CS 33

More Libraries
Pipes
Locking Files

A Problem

- You've put together a library of useful functions
  - libgoodstuff.so
- Lots of people are using it
- It occurs to you that you can make it even better by adding an extra argument to a few of the functions
  - doing so will break all programs that currently use these functions
- You need a means so that old code will continue to use the old version, but new code will use the new version

A Solution

- The two versions of your program coexist
  - libgoodstuff.so.1
  - libgoodstuff.so.2
- You arrange so that old code uses the old version, new code uses the new
- Most users of your code don't really want to have to care about version numbers
  - they want always to link with libgoodstuff.so
  - and get the version that was current when they wrote their programs

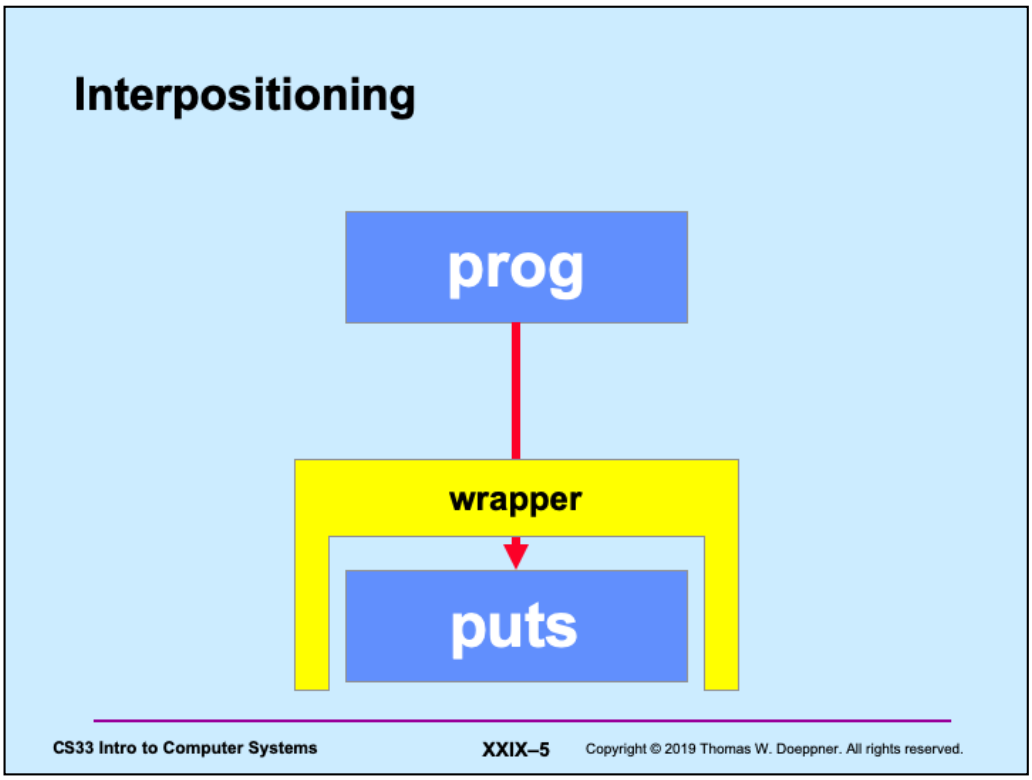
Versioning

```
$ gcc -fPIC -c goodstuff.c
$ ld -shared -soname libgoodstuff.so.1 \
-o libgoodstuff.so.1 goodstuff.o
$ ln -s libgoodstuff.so.1 libgoodstuff.so
$ gcc -o prog1 prog1.c -L. -lgoodstuff \
-Wl,-rpath .
$ vi goodstuff.c
$ gcc -fPIC -c goodstuff.c
$ ld -shared -soname libgoodstuff.so.2 \
-o libgoodstuff.so.2 goodstuff.o
$ rm -f libgoodstuff.so
$ ln -s libgoodstuff.so.2 libgoodstuff.so
$ gcc -o prog2 prog2.c -L. -lgoodstuff \
-Wl,-rpath .
```


Here we are creating two versions of libgoodstuff, in libgoodstuff.so.1 and in libgoodstuff.so.2. Each is created by invoking the loader directly via the "ld" command. The "-soname" flag tells the loader to include in the shared object its name, which is the string following the flag ("libgoodstuff.so.1" in the first call to ld). The effect of the "ln –s" command is to create a new name (its last argument) in the file system that refers to the same file as that referred to by ln's next-to-last argument. Thus, after the first call to ln –s, libgoodstuff.so refers to the same file as does libgoodstuff.so.1. Thus the second invocation of gcc, where it refers to –lgoodstuff (which expands to libgoodstuff.so), is actually referring to libgoodstuff.so.1.

Then we create a new version of goodstuff and from it a new shared object called libgoodstuff.so.2 (i.e., version 2). The call to "rm" removes the name libgoodstuff.so (but not the file it refers to, which is still referred to by libgoodstuff.so.1). Then ln is called again to make libgoodstuff.so now refer to the same file as does libgoodstuff.so.2. Thus when prog2 is linked, the reference to –lgoodstuff expands to libgoodstuff.so, which now refers to the same file as does libgoodstuff.so.2.

If prog1 is now run, it refers to libgoodstuff.so.1, so it gets the old version (version 1), but if prog2 is run, it refers to libgoodstuff.so.2, so it gets the new version (version 2). Thus programs using both versions of goodstuff can coexist.



You'd like to have a "wrapper" that's called when programs call puts: they call your wrapper, then your wrapper calls the real puts. If you name your wrapper "puts", then if programs call puts, they'll call your wrapper. But how does your wrapper call the real puts, when its name is wrapper?

How To ...

```
int __wrap_puts(const char *s) {
  int __real_puts(const char *);

  write(2, "calling myputs: ", 16);
  return __real_puts(s);
}
```


__wrap_puts is the "wrapper" for *puts*. *__real_puts* is the "real" *puts* routine. What we want is for calls to *puts* to go to *__wrap_puts*, and calls to *__real_puts* to go to the real *puts* routine (in stdio).

Compiling/Linking It

```
$ cat tputs.c
int main() {
  puts("This is a boring message.");
  return 0;
}
$ gcc -o tputs -Wl,--wrap=puts tputs.c myputs.c
$ ./tputs
calling myputs: This is a boring message.
$
```


The arguments to gcc shown in the slide cause what we asked for in the previous slide to actually happen. Calls to *puts* go to *__wrap_puts*, and calls to *__real_puts* go to the real *puts* routine.

How To (Alternative Approach) …

```
#include <dlfcn.h>

int puts(const char *s) {
  int (*pptr)(const char *);

  pptr = (int(*)())dlsym(RTLD_NEXT, "puts");

  write(2, "calling myputs: ", 16);
  return (*pptr)(s);
}
```


An alternative approach to wrapping is to invoke ld-linux.so directly from the program, and have it find the real puts routine. The call to dlsym above directly invokes ld-linux.so, asking it (as given by the first argument) to find the next definition of puts in the list of libraries. It returns the location of that routine, which is then called (*pptr).

