Security Part 3

Live Anonymous Q&A:
https://tinyurl.com/cs1670feedback

Recap: Access Control

- Two approaches
  - who you are
    - subjects' identity attributes determine access to objects
  - what you have
    - capabilities possessed by subjects determine access to objects

$\Big\}$ *today*

$\Big\}$ *revisit later*

Who-You-Are-Based Access Control

- Discretionary access control (DAC)
  - objects have owners
  - owners determine who may access objects and how they may access them
- Mandatory access control (MAC)
  - system-wide policy on who may access what and how
  - object owners have no say

Access Control in Traditional Systems

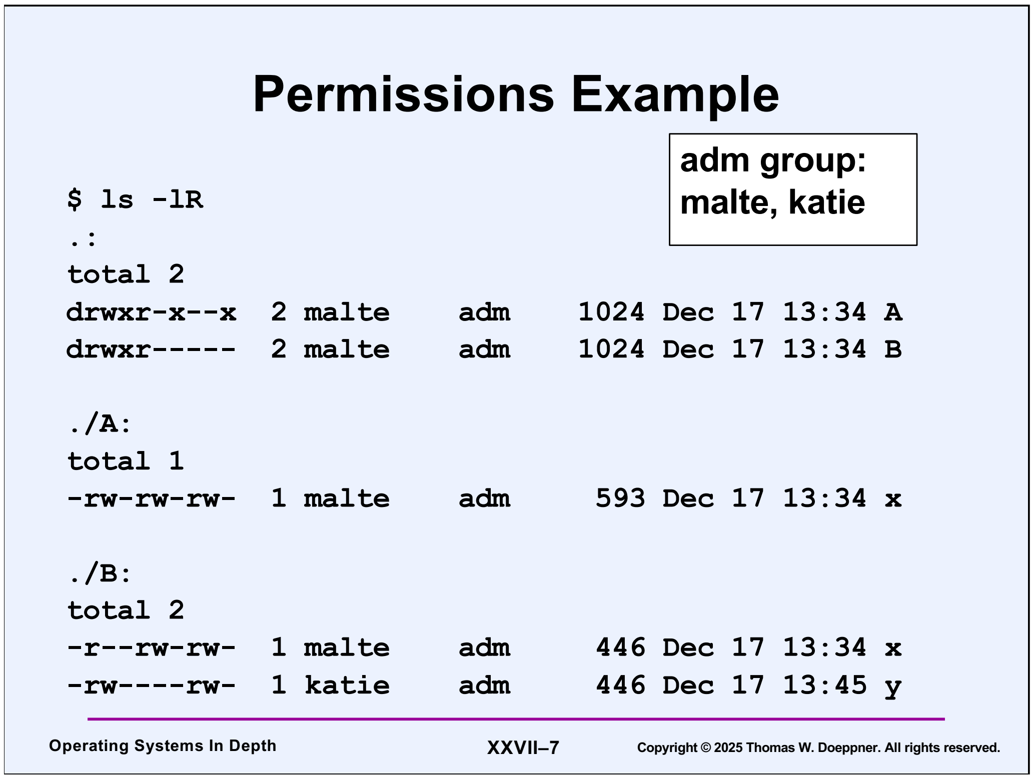
- Unix and Windows
  - primarily DAC
  - file descriptors and file handles provide capabilities
  - MAC becoming more popular
    - SELinux
    - Windows

$\left.\begin{array}{l}\text{MAC becoming more popular}\\ \text{SELinux}\\ \text{Windows}\end{array}\right\}$ *will discuss in later lecture*

Case Study 1: Unix Permissions

Unix

- Process’s security context
  - user ID
  - set of group IDs
  - more discussed later
- Object’s authorization information
  - owner user ID
  - group owner ID
  - permission vector



Permissions Example

adm group:
malte, katie

```
$ ls -lR
.:
total 2
drwxr-x--x  2 malte    adm    1024 Dec 17 13:34 A
drwxr-----  2 malte    adm    1024 Dec 17 13:34 B

./A:
total 1
-rw-rw-rw-  1 malte    adm     593 Dec 17 13:34 x

./B:
total 2
-r--rw-rw-  1 malte    adm     446 Dec 17 13:34 x
-rw----rw-  1 katie    adm     446 Dec 17 13:45 y
```

The `ls -lR` command lists the contents of the current directory, its subdirectories, their subdirectories, etc. in long format (the `l` causes the latter, the `R` the former).

In the current directory are two subdirectories, **A** and **B**, with access permissions as shown in the slide. Note that the permissions are given as a string of characters: the first character indicates whether or not the file is a directory, the next three characters are the permissions for the owner of the file, the next three are the permissions for the members of the file's group's members, and the last three are the permissions for the rest of the world.

Quiz: the users **malte** and **katie** are members of the **adm** group; **leo** is not.

May **leo** list the contents of directory *A*?

May **leo** read *A/x*?

May **katie** list the contents of directory *B*?

May **katie** modify *B/y*?

May **malte** modify *B/x*?

May **malte** read *B/y*?

May **leo** read *B/y*?

Quiz 1

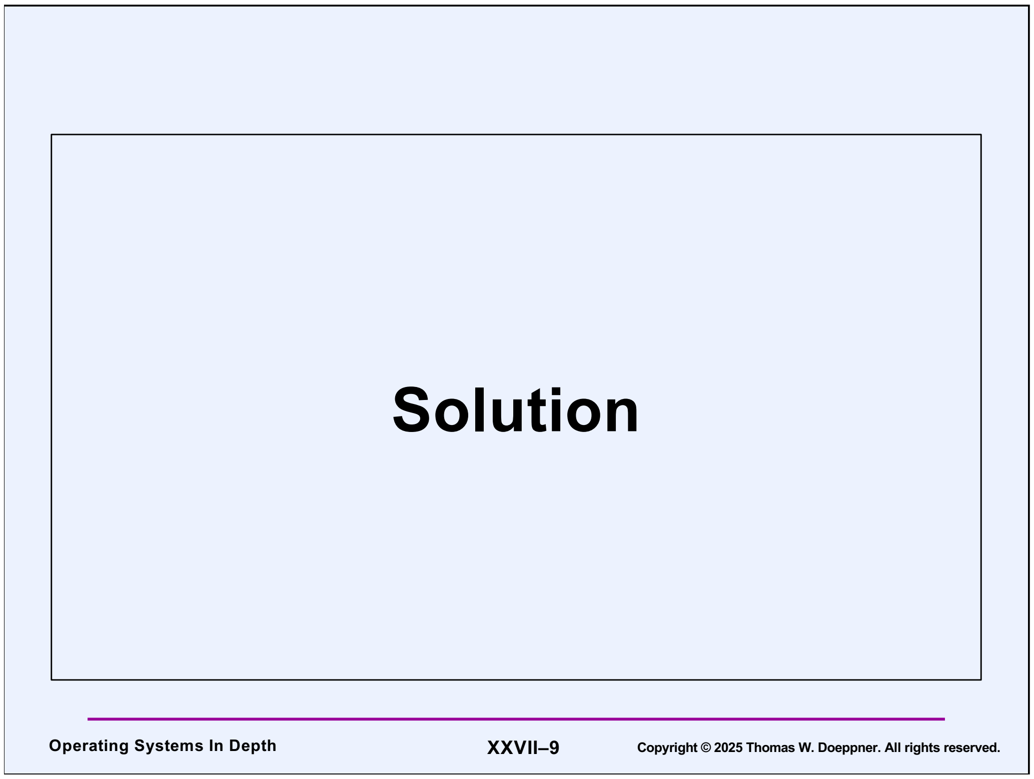
Recall that in Unix, each file/directory has an owner and a group (e.g., owner “joe”, group “adm”).

Is there a means in Unix to specify that members of *two* different groups have read access to a file, without resorting to features we haven’t yet discussed?

a) No, it can’t be done.

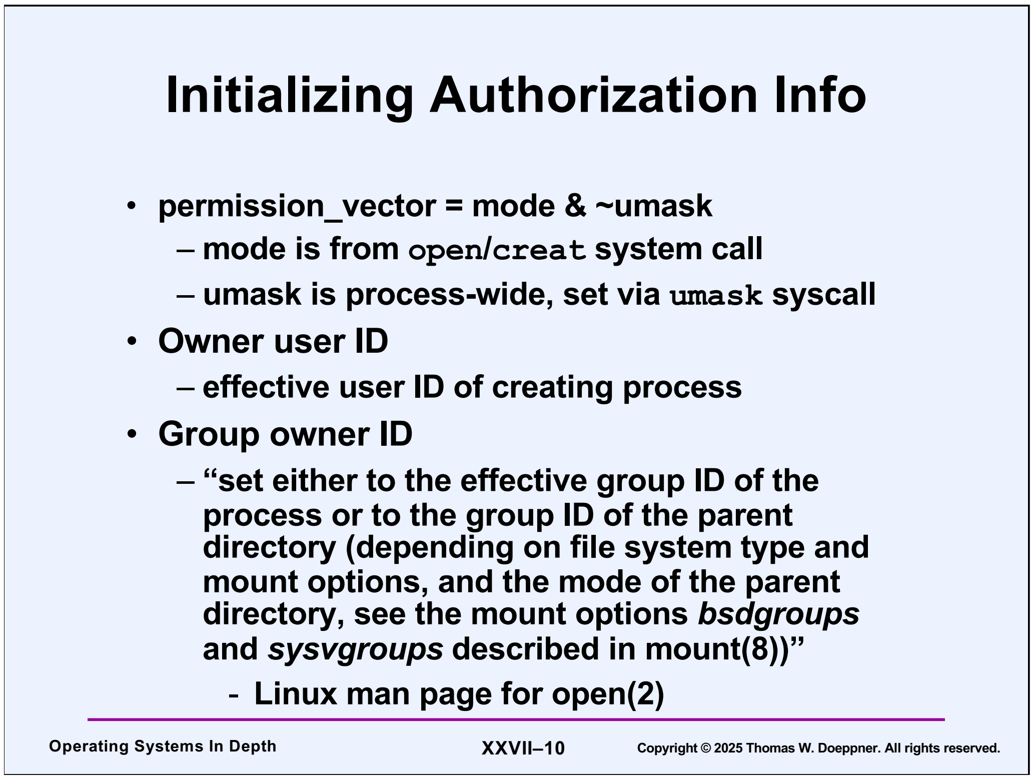
b) Yes, but it’s complicated.

c) Yes, it can be done in a single command just like setting the file readable by just one group.



