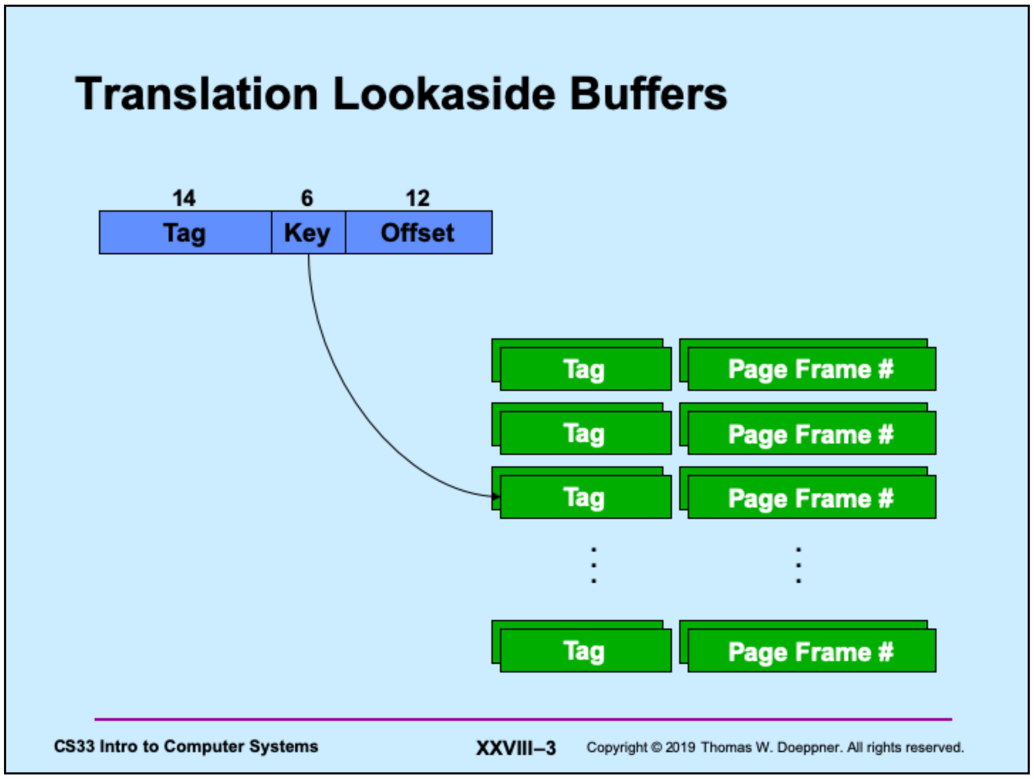
CS 33

More VM
Libraries

Performance

- Page table resides in real memory (DRAM)
- A 32-bit virtual-to-real translation requires two accesses to page tables, plus the access to the ultimate real address
  - three real accesses for each virtual access
  - 3X slowdown!
- A 64-bit virtual-to-real translation requires four accesses to page tables, plus the access to the ultimate real address
  - 5X slowdown!



Translation Lookaside Buffers

To speed-up virtual-to-real translation, a special cache is maintained of recent translations — it's called the translation lookaside buffer (TLB). It resides in the chip, one per core and hyperthread. The TLB shown in the slide is a two-way set associative cache, as discussed in lecture 17. This one assumes a 32-bit virtual address with a 4k page. Things are more complicated when multiple page sizes are supported. For example, is there just one entry for a large page that covers its entire range of addresses, or is a large page dealt with by putting into the cache multiple entries covering the large page, but each for the size of a small page? Both approaches are not only possible, but done.

Quiz 1

Recall that there is a 5x slowdown on memory references via virtual memory on the x86-64. If all references are translated via the TLB, the slowdown will be

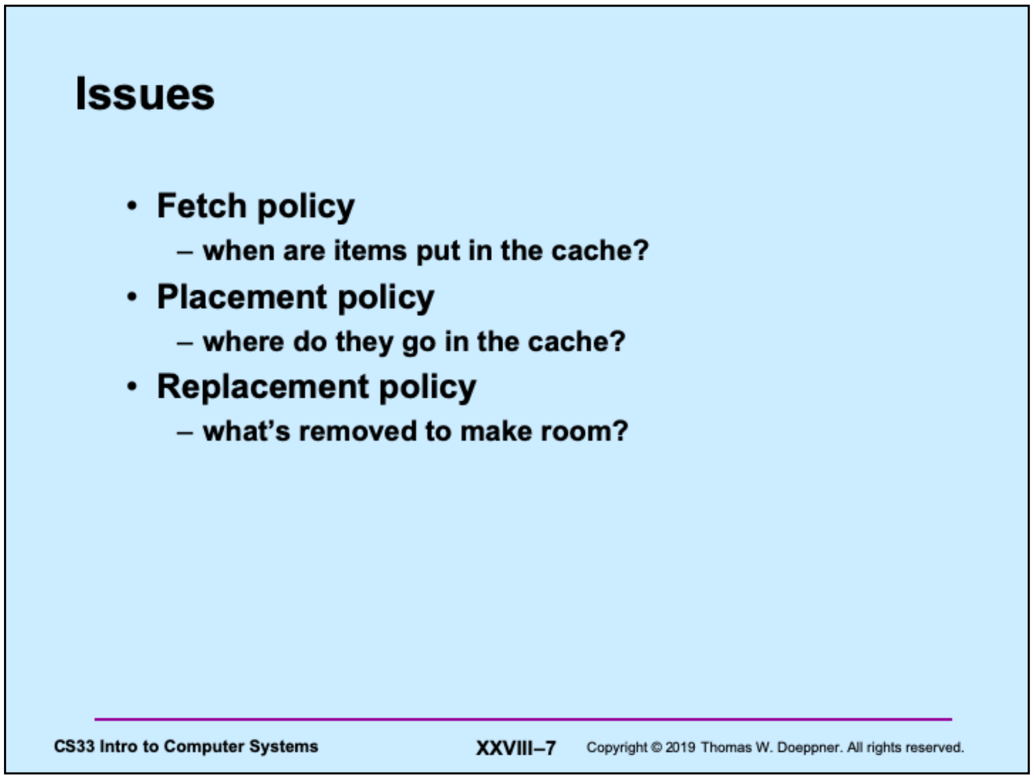
a) 1x
b) 2x
c) 3x
d) 4x

OS Role in Virtual Memory

- **Memory is like a cache**
  - quick access if what's wanted is mapped via page table
  - slow if not — OS assistance required
- **OS**
  - make sure what's needed is mapped in
  - make sure what's no longer needed is not mapped in

Mechanism

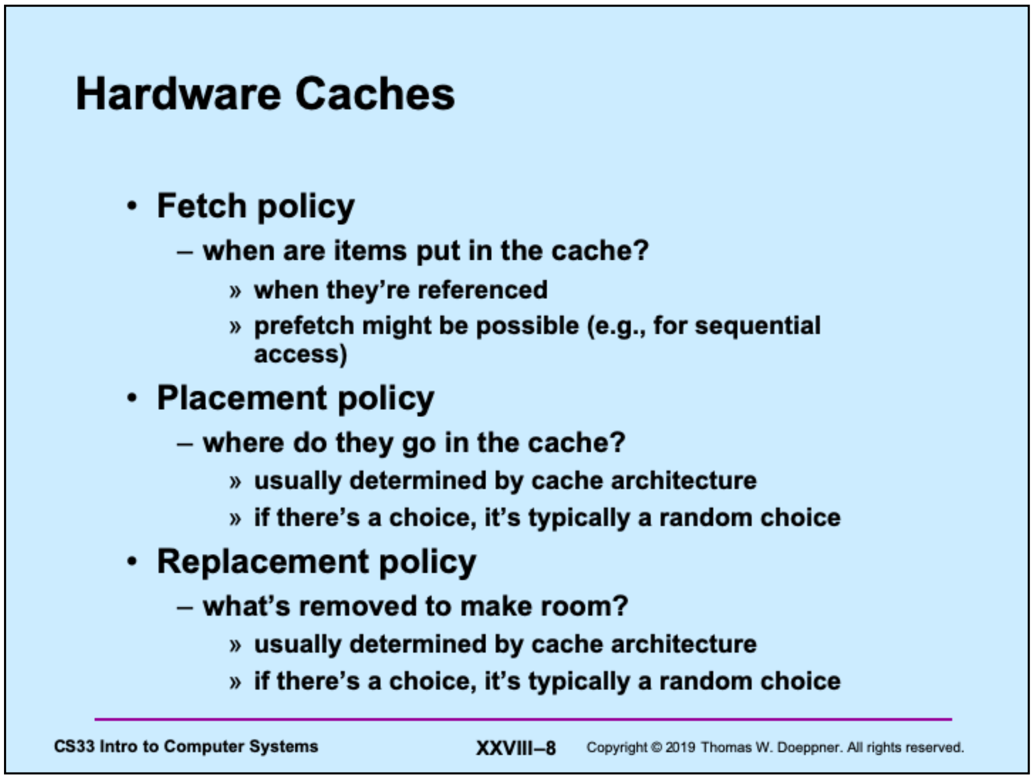
- Program references memory
  - if reference is mapped, access is quick
    - » even quicker if translation in TLB and referent in on-chip cache
  - if not, page-translation fault occurs and OS is invoked
    - » determines desired page
    - » maps it in, if legal reference



Issues

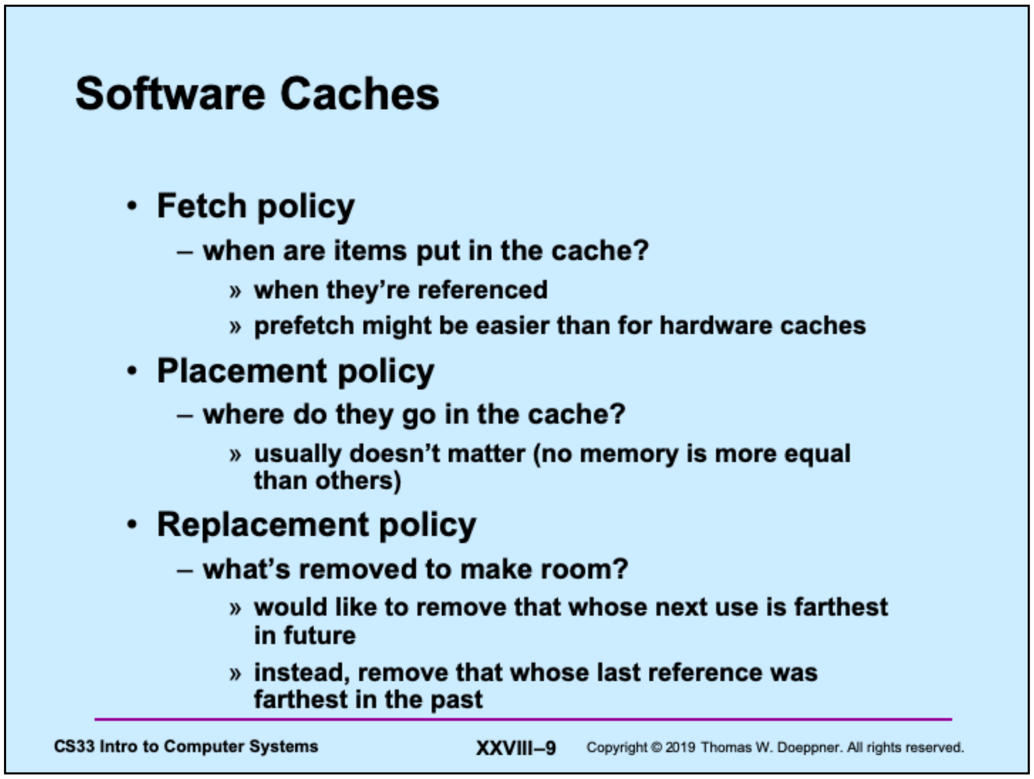
- **Fetch policy**
  - when are items put in the cache?
- **Placement policy**
  - where do they go in the cache?
- **Replacement policy**
  - what's removed to make room?

Three issues concerning the mechanism for caching are the following: the *fetch policy*, which governs when item are fetched to go into the cache, the *placement policy*, which governs where the fetched items are placed in the cache, and the *replacement policy*, which governs when and which items are removed from the cache (and perhaps written back to their source).



Hardware Caches

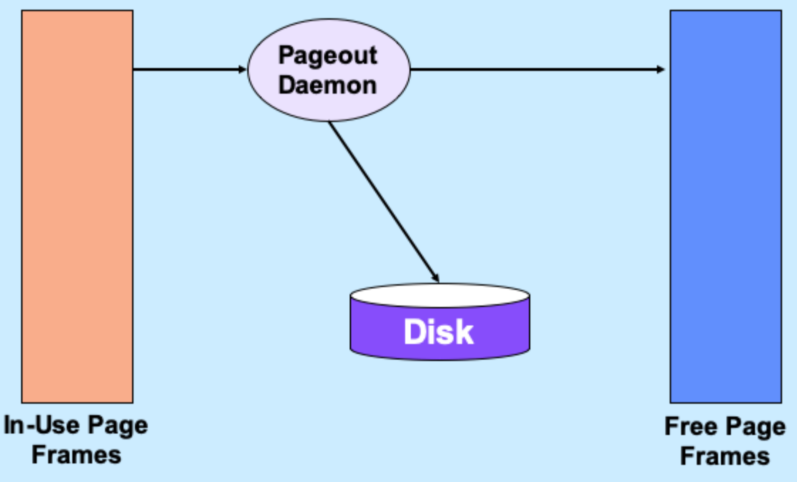
- Fetch policy
  - when are items put in the cache?
    - » when they're referenced
    - » prefetch might be possible (e.g., for sequential access)
- Placement policy
  - where do they go in the cache?
    - » usually determined by cache architecture
    - » if there's a choice, it's typically a random choice
- Replacement policy
  - what's removed to make room?
    - » usually determined by cache architecture
    - » if there's a choice, it's typically a random choice



Software Caches

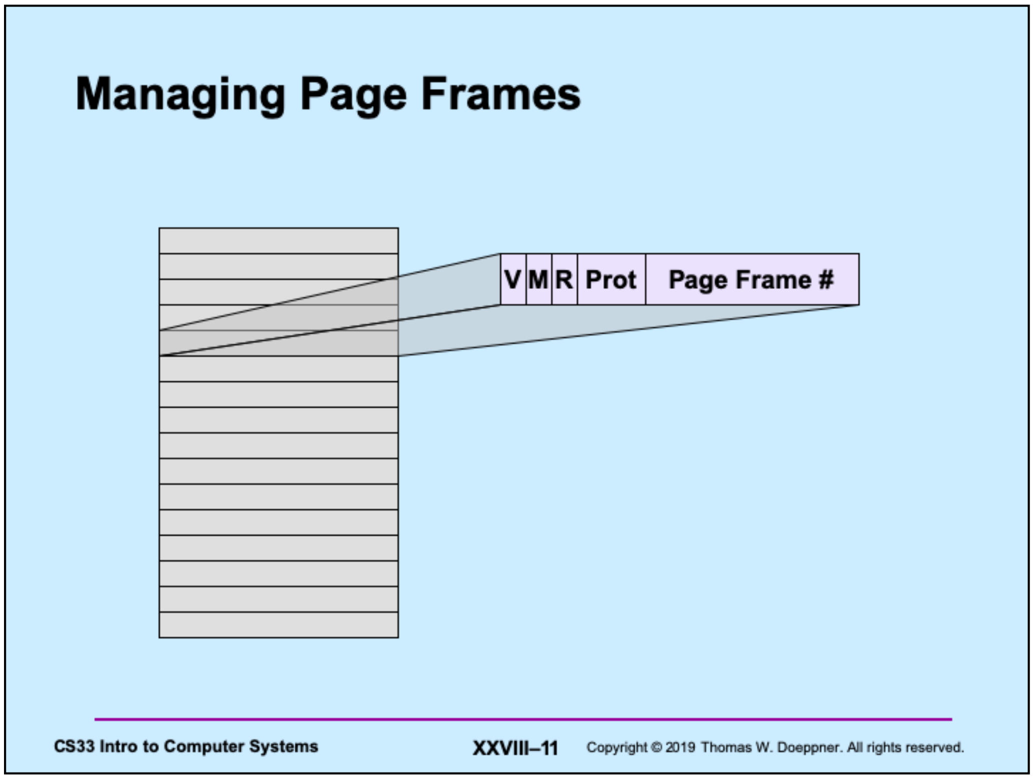
- Fetch policy
  - when are items put in the cache?
    - » when they're referenced
    - » prefetch might be easier than for hardware caches
- Placement policy
  - where do they go in the cache?
    - » usually doesn't matter (no memory is more equal than others)
- Replacement policy
  - what's removed to make room?
    - » would like to remove that whose next use is farthest in future
    - » instead, remove that whose last reference was farthest in the past