What's Going On …

- gcc/ld
  - compiles code
  - does static linking
    - » searches list of libraries
    - » adds references to shared objects
- runtime
  - program invokes *ld-linux.so* to finish linking
    - » maps in shared objects
    - » does relocation and procedure linking as required
  - *dlsym* invokes *ld-linux.so* to do more linking
    - » RTLD_NEXT says to use the next (second) occurrence of the symbol

Delayed Wrapping

- LD_PRELOAD
  - environment variable checked by *ld-linux.so*
  - specifies additional shared objects to search (first) when program is started

Environment Variables

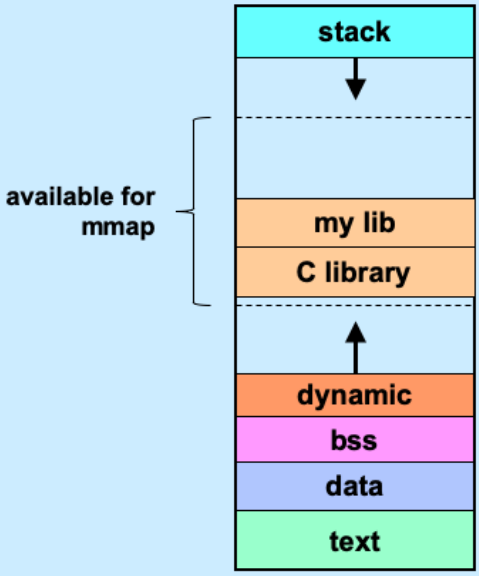
- Another form of exec
  - `int execve(const char *filename, char *const argv[], char *const envp[]);`
- envp is an array of strings, of the form
  - key=value
- programs can search for values, given a key
- example
  - PATH=~/bin:/bin:/usr/bin:/course/cs0330/bin

Example

```
$ gcc -o tputs tputs.c
$ ./tputs
This is a boring message.
$ LD_PRELOAD=./libmyputs.so.1; export LD_PRELOAD
$ ./tputs
calling myputs: This is a boring message.
$
```

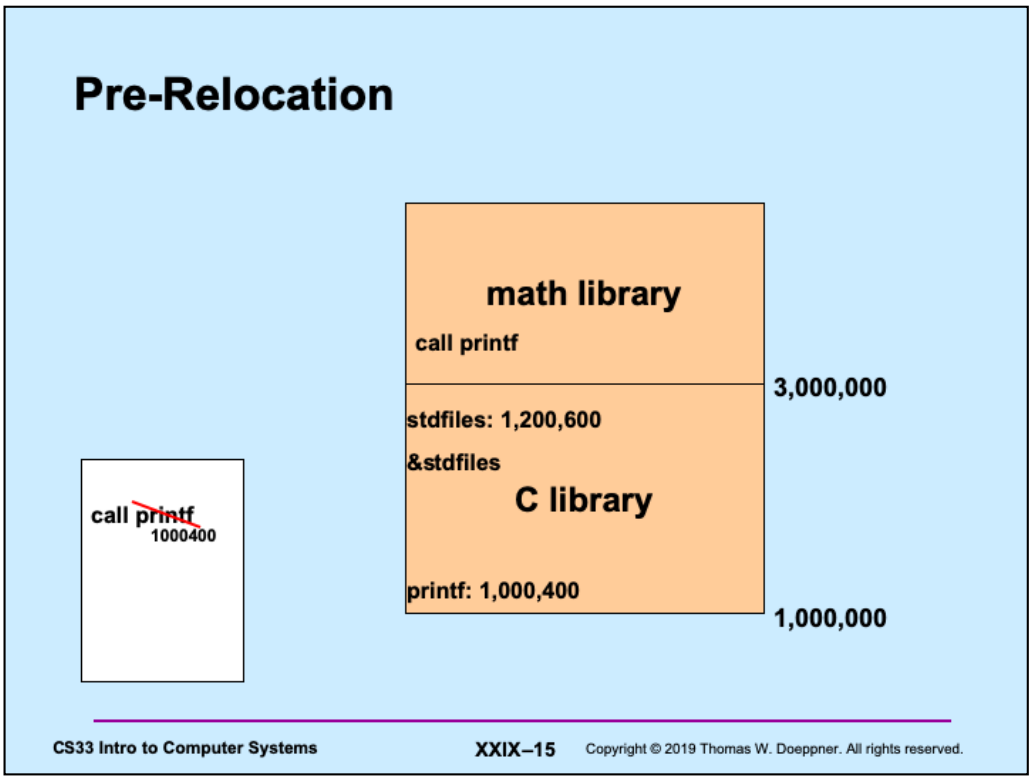

Here we add "LD_PRELOAD=./libmyputs.so.1" to the environment. The export command instructs the shell to supply this as part of the environment for the commands it runs.

Mmapping Libraries

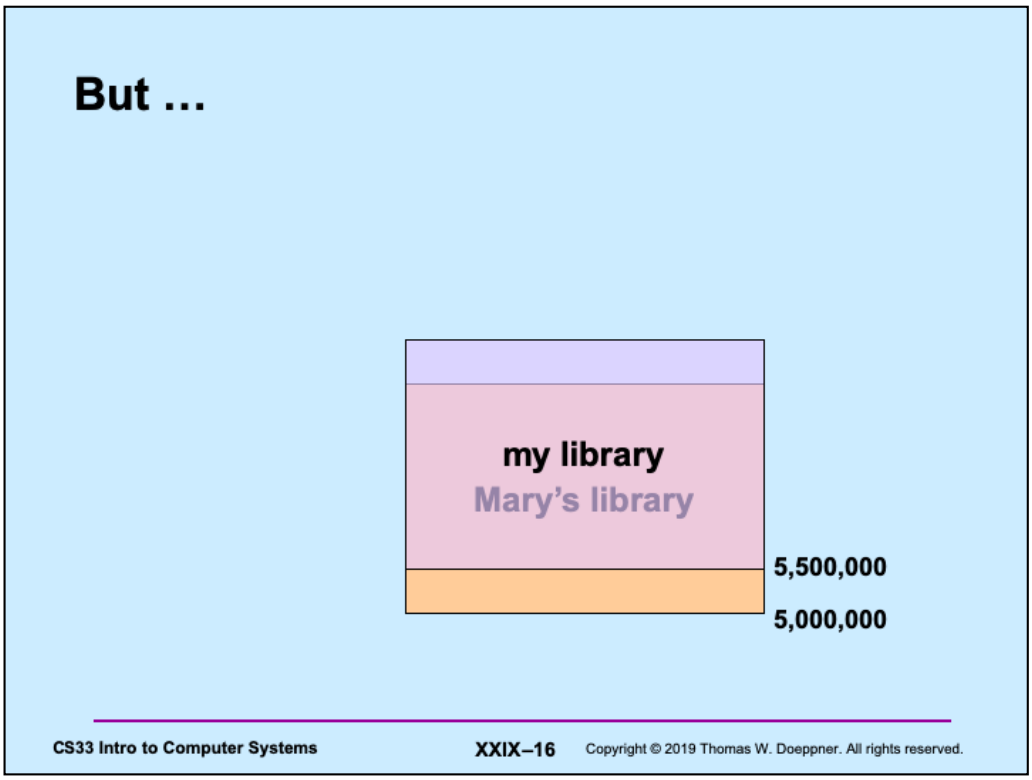


Problem

- How is relocation handled?

