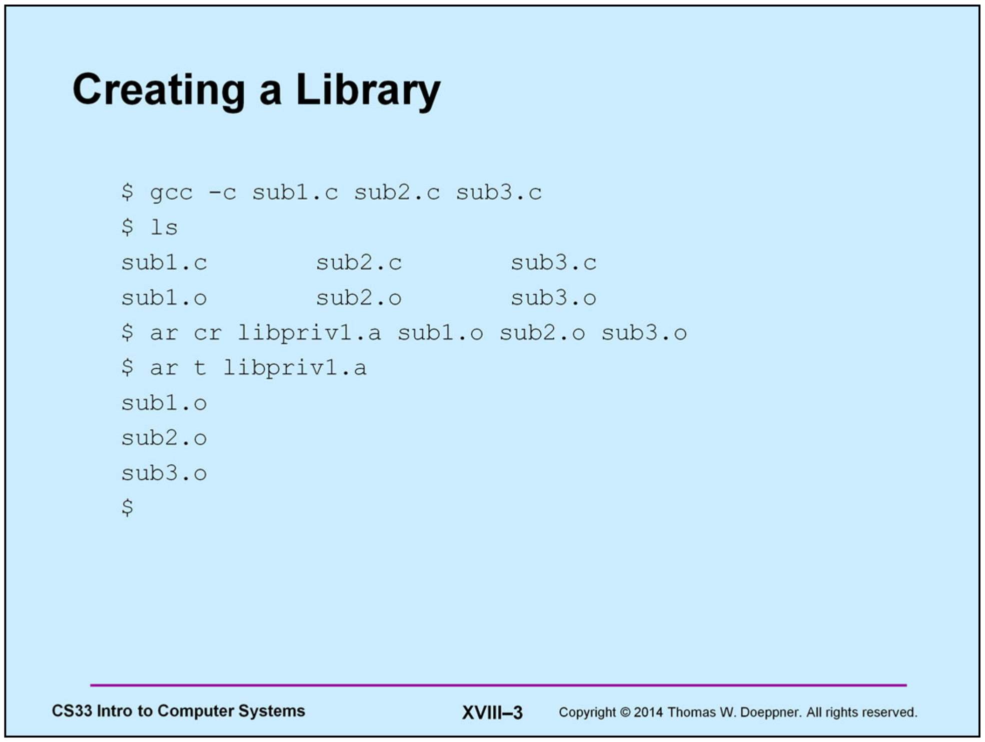
CS 33

Libraries

Libraries

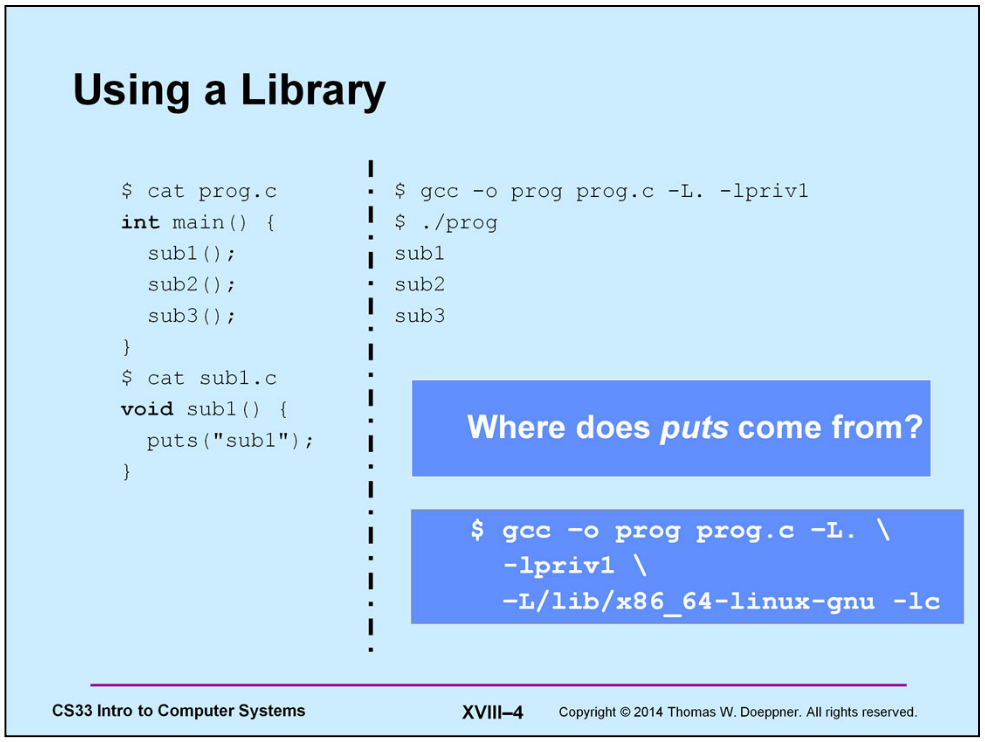
- Collections of useful stuff
- Incorporate items into your program
- Replace existing items with new stuff
- Often ugly …