The “Pageout Daemon”

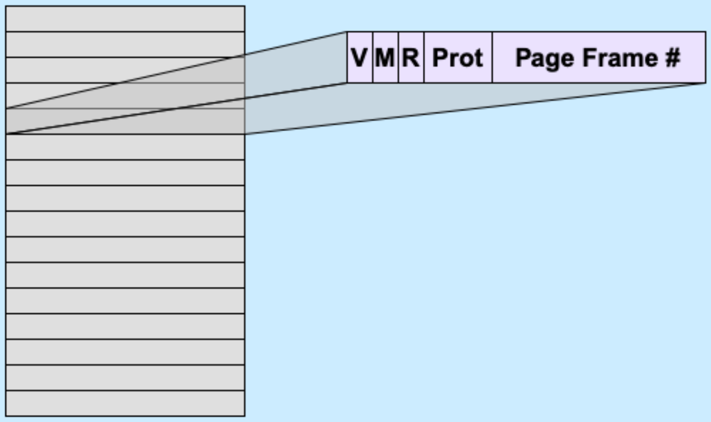


The (kernel) thread that maintains the free page-frame list is typically called the *pageout daemon*. Its job is to make certain that the free page-frame list has enough page frames on it. If the size of the list drops below some threshold, then the pageout daemon examines those page frames that are being used and selects a number of them to be freed. Before freeing a page, it must make certain that a copy of the current contents of the page exists on secondary storage. So, if the page has been modified since it was brought into primary storage (easily determined if there is a hardware-supported *modified bit*), it must first be written out to secondary storage. In many systems, the pageout daemon groups such pageouts into batches, so that a number of pages can be written out in a single operation, thus saving disk time. Unmodified, selected pages are transferred directly to the free page-frame list, modified pages are put there after they have been written out.

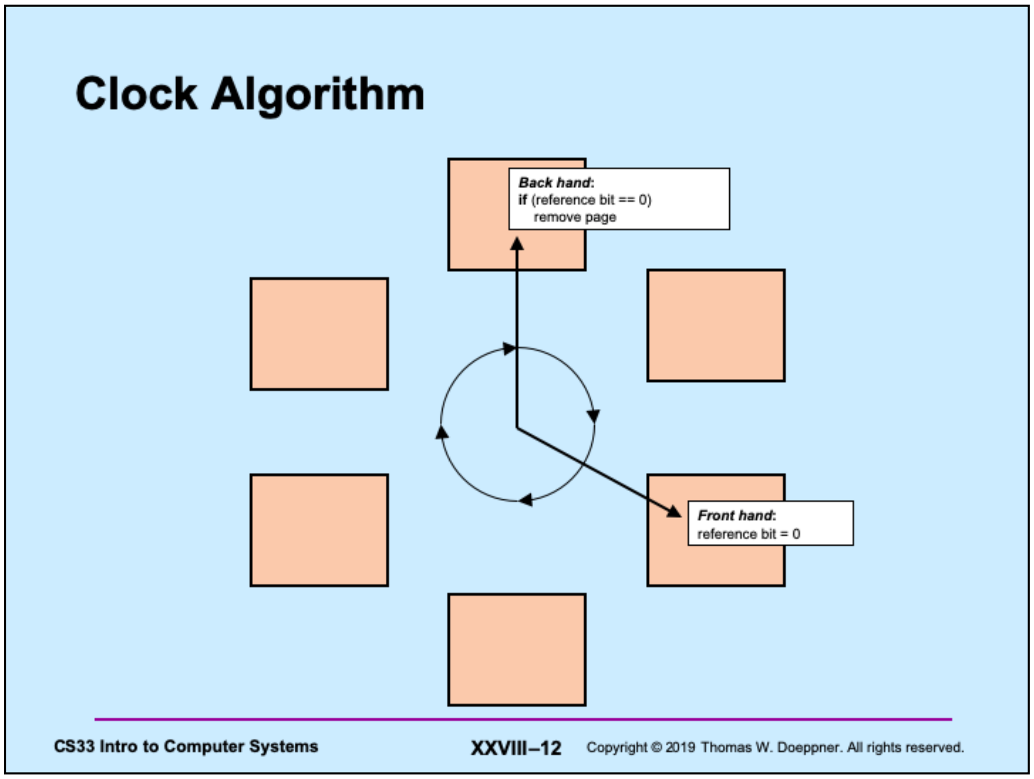
In most systems, pages in the free list get a “second chance” — if a thread in a process references such a page, there is a page fault (the page frame has been freed and could be used to hold another page), but the page-fault handler checks to see if the desired page is still in primary storage, but in the free list. If it is in the free list, it is removed and given back to the faulting process. We still suffer the overhead of a trap, but there is no wait for I/O.



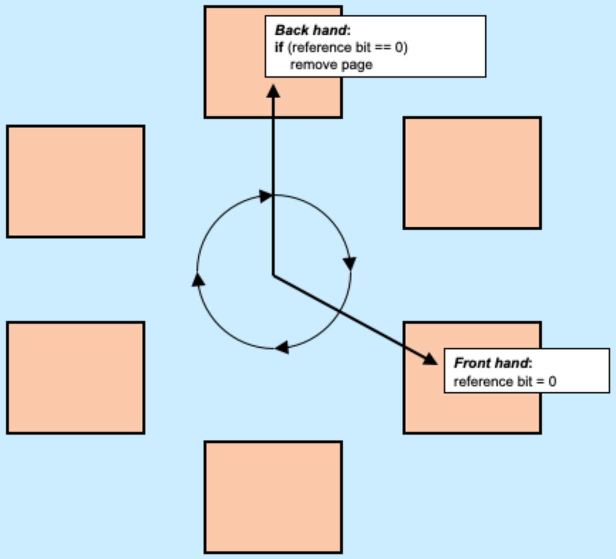
Managing Page Frames



The OS can keep track of the history of page frame by use of two bits in each page-table entry: the *modify* bit, which is set by hardware whenever the associated page frame is modified, and the *referenced* bit, which is set by hardware whenever the associated page is accessed (via either a load or a store).



Clock Algorithm

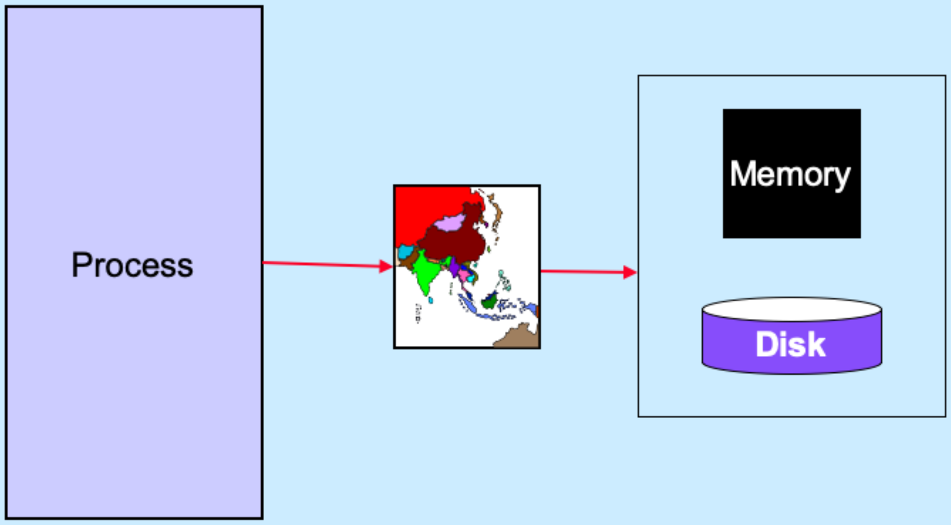


A common approach for determining which page frames are not in use is known as the clock algorithm. All active page frames are conceptually arranged in a circularly linked list. The page-out thread slowly traverses the list. The "one-handed" version of the clock algorithm, each time it encounters a page, checks the reference bit in the corresponding translation entry: if the bit is set, it clears it. If the bit is clear, it adds the page to the free list (writing it back to secondary storage first, if necessary).

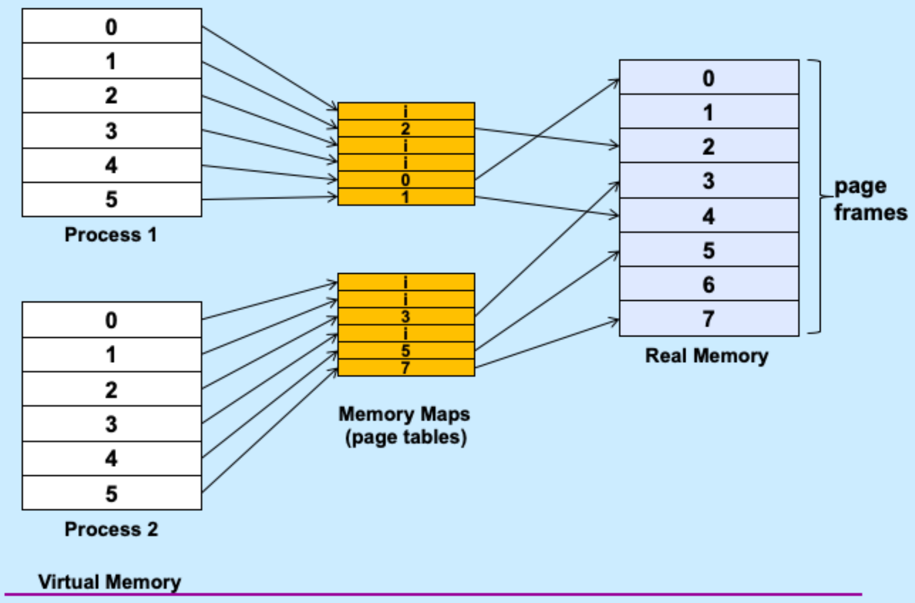
A problem with the one-handed version is that, in systems with large amounts of primary storage, it might take too long for the page-out thread to work its way all around the list of page frames before it can recognize that a page has not been recently referenced. In the two-handed version of the clock algorithm, the page-out thread implements a second hand some distance behind the first. The front hand simply clears reference bits. The second (back) hand removes those pages whose reference bits have not been set to one by the time the hand reaches the page frame.

Why is virtual memory used?

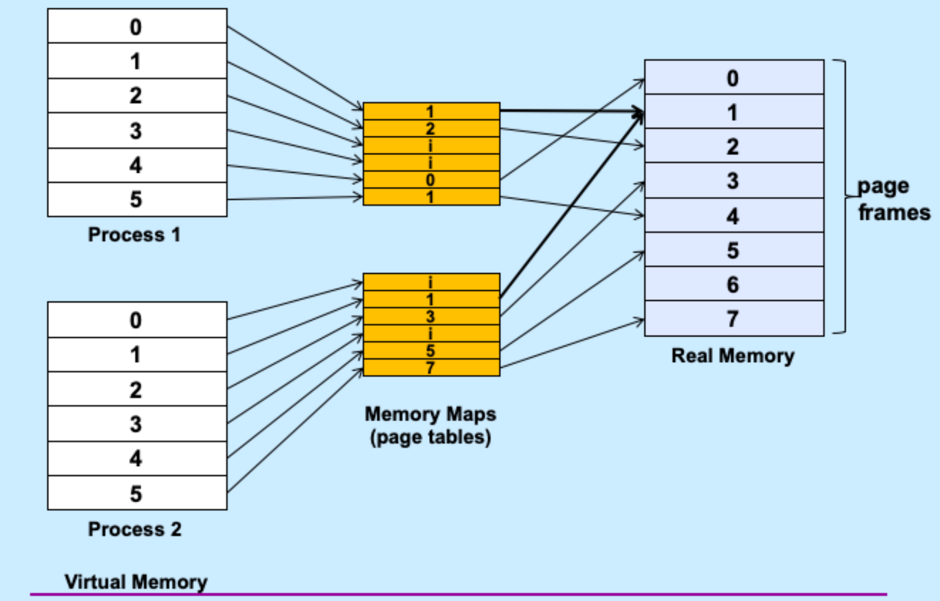
More VM than RM

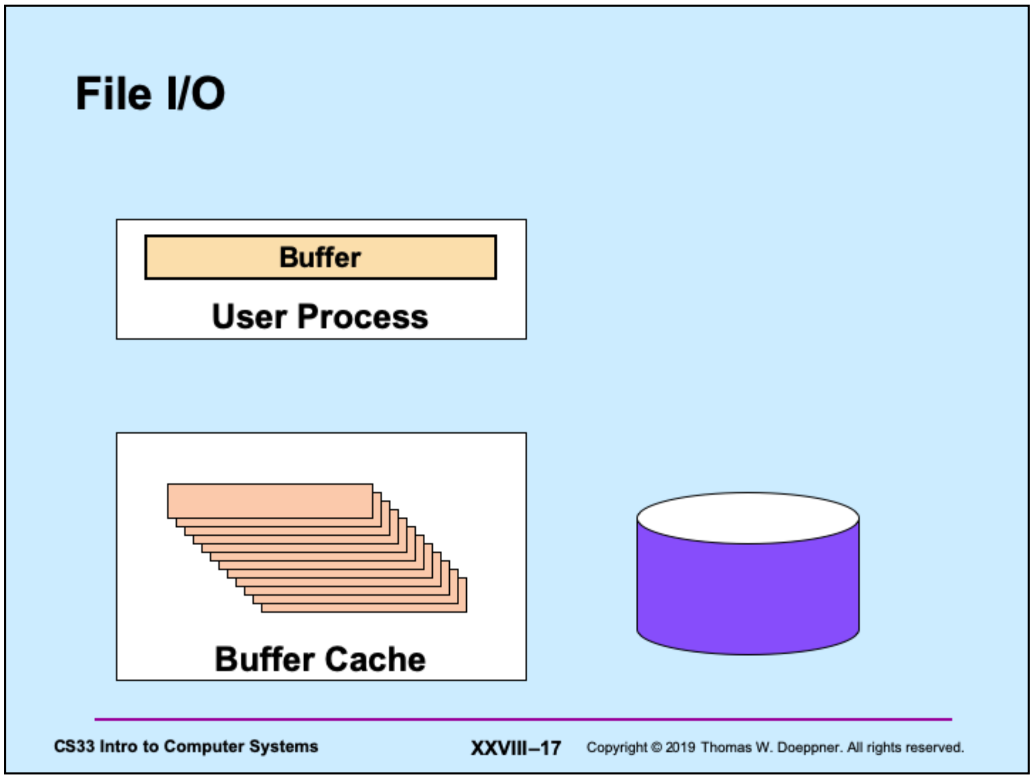


Isolation


0
1
2
3
4
5
Process 1
0
1
2
3
4
5
Process 2
Virtual Memory
i
2
i
i
0
1
i
i
3
i
5
7
Memory Maps
(page tables)
0
1
2
3
4
5
6
7
Real Memory
page
frames

Sharing


0
1
2
3
4
5
Process 1
0
1
2
3
4
5
Process 2
Virtual Memory
1
2
i
i
0
1
i
1
3
i
5
7
Memory Maps
(page tables)
0
1
2
3
4
5
6
7
Real Memory
page
frames

