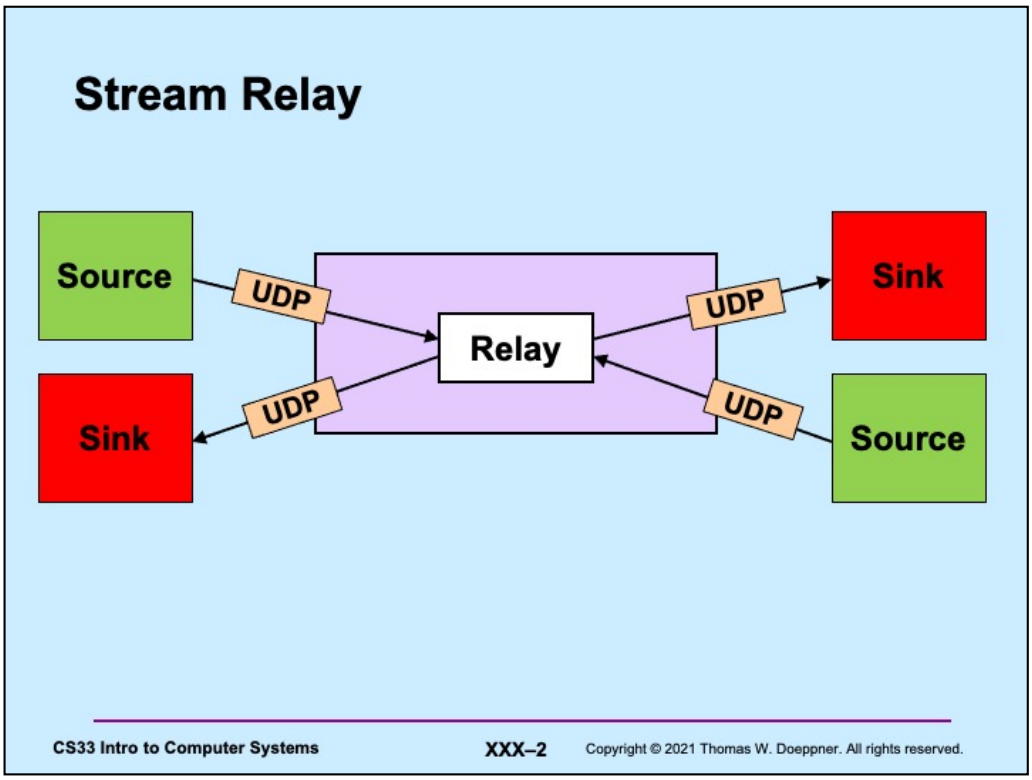
CS 33

Event-Based Programming



We look at what's known as **event-based programming**: we write code that responds to events coming from a number of sources. As a simple example we examine a simple relay: we want to write a program that takes data received via UDP from a source on the left and forwards it (via UDP) to a sink on the right. At the same time, it's taking data received from a source on the right and forwards it (via UDP) to a sink on the left.

Solution?

```
while(…) {
   size = read(left, buf, sizeof(buf));
   write(right, buf, size);
   size = read(right, buf, sizeof(buf));
   write(left, buf, size);
}
```


This solution is probably not what we'd want, since it strictly alternates between processing the data stream in one direction and then the other.

Note that to simply the slides a bit, even though we're using UDP, we'll use read and write system calls – the source and destination are assumed in each case.

Select System Call

```
int select(
  int nfds,          // size of fd_sets
  fd_set *readfds,   // descriptors of interest
                     // for reading
  fd_set *writefds,  // descriptors of interest
                     // for writing
  fd_set *excpfds,   // descriptors of interest
                     // for exceptional events
  struct timeval *timeout
                     // max time to wait
);
```


The **select** system call operates on three sets of file descriptors: one of fie descriptors we're interested in reading from, one of file descriptors we're interested in writing to, and one of file descriptors that might have exceptional conditions pending (we haven't covered any examples of such things – they come up as a peculiar feature of TCP known as out-of-band data, which is beyond the scope of this course). A call to **select** waits until at least one of the file descriptors in the given sets has something of interest. In particular, for a file descriptor in the read set, it's possible to read data from it; for a file descriptor in the write set, it's possible to write data to it. The **nfds** parameter indicates the maximum file descriptor number in any of the sets. The **timeout** parameter may be used to limit how long **select** waits. If set to zero, select waits indefinitely.