Solution

The “none” group is a group with no members.

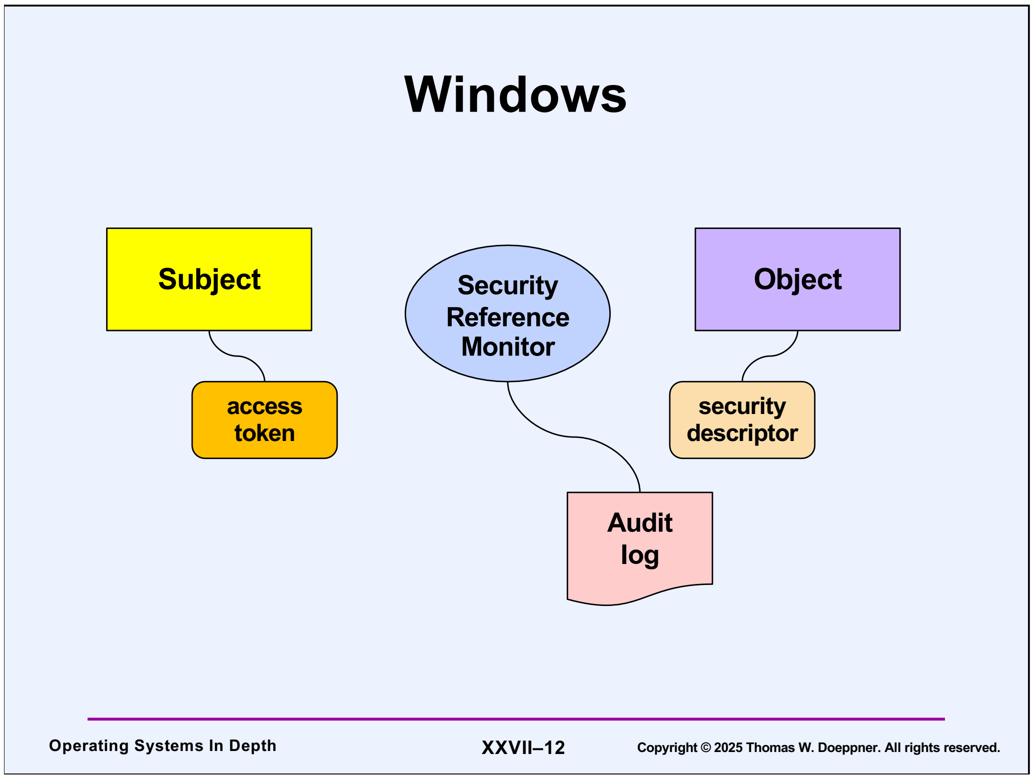


Initializing Authorization Info

- permission_vector = mode & ~umask
  - mode is from `open`/`creat` system call
  - umask is process-wide, set via `umask` syscall
- Owner user ID
  - effective user ID of creating process
- Group owner ID
  - "set either to the effective group ID of the process or to the group ID of the parent directory (depending on file system type and mount options, and the mode of the parent directory, see the mount options *bsdgroups* and *sysvgroups* described in mount(8))"
    - Linux man page for open(2)

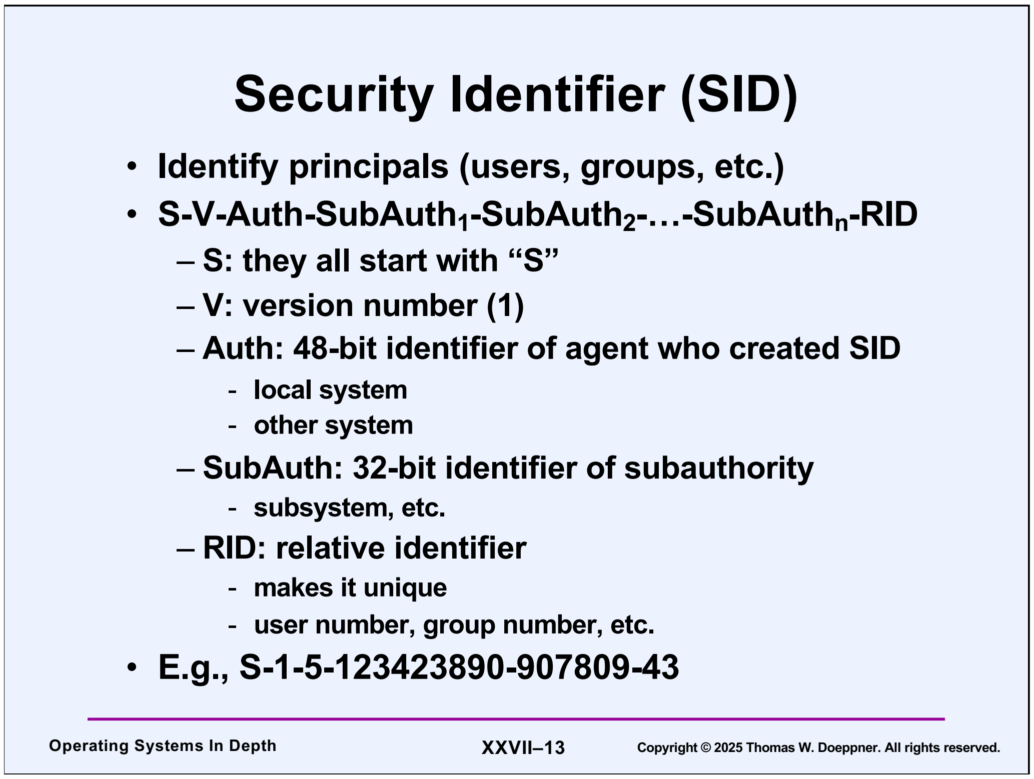
We explain "effective user ID" in the next lecture. For now, simply think of it as "user ID".

Case Study 2: Windows Security Architecture



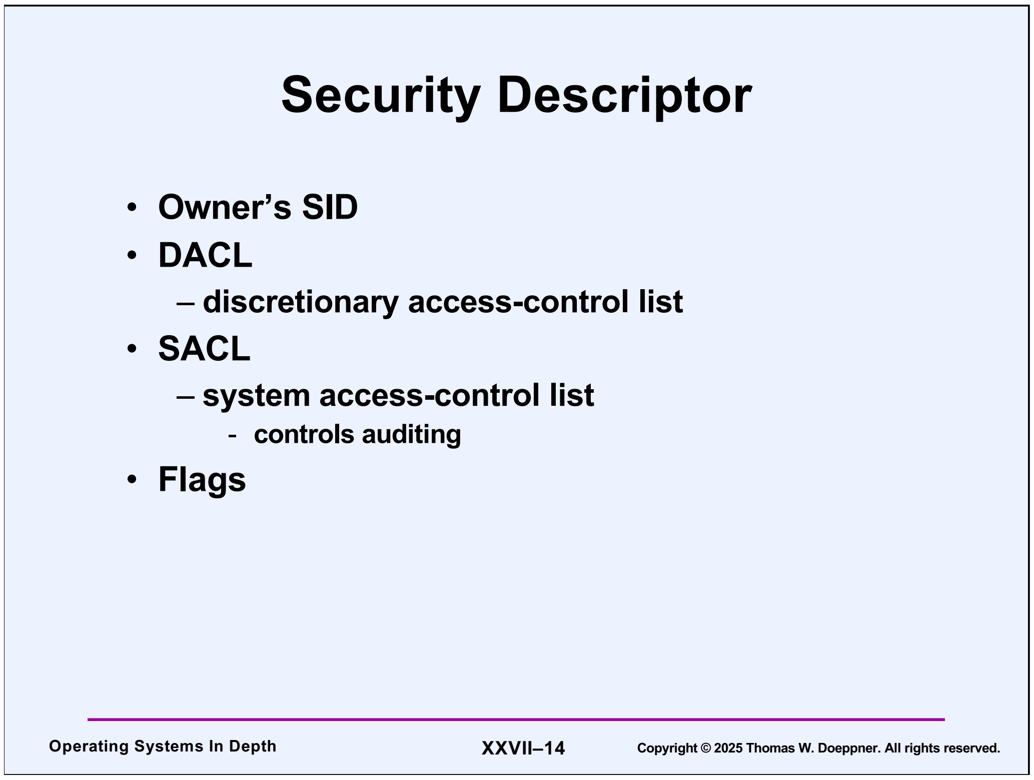
Windows

A subject (a process) is attempting to access an object (perhaps a file). The **access token** represents the identity of the subject. The **security descriptor** describes who may access the object in what ways. The reference monitor, using the **access token** and **security descriptor** as input, determines whether the desired access should be permitted. These decisions may be recorded in an **audit log**.