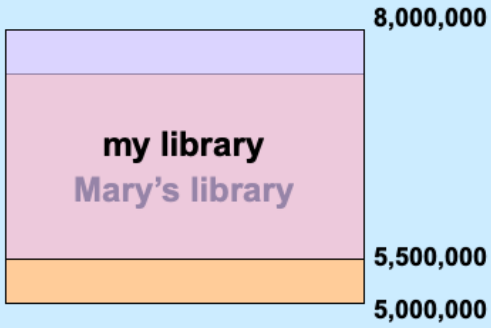


Assuming we're using pre-relocation, the C library and the math library would be assumed to be in virtual memory at their pre-assigned locations. In the slide, these would be starting at locations 1,000,000 and 3,000,000, respectively. Let's suppose printf, which is in the C library, is at location 1,000,400. Thus calls to printf at static link time could be linked to that address. If the math library also contains calls to printf, these would be linked to that address as well. The C library might contain a global identifier, such as stdfiles. Its address would also be known.



Pre-relocation doesn't work if we have two libraries pre-assigned such that they overlap. If so, at least one of the two will have to be moved, necessitating relocation.

But …



Quiz 1

We need to relocate all references to Mary's library in *my library*. What option should we give to *mmap* when we map *my library* into our address space?

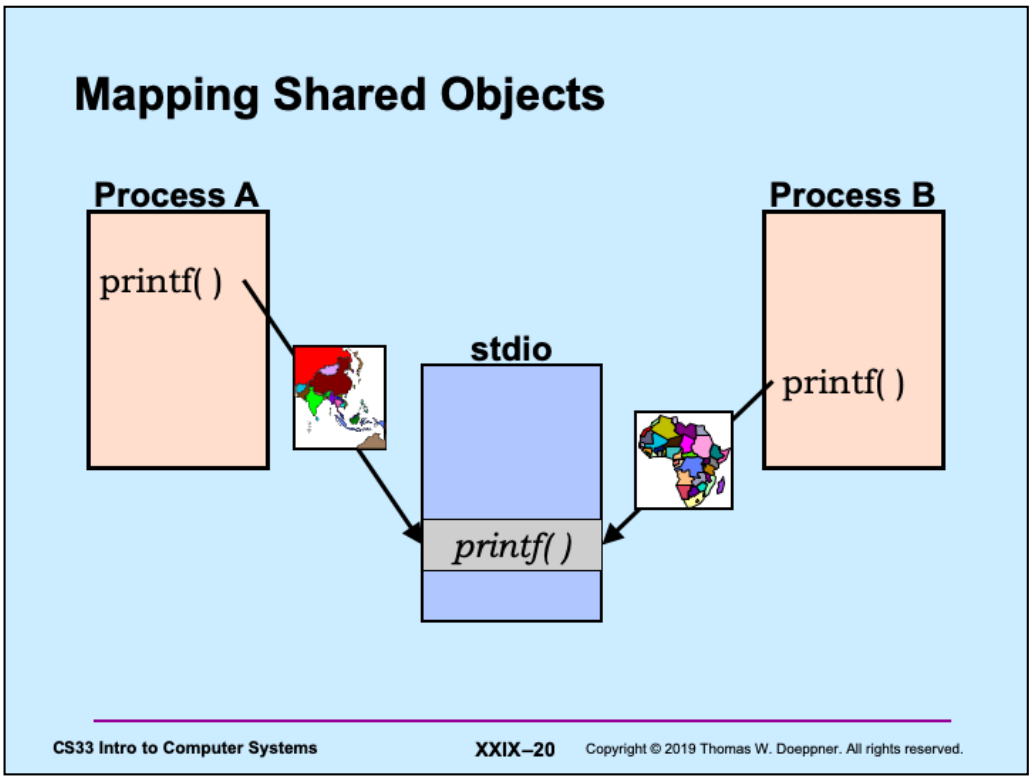
a) **the MAP_SHARED option**

b) **the MAP_PRIVATE option**

c) **mmap can't be used in this situation**

Relocation Revisited

- Modify shared code to effect relocation
  - result is no longer shared!
- Separate shared code from (unshared) addresses
  - position-independent code (PIC)
  - code can be placed anywhere
  - addresses in separate private section
    - » pointed to by a register



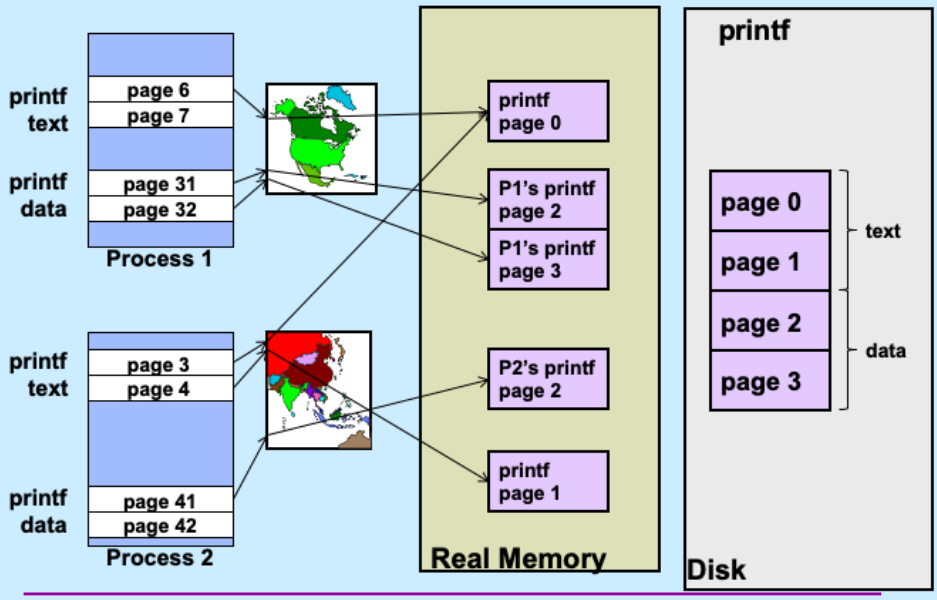
The C library (and other libraries) can be mapped into different locations in different processes' address spaces.