Relay Sketch

```
void relay(int left, int right) {
   fd_set rd, wr;
   int maxFD = max(left, right) + 1;
   FD_ZERO(&rd); FD_SET(left, &rd); FD_SET(right, &rd);
   FD_ZERO(&wr); FD_SET(left, &wr); FD_SET(right, &wr);
   while (1) {
      select(maxFD, &rd, &wr, 0, 0);
      if (FD_ISSET(left, &rd))
         read(left, bufLR, sizeof(message_t));
      if (FD_ISSET(right, &rd))
         read(right, bufRL, sizeof(message_t));
      if (FD_ISSET(right, &wr))
         write(right, bufLR, sizeof(message_t));
      if (FD_ISSET(left, &rd))
         write(left, bufRL, sizeof(message_t));
   }
}
```


Here a simplified version of a program to handle the relay problem using *select*. An **fd_set** is a data type that represents a set of file descriptors. FD_ZERO, FD_SET, and FD_ISSET are macros for working with fd_sets; the first makes such a set represent the null set, the second sets a particular file descriptor to be included in the set, the last checks to see if a particular file descriptor is included in the set.

This sketch doesn't quite work because it doesn't take into account the fact that we have limited buffer space: we can't read two messages in a row from one side without writing the first to the other side before reading the second. Furthermore, even though select may say it's possible to write to either the left or the right side, we can't do so until we're read in some data from the other side. Also, the fd_sets that are select's arguments are modified on return from select to indicate if it's now possible to read or write on the associated file descriptor. Thus if, on return from select, it's not possible to use that file descriptor, its associated bit will be zero. We need to explicitly set it to one for the next call so that select knows we're still interested.

Relay (1)

```
void relay(int left, int right) {
  fd_set rd, wr;
  int left_read = 1, right_write = 0;
  int right_read = 1, left_write = 0;
  message_t bufLR;
  message_t bufRL;
  int maxFD = max(left, right) + 1;
```


This and the next three slides give a more complete version of the relay program.

Initially our program is prepared to read from either the left or the right side, but it's not prepared to write, since it doesn't have anything to write. The variables **left_read** and **right_read** are set to one to indicate that we want to read from the left and right sides. The variables **right_write** and **left_write** are set to zero to indicate that we don't yet want to write to either side.

The two variables of type **message_t** are used as buffers to hold a messages received from the left and to be written to the right, or vice versa.

Relay (2)

```
while(1) {
  FD_ZERO(&rd);
  FD_ZERO(&wr);
  if (left_read)
    FD_SET(left, &rd);
  if (right_read)
    FD_SET(right, &rd);
  if (left_write)
    FD_SET(left, &wr);
  if (right_write)
    FD_SET(right, &wr);

  select(maxFD, &rd, &wr, 0, 0);
```


We set up the fd_sets **rd** and **wr** to indicate what we are interested in reading from and writing to (initially we have no interest in writing, but are interested in reading from either side).

Relay (3)

```
if (FD_ISSET(left, &rd)) {
  read(left, bufLR, sizeof(message_t));
  left_read = 0;
  right_write = 1;
}
if (FD_ISSET(right, &rd)) {
  read(right, bufRL, sizeof(message_t));
  right_read = 0;
  left_write = 1;
}
```


If there is something to read from the left side, we read it. Having read it, we're temporarily not interested in reading anything further from the left side, but now want to write to the right side.

In a similar fashion, if there is something to read from the right side, we read it.

Relay (4)

```
    if (FD_ISSET(right, &wr)) {
      write(right, bufLR, sizeof(message_t));
      left_read = 1;
      right_write = 0;
    }
    if (FD_ISSET(left, &wr)) {
      write(left, bufRL, sizeof(message_t));
      right_read = 1;
      left_write = 0;
    }
  }
  return 0;
}
```


Similarly for writing: if we've written something to one side, we have nothing more to write to that side, but are now interested in reading from the other side.

