges
    • Domains are the granularity at which privileges are specified
What is a Domain?

- Formally, a domain is a set of **access rights**
  - an access right is a pair: <object, rights>
    - Example could be <payroll-file, read-only>

- A process can only be in one particular domain at any given time
  - Association can be …
  - … **static**: need OS mechanisms to modify access rights
  - or **dynamic**: need OS mechanisms to switch domains

- Domains can
  - contain each other
  - be non-overlapping
  - change over time, i.e. can be modified
Examples

• Dual-mode model of operating system execution
  – Two domains: system (monitor) and user
  – System domain has complete rights
  – User domain cannot execute privileged instructions

• Unix
  – Three kinds of domains: users, groups, “other”
  – Access rights are specified for each such domain
    • E.g., chmod operations for setting file access rights
  – Switching the domain corresponds to changing the user identification or group identification temporarily
Access Matrices

• Used to describe domains
  – Each row is a domain
  – Each column is an object
  – Entry \((i, j)\) indicates the privileges associated with domain \(i\) to access object \(j\)

• Example

<table>
<thead>
<tr>
<th></th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>printer</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>r</td>
<td></td>
<td>r</td>
<td></td>
</tr>
<tr>
<td>D2</td>
<td></td>
<td></td>
<td>pr</td>
<td></td>
</tr>
<tr>
<td>D3</td>
<td></td>
<td>r</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>D4</td>
<td>r</td>
<td></td>
<td>rw</td>
<td></td>
</tr>
</tbody>
</table>
Mechanism vs. Policy

- The OS supports means of specifying and enforcing preferred access patterns via the access matrix and hence supports the mechanism.
- The actual decisions are entries in the matrix and denote policy.
- Typically, the owner of an object determines the corresponding privileges and hence entries in the appropriate column.
Implementing Access Matrices

• Implementing the whole matrix
  – Allows very fast direct access
  – Extremely space **inefficient**
    • most objects are accessed by only a few domains
    • matrices are very sparse

• By columns
  – Each object has an **access list**
    • Which domains have what privileges
  – Advantages
    • Easily extensible
    • e.g. to add new domains that can access a given object
  – Used in condensed form in Unix
Implementing Access Matrices (contd.)

• By rows
  – Each domain has a capability list
    • What privileges does this domain have for each kind of object
  – Easily accessed: indexed by "object"

• These lists themselves
  – can be viewed as special objects
    • Typically, only an OS-state induced domain can change things in them
  – can also be supported in hardware and tagged as such, to distinguish it
  – segmentation is a good way of realizing this
Lock-Key Mechanisms

• Compromise between access lists and capability lists
  – Each object has a unique list of bit patterns called locks
  – Domain has a list of bit patterns called keys
  – To access an object, the domain must have a key that matches one of its locks

• Example: Unix
  – Domain is decided by effective UID, GID
    • setuid bit permits switching of domains (can become root if not careful)
  – Objects are files or directories
  – lock is the 9-bit access code
    • 3 bits for each of three domains: owner, group, other
    • Rights: read, write, execute (interpreted as search, create, use for directories)
      – E.g., rw+xrw-r--
  – key is effective UID and effective GID
    • Normalized with respect to the owner and group of the file/directory
Access Matrix Implementations In Practice

• Most systems use a mixture of
  – access lists
  – capability lists

• For example, in Unix, we can view
  – File permissions as access lists
  – File handles (FIDs) as capabilities
  – OS calls such as `open`
    • Check that the process has the appropriate permissions
      – Enforced by the system call mechanism
      – Requires comparing against the permissions associated with the file
    • Return a capability, the file handle
    • This capability can be used in subsequent system calls to avoid repeated protection checks
Another Example: MULTICS

- Protection domains are hierarchically arranged and are organized such that $D_j$ is a subset of $D_i$ whenever $i > j$
  - These domains are called **rings**
  - $D_0$ has the most privileges

- MULTICS has a segmented address space
  - Each segment is associated with one of the rings and a set of permissions
  - A process in ring $D_i$ can only access segments in rings $D_j$ ($j \geq i$)