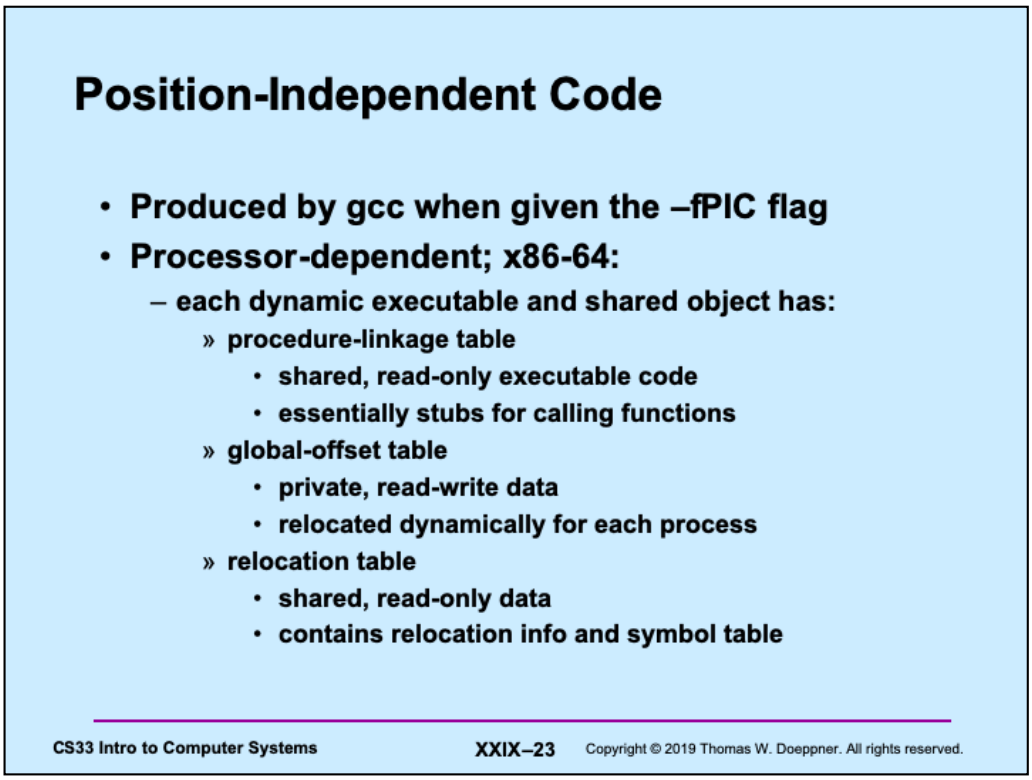
Mapping printf into the Address Space

- Printf's text
  - read-only
  - can it be shared?
    - » yes: use MAP_SHARED
- Printf's data
  - read-write
  - not shared with other processes
  - initial values come from file
  - can mmap be used?
    - » MAP_SHARED wouldn't work
      - changes made to data by one process would be seen by others
    - » MAP_PRIVATE does work!
      - mapped region is initialized from file
      - changes are private

For this slide, we assume relocation is dealt with through the use of position-independent code (PIC).

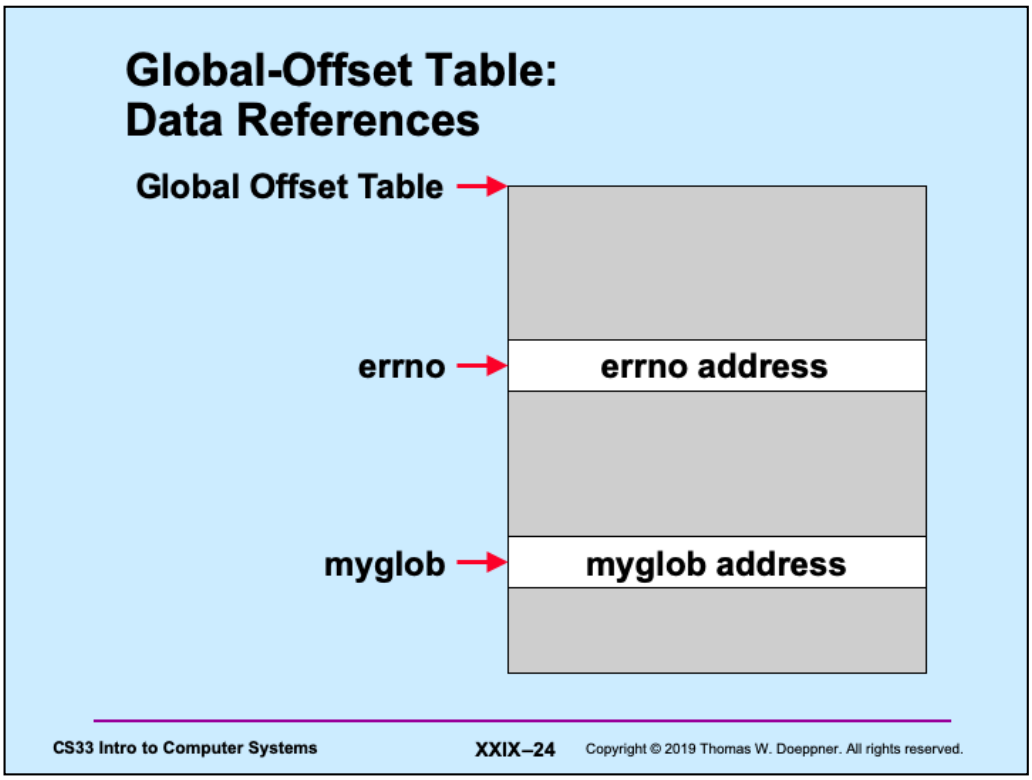
Mapping printf





To provide position-independent code on x86-64, ELF requires three data structures for each dynamic executable (i.e., the program binary loaded by *exec*) and shared object: the *procedure-linkage table*, the *global-offset table*, and the *relocation table*. To simplify discussion, we refer to dynamic executables and shared objects as *modules*. The procedure linkage table contains the code that's actually called when control is to be transferred to an externally defined routine. It is shared by all processes using the associated executable or object, and makes use of data in the global-object table to link the caller to the called program. Each process has its own private copy of each global-object table. It contains the relocated addresses of all externally defined symbols. Finally, the relocation table contains much information about each module. What is used for linking is relocation information and the symbol table, as we explain in the next few slides.

How things work is similar for other architectures, but definitely not the same.



To establish position-independent references to global variables, the compiler produces, for each module, a *global-offset table*. Modules refer to global variables indirectly by looking up their addresses in the table, using PC-relative addressing. The item needed is at some fixed offset from the beginning of the table. When the module is loaded into memory, ld-linux.so is responsible for putting into it the actual addresses of all the needed global variables.