CS 33

Linking and Libraries

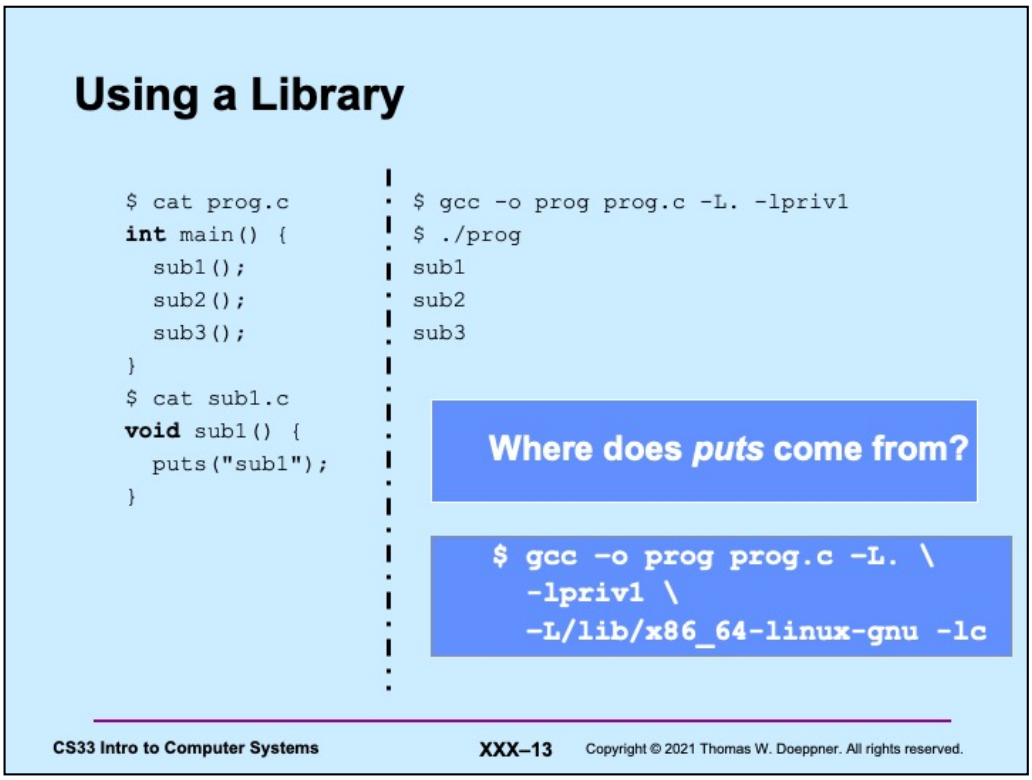
Libraries

- 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**.



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

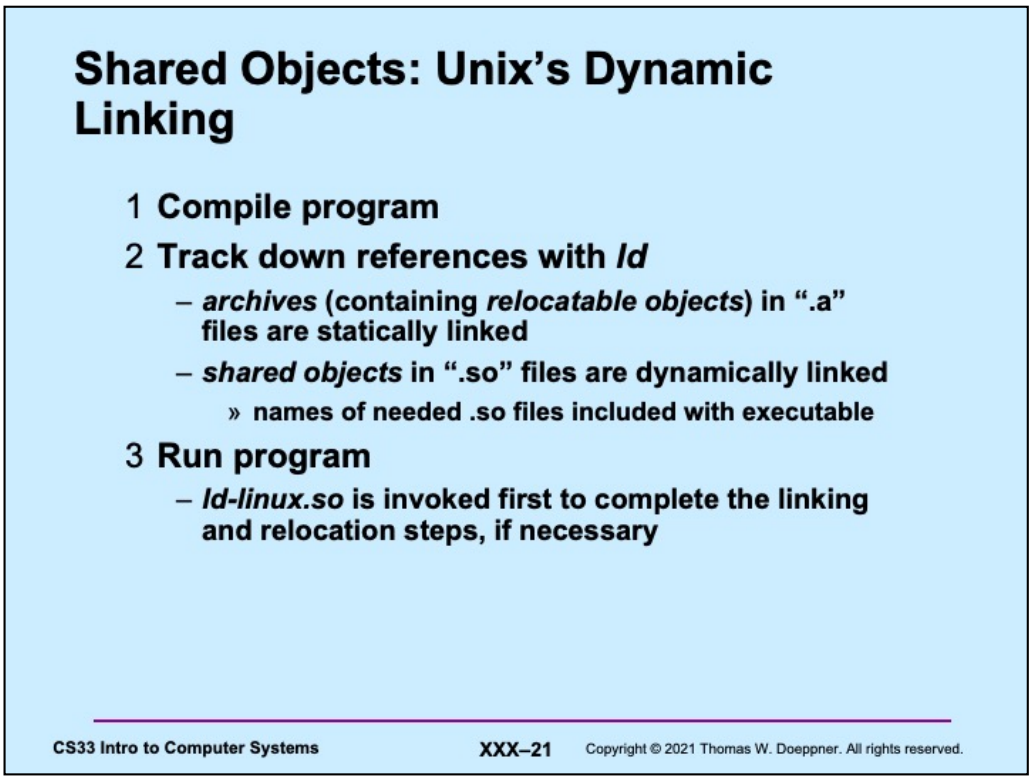
- printf is found to have a bug
  - perhaps a security problem
- All existing instances must be replaced
  - there are zillions of instances ...
- Do we have to re-link all programs that use printf?