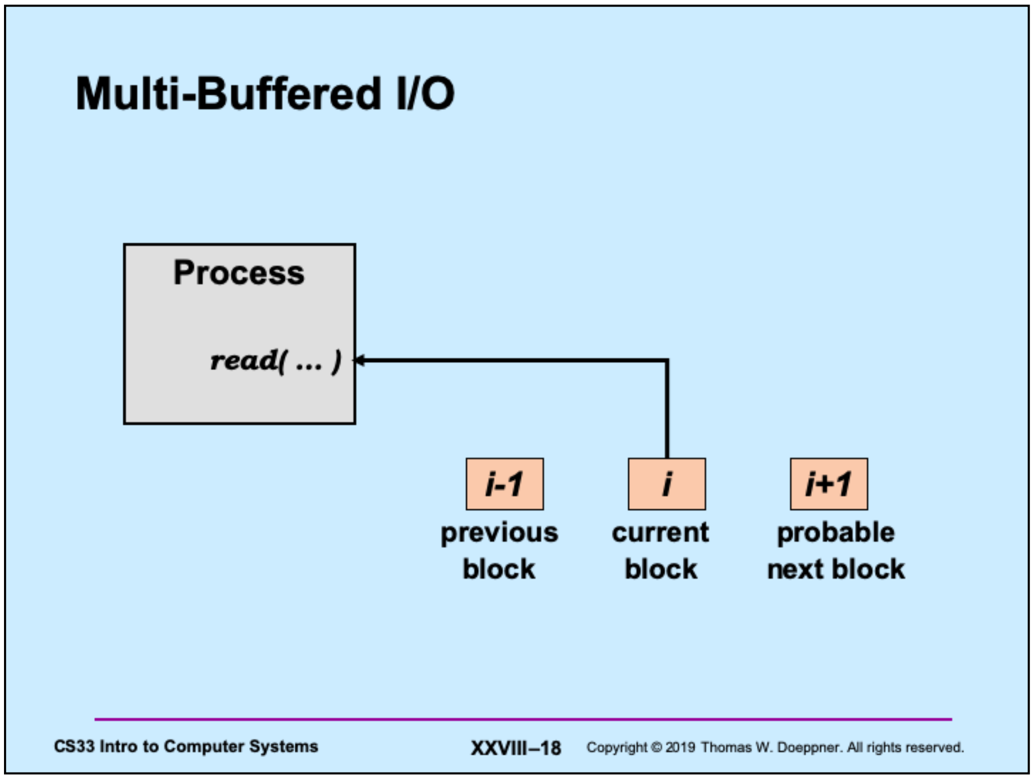


File I/O

File I/O in Unix, and in most operating systems, is not done directly to the disk drive, but through intermediary buffers, known as the buffer cache, in the operating system's address space. This cache has two primary functions. The first, and most important, is to make possible concurrent I/O and computation within a Unix process. The second is to insulate the user from physical disk-block boundaries.

From a user process's point of view, I/O is *synchronous*. By this we mean that when the I/O system call returns, the system no longer needs the user-supplied buffer. For example, after a write system call, the data in the user buffer has either been transmitted to the device or copied to a kernel buffer — the user can now scribble over the buffer without affecting the data transfer. Because of this synchronization, from a user process's point of view, no more than one I/O operation can be in progress at a time.

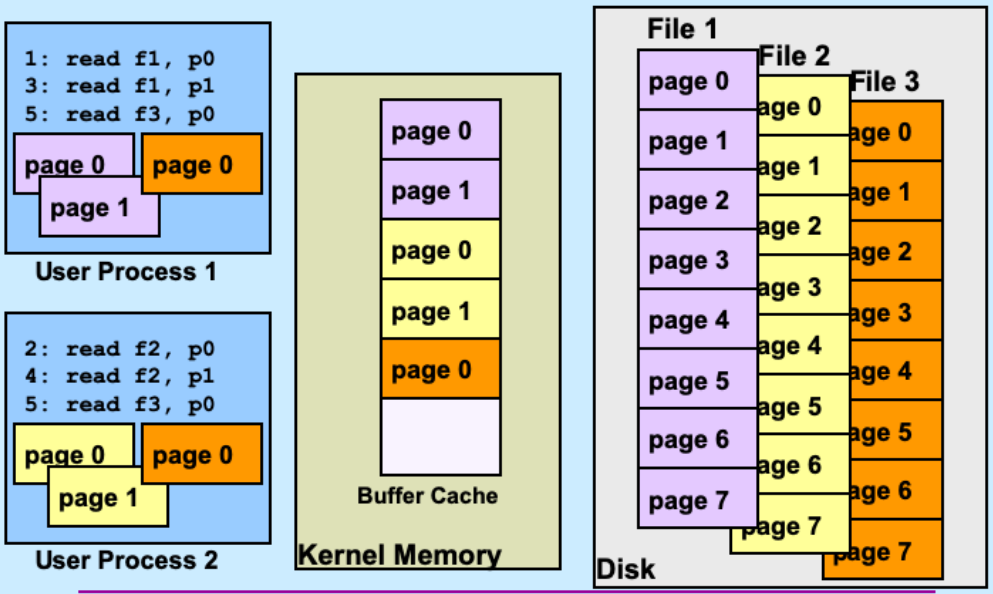
The buffer cache provides a kernel implementation of multibuffered I/O, and thus concurrent I/O and computation are made possible.

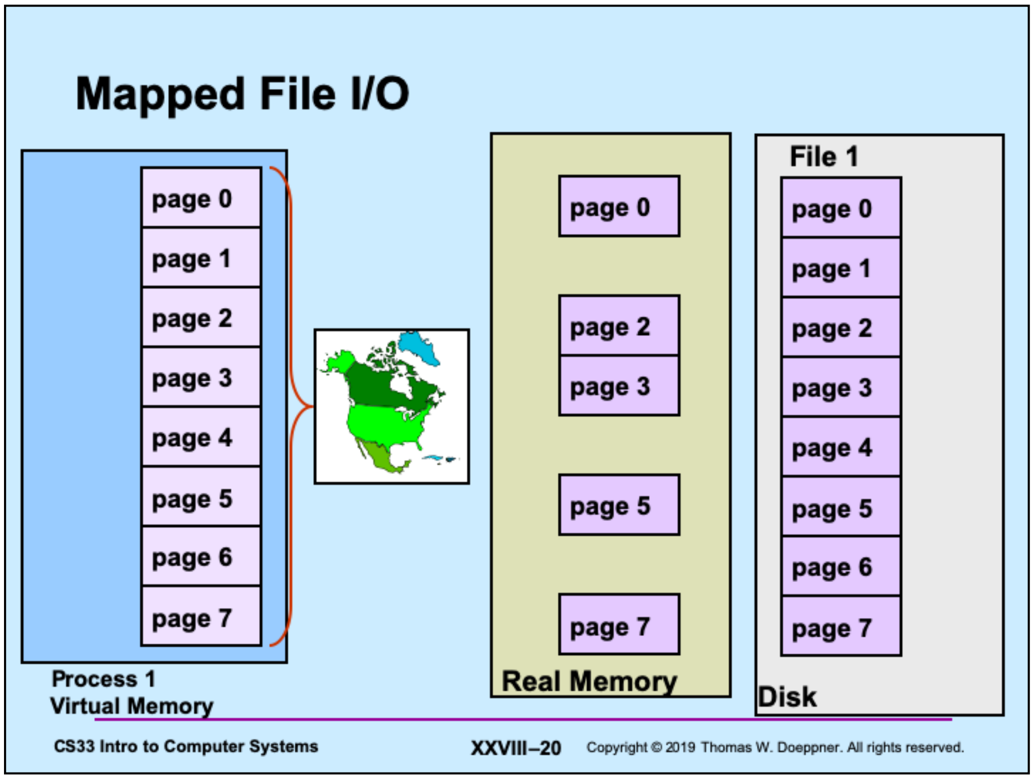


Multi-Buffered I/O

The use of *read-aheads* and *write-behinds* makes possible concurrent I/O and computation: if the block currently being fetched is block *i* and the previous block fetched was block *i-1*, then block *i+1* is also fetched. Modified blocks are normally written out not synchronously but instead sometime after they were modified, asynchronously.

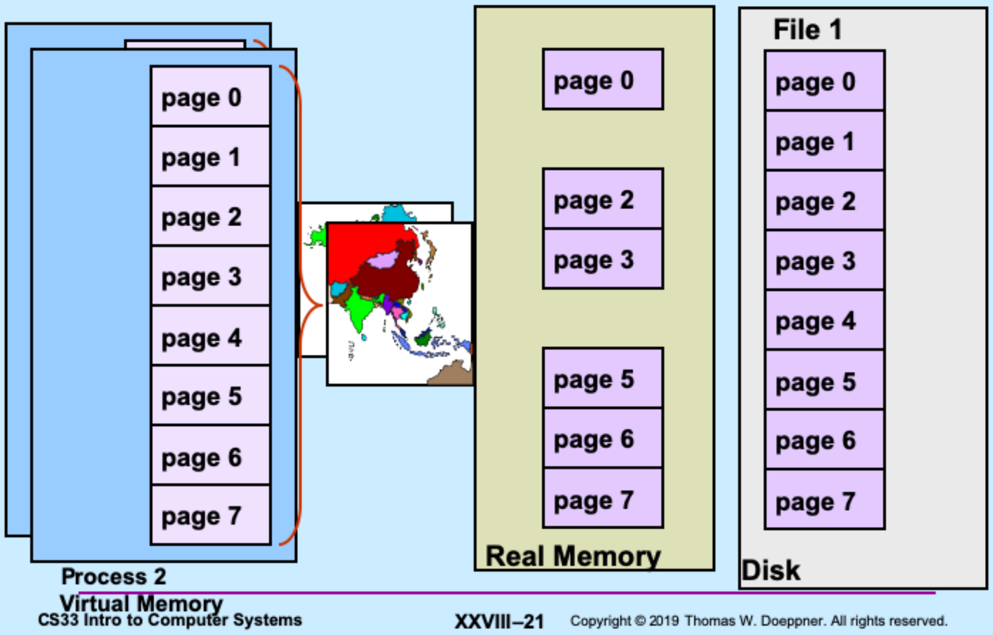
Traditional I/O

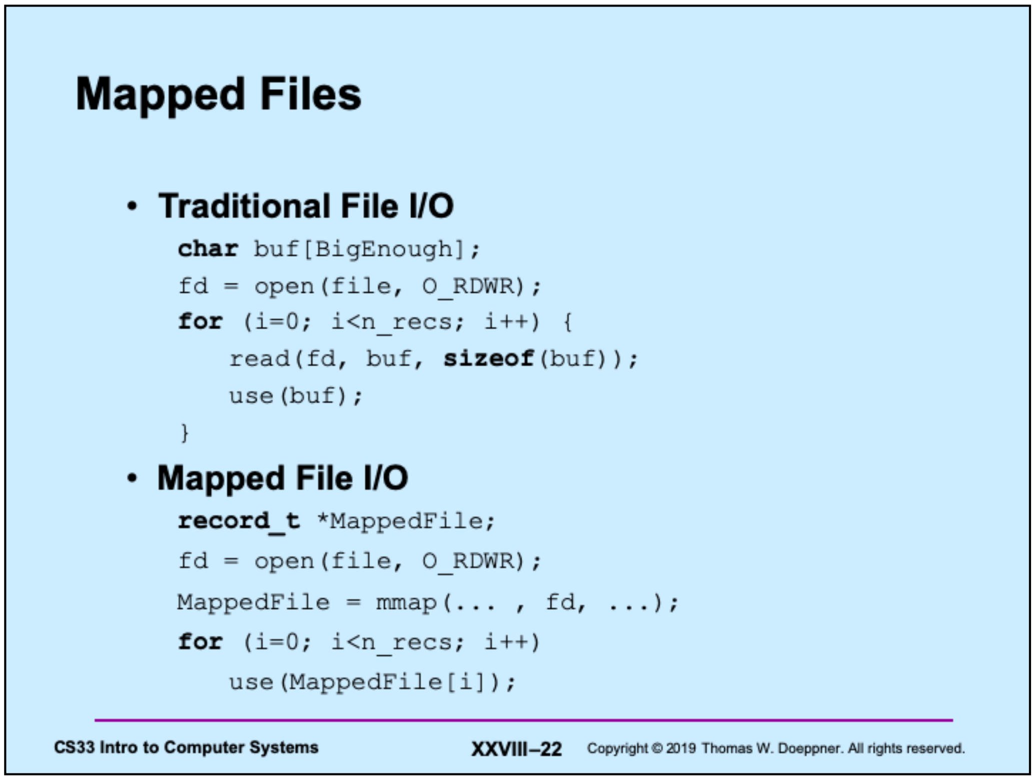




Mapped File I/O

Multi-Process Mapped File I/O





Mapped Files

- **Traditional File I/O**

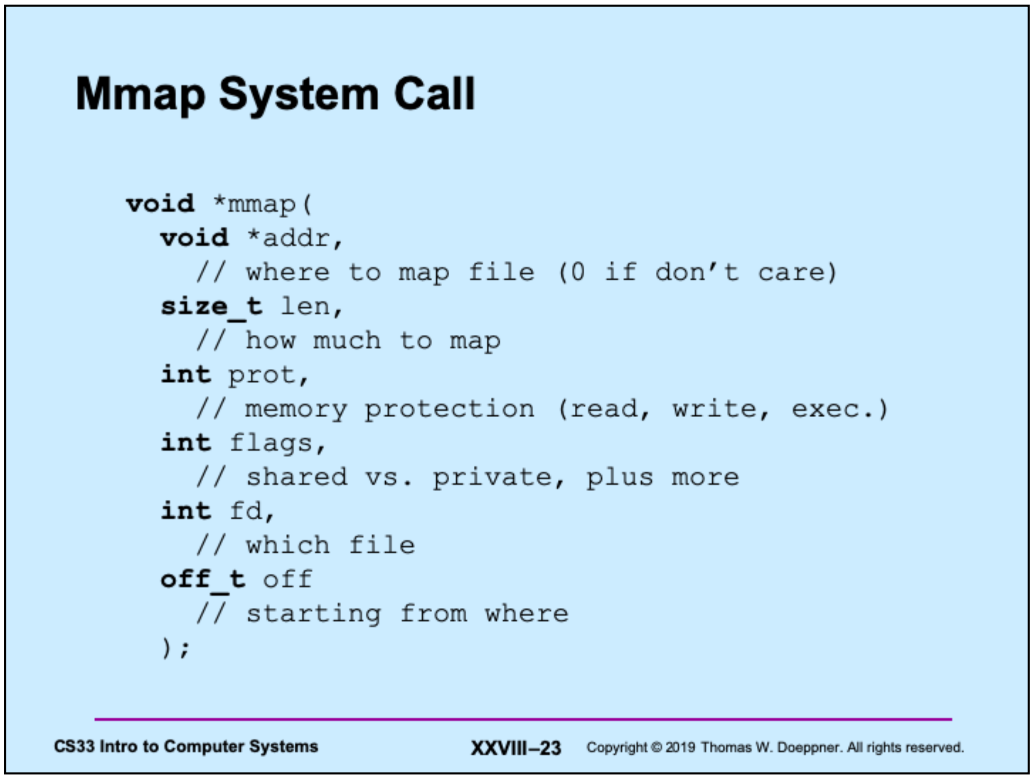
```
char buf[BigEnough];
fd = open(file, O_RDWR);
for (i=0; i<n_recs; i++) {
    read(fd, buf, sizeof(buf));
    use(buf);
}
```

- **Mapped File I/O**

```
record_t *MappedFile;
fd = open(file, O_RDWR);
MappedFile = mmap(... , fd, ...);
for (i=0; i<n_recs; i++)
    use(MappedFile[i]);
```

Traditional I/O involves explicit calls to read and write, which in turn means that data is accessed via a buffer; in fact, two buffers are usually employed: data is transferred between a user buffer and a kernel buffer, and between the kernel buffer and the I/O device.

An alternative approach is to *map* a file into a process's address space: the file provides the data for a portion of the address space and the kernel's virtual-memory system is responsible for the I/O. A major benefit of this approach is that data is transferred directly from the device to where the user needs it; there is no need for an extra system buffer.