- Processes switch domains to change access privileges
  - Each segment also has associated with it
    - An **access bracket** $(b_1, b_2)$: Call is allowed if $b_1 \leq i \leq b_2$, else traps
      - A trapped call is permitted to proceed if $i \leq b_1$
    - A **limit** $(b_3)$ and a **list of gates** defining valid entry points
      - Trapped calls where $i \geq b_2$ are allowed only to a gate and if $i \leq b_3$
MULTICS Protection Scheme (cont’d)

• Significant weaknesses
  – As a process moves from an “outer" ring to an “inner" one
    • it strictly gains new privileges while maintaining all of its old
      • contravenes the need to know principle
      • an important rule in designing protection policies
    – Cannot create partially overlapping domains
  – Also, relatively heavyweight
Access Matrices: More General Operations

- In general, also want some way of
  - Dynamically changing domains
    - Treat domains also as objects
    - Rights associated with this object indicate whether or not a switch is allowed
  - Increasing privileges (capabilities) and passing them on
  - See Section 18.3 for details

- **Revocation** of access rights
  - Several dimensions: immediate vs. delayed, selective (w.r.t. domain members), partial (w.r.t. rights on object), temporary vs. permanent
  - **Access Lists**: Just scan the list and modify
    - Does not work as well when there are too many objects
  - **Capabilities**: Harder problem, since they are distributed to members
    - Require periodic reacquisition of capabilities
    - Capabilities point to actual object via an indirect “object table”
Language-based Protection: Rationale

• Protection solutions discussed so far require involvement of OS kernel
  – To validate access rights or capabilities at run time
  – Tend to be high overhead
  – Are inflexible in the objects and operations being protected
    • System-defined functions: e.g., file operations
  – Require hardware support for efficiency
    • E.g., virtual-memory page protection
    • Not generally applicable

• Current-day OSes need additional flexibility
  – To protect arbitrary (user-defined) objects and operations
    • E.g., a server might want to restrict access to certain services
  – Need these checks to be performed efficiently

• Solutions involve language features and their trusted implementation
Language-based Protection: Solution

- Shared resources/services defined as objects
  - Abstract data types
  - Protection enforced on access to an object’s methods

- Goal of protection: Ensure that only allowed methods are invoked
  - Definition of allowed depends on context
    - More generally, the software module that is doing the call

- Object-oriented languages already provide such support …
- … for enforcing abstraction boundaries
  - E.g., private variables and methods are not accessible outside their class
  - Enforced statically by the compiler (e.g., Java)
  - Sometimes, also requires run-time support (e.g., C++)

- General problem is known as type safety
  - Languages where type-based access is enforceable: safe languages
Language-based Protection: Two Issues

• “Memory” protection
  – A program will not access memory or execute code for which it is not authorized
  – Builds on top of language type safety

• Secure system services
  – A program will not access unauthorized system services (files, graphics, …)
  – Type safety is not sufficient
    • Type safety performs local checks at the call interface
    • Not convenient to enforce global properties such as:
      Program A can access all files, while program B can only access /tmp
    • Particularly when programs access resources using intermediate modules
Java 2 Security

- Based on URL from which the class was loaded, digital signatures …
- Class placed in a protection domain
  - Privileges of a protection domain specified in a policy file
  - Maintained as run-time state
- Stack introspection algorithm when method M requires permission P
  - Functionality implemented in SecurityManager, AccessController classes
  - Unwind each frame and check if P is allowed for each protection domain
    - Intersection of permissions
    - doPrivileged tags caller’s frame: algorithm terminates on encountering tag
- Standard permissions
  - Filesystem, Sockets, System properties, …
- Guarded Objects permit access control for individual object instances
  - Above mechanisms control access for all instances of a class
Language-based Protection: Pros and Cons

• Pros
  – Precision and flexibility: Only some software modules can have privileges
    • Rights amplification: As required, rights can be increased/decreased
      – E.g., an object’s method can have access to its private members
  – Security: Static checking can enforce more general security policies
    – E.g., a lock that is being acquired will always be released
  – Efficiency

• Cons
  – Need for a trusted compiler and run-time system
    • Particularly, in the presence of multiple sources
  – Single language restriction
  – Garbage collection: What happens to deallocated pointers?
  – Revocation
  – Performance of regular code