Security Identifier (SID)

- Identify principals (users, groups, etc.)
- S-V-Auth-SubAuth$_1$-SubAuth$_2$-…-SubAuth$_n$-RID
  - S: they all start with “S”
  - V: version number (1)
  - Auth: 48-bit identifier of agent who created SID
    - local system
    - other system
  - SubAuth: 32-bit identifier of subauthority
    - subsystem, etc.
  - RID: relative identifier
    - makes it unique
    - user number, group number, etc.
- E.g., S-1-5-123423890-907809-43

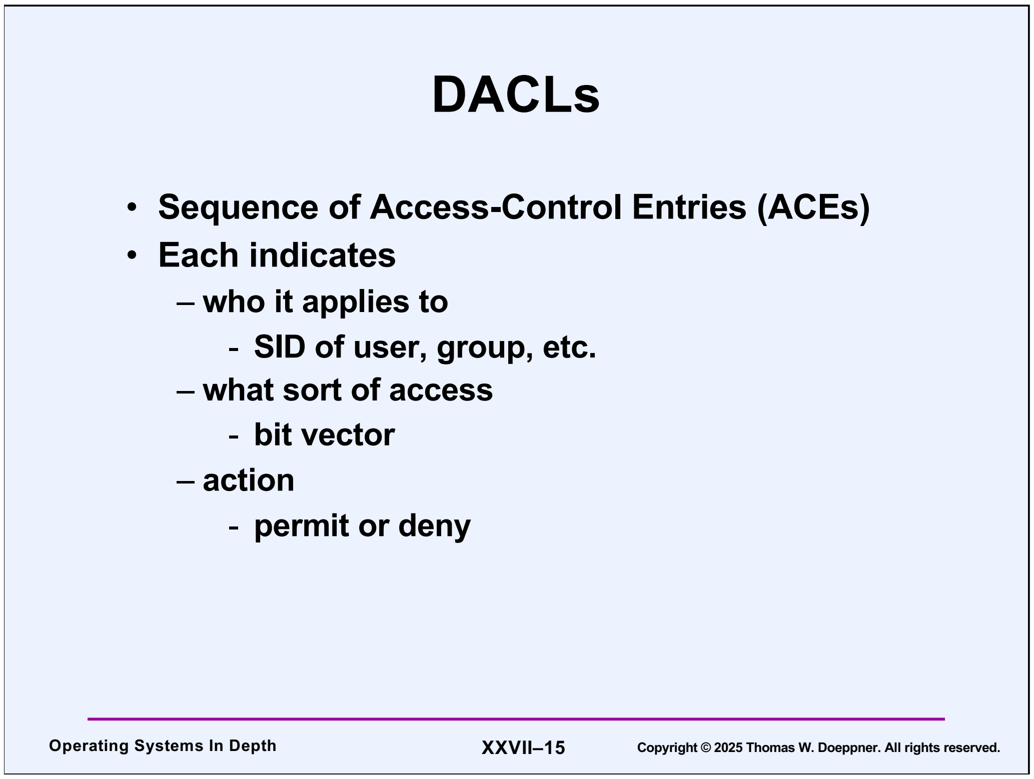
Security **principals** are the users and groups of users for whom access is to be mediated. They’re uniquely identified by **Security Identifiers**, as shown in the slide.



Security Descriptor

- Owner's SID
- DACL
  - discretionary access-control list
- SACL
  - system access-control list
    - controls auditing
- Flags

A security descriptor is provided for each object. It indicates who owns the object, who is allowed to access it, and what sorts of accesses should be audited.

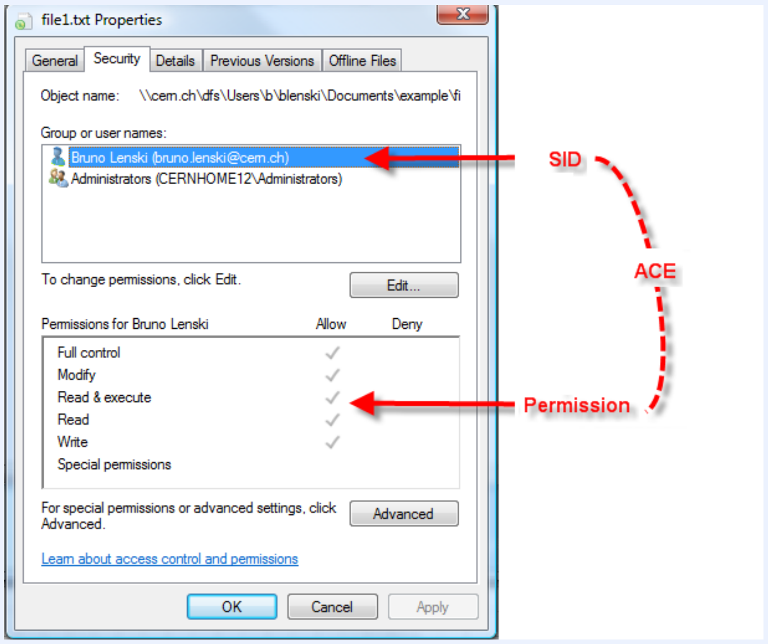


DACLs

- Sequence of Access-Control Entries (ACEs)
- Each indicates
  - who it applies to
    - SID of user, group, etc.
  - what sort of access
    - bit vector
  - action
    - permit or deny

A DACL (discretionary access control list) describe which principals may access an object and how they may access it.

See http://msdn.microsoft.com/en-us/library/aa374876(VS.85).aspx for more details about ACLs.


file1.txt Properties

General | Security | Details | Previous Versions | Offline Files

Object name: \\cern.ch\dfs\Users\b\blenski\Documents\example\fi

Group or user names:

Bruno Lenski (bruno.lenski@cern.ch)
Administrators (CERNHOME12\Administrators)

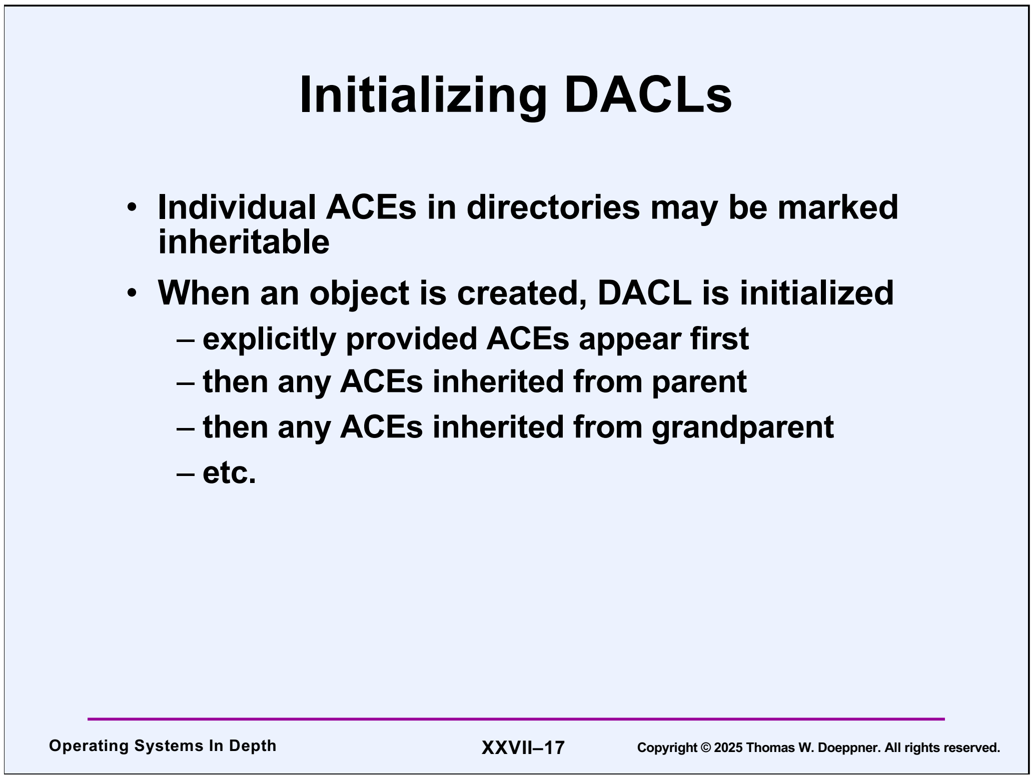
To change permissions, click Edit.

Permissions for Bruno Lenski	Allow	Deny
Full control	✓	
Modify	✓	
Read & execute	✓	
Read	✓	
Write	✓	
Special permissions		

For special permissions or advanced settings, click Advanced.

Learn about access control and permissions

Image credit: https://devices.docs.cern.ch/devices/windows/permissions/



Initializing DACLs

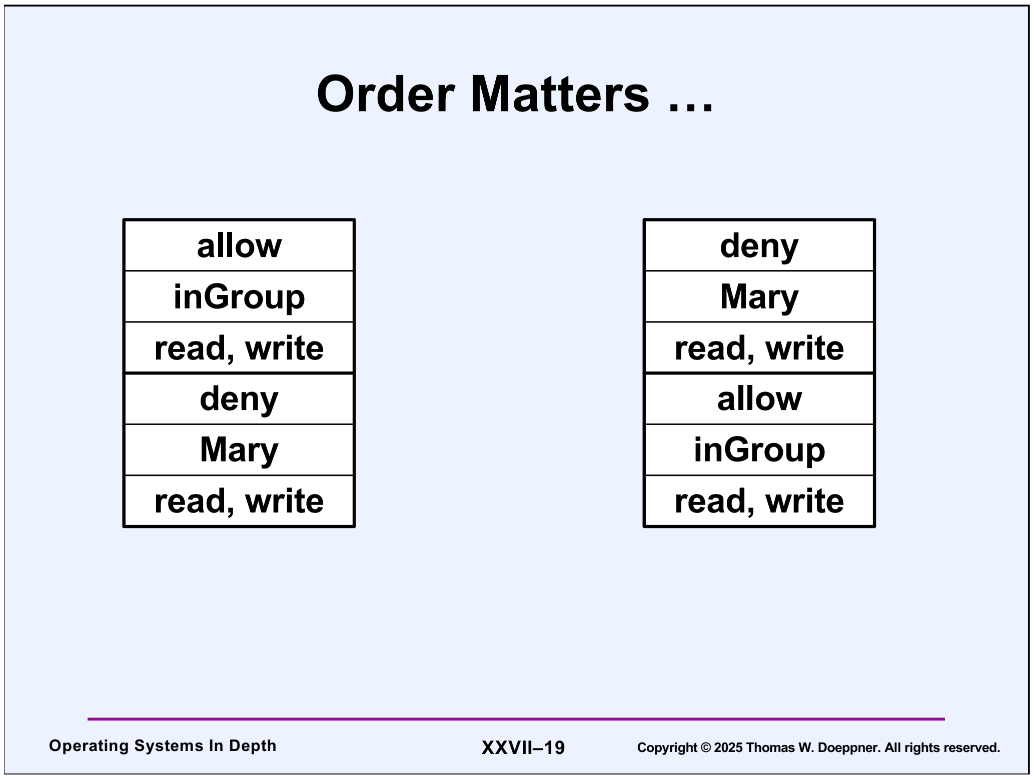
- Individual ACEs in directories may be marked inheritable
- When an object is created, DACL is initialized
  - explicitly provided ACEs appear first
  - then any ACEs inherited from parent
  - then any ACEs inherited from grandparent
  - etc.