Mmap System Call

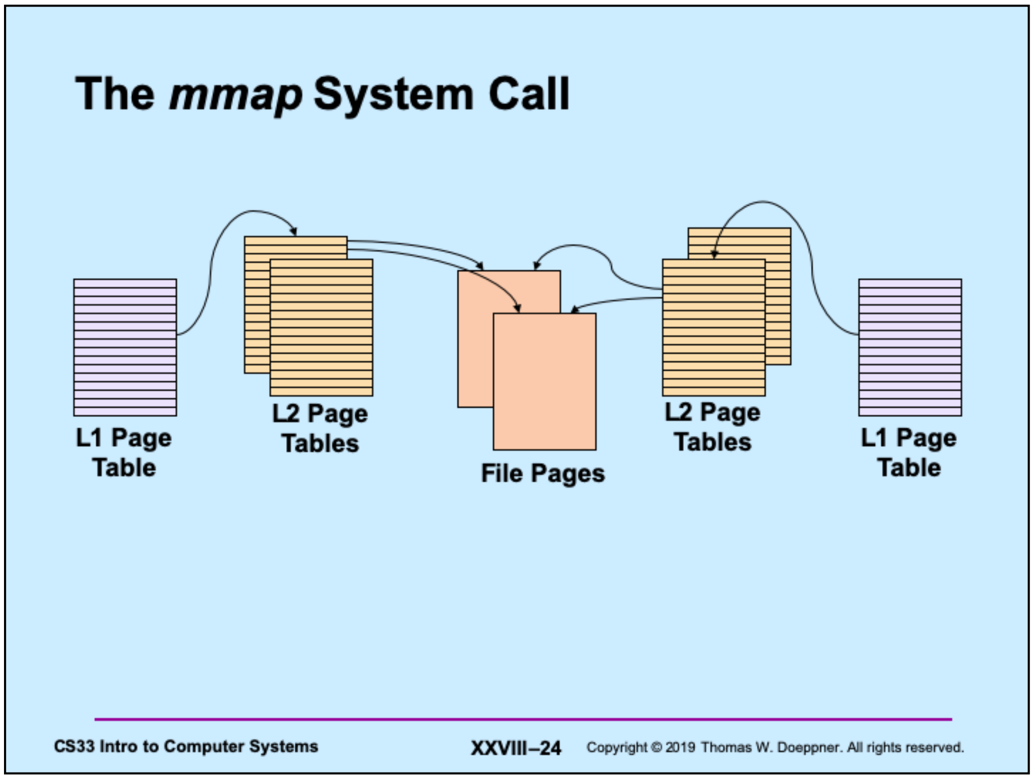
```
void *mmap(
  void *addr,
    // where to map file (0 if don't care)
  size_t len,
    // how much to map
  int prot,
    // memory protection (read, write, exec.)
  int flags,
    // shared vs. private, plus more
  int fd,
    // which file
  off_t off
    // starting from where
  );
```

Mmap maps the file given by *fd*, starting at position *off,* for *len* bytes, into the caller's address space starting at location *addr*

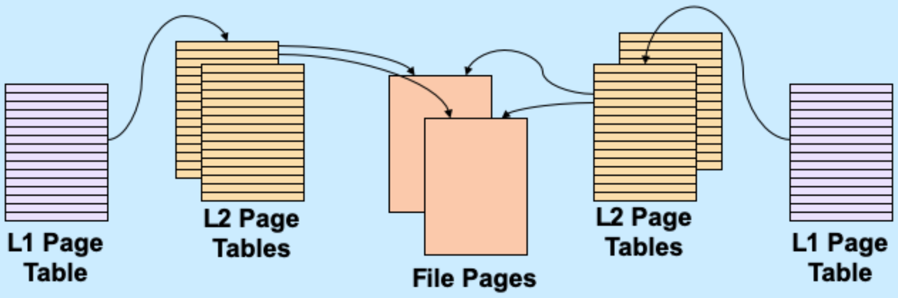
- *len* is rounded up to a multiple of the page size
- *off* must be page-aligned
- if *addr* is zero, the kernel assigns an address
- if *addr* is positive, it is a suggestion to the kernel as to where the mapped file should be located (it usually will be aligned to a page). However, if *flags* includes MAP_FIXED, then *addr* is not modified by the kernel (and if its value is not reasonable, the call fails)
- the call returns the address of the beginning of the mapped file

The flags argument must include either MAP_SHARED or MAP_PRIVATE (but not both). If it's MAP_SHARED, then the mapped portion of the caller's address space contains the current contents of the file; when the mapped portion of the address space is modified by the process, the corresponding portion of the file is modified.

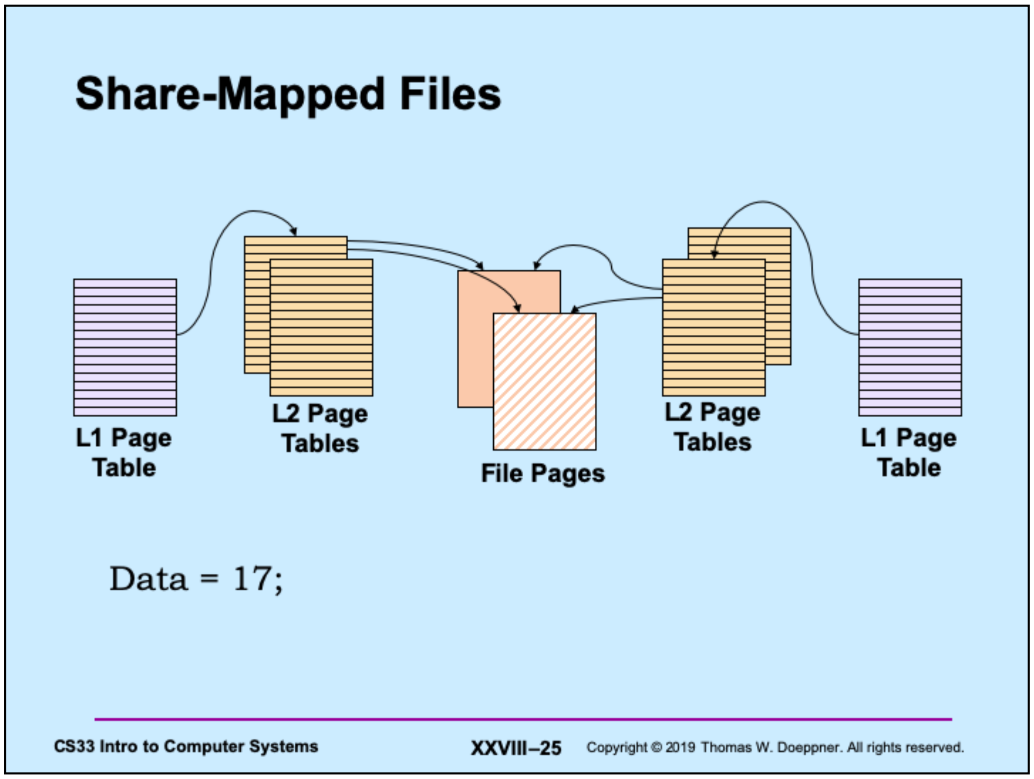
However, if *flags* includes MAP_PRIVATE, then the idea is that the mapped portion of the address space is initialized with the contents of the file, but that changes made to the mapped portion of the address space by the process are private and not written back to the file. The details are a bit complicated: as long as the mapping process does not modify any of the mapped portion of the address space, the pages contained in it contain the current contents of the corresponding pages of the file. However, if the process modifies a page, then that particular page no longer contains the current contents of the corresponding file page, but contains whatever modifications are made to it by the process. These changes are not written back to the file and not shared with any other process that has mapped the file. It's unspecified what the situation is for other pages in the mapped region after one of them is modified. Depending on the implementation, they might continue to contain the current contents of the corresponding pages of the file until they, themselves, are modified. Or they might also be treated as if they'd just been written to and thus no longer be shared with others.



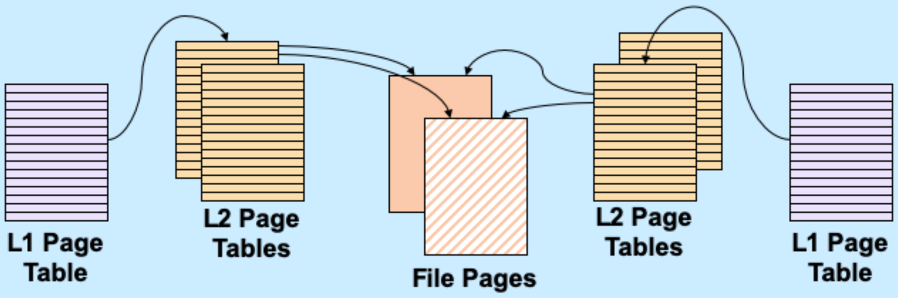
The *mmap* System Call



The *mmap* system call maps a file into a process's address space. All processes mapping the same file can share the pages of the file.

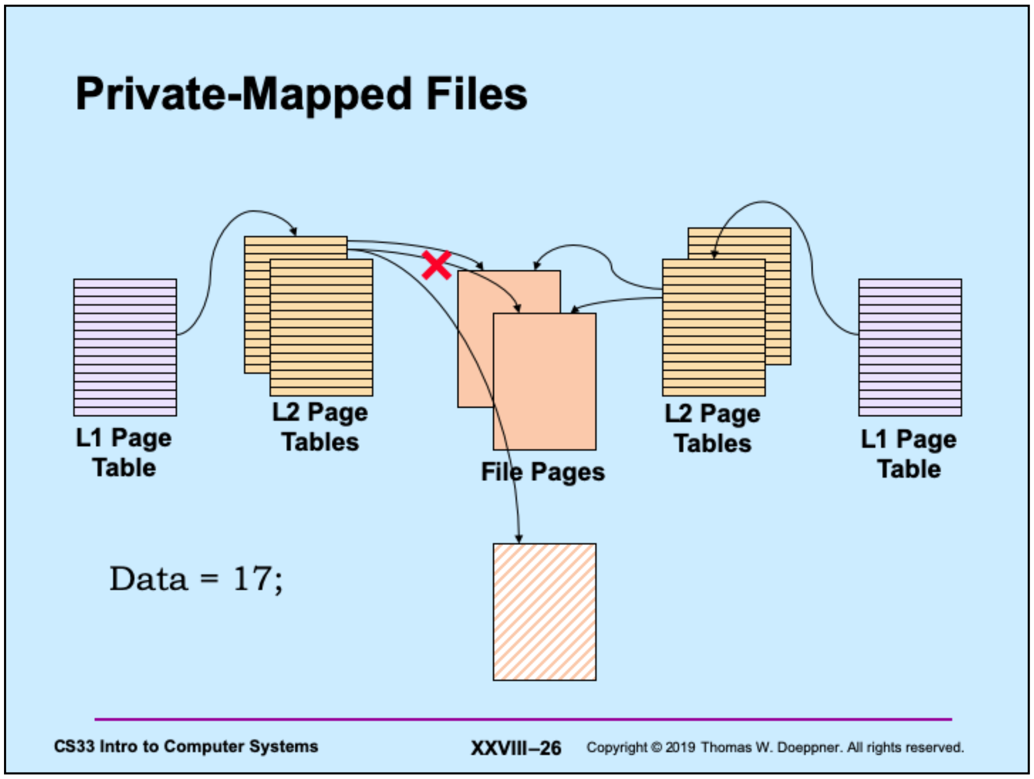


Share-Mapped Files



Data = 17;

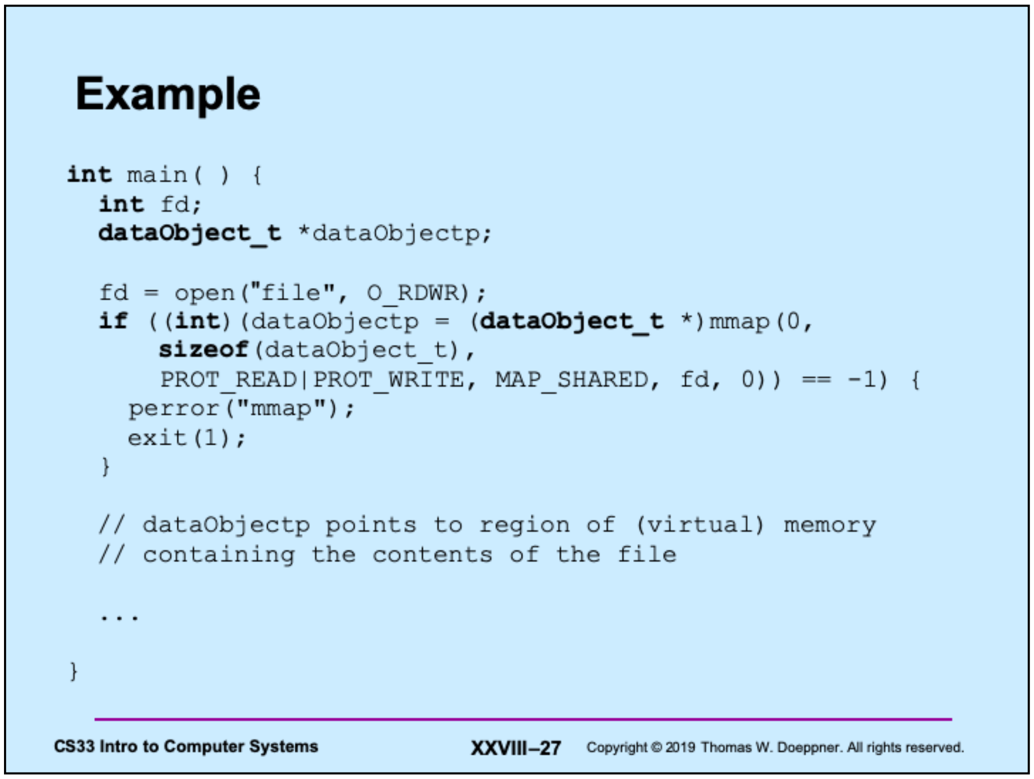
There are a couple options for how modifications to mmapped files are dealt with. The most straightforward is the *share* option in which changes to mmapped file pages modify the file and hence the changes are seen by the other processes who have share-mapped the file.



Private-Mapped Files

The other option is to *private*-map the file: changes made to mmapped file pages do not modify the file. Instead, when a page of a file is first modified via a private mapping, a copy of just that page is made for the modifying process, but this copy is not seen by other processes, nor does it appear in the file.

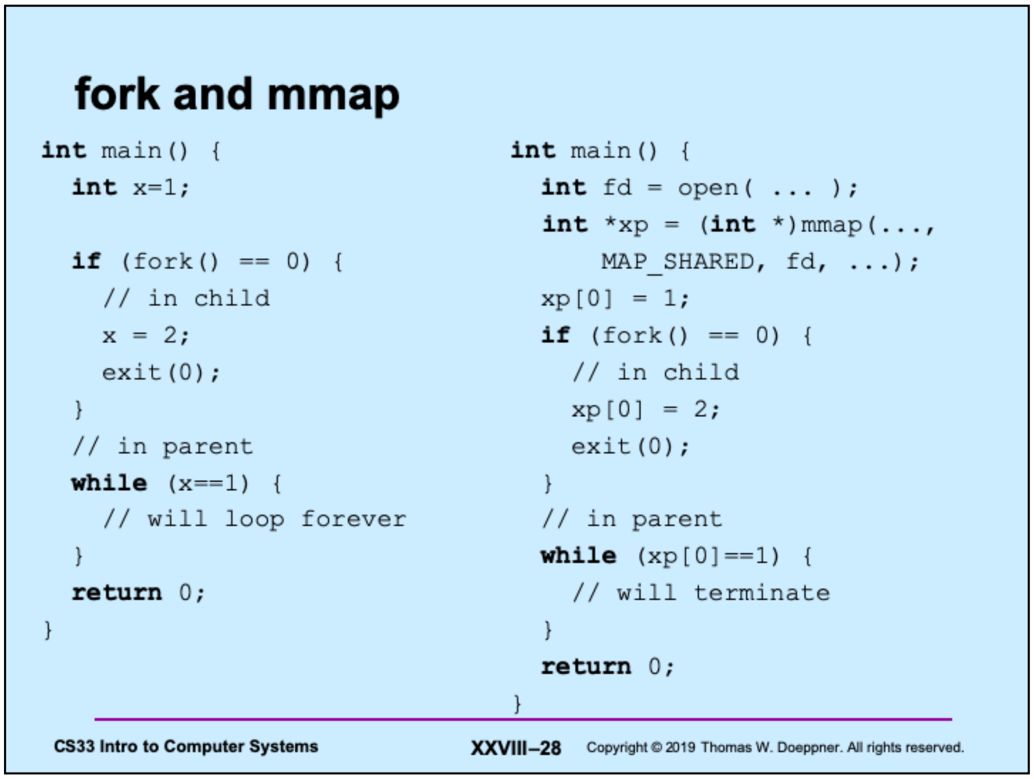
In the slide, the process on the left has private-mapped the file. Thus its changes to the mapped portion of the address space are made to a copy of the page being modified.