Functions in Shared Objects

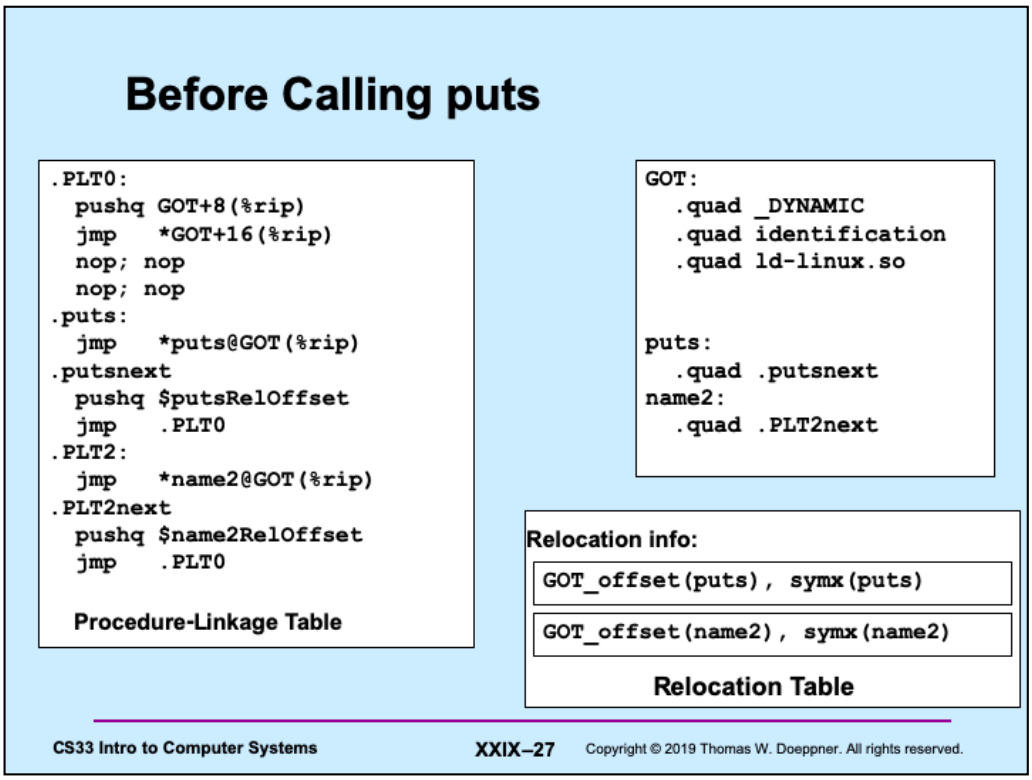
- Lots of them
- Many are never used
- Fix up linkages on demand

An Example

```
int main( ) {
   puts("Hello world\n");
   …
   return 0;
}
```

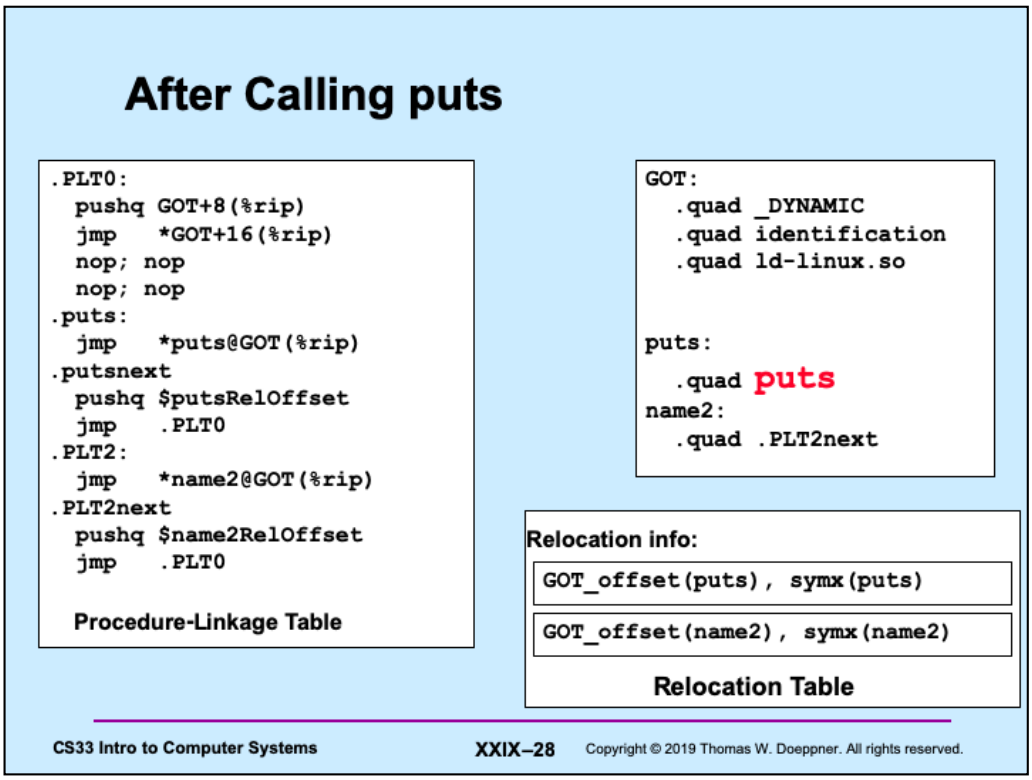
```
00000000000006b0 <main>:
 6b0: 55                      push   %rbp
 6b1: 48 89 e5                mov    %rsp,%rbp
 6b4: 48 8d 3d 99 00 00 00    lea    0x99(%rip),%rdi
 6bb: e8 a0 fe ff ff          callq  560 <puts@plt>
 …
```


The top half of the slide contains an excerpt from a C program. For the bottom half, we've compiled the program and have printed what "objdump –d" produces for main. Note that the call to puts is actually a call to "puts@plt", which is a reference to the procedure linkage table.