ACE = access control entry.

An ACL is an ordered list of ACEs.

Decision Algorithm

```
accesses_permitted = null
walk through the ACEs in order
    if access token's user SID or group SID match ACE's SID
        if ACE is of type access-deny
            if a requested access type is denied
                Stop — access is denied
        if ACE is of type access-allow
            if a requested access type is permitted
                add access type to accesses_permitted
                if all requested accesses are permitted
                    Stop — access is allowed
if not all requested access types permitted
    Stop — access is denied
```



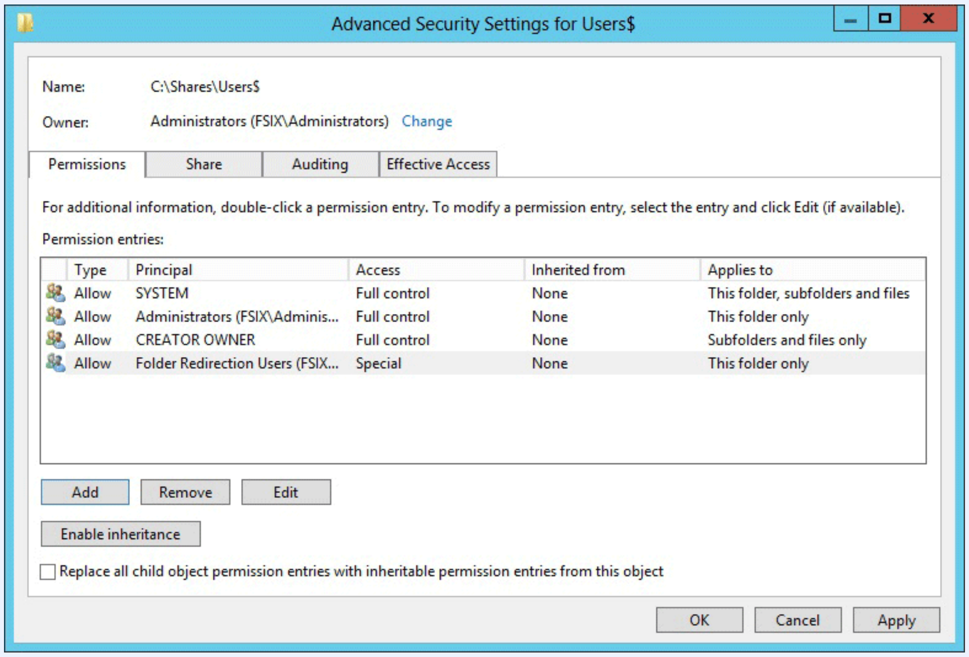
Order Matters …

allow	deny
inGroup	Mary
read, write	read, write
deny	allow
Mary	inGroup
read, write	read, write

Mary is a member of inGroup. Thus she is permitted read/write access in the leftmost ACL, but is denied read/write access in the rightmost ACL.

Preferred Order

- *Access-denied* entries first
- *Access-allowed* entries second
- However …
  - not enforced
  - system GUIs don’t show order
  - only way to find out is to ask for “effective permissions”


Advanced Security Settings for Users$

Name: C:\Shares\Users$

Owner: Administrators (FSIX\Administrators) Change

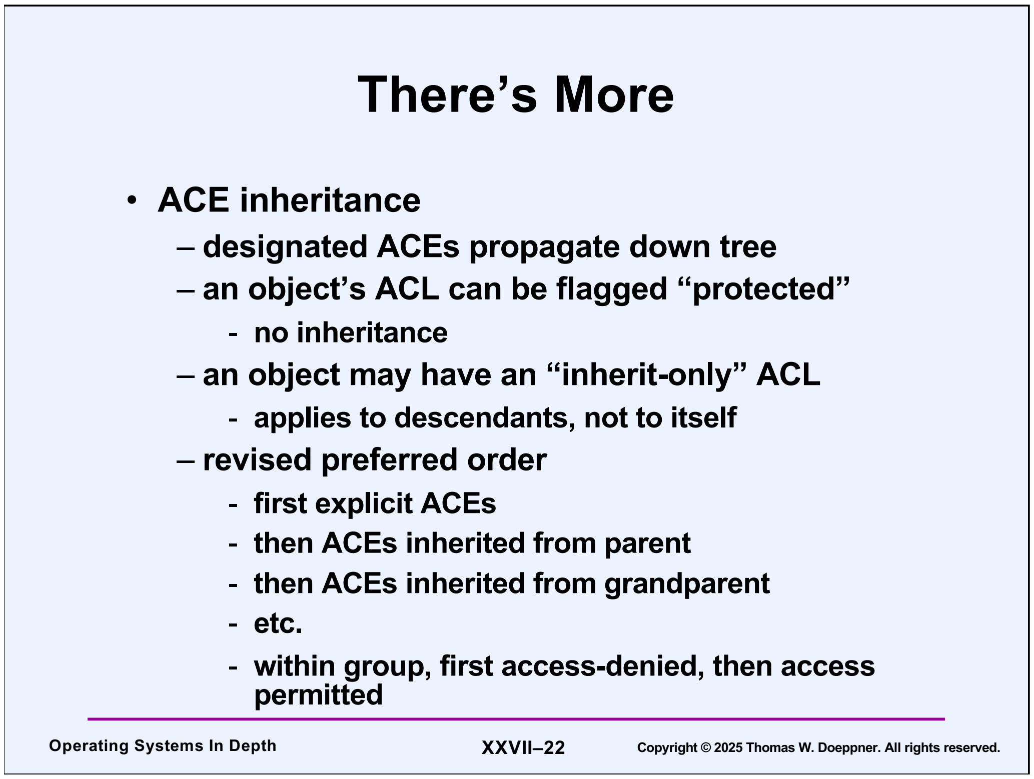
For additional information, double-click a permission entry. To modify a permission entry, select the entry and click Edit (if available).

Permission entries:

Type	Principal	Access	Inherited from	Applies to
Allow	SYSTEM	Full control	None	This folder, subfolders and files
Allow	Administrators (FSIX\Adminis...	Full control	None	This folder only
Allow	CREATOR OWNER	Full control	None	Subfolders and files only
Allow	Folder Redirection Users (FSIX...	Special	None	This folder only

☐ Replace all child object permission entries with inheritable permission entries from this object

Image credit: https://superuser.com/questions/1186187/command-line-to-open-the-advanced-security-settings-dialogue



There's More

- ACE inheritance
  - designated ACEs propagate down tree
  - an object's ACL can be flagged "protected"
    - no inheritance
  - an object may have an "inherit-only" ACL
    - applies to descendants, not to itself
  - revised preferred order
    - first explicit ACEs
    - then ACEs inherited from parent
    - then ACEs inherited from grandparent
    - etc.
    - within group, first access-denied, then access permitted

When the ACL of a directory is modified, its inheritable ACEs are propagated to its descendants.

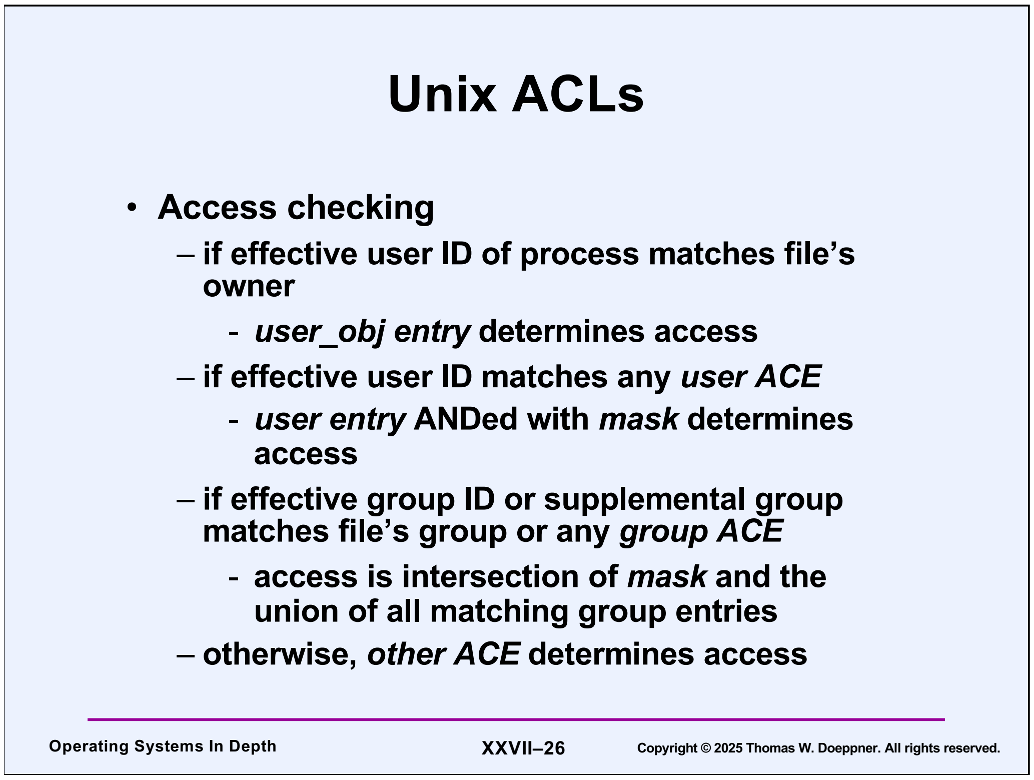
Case Study 3:
POSIX Advanced ACLs

Unix ACLs

- POSIX 1003.1e
  - deliberated for 10 years
    - what to do about backwards compatibility?
  - gave up …
  - but implemented, nevertheless
    - setfacl/getfacl commands in Linux

