

CS 33

Input/Output Miscellany

File Copying

```
int in = open("InFile", O_RDONLY);
int out = open("OutFile", O_CREAT|O_WRONLY|O_TRUNC, 0666);
int count;

while ((count = read(in, buffer, 8192)) > 0)
    write(out, buffer, count);
```

Here's a simple program showing how a file might be copied.

Faster File Copying?

```
int in = open("InFile", O_RDONLY);
int out = open("OutFile", O_CREAT|O_WRONLY|O_TRUNC, 0666);
struct stat sbuf;

fstat(in, &sbuf);

void *src = mmap(0, sbuf.st_size, PROT_READ,
                 MAP_SHARED, in, 0);

write(out, src, sbuf.st_size);
```

Might this be faster than the program of the previous slide? Note that the *fstat* system call obtains information about a file, including its size. The argument *sbuf* is a pointer to a *struct stat*, which contains much information about a file. See the man page for *fstat* for details (i.e., use the command “man 2 fstat”).

Even Faster File Copying?

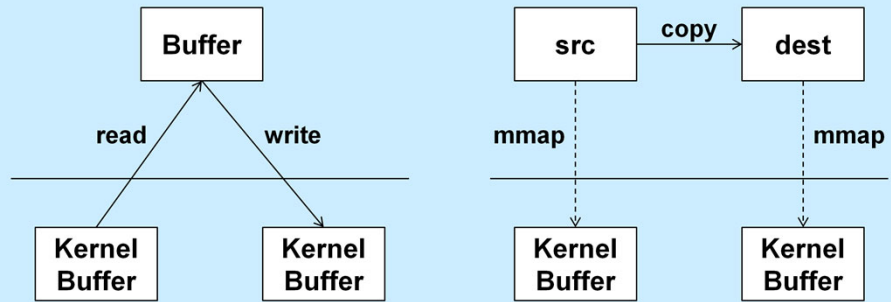
```
int in = open("InFile", O_RDONLY);
int out = open("OutFile", O_CREAT|O_RDWR|O_TRUNC, 0666);
struct stat sbuf;

fstat(in, &sbuf);
void *src= mmap(0, sbuf.st_size, PROT_READ,
               MAP_SHARED, in, 0);
lseek(out, sbuf.st_size-1, SEEK_SET);
write(out, "x", 1);
void *dest= mmap(0, sbuf.st_size, PROT_READ|PROT_WRITE,
                MAP_SHARED, out, 0);

memcpy(dest, src, sbuf.st_size);
```

In our final attempt to speed up file copying, we map both the input file and the output file into the address space, then use `memcpy`, a library routine that copies data as quickly as possible. One restriction on mapping the output file: the file must be at least as big as the area of virtual memory it is mapped into. To make it that big, we write one byte at location `s` in the file, where `s` is the size of the input file. A property of Unix file systems is that if you write to some location in an otherwise empty file, then all locations in the file before the one written to appear to contain zeroes. To write to an arbitrary position within a file, one uses the `lseek` system call, which positions the file pointer, setting the location in the file where the next read or write takes place. See the man page for `lseek` for details.