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 routine "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

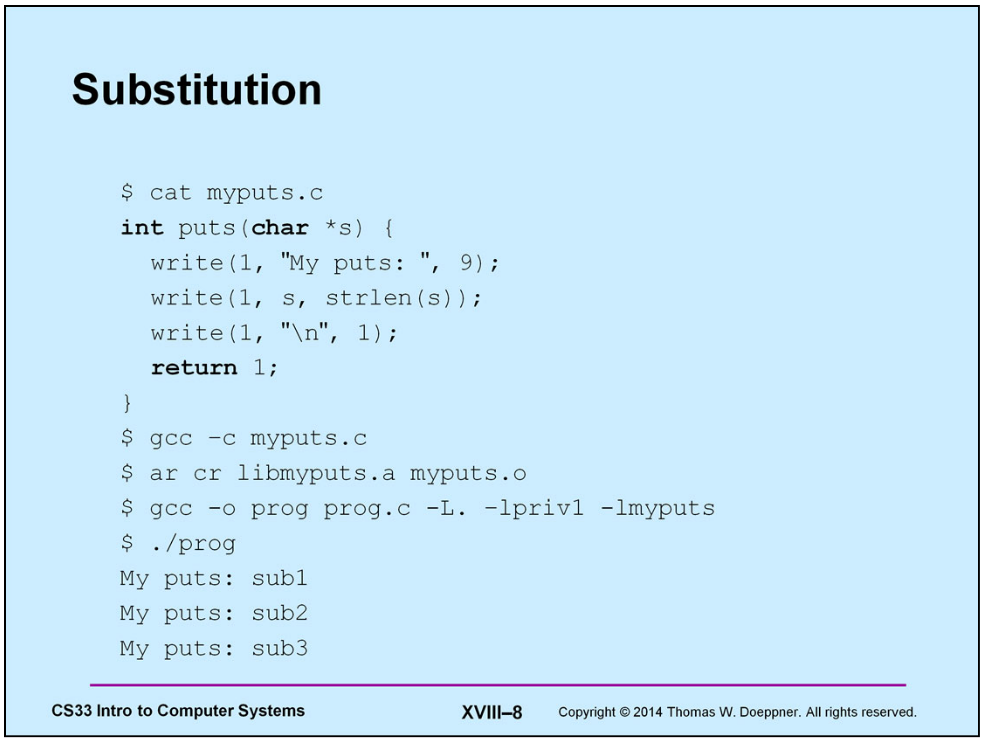
- ld puts in the executable:
  - (assume all .c files have been compiled into .o files)
  - all .o files from argument list (including those newly compiled)
  - .o files from archives as needed to satisfy unresolved references
    - » some may have their own unresolved references that may need to be 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
```

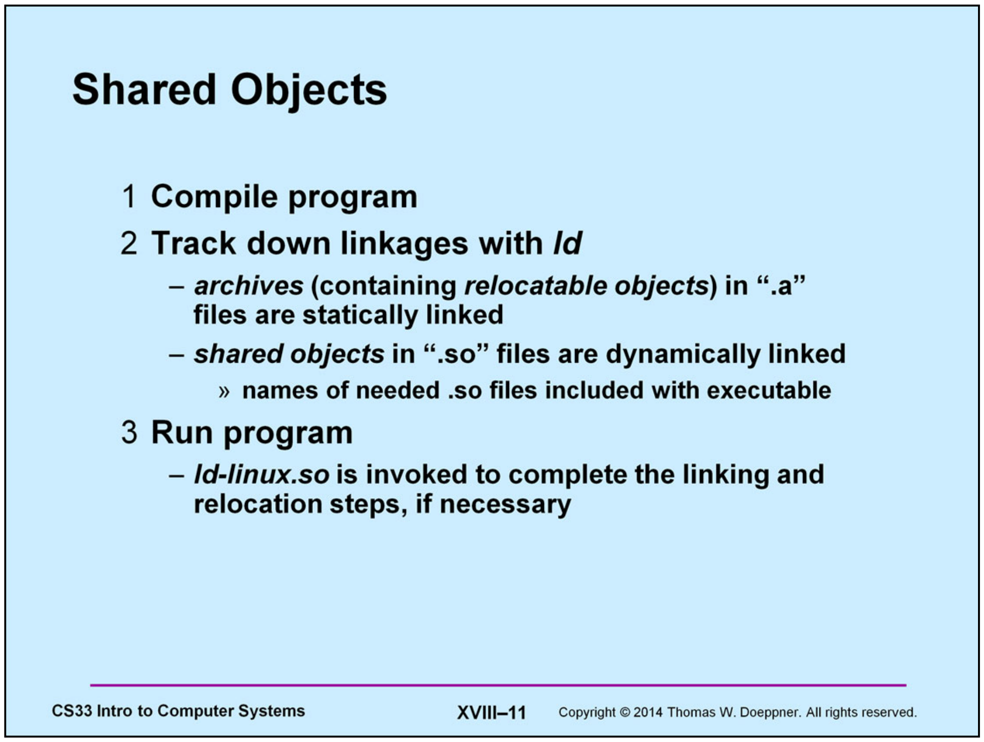
The routine “write” is a low-level I/O routine, used by puts, printf, and anything else that ultimately does output. Its first argument indicates where the data is going: “1” refers to “standard output”, which is normally the display. The second argument is the data to be output, and the third argument is its length. We discuss all this in much more detail in a future lecture.