Example

```
int main( ) {
  int fd;
  dataObject_t *dataObjectp;

  fd = open("file", O_RDWR);
  if ((int)(dataObjectp = (dataObject_t *)mmap(0,
      sizeof(dataObject_t),
      PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0)) == -1) {
    perror("mmap");
    exit(1);
  }

  // dataObjectp points to region of (virtual) memory
  // containing the contents of the file

  ...

}
```

Here we map the contents of a file containing a dataObject_t into the caller's address space, allowing it both read and write access. Note mapping the file into memory does not cause any immediate I/O to take place. The operating system will perform the I/O when necessary, according to its own rules.



fork and mmap

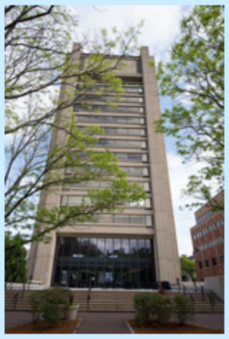
```
int main() {
  int x=1;

  if (fork() == 0) {
    // in child
    x = 2;
    exit(0);
  }
  // in parent
  while (x==1) {
    // will loop forever
  }
  return 0;
}
```

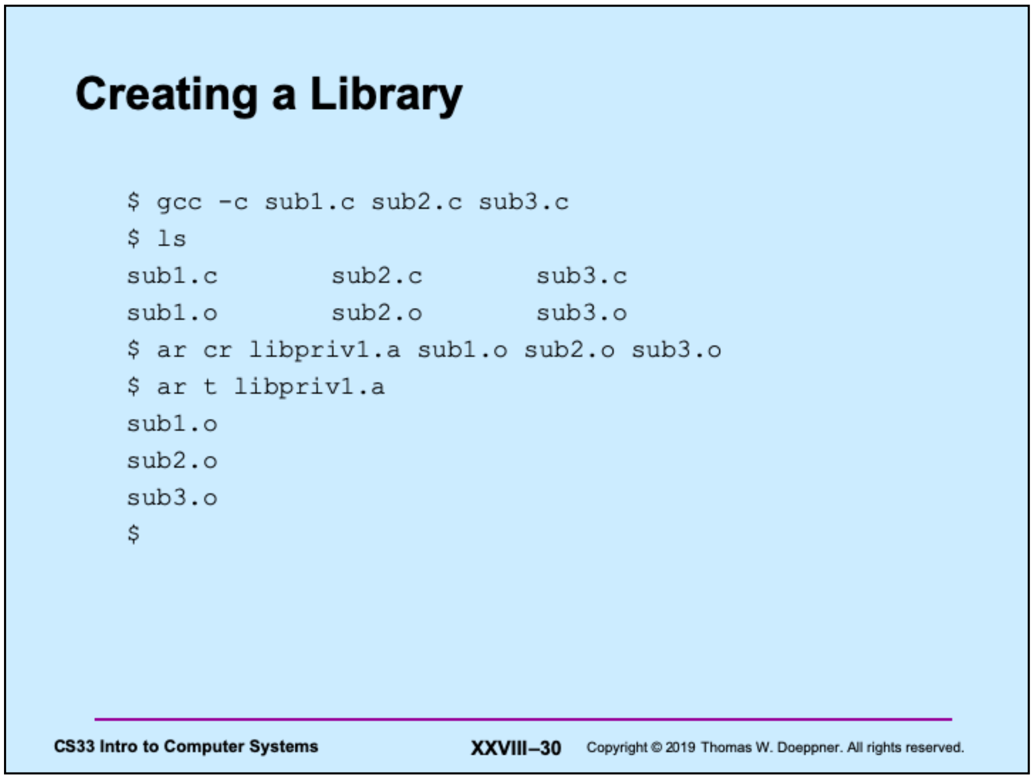
```
int main() {
  int fd = open( ... );
  int *xp = (int *)mmap(...,
      MAP_SHARED, fd, ...);
  xp[0] = 1;
  if (fork() == 0) {
    // in child
    xp[0] = 2;
    exit(0);
  }
  // in parent
  while (xp[0]==1) {
    // will terminate
  }
  return 0;
}
```

When a process calls fork and creates a child, the child's address space is normally a copy of the parent's. Thus changes made by the child to its address space will not be seen in the parent's address space (as shown in the left-hand column). However, if there is a region in the parent's address space that has been mmapped using the MAP_SHARED flag, and subsequently the parent calls fork and creates a child, the mmapped region is not copied but is shared by parent and child. Thus changes to the region made by the child will be seen by the parent (and vice versa).

Libraries

- Collections of useful stuff
- Allow you to:
  - incorporate items into your program
  - substitute new stuff for existing items
- Often ugly …

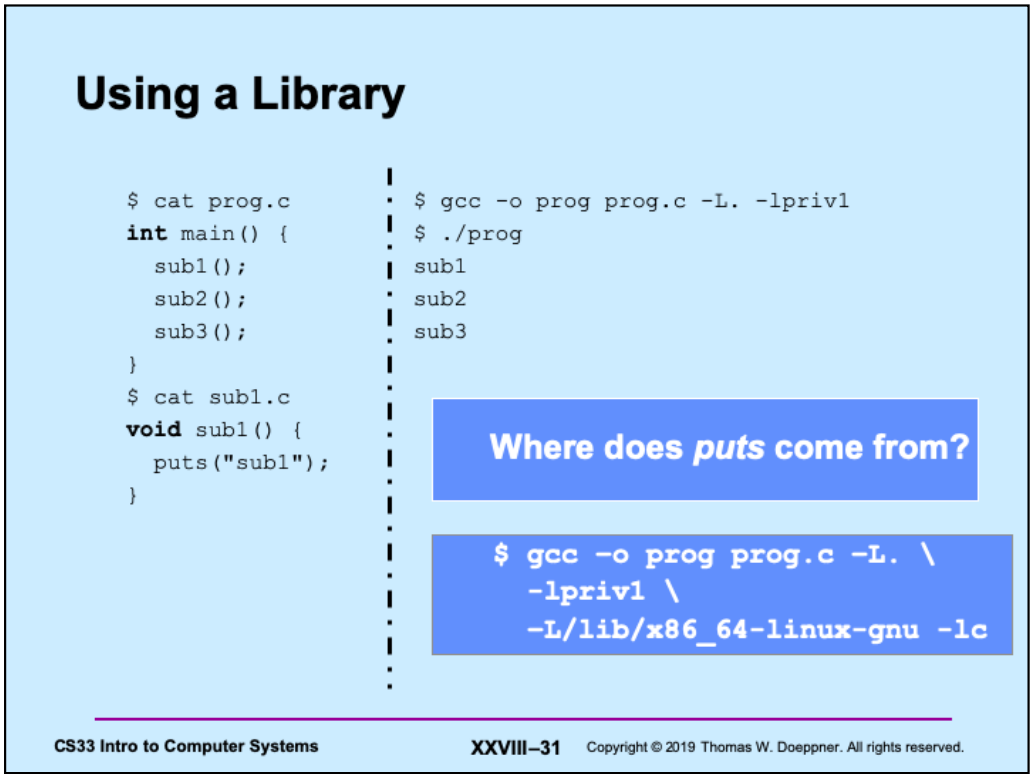




Creating a Library

```
$ gcc -c sub1.c sub2.c sub3.c
$ ls
sub1.c       sub2.c       sub3.c
sub1.o       sub2.o       sub3.o
$ ar cr libpriv1.a sub1.o sub2.o sub3.o
$ ar t libpriv1.a
sub1.o
sub2.o
sub3.o
$
```

Files ending with “.a” are known as *archives* or *static libraries*.



Using a Library

```
$ cat prog.c
int main() {
  sub1();
  sub2();
  sub3();
}
$ cat sub1.c
void sub1() {
  puts("sub1");
}
```

```
$ gcc -o prog prog.c -L. -lpriv1
$ ./prog
sub1
sub2
sub3
```

Where does *puts* come from?

```
$ gcc -o prog prog.c -L. \
  -lpriv1 \
  -L/lib/x86_64-linux-gnu -lc
```

The function "puts" is from the standard-I/O library, just as printf is, but it's far simpler. It prints its single string argument, appending a '\n' (newline) to the end.

Note that "-lpriv1" (the second character of the string is a lower-case L and the last character is the numeral one) is, in this example, shorthand for libpriv1.a, but we'll soon see that it's shorthand for more than that.

Normally, libraries are expected to be found in the current directory. The "-L" flag is used to specify additional directories in which to look for libraries.

Static-Linking: What's in the Executable

- ld puts in the executable:
  - » (assuming all .c files have been compiled into .o files)
  - – all .o files from argument list (including those newly compiled)
  - – .o files from archives as needed to satisfy unresolved references
    - » some may have their own unresolved references that may need to be resolved from additional .o files from archives
    - » each archive processed just once (as ordered in argument list)
      - order matters!

Example

```
$ cat prog2.c
int main() {
  void func1();
  func1();
  return 0;
}
$ cat func1.c
void func1() {
  void func2();
  func2();
}
$ cat func2.c
void func2() {
}
```

Order Matters ...

```
$ ar t libf1.a
func1.o
$ ar t libf2.a
func2.o
$ gcc -o prog2 prog2.c -L. -lf1 -lf2
$
$ gcc -o prog2 prog2.c -L. -lf2 -lf1
./libf1.a(sub1.o): In function `func1':
func1.c:(.text+0xa): undefined reference to `func2'
collect2: error: ld returned 1 exit status
```

Substitution

```
$ cat myputs.c
int puts(char *s) {
  write(1, "My puts: ", 9);
  write(1, s, strlen(s));
  write(1, "\n", 1);
  return 1;
}
$ gcc -c myputs.c
$ ar cr libmyputs.a myputs.o
$ gcc -o prog prog.c -L. -lpriv1 -lmyputs
$ ./prog
My puts: sub1
My puts: sub2
My puts: sub3
```

An Urgent Problem

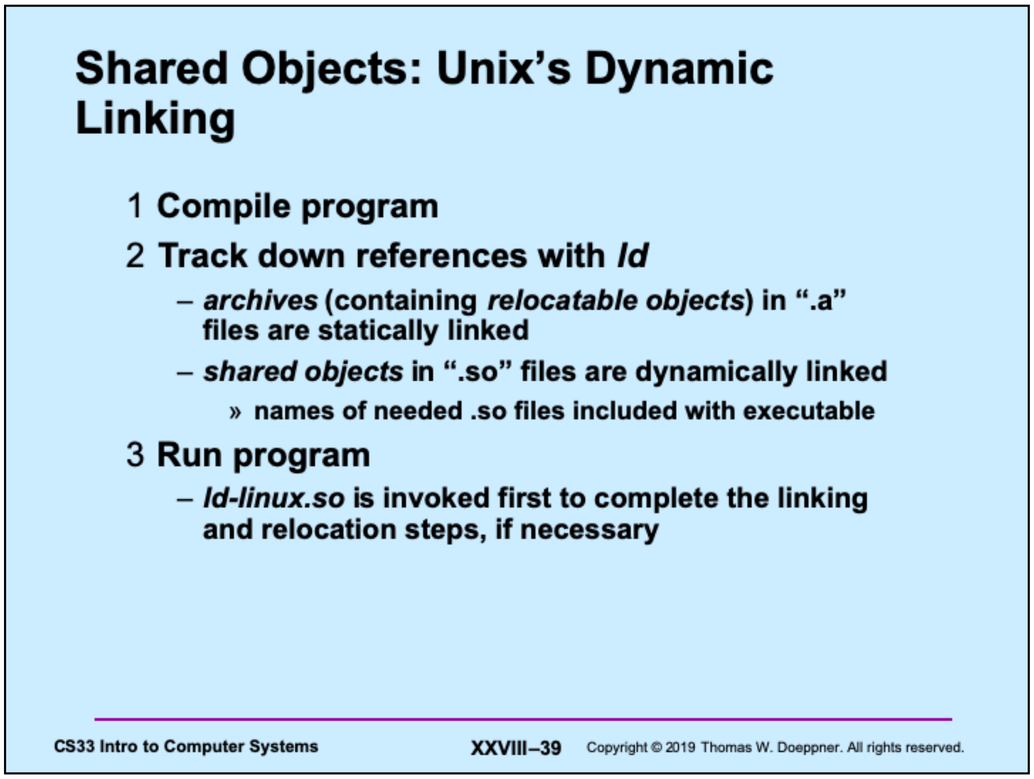
- printf is found to have a bug
  - perhaps a security problem
- All existing instances must be replaced
  - there are zillions of instances ...
- Do we have to re-link all programs that use printf?

Dynamic Linking

- Executable is not fully linked
  - contains list of needed libraries
- Linkages set up when executable is run

Benefits

- Without dynamic linking
  - every executable contains copy of printf (and other stuff)
    - » waste of disk space
    - » waste of primary memory
- With dynamic linking
  - just one copy of printf
    - » shared by all



Shared Objects: Unix's Dynamic Linking

1 **Compile program**

2 **Track down references with *ld***

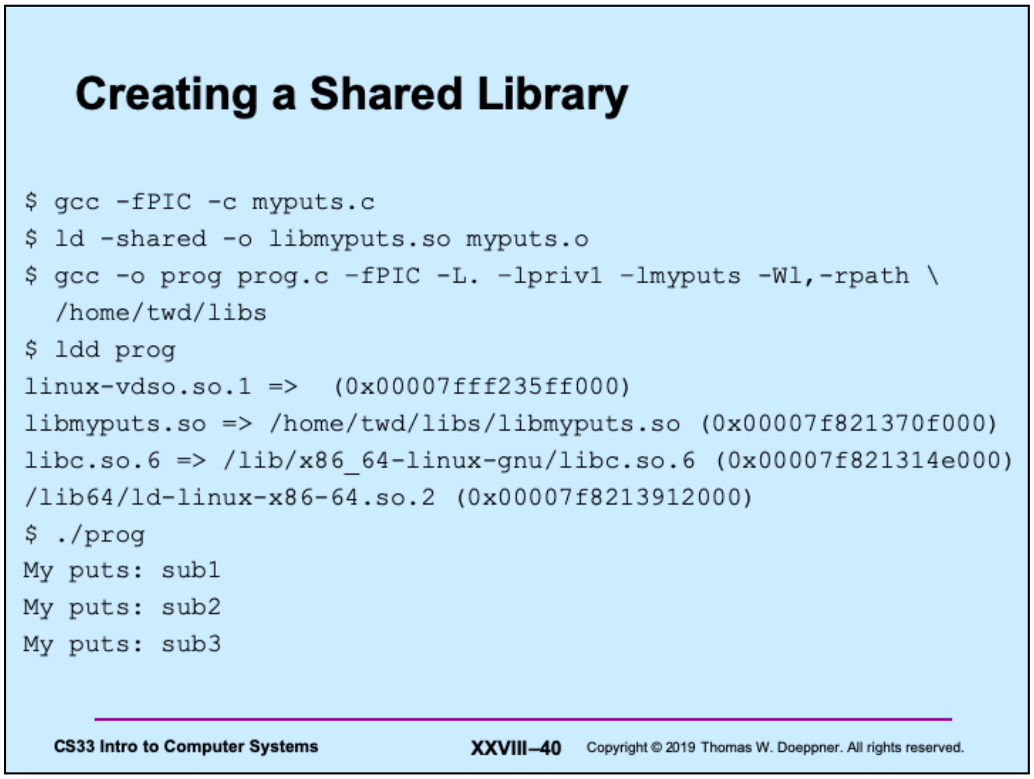
- ***archives*** **(containing *relocatable objects*) in ".a" files are statically linked**
- ***shared objects*** **in ".so" files are dynamically linked**
  - » **names of needed .so files included with executable**

3 **Run program**

- ***ld-linux.so*** **is invoked first to complete the linking and relocation steps, if necessary**

Linux supports two kinds of libraries — static libraries, contained in *archives*, whose names end with ".a" (e.g. *libc.a*) and *shared* objects, whose names end with ".so" (e.g. *libc.so*). When *ld* is invoked to handle the linking of object code, it is normally given a list of libraries in which to find unresolved references. If it resolves a reference within a *.a* file, it copies the code from the file and statically links it into the object code. However, if it resolves the reference within a *.so* file, it records the name of the shared object (not the complete path, just the final component) and postpones actual linking until the program is executed.