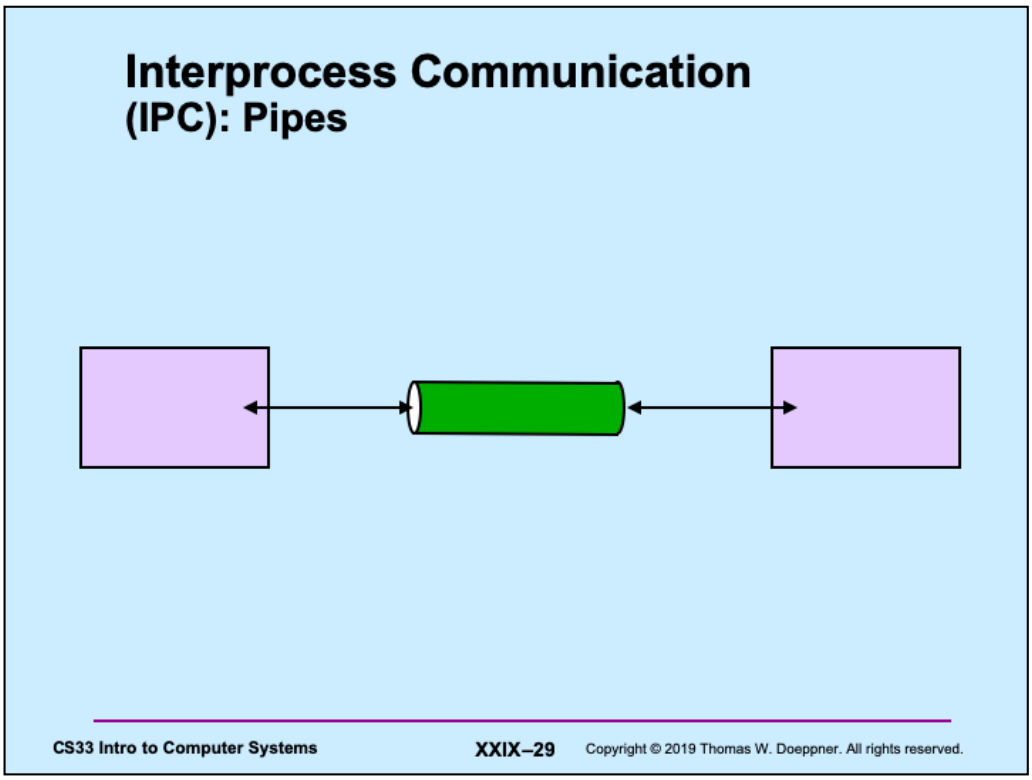


Dealing with references to external procedures is considerably more complicated than dealing with references to external data. This slide shows the procedure linkage table, global offset table, and relocation information for a module that contains references to external procedures *puts* and *name2*. Let's follow a call to procedure *puts*. The general idea is before the first call to puts, the actual address of the puts procedure is not recorded in the global-offset table, Instead, the first call to puts actually invokes ld-linux.so, which is passed parameters indicating what is really wanted. It then finds puts and updates the global-offset table so that things are more direct on subsequent calls.

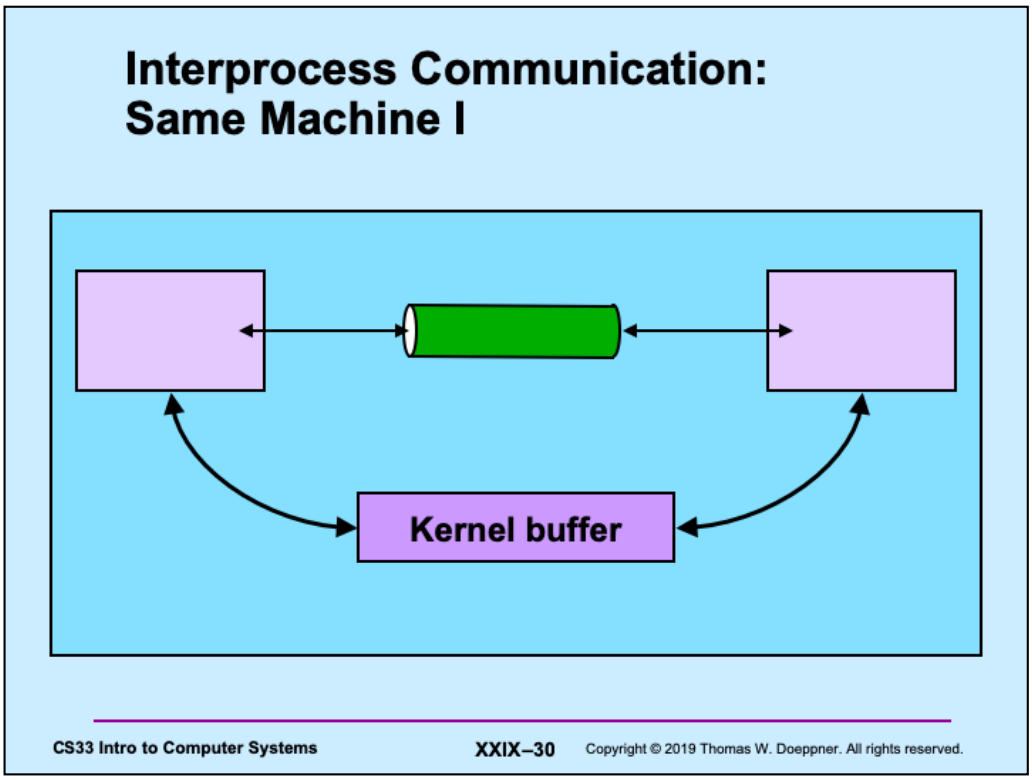
To make this happen, references from the module to *puts* are statically linked to entry .puts in the procedure-linkage table. This entry contains an unconditional jump (via PC-relative addressing) to the address contained in the *puts* offset of the global-offset table. Initially this address is of the instruction following the jump instruction, which contains code that pushes onto the stack the offset of the puts entry in the relocation table. The next instruction is an unconditional jump to the beginning of the procedure-linkage table, entry .PLT0. Here there's code that pushes onto the stack the second 64-bit word of the global-offset table, which contains a value identifying this module. The following instruction is an unconditional jump to the address in the third word of the global-offset table, which is conveniently the address of ld-linux.so. Thus control finally passes to ld-linux.so, which looks back on the stack and determines which module has called it and what that module really wants to call. It figures this out based on the module-identification word and the relocation table entry, which contains the offset of the puts entry in the global-offset table (which is what must be updated) and the index of *puts* in the symbol table (so it knows the name of what it must locate).



Finally, ld-linux.so writes the actual address of the *puts* procedure into the puts entry of the global-offset table, and, after unwinding the stack a bit, passes control to *puts*. On subsequent calls by the module to puts, since the global-offset table now contains puts's address, control goes to it more directly, without an invocation of ld-linux.so.

