CS 33

Files Part 2
Directories

unix  etc  home  pro  dev

passwd  motd

unix  ...

slide1  slide2
## Directory Representation

<table>
<thead>
<tr>
<th>Component Name</th>
<th>Inode Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>directory entry</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>1</td>
</tr>
<tr>
<td>..</td>
<td>1</td>
</tr>
<tr>
<td>unix</td>
<td>117</td>
</tr>
<tr>
<td>etc</td>
<td>4</td>
</tr>
<tr>
<td>home</td>
<td>18</td>
</tr>
<tr>
<td>pro</td>
<td>36</td>
</tr>
<tr>
<td>dev</td>
<td>93</td>
</tr>
</tbody>
</table>
Hard Links

```
$ ln /unix /etc/image
# link system call
```
## Directory Representation

<table>
<thead>
<tr>
<th>Directory</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>1</td>
</tr>
<tr>
<td>..</td>
<td>1</td>
</tr>
<tr>
<td>unix</td>
<td>117</td>
</tr>
<tr>
<td>etc</td>
<td>4</td>
</tr>
<tr>
<td>home</td>
<td>18</td>
</tr>
<tr>
<td>pro</td>
<td>36</td>
</tr>
<tr>
<td>dev</td>
<td>93</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Directory</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>4</td>
</tr>
<tr>
<td>..</td>
<td>1</td>
</tr>
<tr>
<td>image</td>
<td>117</td>
</tr>
<tr>
<td>motd</td>
<td>33</td>
</tr>
</tbody>
</table>
Symbolic Links

% ln -s /unix /home/twd/mylink
% ln -s /home/twd /etc/twd
# symlink system call
Working Directory

• Maintained in kernel for each process
  – paths not starting from “/” start with the working directory
  – changed by use of the chdir system call
    » cd shell command
  – displayed (via shell) using “pwd”
    » how is this done?
Quiz 1

What is the working directory after doing
```
cd /etc/twd/.. ?
```

a) /  
b) /etc  
c) /home  
d) /home/twd
Open

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *path, int options [, mode_t mode])

- options
  » O_RDONLY  open for reading only
  » O_WRONLY  open for writing only
  » O_RDWR    open for reading and writing
  » O_APPEND  set the file offset to end of file prior to each write
  » O_CREAT   if the file does not exist, then create it, setting its mode to mode adjusted by umask
  » O_EXCL    if O_EXCL and O_CREAT are set, then open fails if the file exists
  » O_TRUNC   delete any previous contents of the file
  » O_NONBLOCK don’t wait if I/O can’t be done immediately
```
File Access Permissions

• Who's allowed to do what?
  – who
    » user (owner)
    » group
    » others (rest of the world)
  – what
    » read
    » write
    » execute
Permissions Example

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

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

./B:
 total 2
 -r--rw-rw-  1 tom adm  446 Dec 17 13:34 x
 -rw-----rw-  1 trina adm  446 Dec 17 13:45 y
Setting File Permissions

```c
#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char *path, mode_t mode)
```

- sets the file permissions of the given file to those specified in `mode`
- only the owner of a file and the superuser may change its permissions
- nine combinable possibilities for `mode`
  (read/write/execute for user, group, and others)
  
  » S_IRUSR (0400), S_IWUSR (0200), S_IXUSR (0100)
  » S_IRGRP (040), S_IWGRP (020), S_IXGRP (010)
  » S_IROTH (04), S_IWOTH (02), S_IXOTH (01)
Umask

- Standard programs create files with “maximum needed permissions” as mode
  - compilers: 0777
  - editors: 0666
- Per-process parameter, `umask`, used to turn off undesired permission bits
  - e.g., turn off all permissions for others, write permission for group: set umask to 027
    » compilers: permissions = 0777 & ~(027) = 0750
    » editors: permissions = 0666 & ~(027) = 0640
  - set with `umask` system call or (usually) shell command
Creating a File

• Use either *open* or *creat*
  – *open*(const char *pathname, int flags, mode_t mode)
    » flags must include O_CREAT
  – *creat*(const char *pathname, mode_t mode)
    » open is preferred

• The *mode* parameter helps specify the permissions of the newly created file
  – permissions = mode & ~umask
Link and Reference Counts

int fd = open("n1", O_RDONLY);
// n1’s reference count
// incremented by 1
Link and Reference Counts

```
int fd = open("n1", O_RDONLY);
   // n1’s reference count
   // incremented by 1

unlink("n1");
   // link count decremented by 1
   // same effect in shell via “rm n1”
```
Link and Reference Counts