A 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 tracked down at runtime
  - thus call is not bound to code until program is executed
  - the actual code is shared with other programs



Shared Objects

1 **Compile program**

2 **Track down linkages with *ld***

– ***archives*** **(containing *relocatable objects*) in ".a" files are statically linked**

– ***shared objects*** **in ".so" files are dynamically linked**

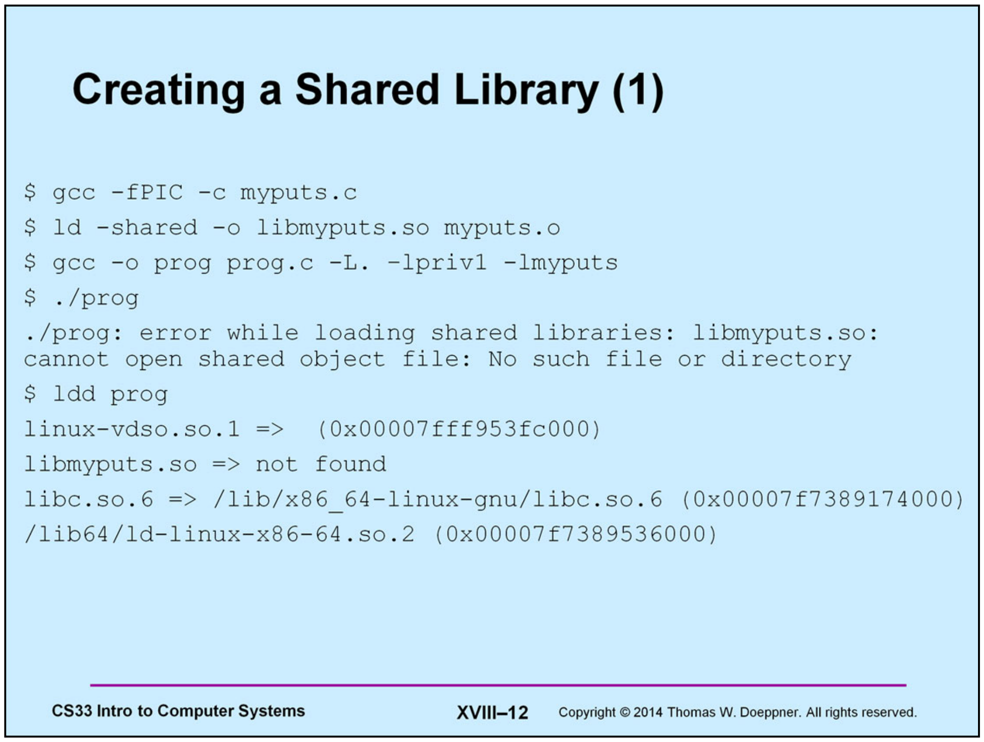
» **names of needed .so files included with executable**

3 **Run program**

– ***ld-linux.so*** **is invoked 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 (we describe how this mapping works in a later lecture) and then calls the main routine.

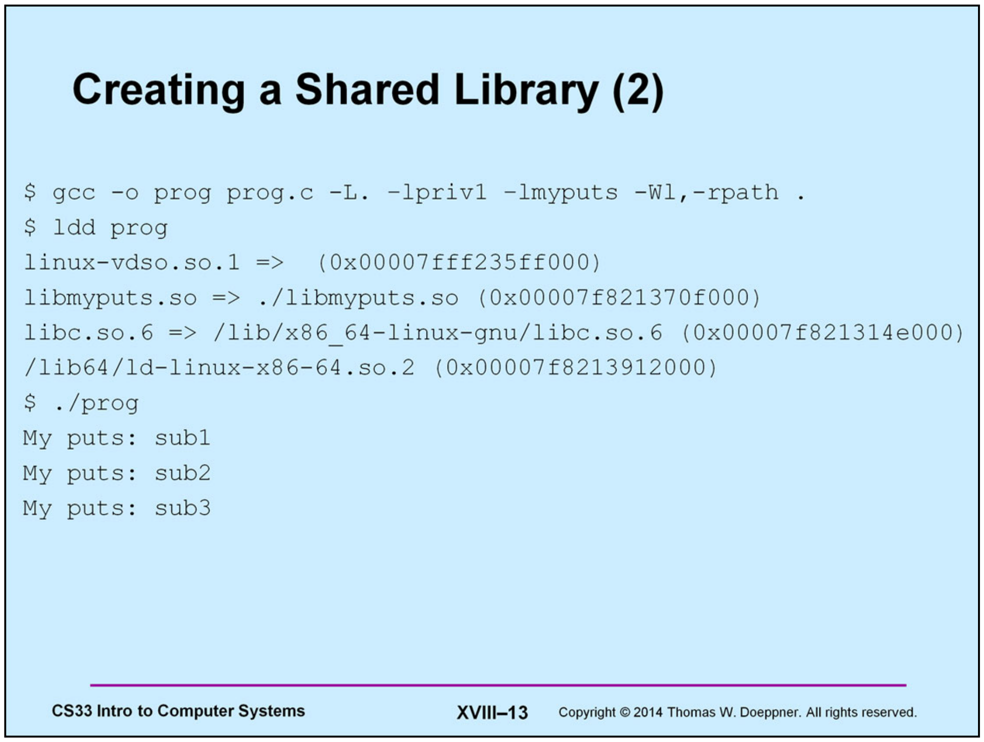


Creating a Shared Library (1)