Read/Write vs. Mmap



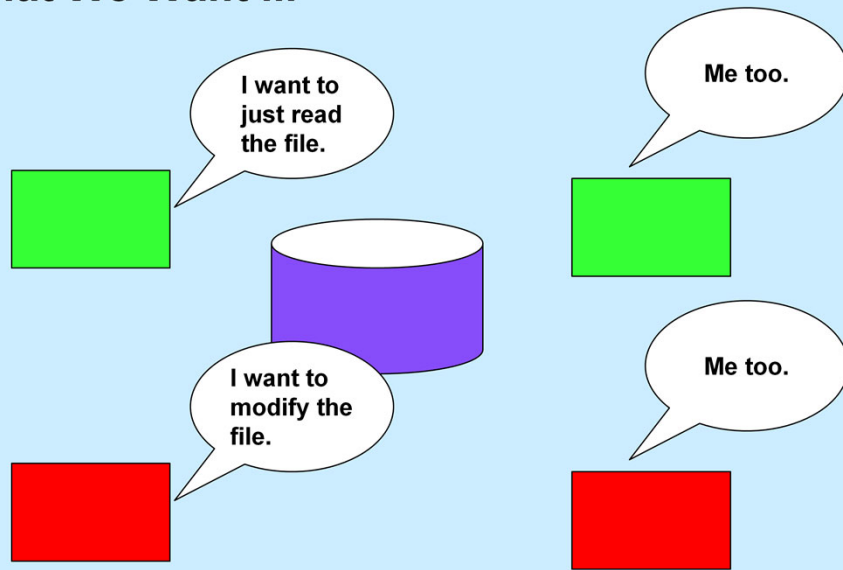
Sharing Files

- 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**

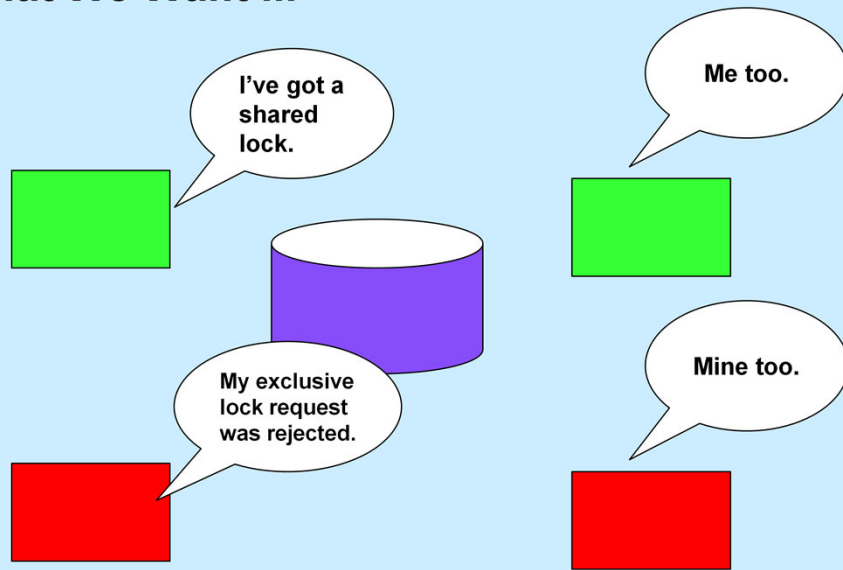
What We Want ...



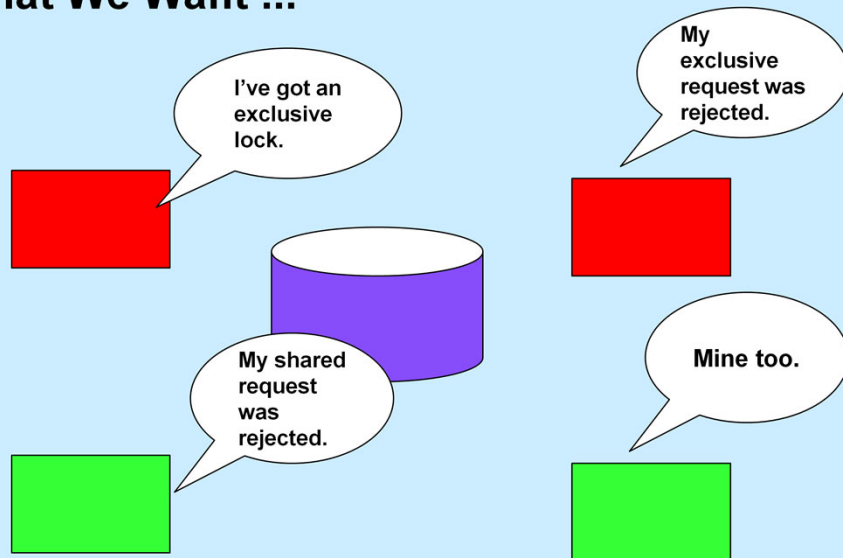
Types of Locks

- **Shared (readers) locks**
 - any number may have them at 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 ...



What We Want ...



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
 - don't ask: no problem
 - » 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!
```

Alternatively, one may use `l_type` values of `F_RDLCKW` and `F_WRLCKW` to wait until the lock may be obtained, rather than to return an error if it can't be obtained.

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)