```
int fd = open("n1", O_RDONLY);
    // n1’s reference count
    // incremented by 1

unlink("n1");
    // link count decremented by 1

close(fd);
    // reference count decremented by 1
```
Link and Reference Counts

```c
int fd = open("n1", O_RDONLY);
    // n1's reference count
    // incremented by 1

unlink("n1");
    // link count decremented by 1

close(fd);
    // reference count decremented by 1
```
Link and Reference Counts

unlink("dir1/n2");
// link count decremented by 1
int main() {
    int fd = creat("file", 0666);
    unlink("file");
    PutStuffInFile(fd);
    ReadStuffFromFile(fd);
    return 0;
}

Assume that \textit{PutStuffInFile} writes to the given file, and \textit{ReadStuffFromFile} reads from the file.

\begin{itemize}
\item[a)] This program is doomed to failure, since the file is deleted before it’s used
\item[b)] Because the file is used after the unlink call, it won’t be deleted
\item[c)] The file will be deleted when the program terminates
\end{itemize}
Interprocess Communication (IPC): Pipes
Interprocess Communication: Same Machine I

Kernel buffer
Interprocess Communication: Same Machine II

process 1

process 2

Shared Memory
Interprocess Communication: Different Machines
Intramachine IPC

\$cslab2e \text{ who} \mid \text{ wc -l}
Intramachine IPC

```c
int fd[2];
pipe(fd);
if (fork() == 0) {
    close(fd[0]);
    close(1);
    dup(fd[1]); close(fd[1]);
    execl("/usr/bin/", "who", 0); // who sends output to pipe
}
if (fork() == 0) {
    close(fd[1]);
    close(0);
    dup(fd[0]); close(fd[0]);
    execl("/usr/bin/wc", "wc", "-l", 0); // wc’s input is from pipe
}
close(fd[1]); close(fd[0]);
// ...
```

```bash
$cslab2e who | wc -l
```
Intermachine Communication

• Can pipes be made to work across multiple machines?
  – covered soon ...
    » what happens when you type
      
```bash
who | ssh cslab3a wc -l
```

?
Sharing Files

• You’re doing a project with a partner
• You code it as one 15,000-line file
  – the first 7,500 lines are yours
  – the second 7,500 lines are your partner’s
• You edit the file, changing 6,000 lines
  – it’s now 5am
• Your partner completes her changes at 5:01am
• At 5:02am you look at the file
  – your partner’s changes are there
  – yours are not
Lessons

• Never work with a partner
• Use more than one file
• Read up on git
• Use an editor and file system that support file locking
What We Want ...

I want to just read the file.

Me too.

I want to modify the file.

Me too.
Types of Locks

• Shared (readers) locks
  – any number may have them at the same time
  – may not be held when an exclusive lock is held

• Exclusive (writers) locks
  – only one at a time
  – may not be held when a shared lock is held
What We Want ...

I’ve got a shared lock.

Me too.

My exclusive lock request was rejected.

Mine too.
What We Want ...

I’ve got an exclusive lock.

My shared request was rejected.

My exclusive request was rejected.

Mine too.
Locking Files

• Early Unix didn’t support file locking
• How did people survive?
  
  - `open("file.lck", O_RDWR|O_CREAT|O_EXCL, 0666);`
  
   » operation fails if `file.lck` exists, succeeds (and creates `file.lck`) otherwise
  
   » requires cooperative programs
Locking Files (continued)

• How it’s done in “modern” Unix
  – “advisory locks” may be placed on files
    » may request shared (readers) or exclusive (writers) lock
    • `fcntl` system call
      » either succeeds or fails
      » `open`, `read`, `write` always work, regardless of locks
      » a lock applies to a specified range of bytes, not necessarily the whole file
      » requires cooperative programs
Locking Files  (still continued)

• How to:

    struct flock fl;
    fl.l_type = F_RDLCK;    // read lock
    // fl.l_type = F_WRLCK;  // write lock
    // fl.l_type = F_UNLCK;  // unlock
    fl.l_whence = SEEK_SET; // starting where
    fl.l_start = 0;        // offset
    fl.l_len = 0;          // how much? (0 = whole file)
    fd = open("file", O_RDWR);
    if (fcntl(fd, F_SETLK, &fl) == -1)  
        if ((errno == EACCES) || (errno == EAGAIN))  
            // didn’t get lock
        else  
            // something else is wrong
    else
        // got the lock!
Locking Files (yet still continued)

- Making locks mandatory:
  - if the file’s permissions have group execute permission off and set-group-ID on, then locking is enforced
    » *read, write* fail if file is locked by someone other than the caller
  - however …
    » doesn’t work on NFSv3 or earlier
  - (we run NFSv3 at Brown CS)
Quiz 3

• Your program currently has a shared lock on a portion of a file. It would like to “upgrade” the lock to be an exclusive lock. Would there be any problems with adding an option to `fcntl` that would allow the holder of a shared lock to wait until it’s possible to upgrade to an exclusive lock, then do the upgrade?

1) at least one major problem
2) either no problems whatsoever or some easy-to-deal-with problems