```
$ gcc -fPIC -c myputs.c
$ ld -shared -o libmyputs.so myputs.o
$ gcc -o prog prog.c -L. -lpriv1 -lmyputs
$ ./prog
./prog: error while loading shared libraries: libmyputs.so:
cannot open shared object file: No such file or directory
$ ldd prog
linux-vdso.so.1 =>  (0x00007fff953fc000)
libmyputs.so => not found
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f7389174000)
/lib64/ld-linux-x86-64.so.2 (0x00007f7389536000)
```

The –fPIC flag tells gcc to produce “position-independent code,” which is something we discuss in a later lecture. 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 error occurs because we haven’t indicated in the executable (prog) where ld-linux.so should look for shared objects. The ldd (list dynamic dependencies) command, which looks at all the shared objects referenced in the executable and prints out where they are found, shows us what the problem is.



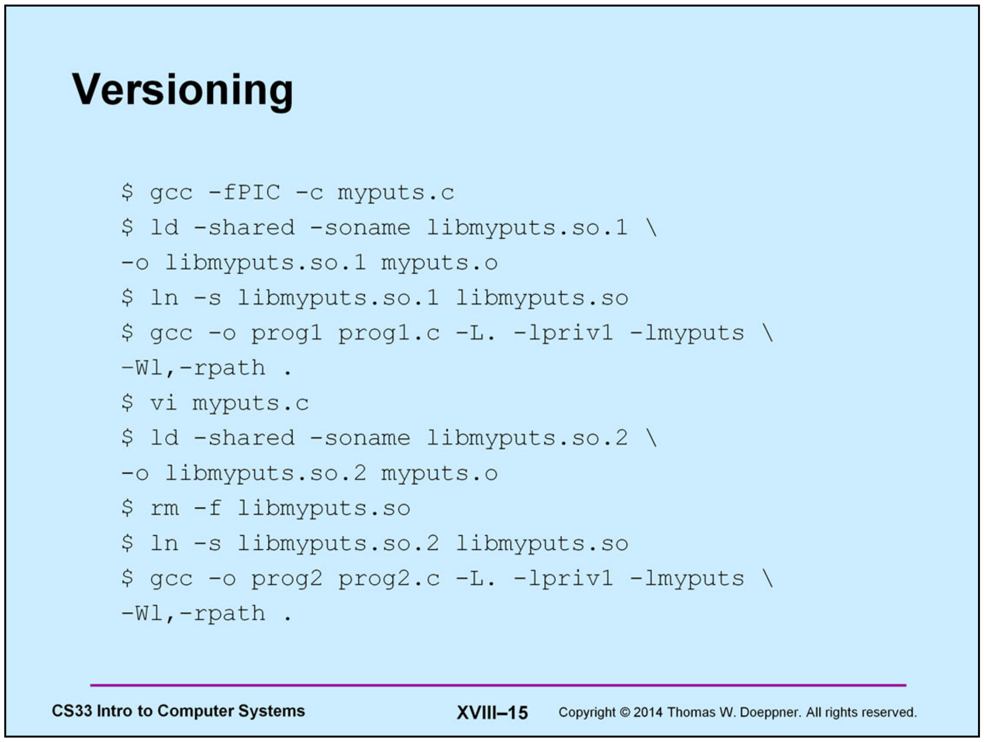
Creating a Shared Library (2)

```
$ gcc -o prog prog.c -L. -lpriv1 -lmyputs -Wl,-rpath .
$ ldd prog
linux-vdso.so.1 =>  (0x00007fff235ff000)
libmyputs.so => ./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 "-Wl,-rpath ." 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 current directory (referred to as ".") for shared objects. (The "-Wl" part of the flag tells gcc to pass the rest of the flag to the loader.)