Unix ACLs

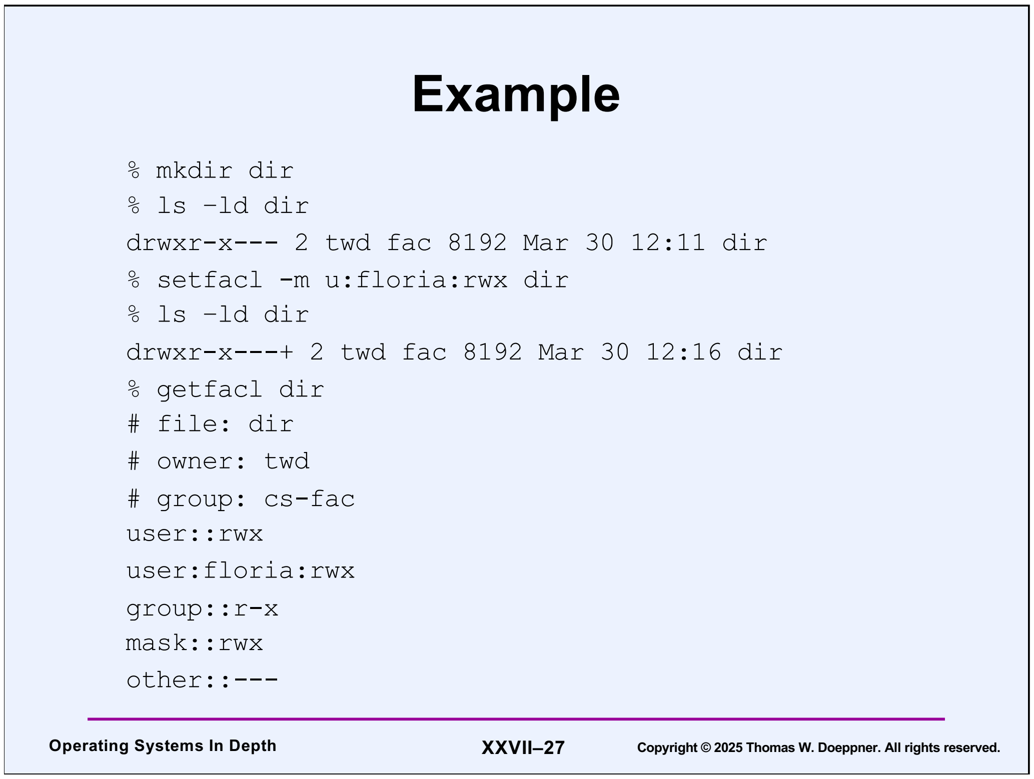
- ACEs
  - user_obj: applies to file's owner
  - group_obj: applies to file's group
  - user: applies to named user
  - group: applies to named group
  - other: applies to everyone else
  - mask: maximum permissions granted to user, group_obj, and group entries



Unix ACLs

- Access checking
  - if effective user ID of process matches file's owner
    - *user_obj entry* determines access
  - if effective user ID matches any *user ACE*
    - *user entry* ANDed with *mask* determines access
  - if effective group ID or supplemental group matches file's group or any *group ACE*
    - access is intersection of *mask* and the union of all matching group entries
  - otherwise, *other ACE* determines access

This is slightly simplified. See the acl man page for details.

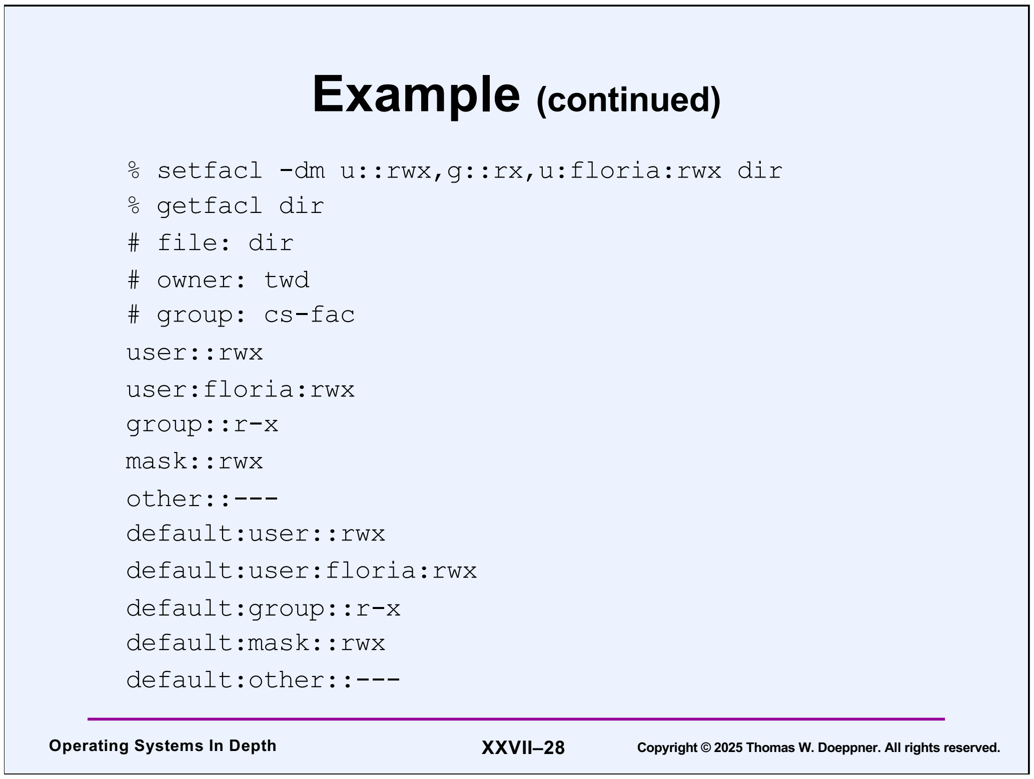


Example

```
% mkdir dir
% ls -ld dir
drwxr-x--- 2 twd fac 8192 Mar 30 12:11 dir
% setfacl -m u:floria:rwx dir
% ls -ld dir
drwxr-x---+ 2 twd fac 8192 Mar 30 12:16 dir
% getfacl dir
# file: dir
# owner: twd
# group: cs-fac
user::rwx
user:floria:rwx
group::r-x
mask::rwx
other::---
```

Here we create a directory, then add an ACL giving floria read, write, and execute permission. (Note that the “-m” flag of setfacl stands for “modify” -- we’re adding a user entry.) The **ls** command, via the “+”, indicates that the permissions on the directory are more complicated than it can show. However, the **getfacl** command shows the complete permissions. Note that a mask is automatically created allowing floria to have the full permissions requested. The user and group entries that don’t explicitly refer to a particular user or group give the access permissions of the user_obj (file owner) and group_obj.

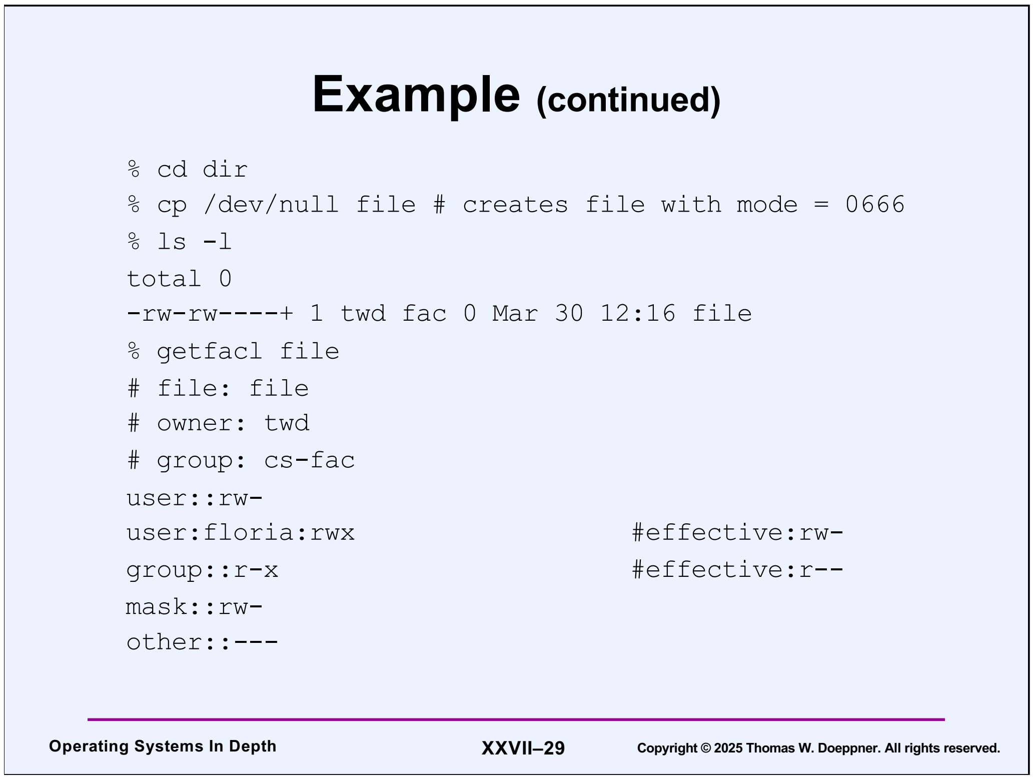
Note that this example assumes that all files (and directories) are on local file systems, not on remote file systems. Thus it cannot be replicated in, for example, your home directory on CS department machines, since home directories are on remote file systems (and ACLs aren’t necessarily implemented on remote file systems).