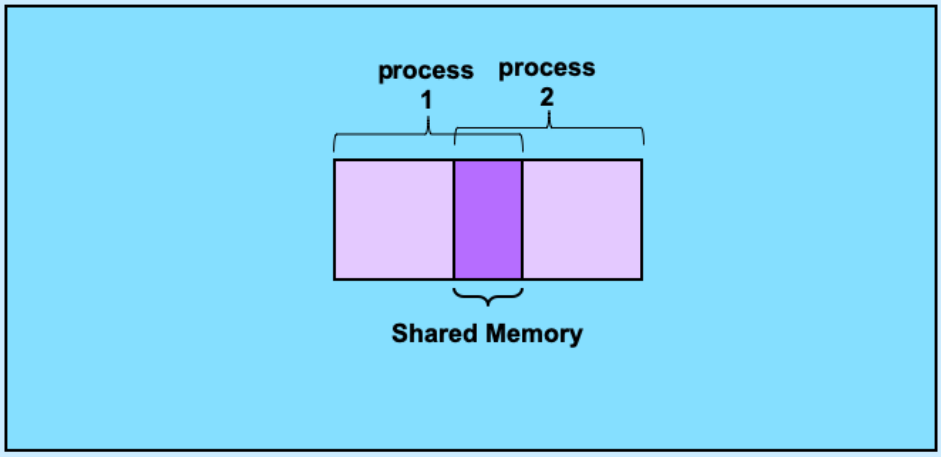


A rather elegant way for different processes to communicate is via a pipe: one process puts data into a pipe, another process reads the data from the pipe.

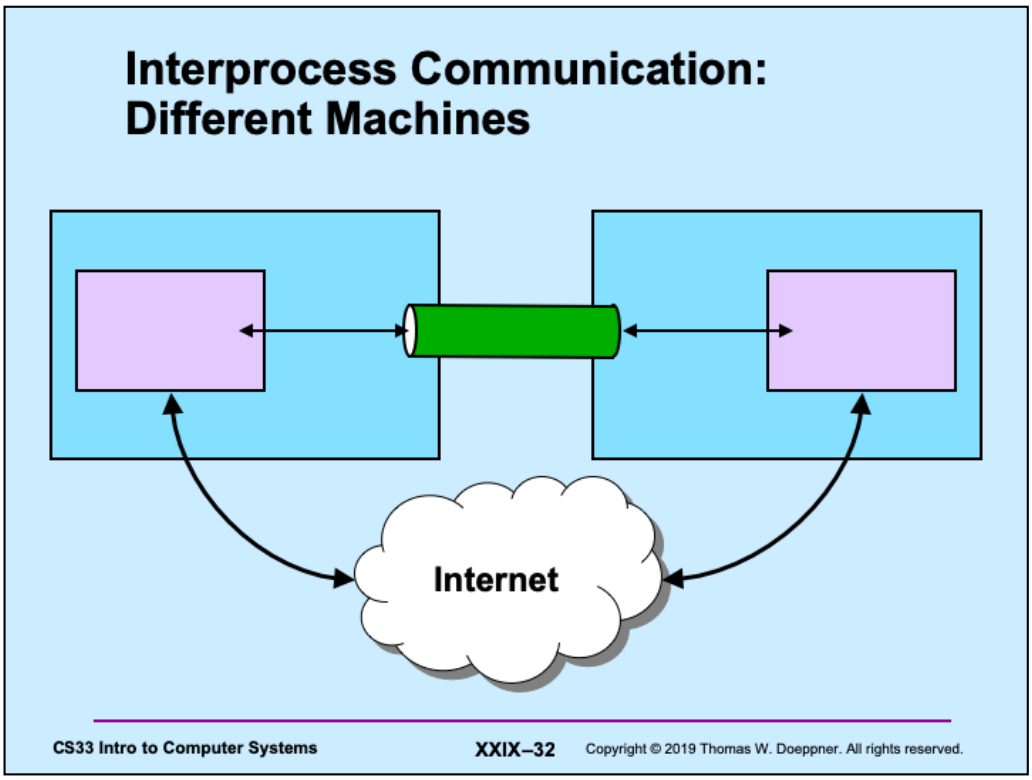


The implementation of a pipe involves the sending process using a write system call to transfer data into a kernel buffer. The receiving process fetches the data from the buffer via a read system call.

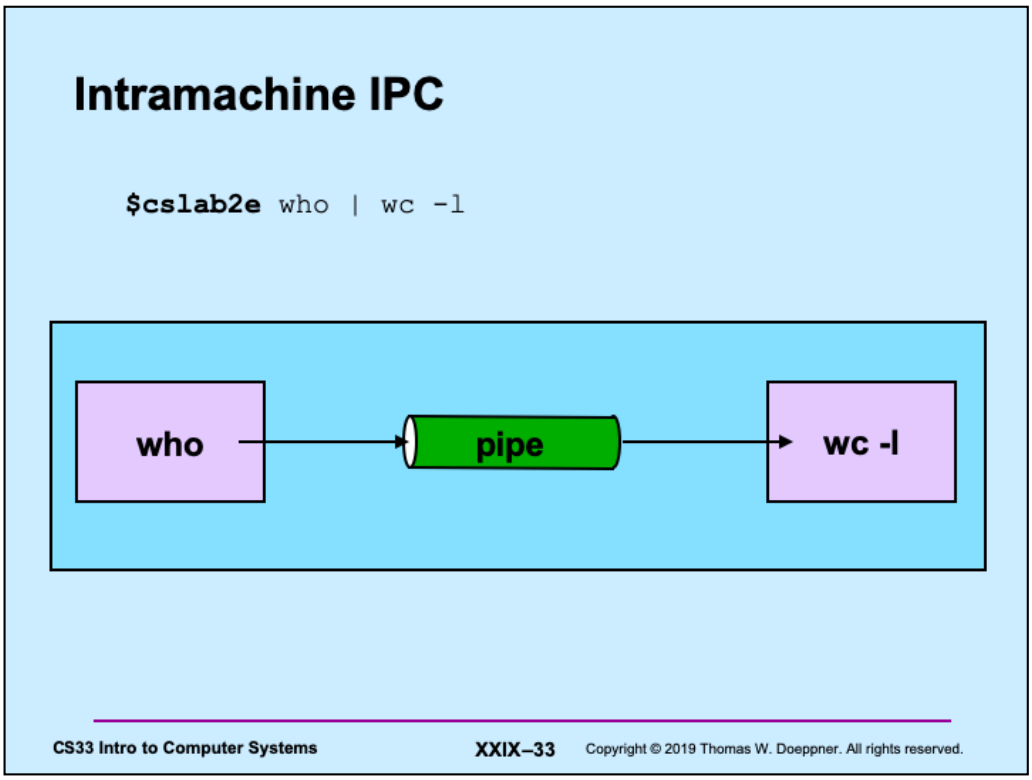
Interprocess Communication: Same Machine II



Another way for processes to communicate is for them to arrange to have some memory in common via which they share information.



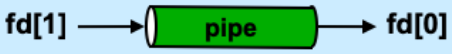
The pipe abstraction can also be made to work between processes on different machines.



The vertical bar ("|") is the pipe symbol in the shell. The syntax shown above represents creating two processes, one running who and the other running wc. The standard output of who is setup to be the pipe; the standard input of wc is setup to be the pipe. Thus the output of who becomes the input of wc. The "-l" argument to wc tells it to count and print out the number of lines that are input to it. The who command writes to standard output the login names of all logged in users. The combination of the two produces the number of users who are currently logged in.