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



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 function, 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

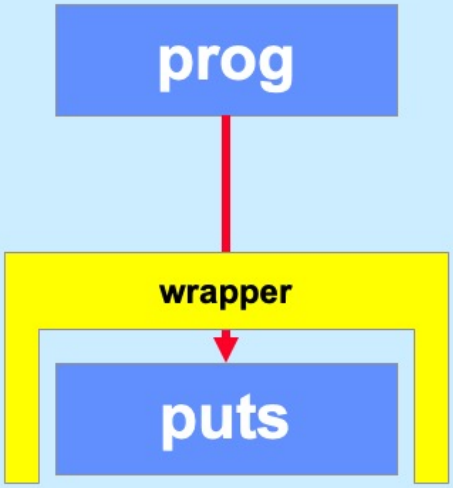
```
$ gcc -fPIC -c goodstuff.c
$ ld -shared -soname libgoodstuff.so.1 \
-o libgoodstuff.so.1 goodstuff.o
$ ln -s libgoodstuff.so.1 libgoodstuff.so
$ gcc -o prog1 prog1.c -L. -lgoodstuff \
-Wl,-rpath .
$ vi goodstuff.c
$ gcc -fPIC -c goodstuff.c
$ ld -shared -soname libgoodstuff.so.2 \
-o libgoodstuff.so.2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

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 function. 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

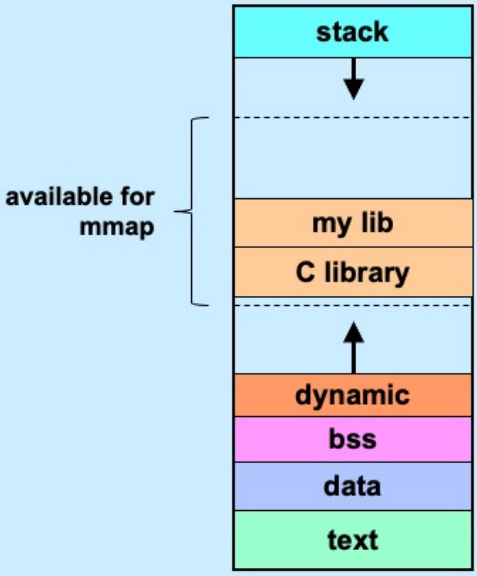
- Another form of exec
  - `int execve(const char *filename, char *const argv[], char *const envp[]);`
- envp is an array of strings, of the form
  - key=value
- programs can search for values, given a key
- example
  - PATH=~/bin:/bin:/usr/bin:/course/cs0330/bin

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.
$
```