Example (continued)

```
% setfacl -dm u::rwx,g::rx,u:floria:rwx dir
% getfacl dir
# file: dir
# owner: twd
# group: cs-fac
user::rwx
user:floria:rwx
group::r-x
mask::rwx
other::---
default:user::rwx
default:user:floria:rwx
default:group::r-x
default:mask::rwx
default:other::---
```

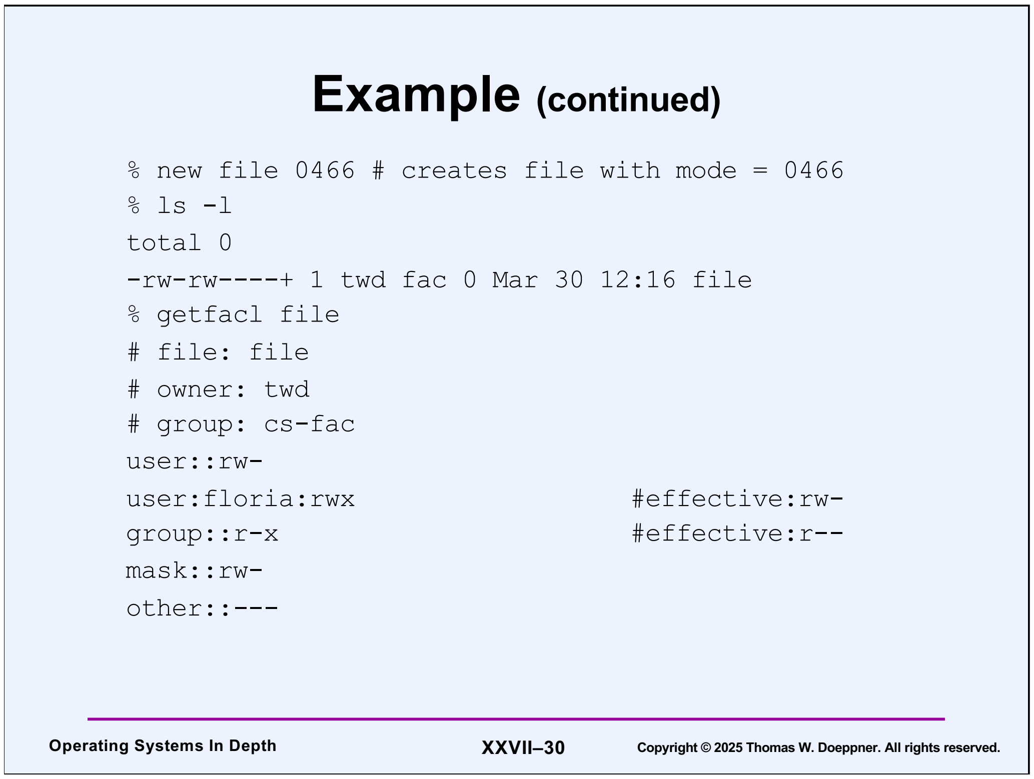
Now we give the directory a default ACL, which is used to initialize the ACLs for files and directories created within the directory.



Example (continued)

```
% cd dir
% cp /dev/null file # creates file with mode = 0666
% ls -l
total 0
-rw-rw----+ 1 twd fac 0 Mar 30 12:16 file
% getfacl file
# file: file
# owner: twd
# group: cs-fac
user::rw-
user:floria:rwx                 #effective:rw-
group::r-x                      #effective:r--
mask::rw-
other::---
```

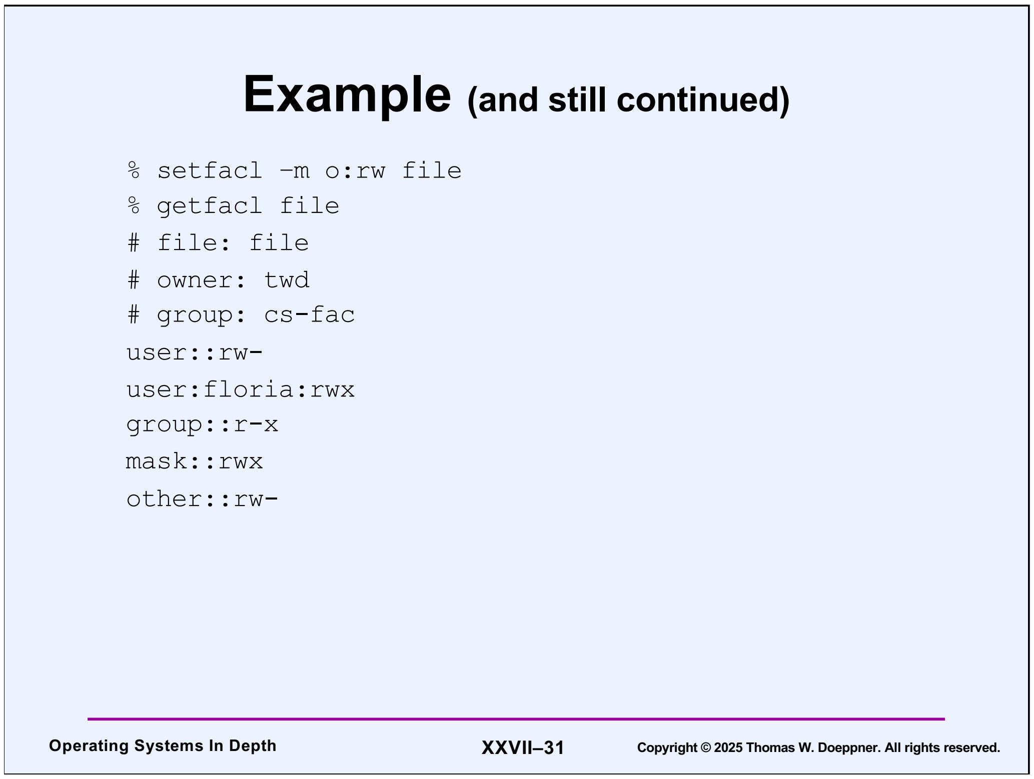
Here we create a file within the directory. The cp command supplies a **mode** argument (requested permissions) of 0666 to the **creat** system call – this is because it's copying a file whose permissions are 0666. (The umask is 007). The **mask ACE** was set automatically to rw (because of the default entry we put in the directory in the previous slide and **mode** having been specified as 0666), thus ensuring that floria's effective permissions are rw.



Example (continued)

```
% new file 0466 # creates file with mode = 0466
% ls -l
total 0
-rw-rw----+ 1 twd fac 0 Mar 30 12:16 file
% getfacl file
# file: file
# owner: twd
# group: cs-fac
user::rw-
user:floria:rwx                  #effective:rw-
group::r-x                       #effective:r--
mask::rw-
other::---
```

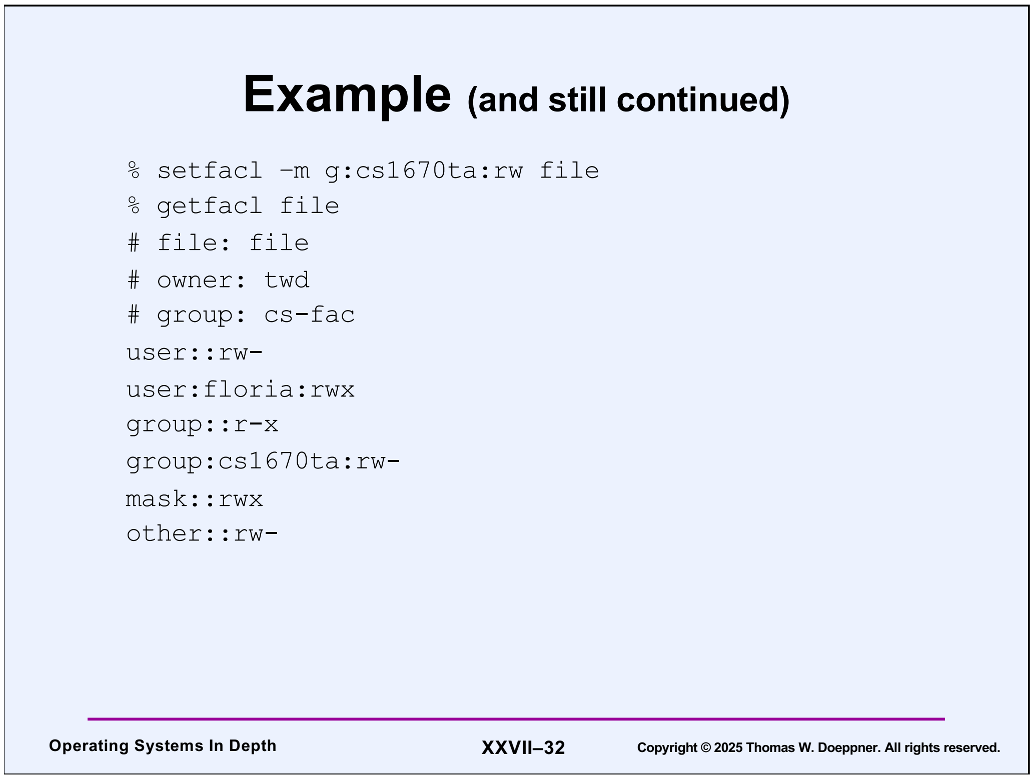
We now create the file again within the directory, but through the use of a program that we wrote, called **new**, with which we can control the mode bits in the **creat** system call. In this example, the file is created with requested permissions of 0466 (read-only for the user, read and write for the group and others), but adjusted by the umask. Strangely, both the owner of the file and floria are given rw permissions. (This is, again, because of the default ACL entry we put in the directory.)