Intramachine IPC

```
$cslab2e who | wc -l
```




```
int fd[2];
pipe(fd);
if (fork() == 0) {
  close(fd[0]);
  close(1);
  dup(fd[1]); close(fd[1]);
  execl("/usr/bin/who", "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]);
// …
```


The *pipe* system call creates a "pipe" in the kernel and sets up two file descriptors. One, in fd[1], is for writing to the pipe; the other, in fd[0], is for reading from the pipe. The input end of the pipe is set up to be *stdout* for the process running *who*, and the output end of the pipe is closed, since it's not needed. Similarly, the input end of the pipe is set up to be *stdin* for the process running *wc*, and the input end is closed. Since the parent process (running the shell) has no further need for the pipe, it closes both ends. When neither end of the pipe is open by any process, the system deletes it. If a process reads from a pipe for which no process has the input end open, the read returns 0, indicating end of file. If a process writes to a pipe for which no process has the output end open, the write returns -1, indicating an error and *errno* is set to EPIPE; the process also receives the SIGPIPE signal.

Intermachine Communication

- Can pipes be made to work across multiple machines?
  - covered soon ...
    - » what happens when you type

```
who | ssh cslab3a wc -l
```

?

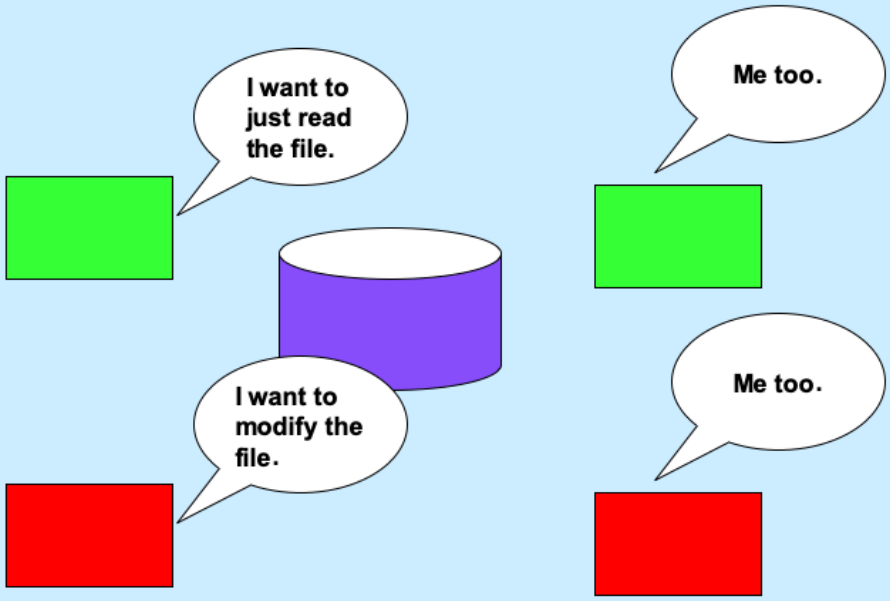
Sharing Files

- 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

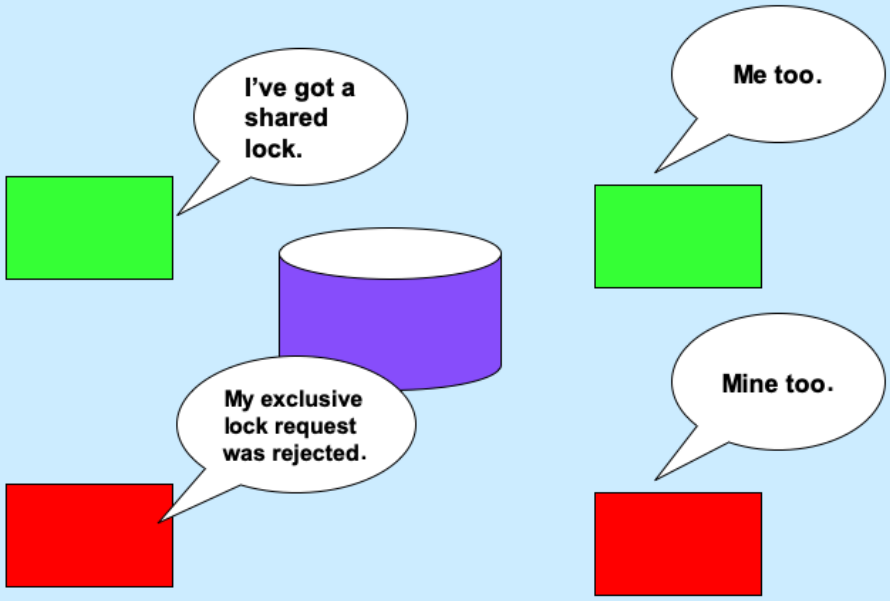
What We Want ...



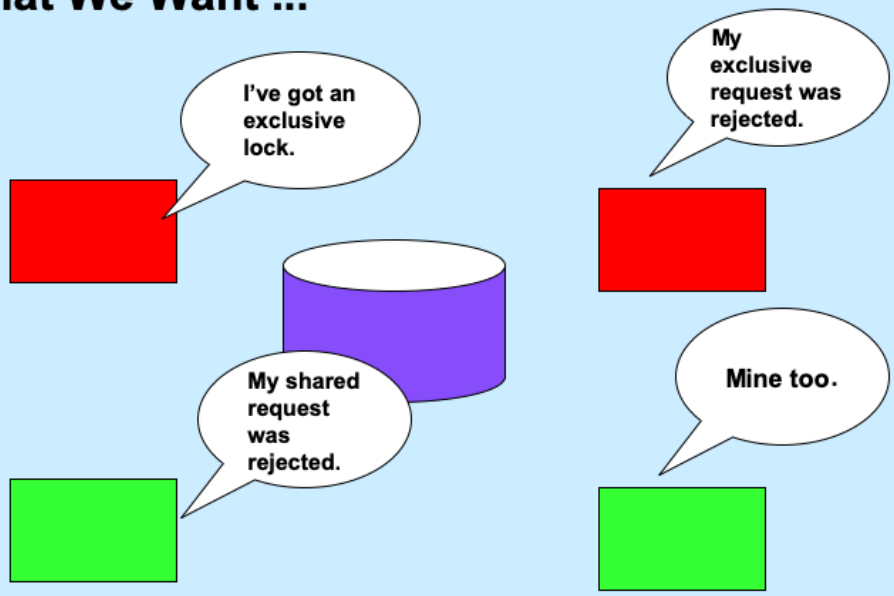
Types of Locks

- 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
    - » 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 …
    - » difficult to implement on distributed file systems (such as used at Brown CS)

Quiz 2

- 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?

  a) at least one major problem

  b) either no problems whatsoever or some easy-to-deal-with problems