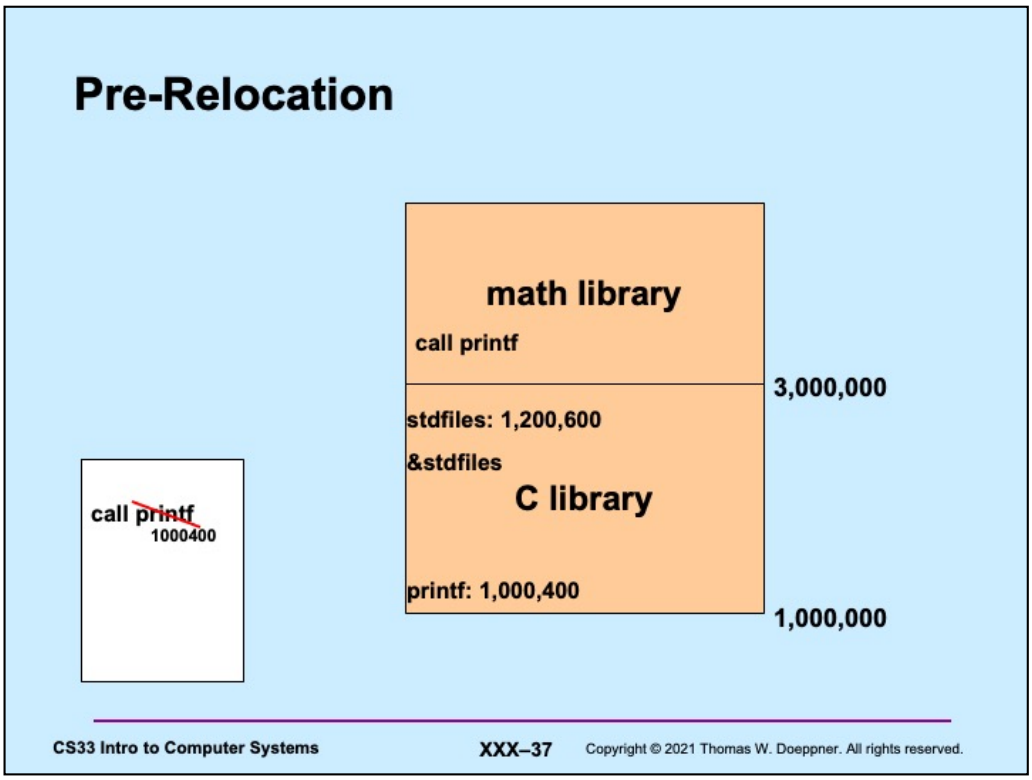

Here we add "LD_PRELOAD=./libmyputs.so.1" to the environment. The export command instructs the shell to supply this as part of the environment for the commands it runs.

Mmapping Libraries

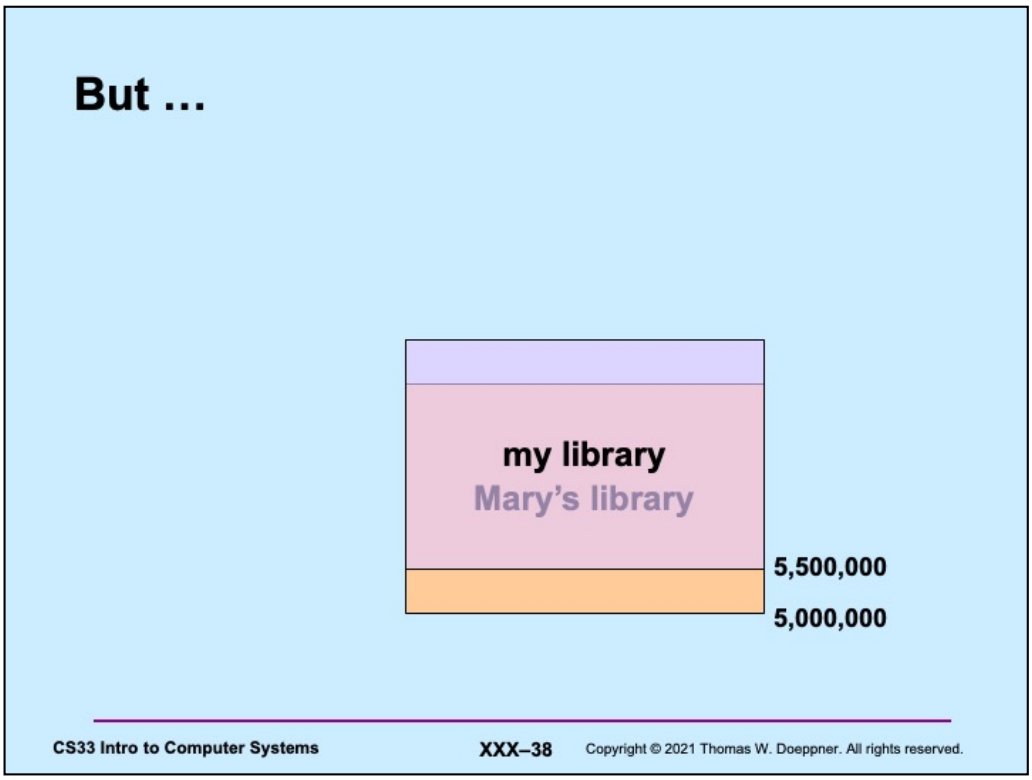


Problem

- How is relocation handled?