Example (and still continued)

```
% setfacl -m o:rw file
% getfacl file
# file: file
# owner: twd
# group: cs-fac
user::rw-
user:floria:rwx
group::r-x
mask::rwx
other::rw-
```

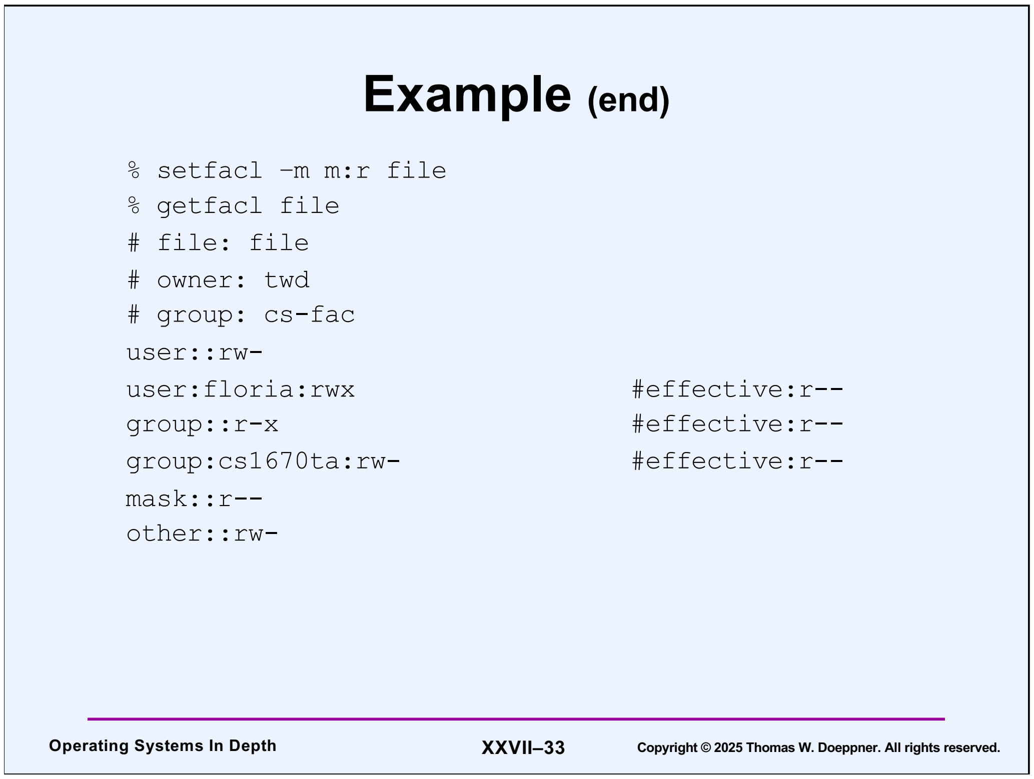
We further modify the ACL by setting the permissions for others to be rw. This appears to have the side effect of changing the mask to rwx, which gives both floria and the file's group execute access to the file. It's not at all clear why this happens and is likely a bug.



Example (and still continued)

```
% setfacl –m g:cs1670ta:rw file
% getfacl file
# file: file
# owner: twd
# group: cs-fac
user::rw-
user:floria:rwx
group::r-x
group:cs1670ta:rw-
mask::rwx
other::rw-
```

We further modify the ACL by adding a group entry for cs1670ta and setting it to rw.



Example (end)

```
% setfacl -m m:r file
% getfacl file
# file: file
# owner: twd
# group: cs-fac
user::rw-
user:floria:rwx                   #effective:r--
group::r-x                        #effective:r--
group:cs1670ta:rw-                #effective:r--
mask::r--
other::rw-
```

Finally, we change the mask to read-only. This affects the explicit user entry and the group entries, but does not affect the owner of the file or others.

Quiz 2

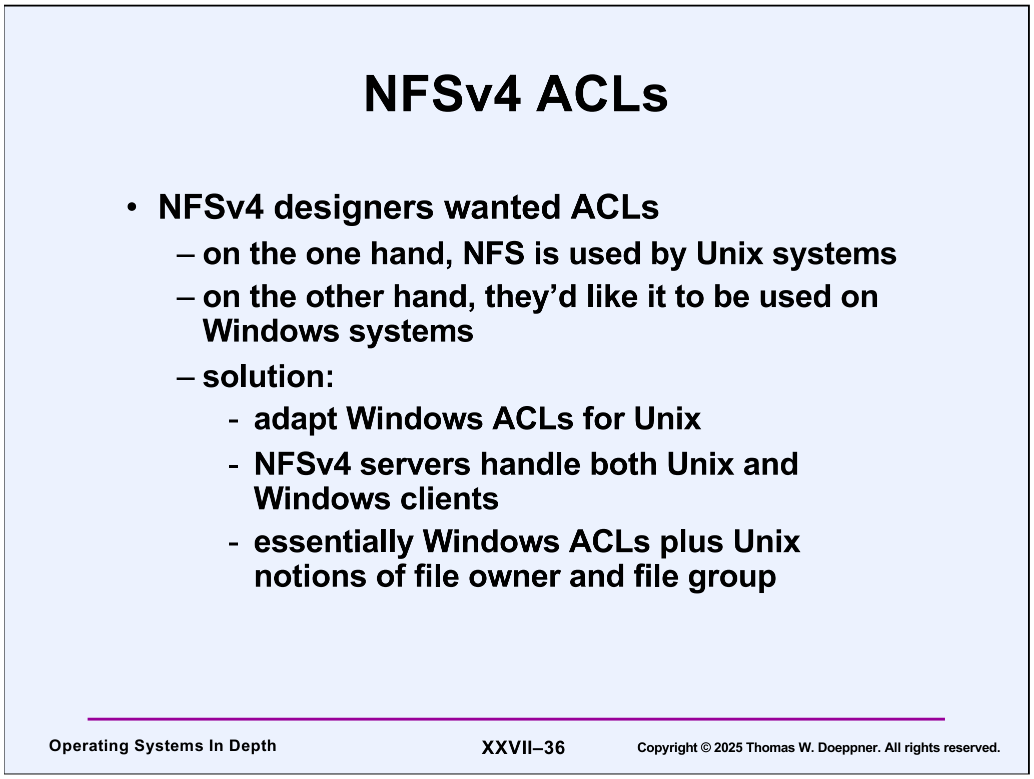
Unlike Windows ACLs, UNIX ACLs have no deny entries. Is it possible to set up an ACL that specifies that everyone in a particular group has rw access, except that a certain group member has no access at all?

a) No, it can't be done

b) Yes, it can be done in two commands

c) Yes, but it's complicated and requires more than two commands

Real-world Problem: Cross-OS ACL Interoperability

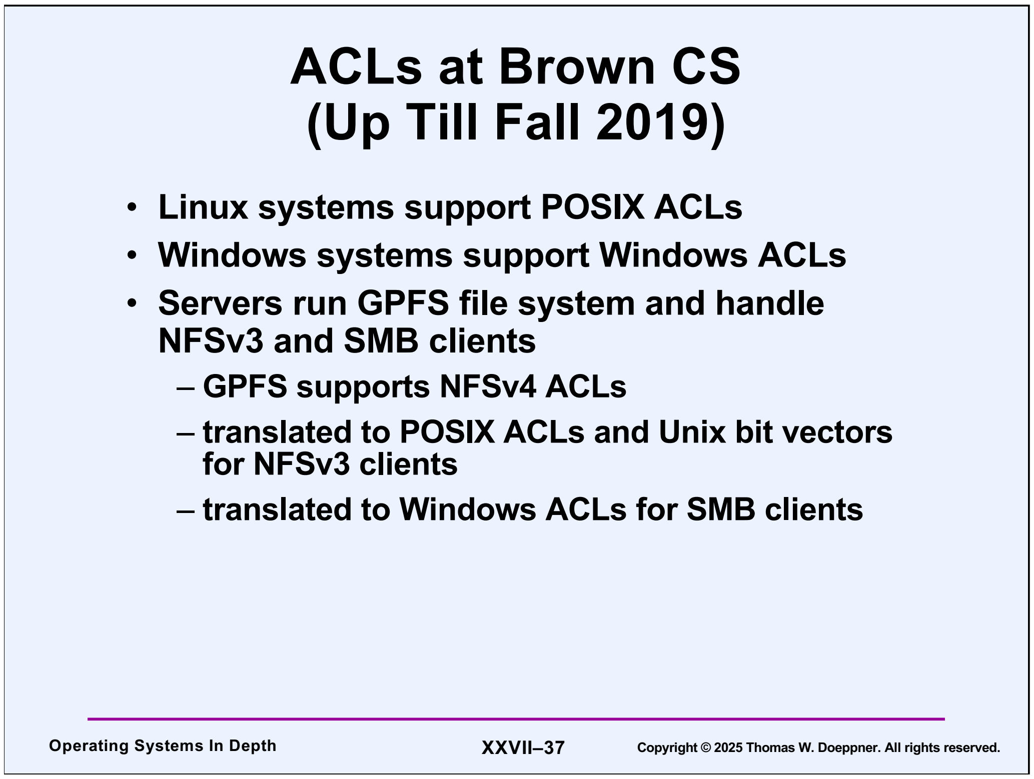


NFSv4 ACLs

- NFSv4 designers wanted ACLs
  - on the one hand, NFS is used by Unix systems
  - on the other hand, they'd like it to be used on Windows systems
  - solution:
    - adapt Windows ACLs for Unix
    - NFSv4 servers handle both Unix and Windows clients
    - essentially Windows ACLs plus Unix notions of file owner and file group

For details on NFSv4 ACLs, see RFC 3530 (http://www.ietf.org/rfc/rfc3530.txt), section 5.11.

We discuss NFS in later lectures.



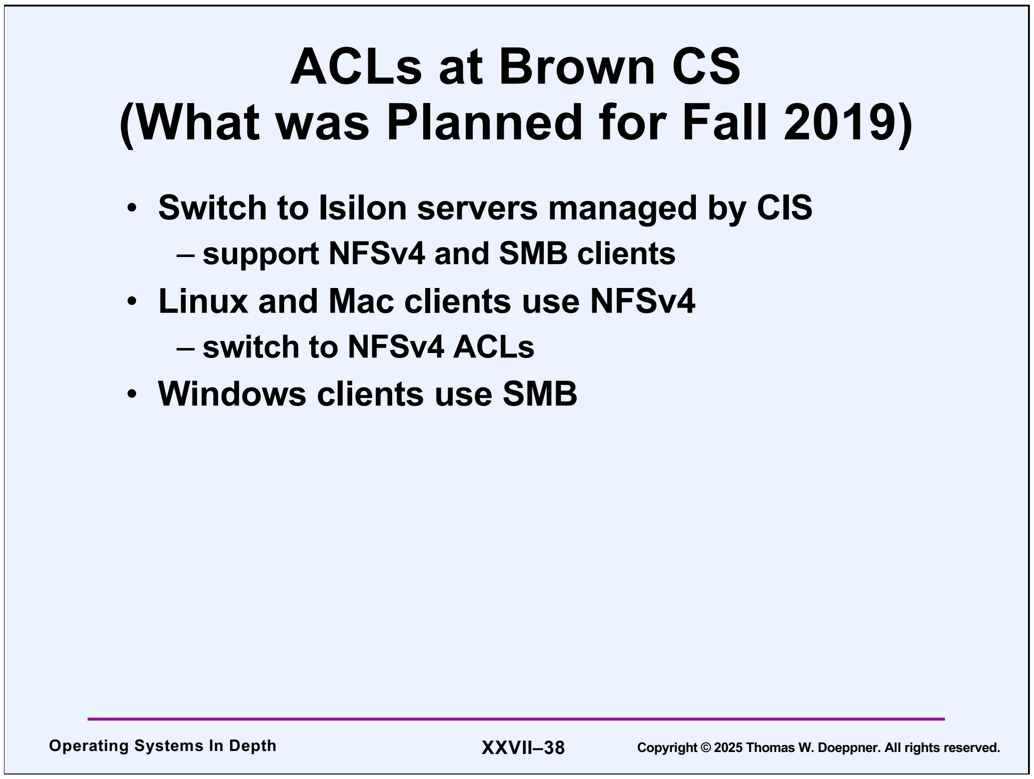
ACLs at Brown CS (Up Till Fall 2019)

- Linux systems support POSIX ACLs
- Windows systems support Windows ACLs
- Servers run GPFS file system and handle NFSv3 and SMB clients
  - GPFS supports NFSv4 ACLs
  - translated to POSIX ACLs and Unix bit vectors for NFSv3 clients
  - translated to Windows ACLs for SMB clients

GPFS stands for general parallel file system. It was developed by IBM for high-performance distributed file services.

