Protection Concepts and Methods

COMP 321

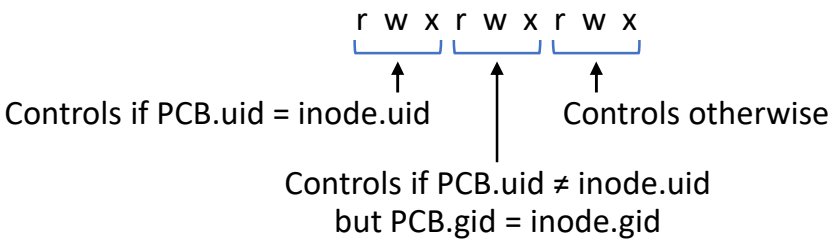
Dave Johnson

1

File Protection: Comparing Alternatives

***Unix* file protection**

- Each process PCB has a ***user id*** and a ***group id***, set at login and from fork()
- Each inode has fields to store the owner and protection for the file
  - a ***user id*** and ***group id*** owner, set from process that created it
  - a "***mode***" with 3 sets of 3 protection bits for read/write/execute protection

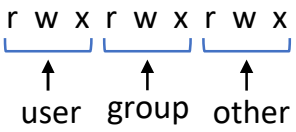


2

The Meaning of Unix Read/Write/Execute Bits

The meaning of "read" and "write" protection is simple

- ***Read*** means can read the contents of the file
- ***Write*** means can modify the contents of the file



But "execute" protection is less straightforward

- For a regular file
  - Can use file on an execve() kernel call, independent of ***read*** protection
- For a directory
  - Can ***use*** names within that directory (this is sometimes called ***search*** or ***traverse*** protection)
  - ***Read*** protection on a directory allows you to ***read*** the directory to learn the names in the directory, but ***execute*** allows you to ***use*** those names

Changing Permissions on a File

```
int  chmod(const char  *pathname,  mode_t  mode);
```

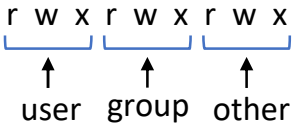
Changes the permissions on the existing file given by pathname

- pathname may refer to a regular file or a directory or a symbolic link
  - If a symbolic link, it follows through and changes permissions on the target
- Example: chmod("/abc/def/ghi", 0755)
  - Changes the permissions on the inode specified by "ghi" in the directory "/abc/def"

Unix File Protection Examples

Simple common examples

- Regular file: 600 or 644
- Program file: 700 or 755
- Directory: 700 or 755



A more complex example

- Imagine a directory somewhere named ***parentdir/secretdir***
- Protection on ***secretdir*** is 777 – everybody can write into it (create files in it)
- The real name of ***secretdir*** is only inside some program ***secretprog***
  - ***secretprog*** is 711 – everybody can execve( ) it but can't see its contents
- Protection on ***parentdir*** is 711 – everybody can use names in it but not "ls" it

Changing a File's Ownership

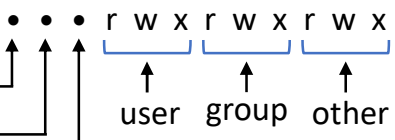
```
int chown(const char  *pathname,  uid_t  owner,  gid_t  group);
```

Changes the ownership of the existing file given by pathname

- pathname may refer to a regular file or a directory or a symbolic link
  - If a symbolic link, it follows through and changes ownership of the target
- Example: chown("/abc/def/ghi", 987, 654)
  - Changes ownership on the inode specified by "ghi" in the directory "/abc/def"
  - The new user id owner will be 987, the new group id owner will be 654
- If owner is -1, then the owner id on the inode is not changed
- If group is -1, then the group id on the inode is not changed

Other Unix Protection Concepts

Three additional protection bits in Unix "mode"



setuid: set PCB.uid to inode.uid on execve()

setgid: set PCB.gid to inode.gid on execve()

On a ***directory*** inode = ***restricted delete***
(can only delete a file in it if PCB.uid = file.uid)
On a regular ***inode*** = ***sticky*** bit (but now unused)

Changing a file (inode) owner uid or gid

- chown( ) kernel call can ony be done if PCB.uid == file.uid

Unix superuser (similar to, e.g., Windows Administrator)

- If PCB.uid == 0 ("root"), user can do "anything"

Setting a Process's Creation Mode Mask

```
mode_t  umask(mode_t  mask);
```

Sets the mask in the calling process's PCB, returns its old value

- Affects the protection on new files or directories created by this process
  - Useful using programs that create files using their own fixed mode value
- Any bit that is SET in "mask" will NOT be set for the new file or directory
  - Result is (requested mode) & ~mask
- Examples after doing umask(022)
  - creat("abc/def", 0666) – resulting protection will be 0644
  - open("abc/def", O_CREAT|O_TRUNC|O_RDWR, 0666) – the same
  - mkdir("/123/456", 0777) – resulting protection will be 0755

File Protection: Comparing Alternatives

***Windows* file protection**

- Equivalent of an inode is a ***record*** in the ***Master File Table*** (***MFT***)
- Each file (i.e., MFT record) is protected by an explicit ***access control list*** (ACL)
- Lists individual explicit users ("principals" or "trustees")
  - user name or group name
  - For each, what operations they can/can't do ("rights" or "permissions") on the file using that ACL
- Two models for NTFS permissions ("basic" or "advanced"):
  - basic: 6 different permissions defined (only 5 apply to files, 6 to directories)
  - advance: 13 different permissions defined
  - The 6 basic ones overlap each other, and the advanced ones even more so!