If the program is fully bound and relocated, then it is ready for direct execution. However, if it is not fully bound and relocated, then *ld* arranges things so that when the program is executed, rather than starting with the program's main routine, a runtime version of *ld*, called *ld-linux.so*, is called first. *ld-linux.so* maps all the required libraries into the address space and then calls the main routine.


Creating a Shared Library

```
$ gcc -fPIC -c myputs.c
$ ld -shared -o libmyputs.so myputs.o
$ gcc -o prog prog.c -fPIC -L. -lpriv1 -lmyputs -Wl,-rpath \
  /home/twd/libs
$ ldd prog
linux-vdso.so.1 =>  (0x00007fff235ff000)
libmyputs.so => /home/twd/libs/libmyputs.so (0x00007f821370f000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f821314e000)
/lib64/ld-linux-x86-64.so.2 (0x00007f8213912000)
$ ./prog
My puts: sub1
My puts: sub2
My puts: sub3
```

The –fPIC flag tells gcc to produce "position-independent code," which is something we discuss soon. The ld command invokes the loader directly. The –shared flag tells it to created a shared object. In this case, it's creating it from the object file *myputs.o* and calling the shared object *libmyputs.so*.

The "-Wl,-rpath /home/twd/libs" flag (the third character of the string is a lower-case L) tells the loader to indicate in the executable (prog) that ld-linux.so should look in the indicated directory for shared objects. (The "-Wl" part of the flag tells gcc to pass the rest of the flag to the loader.)

Order Still Matters

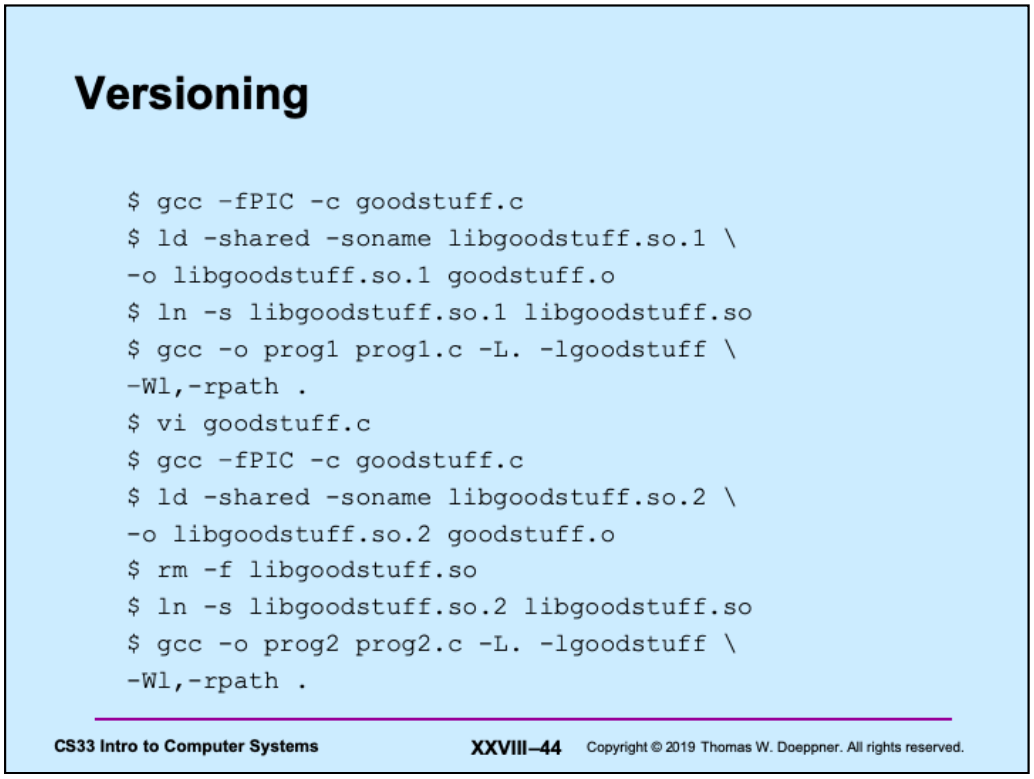
- All shared objects listed in the executable are loaded into the address space
  - whether needed or not
- ld-linux.so will find anything that's there
  - looks in the order in which shared objects are listed

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

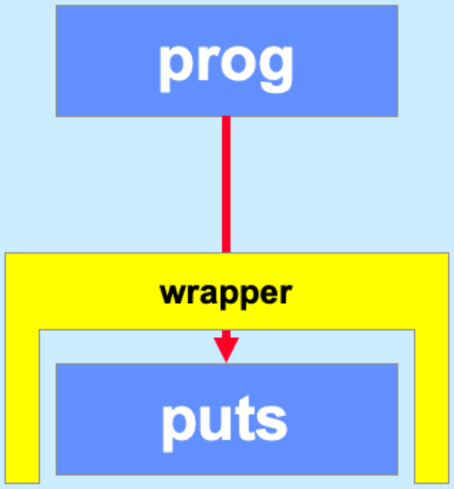
```
$ gcc -fPIC -c goodstuff.c
$ ld -shared -soname libgoodstuff.so.1 \
-o libgoodstuff.so.1 goodstuff.o
$ ln -s libgoodstuff.so.1 libgoodstuff.so
$ gcc -o prog1 prog1.c -L. -lgoodstuff \
-Wl,-rpath .
$ vi goodstuff.c
$ gcc -fPIC -c goodstuff.c
$ ld -shared -soname libgoodstuff.so.2 \
-o libgoodstuff.so.2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

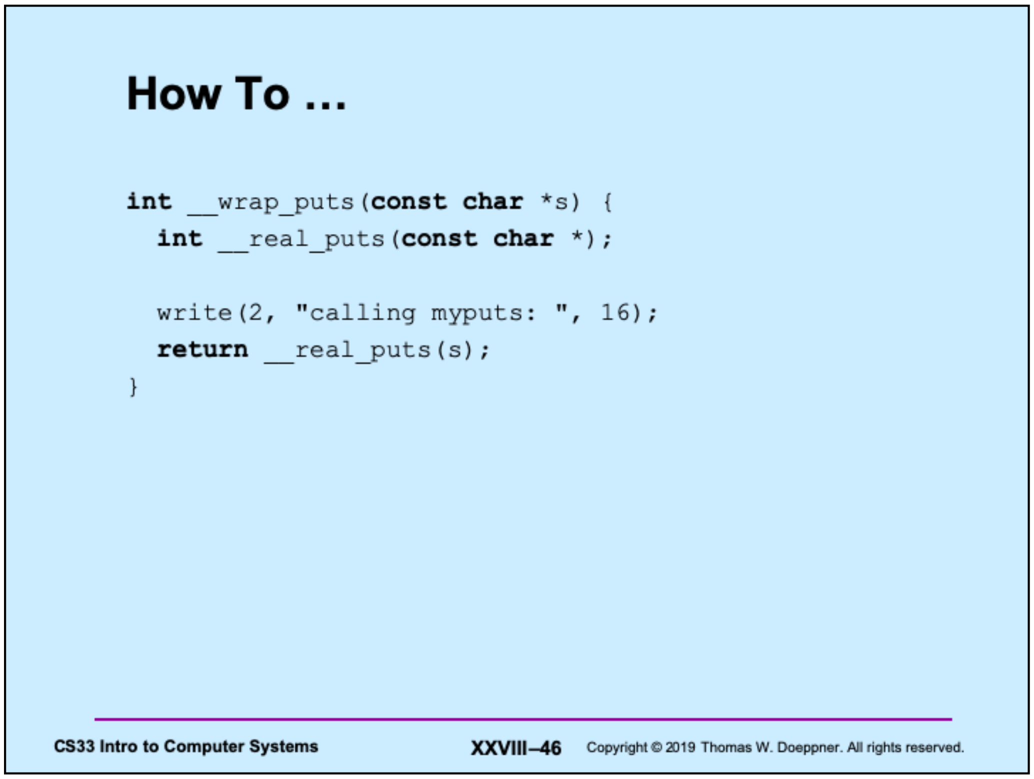
Here we are creating two versions of libgoodstuff, in libgoodstuff.so.1 and in libgoodstuff.so.2. Each is created by invoking the loader directly via the "ld" command. The "-soname" flag tells the loader to include in the shared object its name, which is the string following the flag ("libgoodstuff.so.1" 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.

Interpositioning

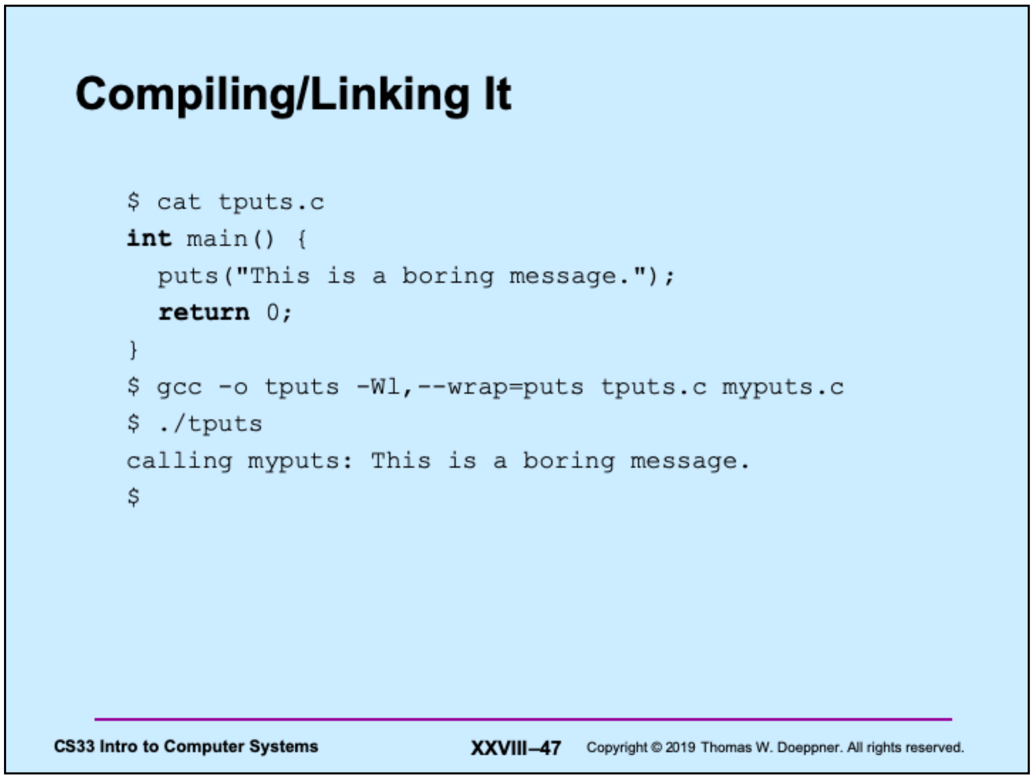




How To ...

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

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

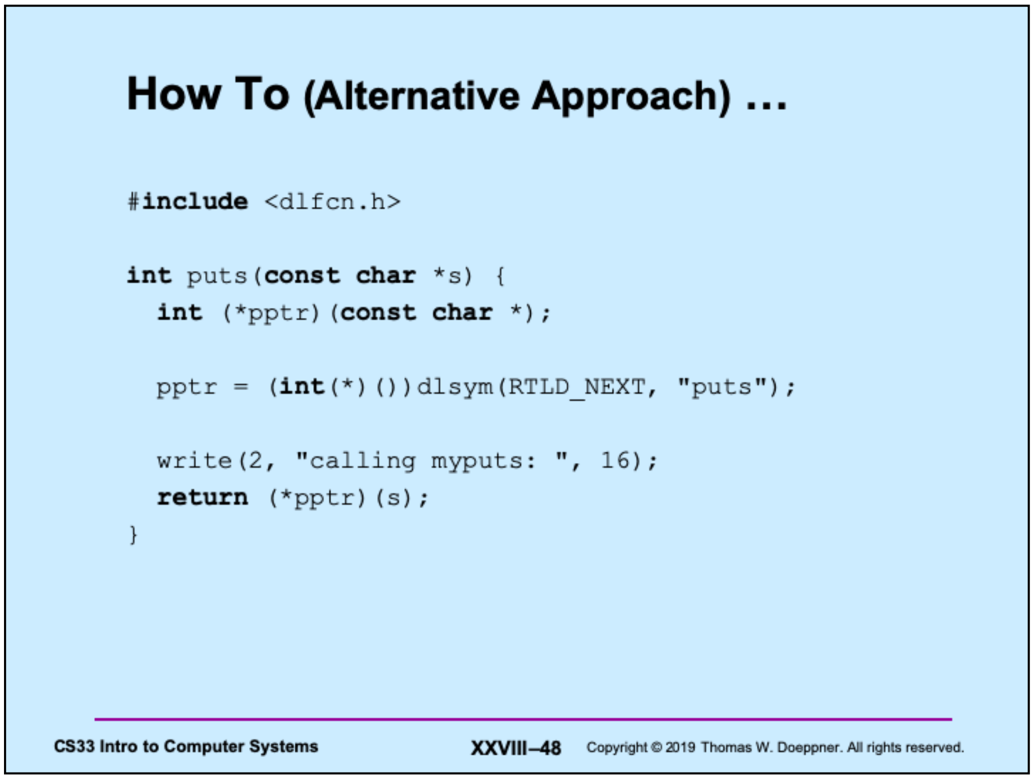
__wrap_puts is the "wrapper" for *puts*. *__real_puts* is the "real" *puts* routine. What we want is for calls to *puts* to go to *__wrap_puts*, and calls to *__real_puts* to go to the real *puts* routine (in stdio).



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

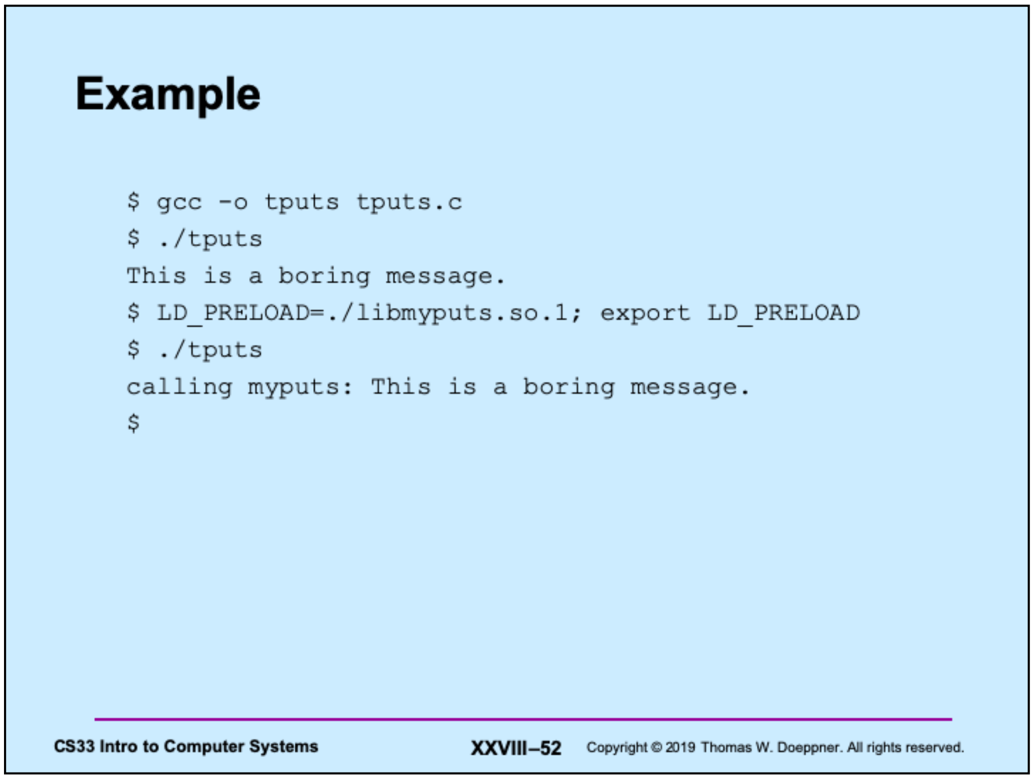
Delayed Wrapping

- 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

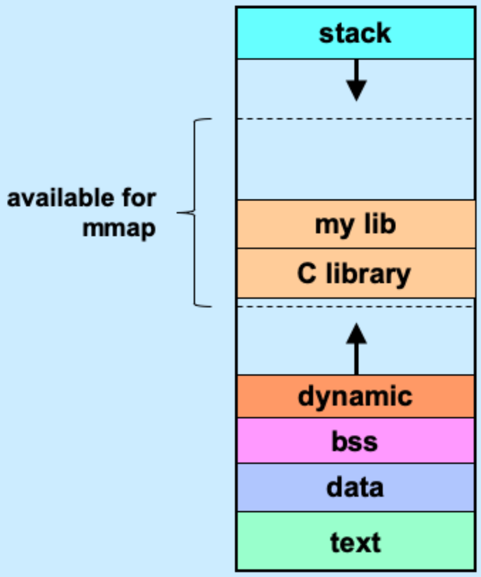


Example