SMB stands for server message block and is the means for handling distributed file systems on Windows (and others that have adopted it).

We will discuss both GPFS and SMB in later lectures.



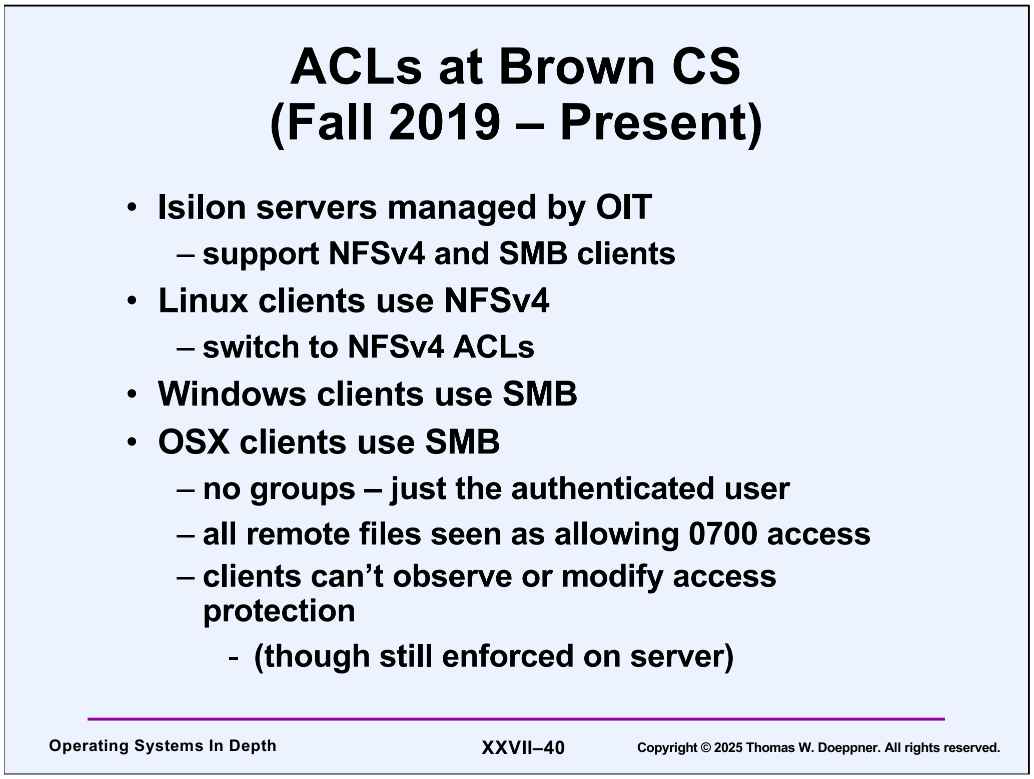
ACLs at Brown CS (What was Planned for Fall 2019)

- Switch to Isilon servers managed by CIS
  - support NFSv4 and SMB clients
- Linux and Mac clients use NFSv4
  - switch to NFSv4 ACLs
- Windows clients use SMB

This was the plan …

OSX (Macs)

- Native support for NFSv4 ACLs
  - no setfacl/getfacl commands, but built into chmod
- No NFSv4 client support
  - third-party implementations exist, but they don't work



ACLs at Brown CS (Fall 2019 – Present)

- Isilon servers managed by OIT
  - support NFSv4 and SMB clients
- Linux clients use NFSv4
  - switch to NFSv4 ACLs
- Windows clients use SMB
- OSX clients use SMB
  - no groups – just the authenticated user
  - all remote files seen as allowing 0700 access
  - clients can't observe or modify access protection
    - (though still enforced on server)

Brown's OIT (Office of Information Technology) is the new name for what was called CIS (Computer and Information Services).

Advanced Access Control

setuid and friends

Extending the Basic Models

- Provide a file that others may write to, but only if using code provided by owner
- Print server
  - pass it file names
  - print server may access print files if and only if client may
- Password-changing program (`passwd`)

Superuser (Unix)

- User ID == 0
  - bypasses all access checks
  - can send signals to any process

Attaining Super (or Lesser) Powers

- Setuid protection bit
  - the exec'ing process's UID is set to owner of file

The setuid bit was patented in 1973 by Dennis Ritchie.

User and Group IDs

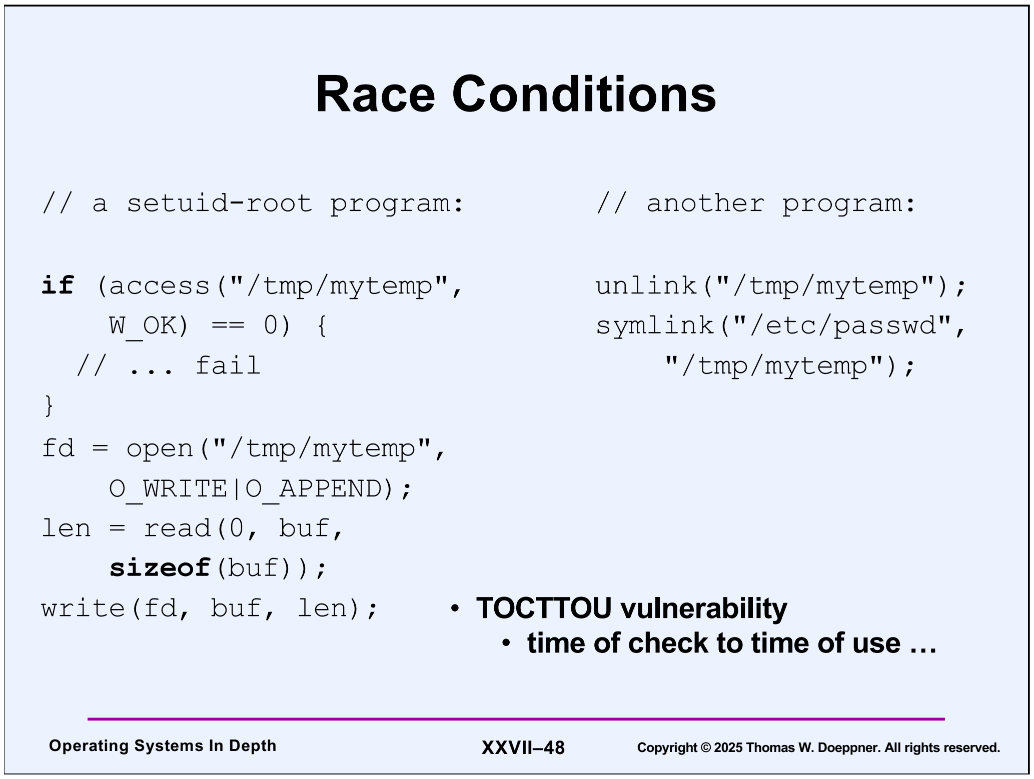
- *Real* user and group IDs — usually used to identify who created the process
- *Effective* user and group IDs — used to determine access rights to files
- Saved user and group IDs — hold the initial effective user and group IDs established at the time of the *exec*, allowing one to revert back to them

Exec

- Normally the real and effective IDs are the same
  - they are copied to the child from the parent during a *fork*
- *exec*s done on files marked *setuid* or *setgid* change this
  - if the file is marked *setuid*, then the effective and saved user IDs become the ID of the owner of the file
  - if the file is marked *setgid*, then the effective and saved group IDs become the ID of the group of the file

Exercise of Powers

- Permission to access a file depends on a process's effective IDs
  - the *access* system call checks permissions with respect to a process's real IDs
    - this allows *setuid*/*setgid* programs to determine the privileges of their invokers
- The *kill* system call makes use of both forms of user ID; for process *A* to send a signal to process *B*, one of the following must be true:
  - *A*'s real user ID is the same as *B*'s real or saved user ID
  - *A*'s effective user ID is the same as *B*'s real or saved user ID
  - *A*'s effective user ID is 0



Race Conditions

```
// a setuid-root program:

if (access("/tmp/mytemp",
    W_OK) == 0) {
  // ... fail
}
fd = open("/tmp/mytemp",
    O_WRITE|O_APPEND);
len = read(0, buf,
    sizeof(buf));
write(fd, buf, len);
```

```
// another program:

unlink("/tmp/mytemp");
symlink("/etc/passwd",
    "/tmp/mytemp");
```

- **TOCTTOU vulnerability**
  - **time of check to time of use …**

The intent of this program is to copy data read from stdin and append it to the end of /tmp/mytemp. But with careful timing and some luck, one can replace /tmp/mytemp with a symbolic link that refers to /etc/passwd, and thus one can add data to the end of the system’s password file.

Note that superuser privileges aren’t necessary in the code of the slide – assume that this code is an excerpt from code that requires superuser privileges.