Windows Basic ACL Permissions

	For a File	***For a Directory***
Read	Read contents, View attrib/owner/perm	List contents, View attrib/owner/perm
Write	Change contents, Change attrib/owner/perm	Create new names in, Change atrrib/owner/perm
List Contents		List names in
Read + Execute	***Read***, Execute (run)	***Read***, ***List Contents***, Traverse
Modify	***Write***, Delete file	Delete dir, ***Read*** + ***Execute***
Full Control	Anything	Anything

Implementing NTFS Access Control Lists

NTFS originally stored ACL in Security Descriptor attribute in each MFT record

- Problem: Many files have the same ACL, so wastes space repeating them

Instead, NTFS 3.0 moved all ACLs into a single global "file"

- Added a ***SecurityId*** to the "***standard information***" attribute in each MFT record = as a hash of the full ACL for the file
- All actual ACLs are now stored together in the "security file" at MFT record 9
  - (Equivalent to using a fixed inode number 9, like the root inode number)
  - Only one copy of each ***different*** ACL needs to be stored
  - Contents organized as a B-Tree (similar to an NTFS directory), sorted by the ***SecurityID*** hash value
  - If new ACL is not found in the B-Tree, it is added (once)

Generalized Protection Concepts

Protection is enforced on objects

- Examples: files, processes, memory, devices, . . .

Each process is in exactly one protection domain

- A list of all ***objects***, and for each object, a list of that process's ***rights*** for that object (permission to perform some operation on that object)
- Example: In Unix for most objects, a process's uid and gid in its PCB define which protection domain the process is in
  - Unix protections are based on these two numbers
  - File protections, already discussed in this lecture
  - Kill another process (only if the uid of that process matches yours)
  - Uid = 0 means that you are the "superuser" and can do "anything"

The Access Matrix Represents All Protections

Domain	File 1	File 2	File 3	. . .	Process 1	Process 2	. . .
D1	RWX	-	RX		kill	-	
D1	-	RX	RX		-	-	
D3	-	RWX	-		-	kill	
D4	RX	-	RWX		kill	kill	
. . .				. . .			. . .
D*n*	RWX	-	RX		-	kill	

One row for each protection domain, and one column for each object

- ***Problem***: The access matrix is ***huge***!
- ***Problem***: The access matrix is ***sparse***

Access Control Lists

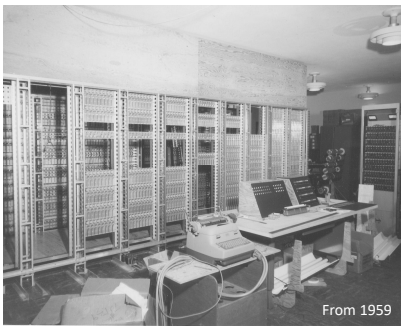
Essentially, is the non-empty entries in a single column of the access matrix

- Store the ACL in a way associated with the object that is protects
- Must, of course, store it in a way that is, itself, protected
- Example: Windows NTFS ACL used for file protection
  - A file's MFT record "standard information" attribute gives ***SecurityID***
  - Serves as an index into the B-Tree in the "security" file at MFT record 9
  - That "security" file at MFT record 9 is also protected by an ACL, stored in the "security" file at MFT record 9
  - So the "security" file protects itself!

Capability Lists

Essentially, is the non-empty entries on a single row of the access matrix

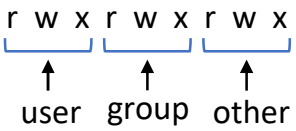
- Each ***capability*** in the list identifies a single (object, rights) for this process
- Store the capability list associated with the process to which it applies
  - Example: In the process's PCB (or linked from the PCB)
  - Capability list contents is thus protected the same as other kernel data
- Or encrypt the capability and store it anywhere
- The basic idea originated for memory protection
  - John K. Iliffe and Jane G. Jodeit (1962)
  - For the ***Rice Institute Computer*** (later known as the Rice R1 computer)!



Does Classical Unix Use Access Control Lists?

Most people would say "no," but they are wrong

- You just have to look at classical Unix protection carefully
- Consider the protection defined in an inode
  - classical 9 protection "mode" bits
  - user id and group id owning the file



- This is a compact representation of a simple ACL
  - Defines a simple set of rights for 3 principals
    - process uid matches inode uid
    - process uid doesn't match indoe uid, but process and inode gid match
    - all other processes

Does Classical Unix Use Capability Lists?

Most people would say "no," but they are wrong

- You just have to look at classical Unix protection carefully
- Consider Unix open(pathname, mode) for mode = O_RDONLY, O_WRONLY, or O_RDWR
- The kernel decides whether or not to allow the open()
  - Using the inode's "ACL" ! (i.e., mode bits, uid, and gid in the inode)
- But what then limits read() and write() based on mode for which opened?
  - Kernel remembers the opened mode O_RDONLY, O_WRONLY, or O_RDWR
  - Kernel checks each read() or write() call based only on that
  - Not based on the indode's mode bits, or inode's uid or gid, or process's uid or gid – Only the O_RDONLY, O_WRONLY, or O_RDWR matters then!

Does Classical Unix Use Capability Lists?

Most people would say "no," but they are wrong

- The open file table in each process's PCB
  - struct file *u_ofile[NOFILE]; // pointers to file structures of open files
  - Indexed by file descriptor number used on read(), write(), etc.
  - If ***pointer*** == ***NULL***, this file descriptor is not open, so ERROR
  - Otherwise
    - That "struct file" remembers the current offset within the open file
    - and ***stores the mode that was specified on the open()***
- The kernel checks file inode protection at open() time using the inode's "ACL"
- Then for each read() or write(), u_ofile array is the ***capability list***, and that pointer and its struct file are the ***capability***, protecting read vs. write for that fd