Order Doesn't Matter (as much) ...

- 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



Versioning

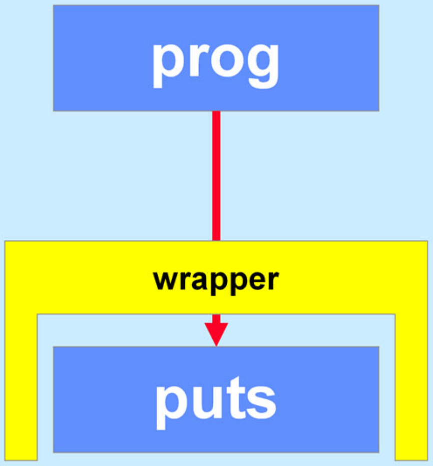
```
$ gcc -fPIC -c myputs.c
$ ld -shared -soname libmyputs.so.1 \
-o libmyputs.so.1 myputs.o
$ ln -s libmyputs.so.1 libmyputs.so
$ gcc -o prog1 prog1.c -L. -lpriv1 -lmyputs \
-Wl,-rpath .
$ vi myputs.c
$ ld -shared -soname libmyputs.so.2 \
-o libmyputs.so.2 myputs.o
$ rm -f libmyputs.so
$ ln -s libmyputs.so.2 libmyputs.so
$ gcc -o prog2 prog2.c -L. -lpriv1 -lmyputs \
-Wl,-rpath .
```

Here we are creating two versions of libmyputs, in libmyputs.so.1 and in libmyputs.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 (“libmyputs.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, libmyputs.so refers to the same file as does libmyputs.so.1. Thus the second invocation of gcc, where it refers to –lmyputs (which expands to libmyputs.so), is actually referring to libmyputs.so.1.

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

If prog1 is now run, it refers to libmyputs.so.1, so it gets the old version (version 1), but if prog2 is run, it refers to libmyputs.so.2, so it gets the new version (version 2). Thus programs using both versions of myputs 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);
}
```

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.
$
```

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);
}
```

What's Going On …

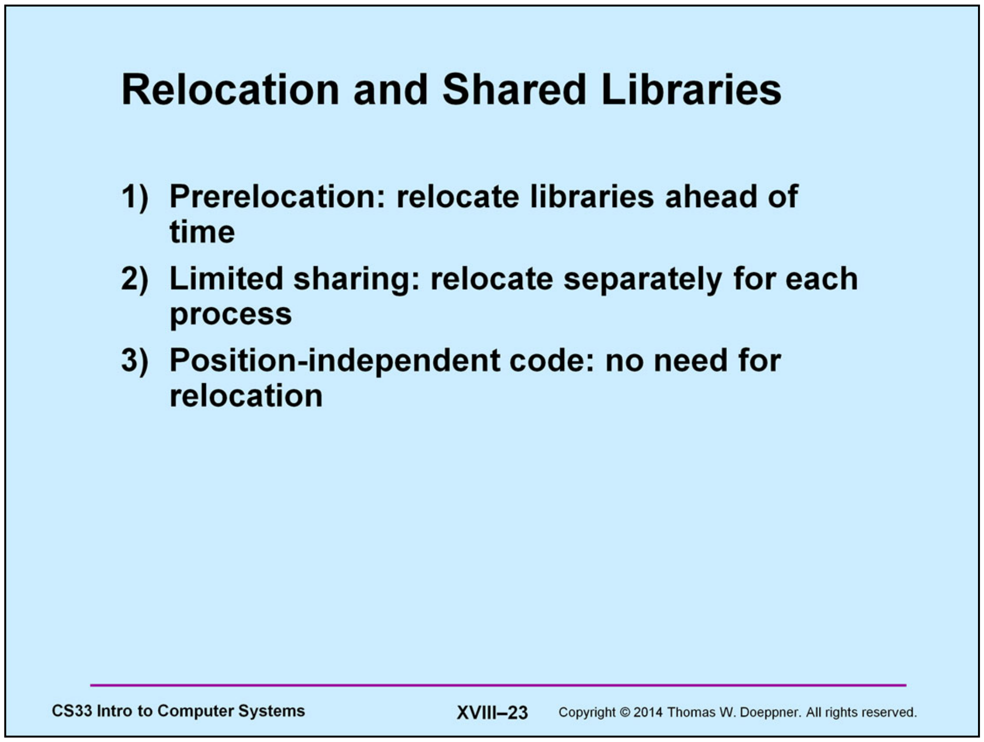
- gcc/ld
  - compiles code
  - does static linking
    - » searches list of libraries
    - » adds references to shared objects
- runtime
  - program invokes *ld-linux.so* to finish linking
    - » maps in shared objects
    - » does relocation and procedure linking as required
  - *dlsym* invokes *ld-linux.so* 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

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.
$
```