```
$ 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.
$
```

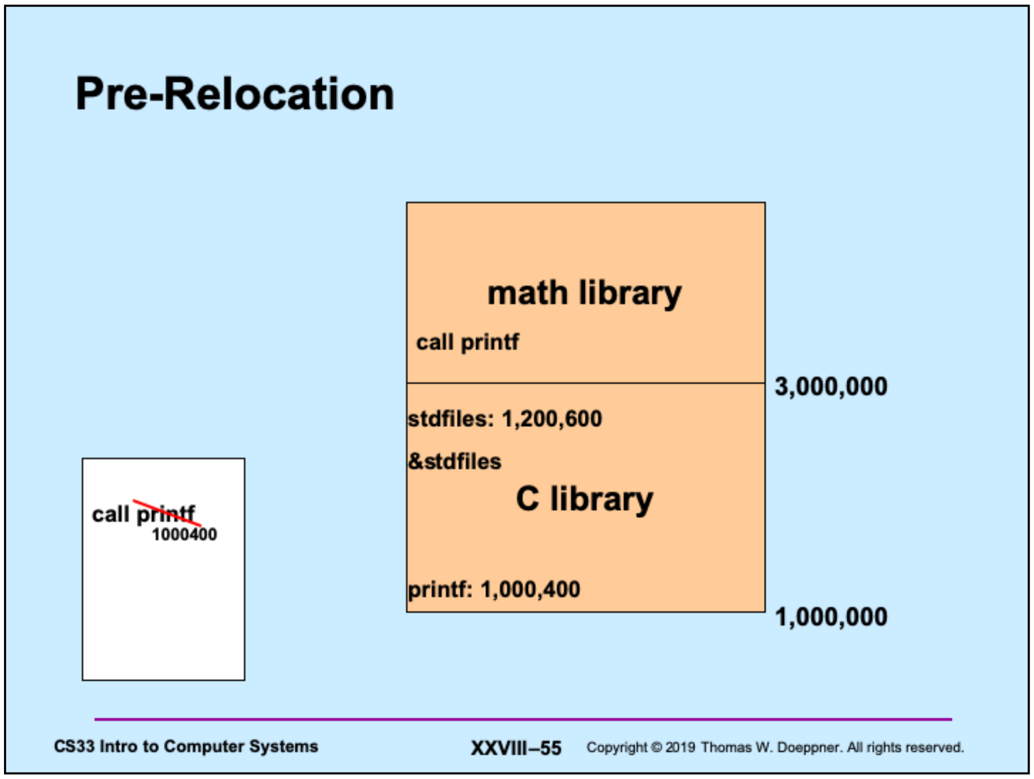
Here we add "LD_PRELOAD=./libmyputs.so.1" to the environment. The export command instructs the shell to supply this as part of the environment for the commands it runs.

Mmapping Libraries



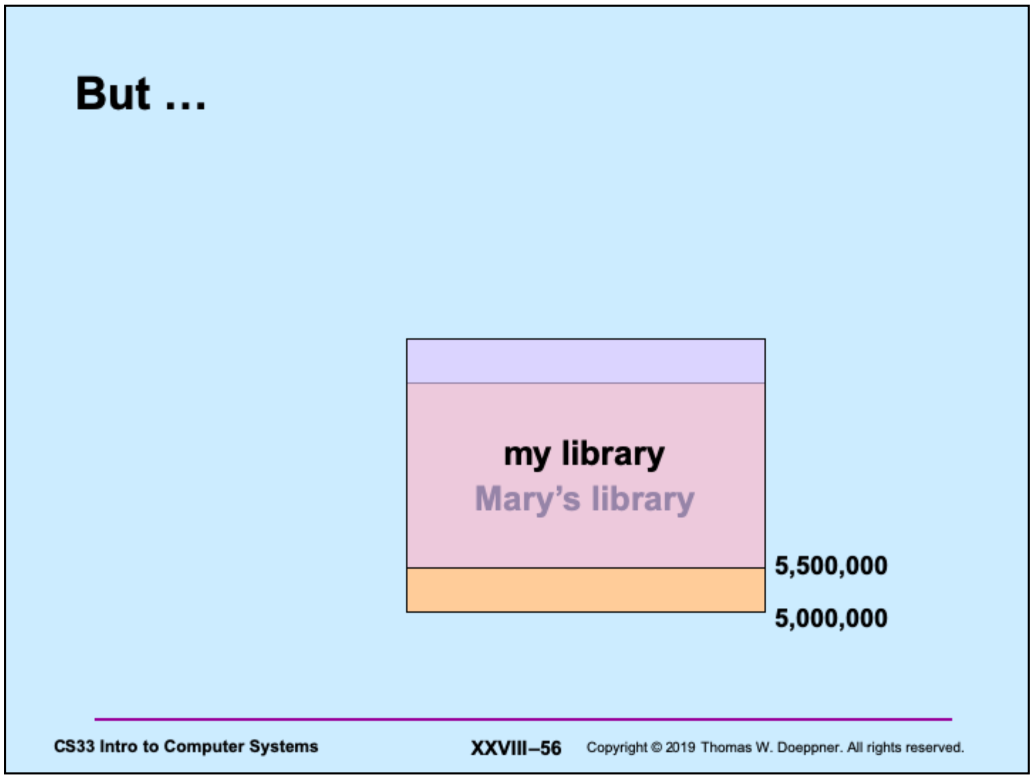
Problem

- **How is relocation handled?**

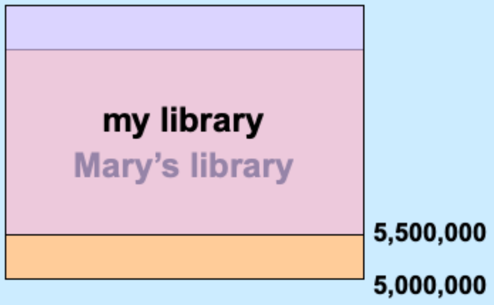


Pre-Relocation

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 identifies, such as stdfiles. Its address would also be known.

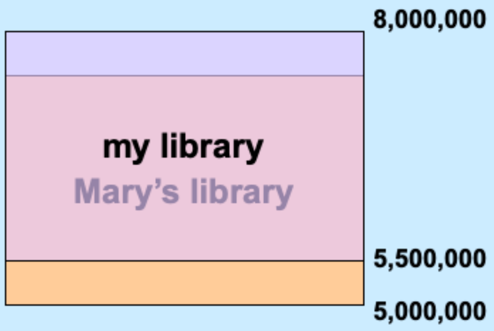


But …



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 2

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?

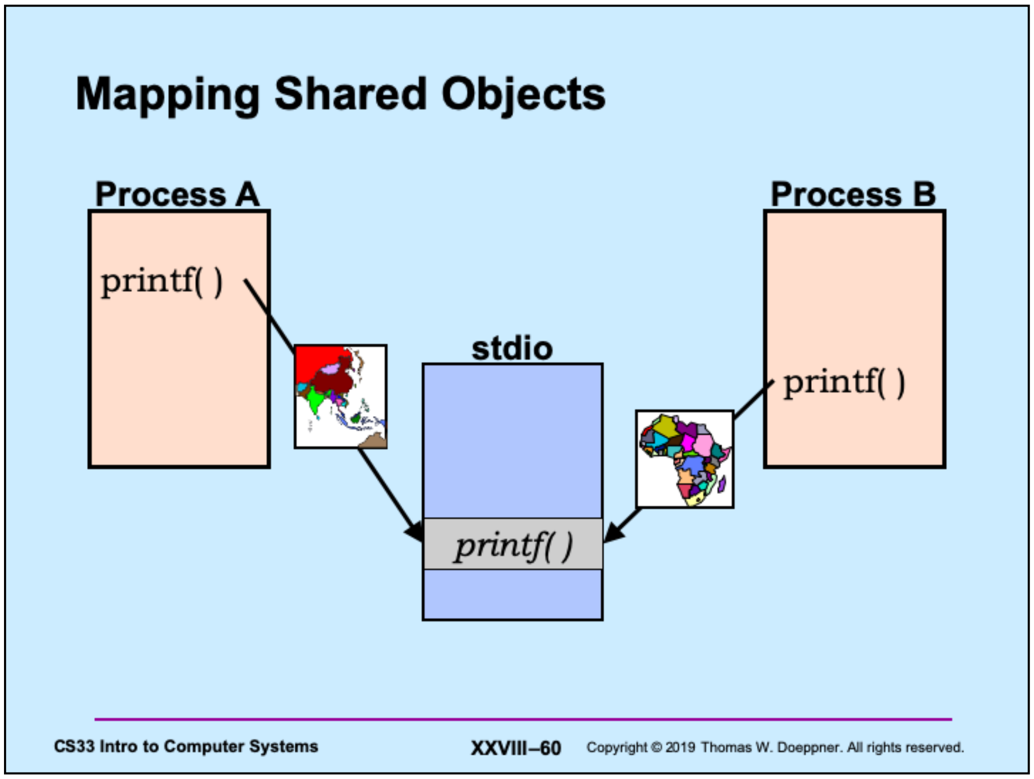
a) the MAP_SHARED option

b) the MAP_PRIVATE option

c) mmap can't be used in this situation

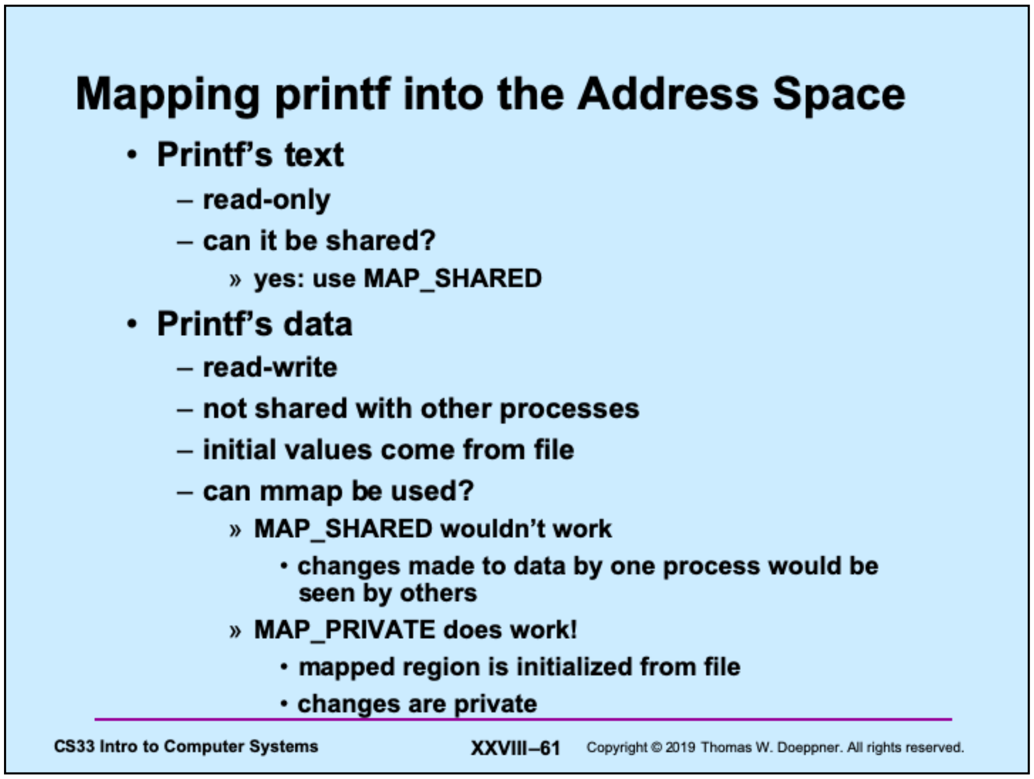
Relocation Revisited

- 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



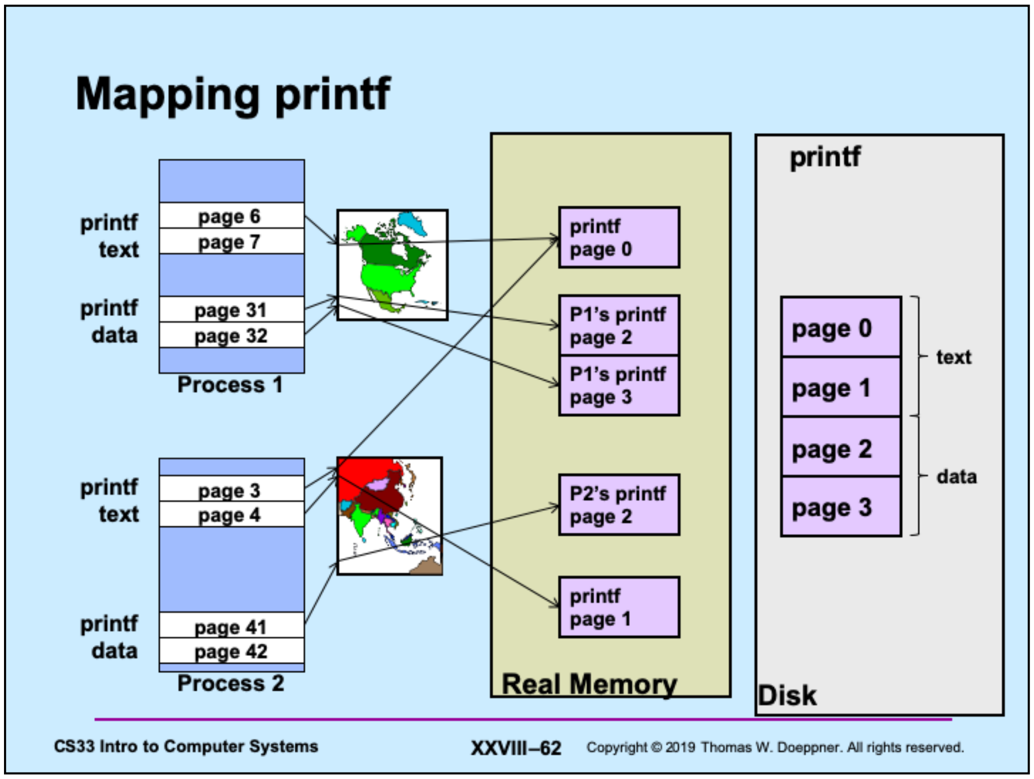
Mapping Shared Objects

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

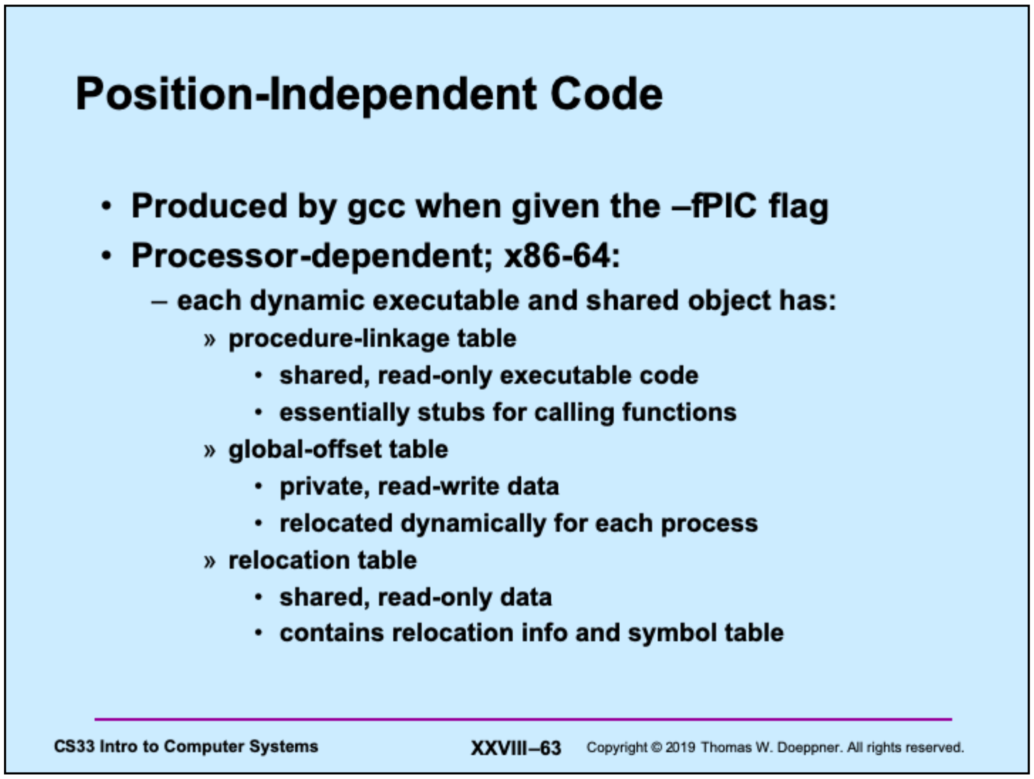

Mapping printf into the Address Space

- 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

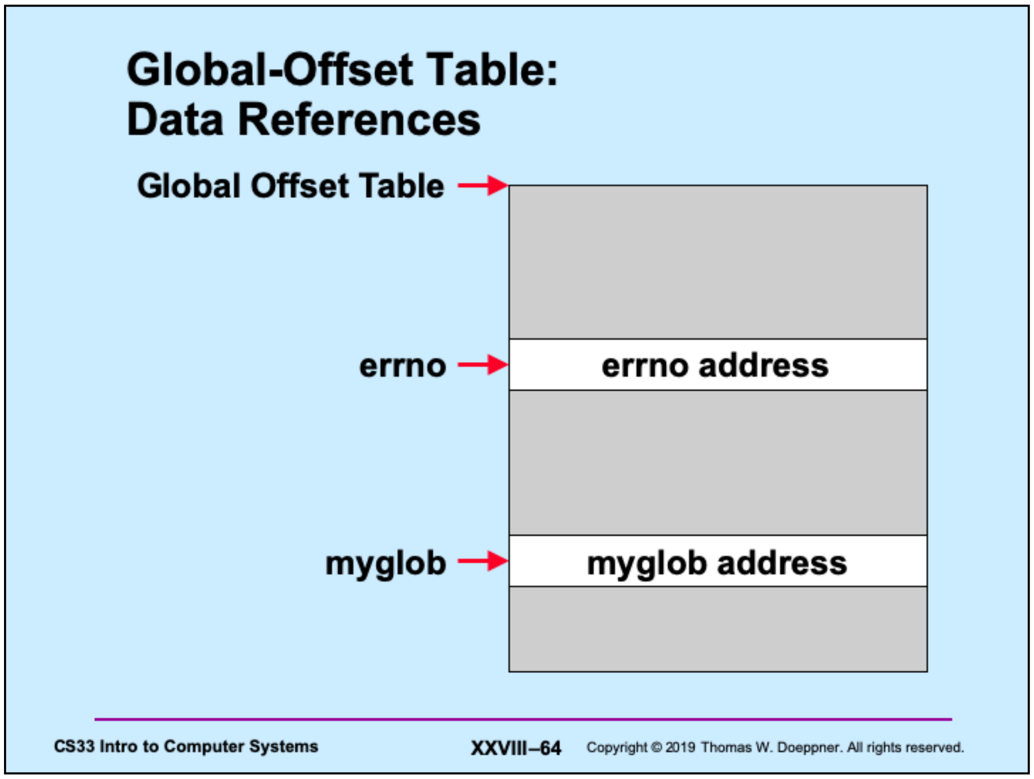


Position-Independent Code

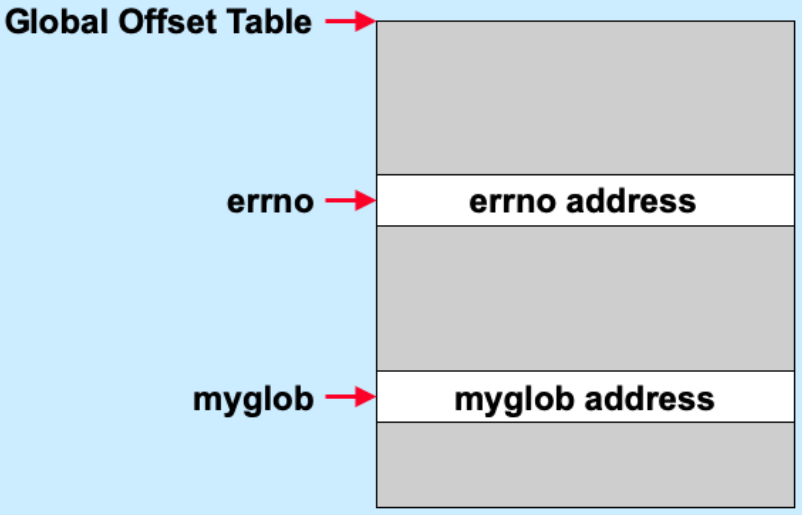
- Produced by gcc when given the –fPIC flag
- Processor-dependent; x86-64:
  - each dynamic executable and shared object has:
    - » procedure-linkage table
      - shared, read-only executable code
      - essentially stubs for calling functions
    - » global-offset table
      - private, read-write data
      - relocated dynamically for each process
    - » relocation table
      - shared, read-only data
      - contains relocation info and symbol table

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.



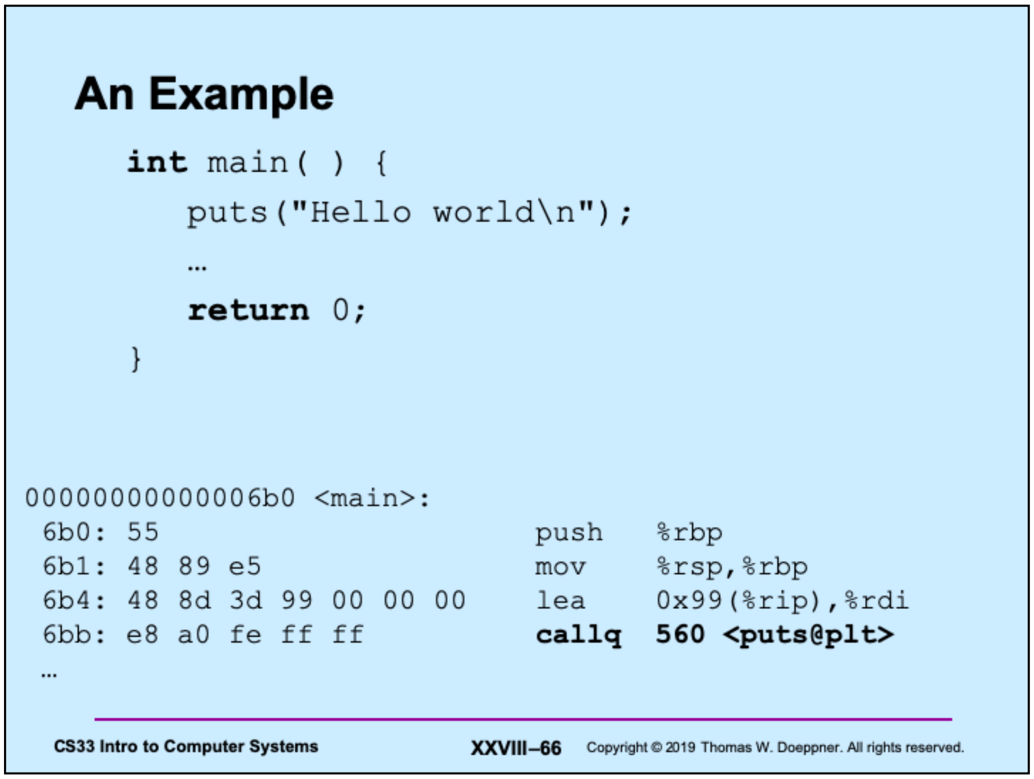
Global-Offset Table: Data References



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

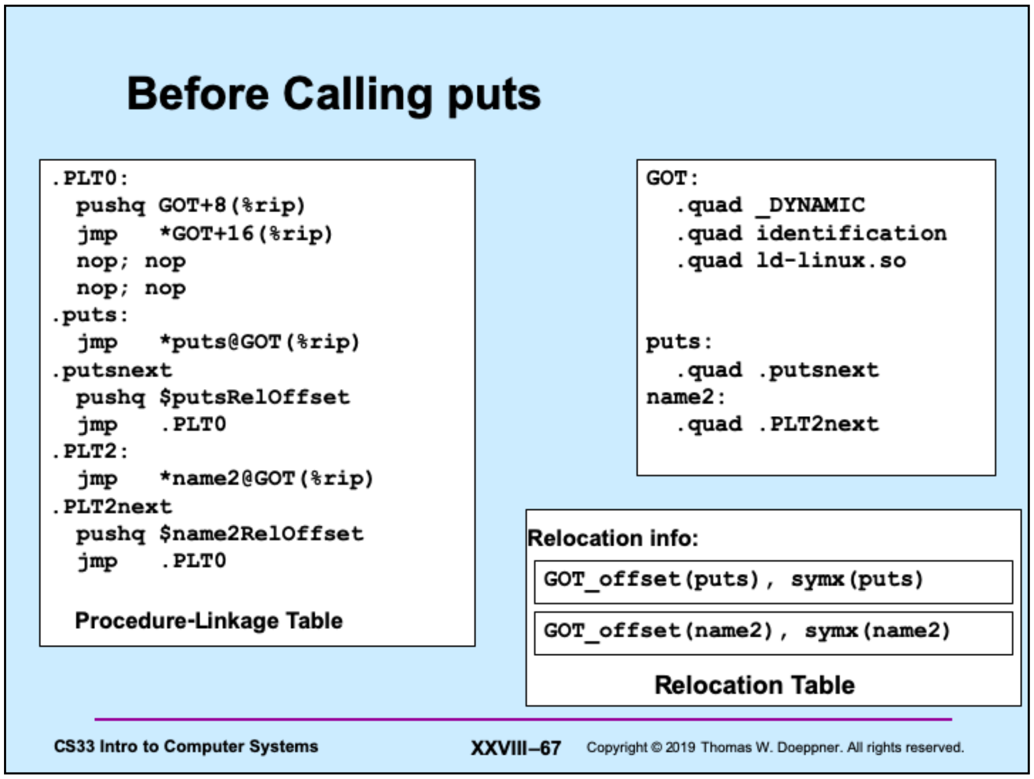


An Example