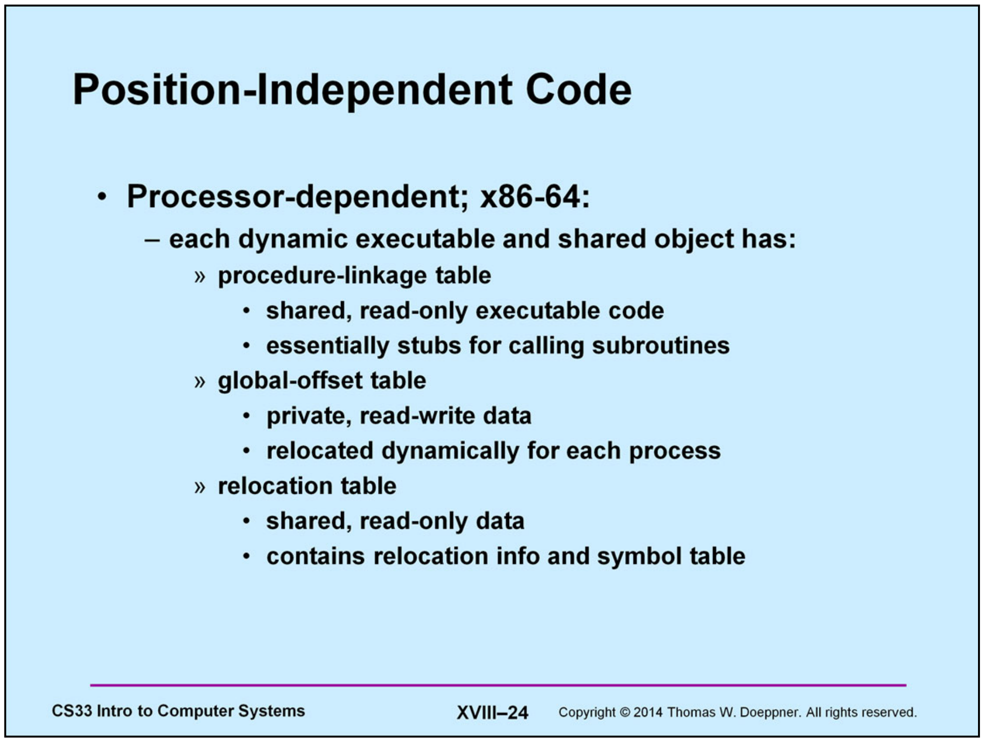
Relocation and Shared Libraries

1) Prerelocation: relocate libraries ahead of time
2) Limited sharing: relocate separately for each process
3) Position-independent code: no need for relocation

Clearly all processes using printf will have to have their own private copies of its data and bss areas. But what about printf's text? If we didn't have to worry about relocation, then all processes could share it. If all users of *printf* agree to load it and everything it references into the same locations in their address spaces, we would have no relocation problem, and thus the text could be shared among all processes. But such agreement is, in general, hard to achieve. It is, however, the approach used in Windows.

A possibility might be for the processes using *printf* to copy it into their address spaces when they start up, then for each process to relocate its copy. This definitely works, but is a bit cumbersome.

Another possibility is for *printf* to be written in such a way that relocation is not necessary. Code written in this fashion is known as *position-independent code* (PIC).

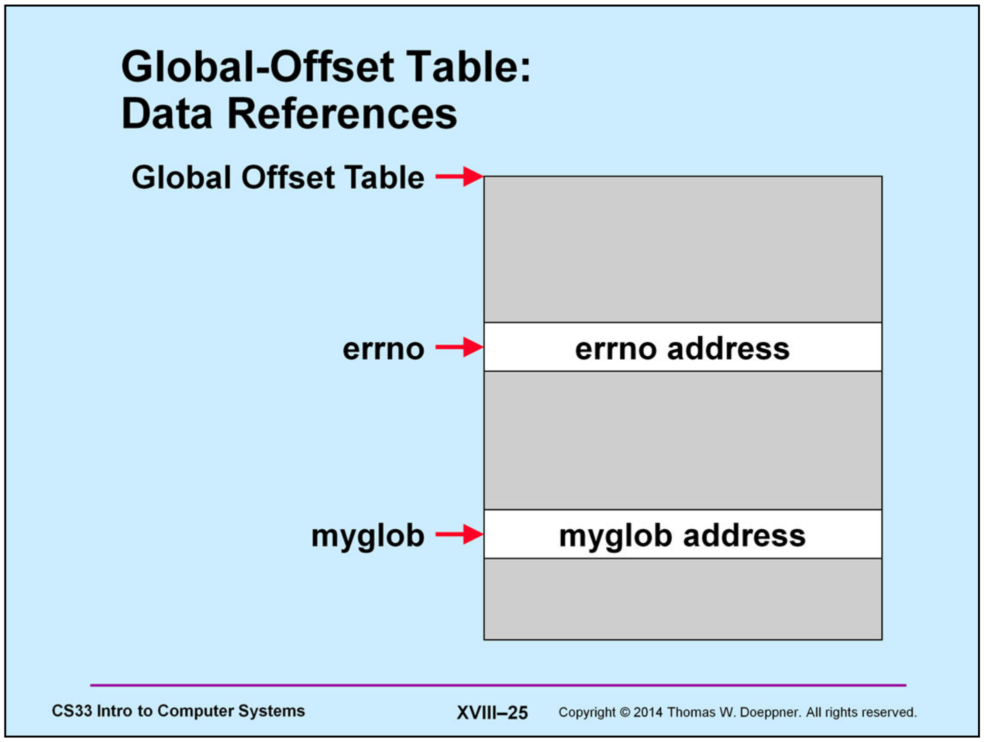


Position-Independent Code

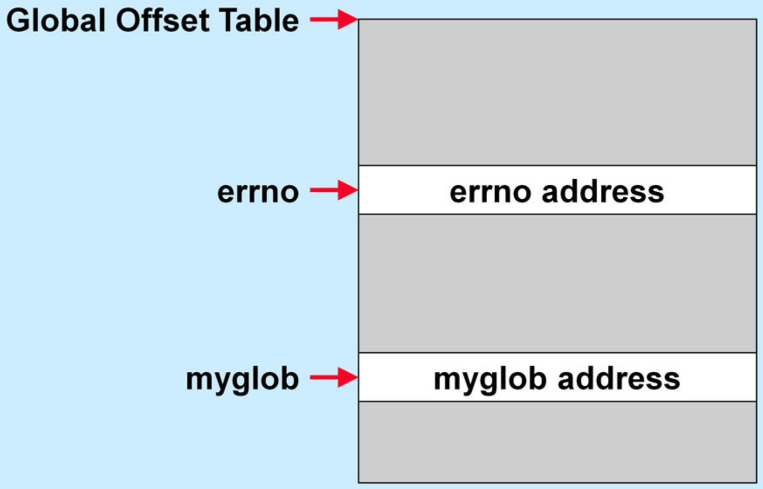
- Processor-dependent; x86-64:
  - each dynamic executable and shared object has:
    - » procedure-linkage table
      - shared, read-only executable code
      - essentially stubs for calling subroutines
    - » global-offset table
      - private, read-write data
      - relocated dynamically for 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.



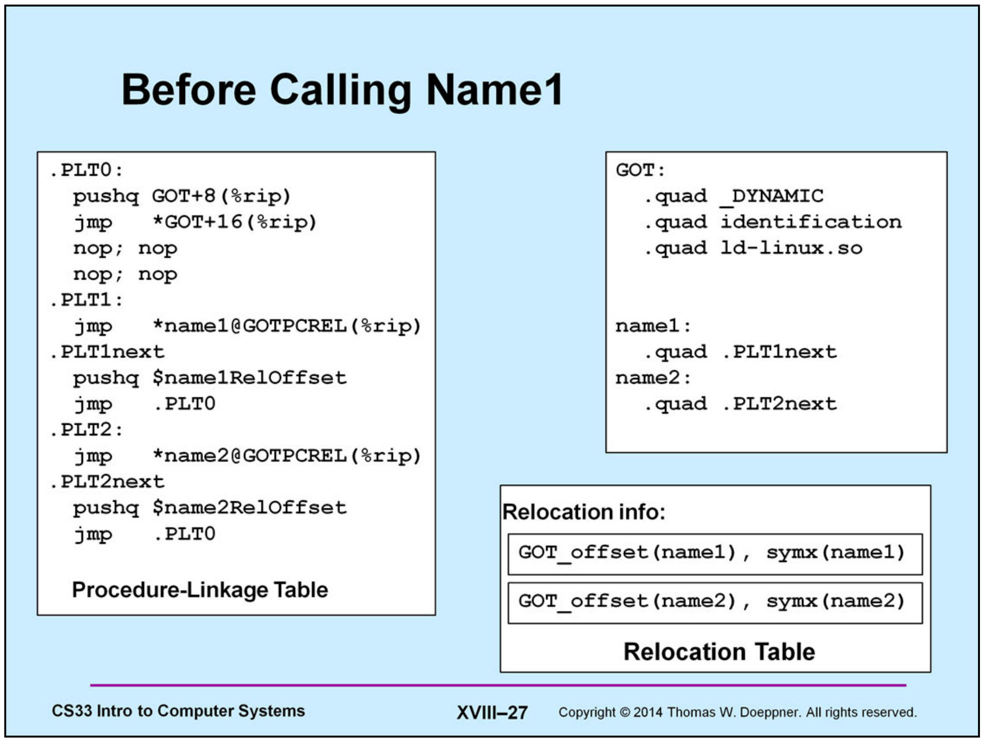
Global-Offset Table: Data References



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.