```
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.



Before Calling puts

```
.PLT0:
  pushq GOT+8(%rip)
  jmp   *GOT+16(%rip)
  nop; nop
  nop; nop
.puts:
  jmp   *puts@GOT(%rip)
.putsnext
  pushq $putsRelOffset
  jmp   .PLT0
.PLT2:
  jmp   *name2@GOT(%rip)
.PLT2next
  pushq $name2RelOffset
  jmp   .PLT0
```

Procedure-Linkage Table

```
GOT:
  .quad _DYNAMIC
  .quad identification
  .quad ld-linux.so


puts:
  .quad .putsnext
name2:
  .quad .PLT2next
```

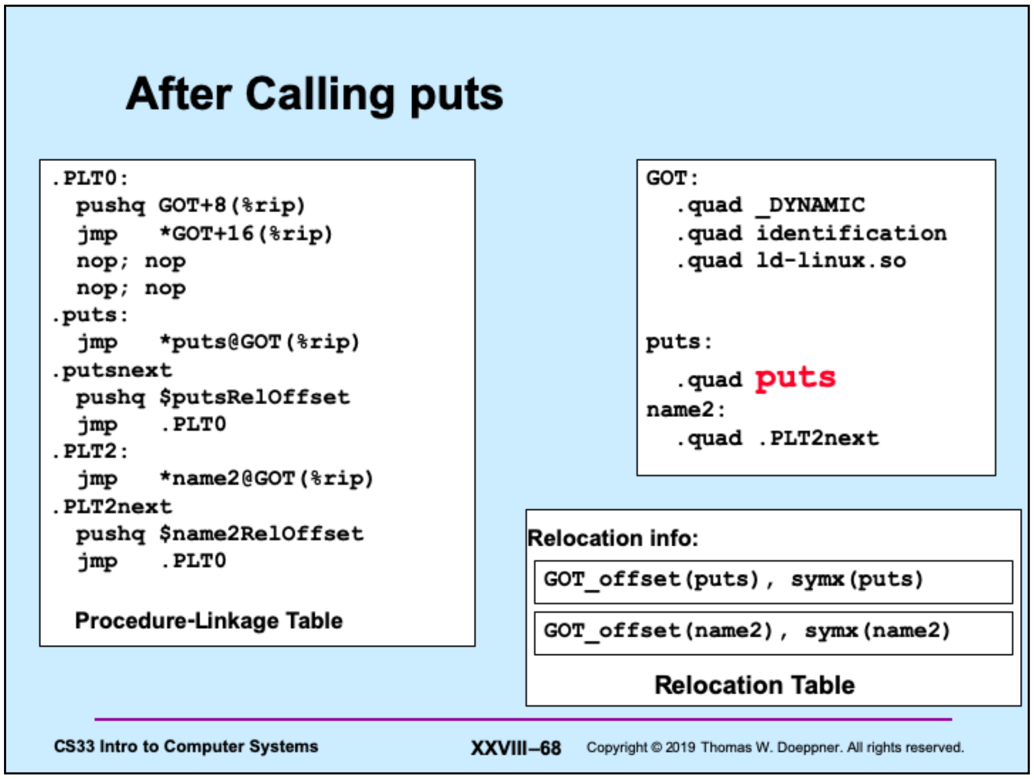
Relocation info:

```
GOT_offset(puts), symx(puts)
GOT_offset(name2), symx(name2)
```

Relocation 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).



After Calling puts

```
.PLT0:
  pushq GOT+8(%rip)
  jmp   *GOT+16(%rip)
  nop; nop
  nop; nop
.puts:
  jmp   *puts@GOT(%rip)
.putsnext
  pushq $putsRelOffset
  jmp   .PLT0
.PLT2:
  jmp   *name2@GOT(%rip)
.PLT2next
  pushq $name2RelOffset
  jmp   .PLT0
```

Procedure-Linkage Table

```
GOT:
  .quad _DYNAMIC
  .quad identification
  .quad ld-linux.so

puts:
  .quad puts
name2:
  .quad .PLT2next
```

Relocation info:

```
GOT_offset(puts), symx(puts)
GOT_offset(name2), symx(name2)
```

Relocation Table

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.

Quiz 2

On the second and subsequent calls to *puts*

a) **control goes directly to *puts***
b) **control goes to an instruction that jumps to *puts***
c) **control still goes to ld-linux.so, but it now transfers control directly to *puts***