Procedures in Shared Objects

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



Before Calling Name1

```
.PLT0:
  pushq GOT+8(%rip)
  jmp   *GOT+16(%rip)
  nop; nop
  nop; nop
.PLT1:
  jmp   *name1@GOTPCREL(%rip)
.PLT1next
  pushq $name1RelOffset
  jmp   .PLT0
.PLT2:
  jmp   *name2@GOTPCREL(%rip)
.PLT2next
  pushq $name2RelOffset
  jmp   .PLT0
```

Procedure-Linkage Table

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


name1:
  .quad .PLT1next
name2:
  .quad .PLT2next
```

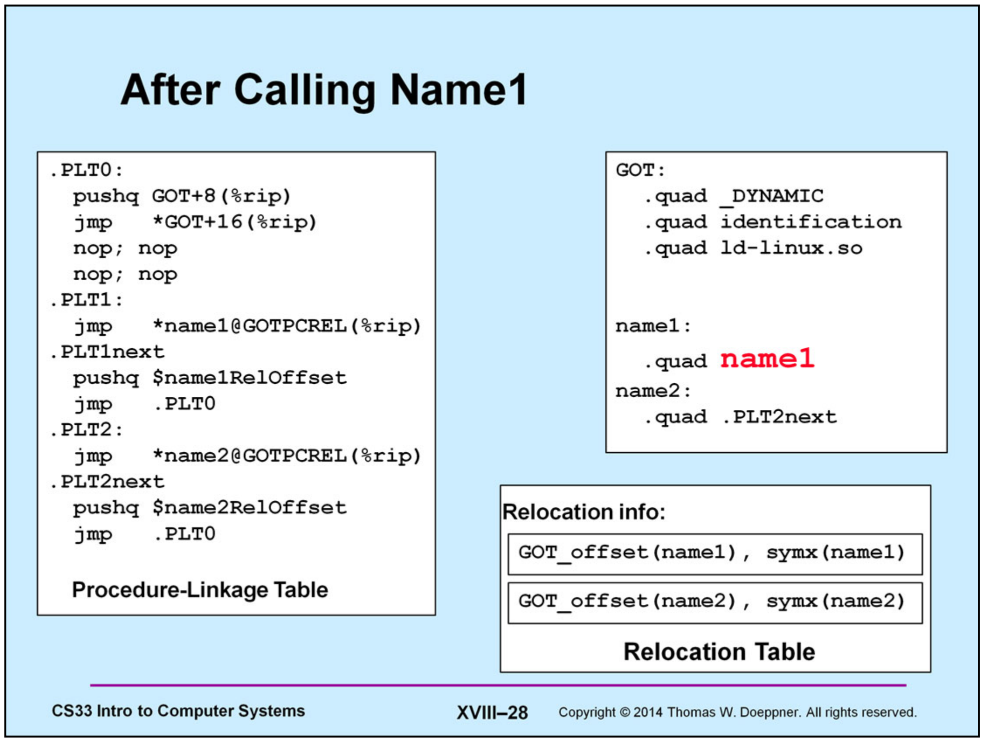
Relocation info:

```
GOT_offset(name1), symx(name1)
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 *name1* and *name2*. Let's follow a call to procedure name1. The general idea is before the first call to name1, the actual address of the name1 procedure is not recorded in the global-offset table, Instead, the first call to name1 actually invokes ld-linux.so, which is passed parameters indicating what is really wanted. It then finds name1 and updates the global-offset table so that things are more direct on subsequent calls.

To make this happen, references from the module to name1 are statically linked to entry .PLT1 in the procedure-linkage table. This entry contains an unconditional jump (via PC-relative addressing) to the address contained in the name1 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 name1 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 32-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 name1 entry in the global-offset table (which is what must be updated) and the index of name1 in the symbol table (so it knows the name of what it must locate).



After Calling Name1

```
.PLT0:
  pushq GOT+8(%rip)
  jmp   *GOT+16(%rip)
  nop; nop
  nop; nop
.PLT1:
  jmp   *name1@GOTPCREL(%rip)
.PLT1next
  pushq $name1RelOffset
  jmp   .PLT0
.PLT2:
  jmp   *name2@GOTPCREL(%rip)
.PLT2next
  pushq $name2RelOffset
  jmp   .PLT0
```

Procedure-Linkage Table

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


name1:
  .quad name1
name2:
  .quad .PLT2next
```

Relocation info:

```
GOT_offset(name1), symx(name1)
GOT_offset(name2), symx(name2)
```

Relocation Table

Finally, ld-linux.so writes the actual address of the name1 procedure into the name1 entry of the global-offset table, and, after unwinding the stack a bit, passes control to name1. On subsequent calls by the module to name1, since the global-offset table now contains name1's address, control goes to it more directly, without an invocation of ld-linux.so.