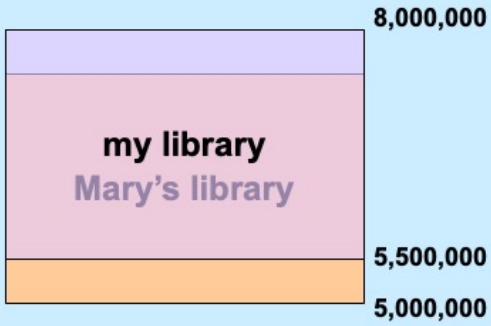


Assuming we're using pre-relocation, the C library and the math library would be assumed to be in virtual memory at their pre-assigned locations. In the slide, these would be starting at locations 1,000,000 and 3,000,000, respectively. Let's suppose printf, which is in the C library, is at location 1,000,400. 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?

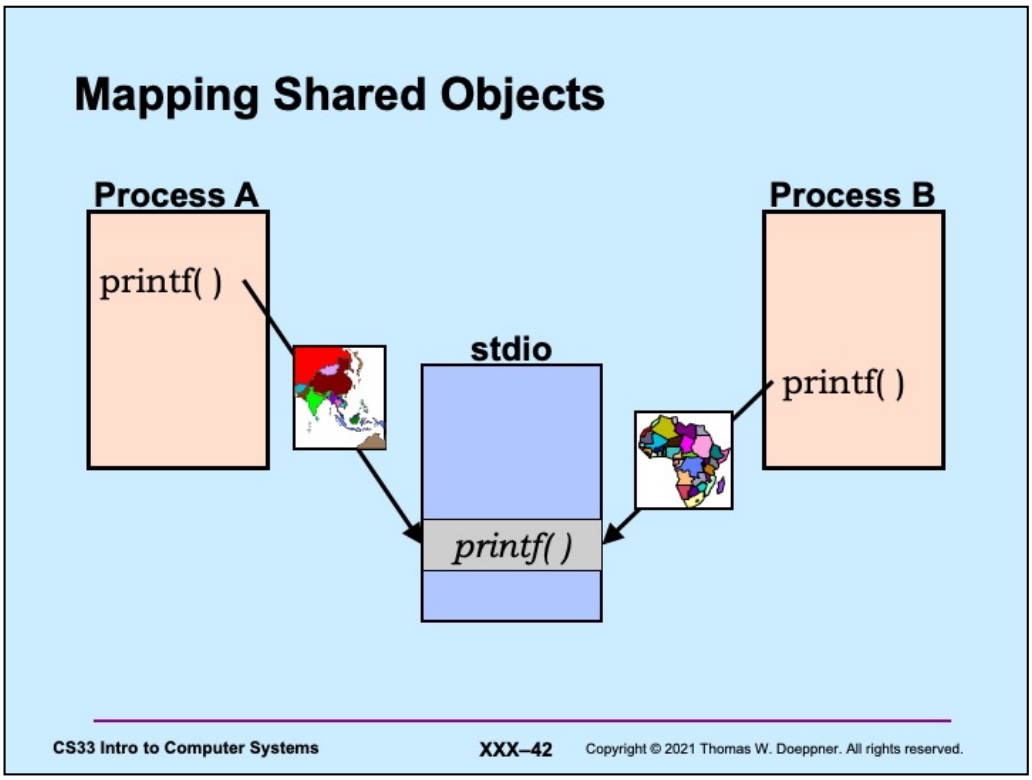
a) the MAP_PRIVATE option

b) the MAP_SHARED 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



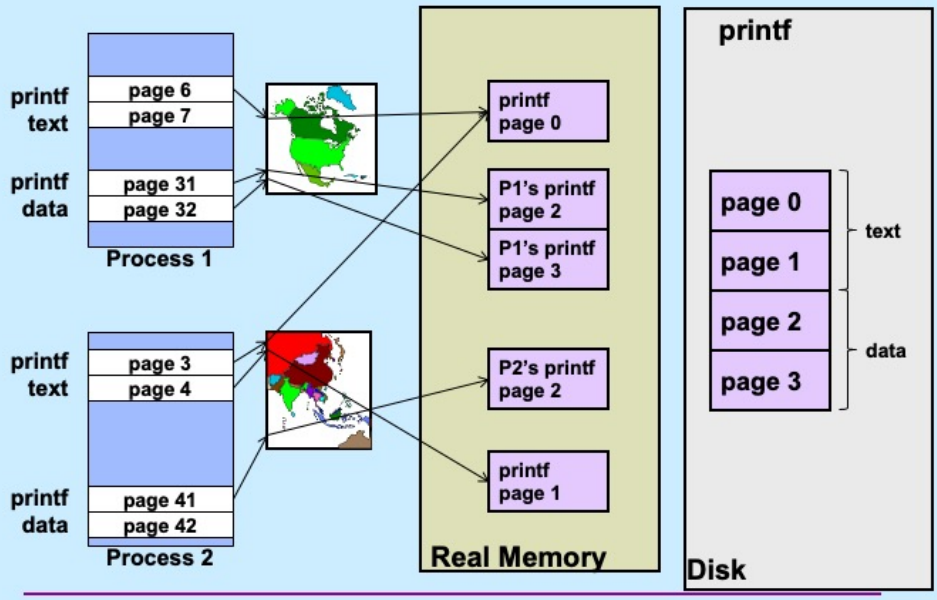
The C library (and other libraries) can be mapped into different locations in different processes' address spaces.

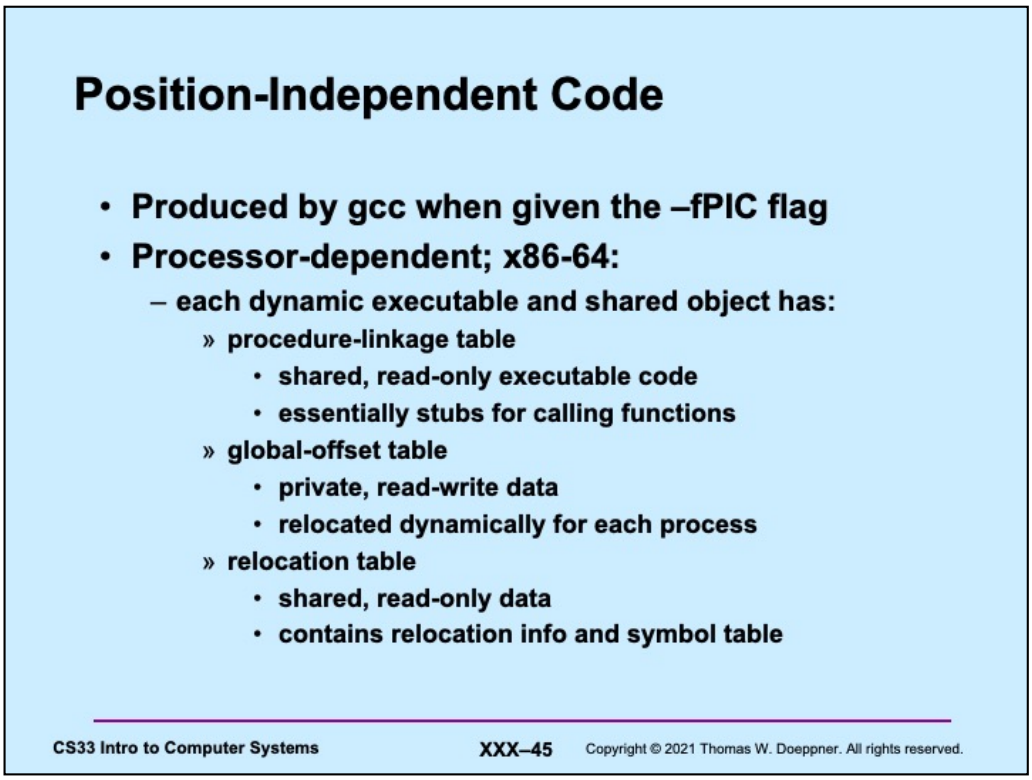
Mapping printf into the Address Space

- 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).

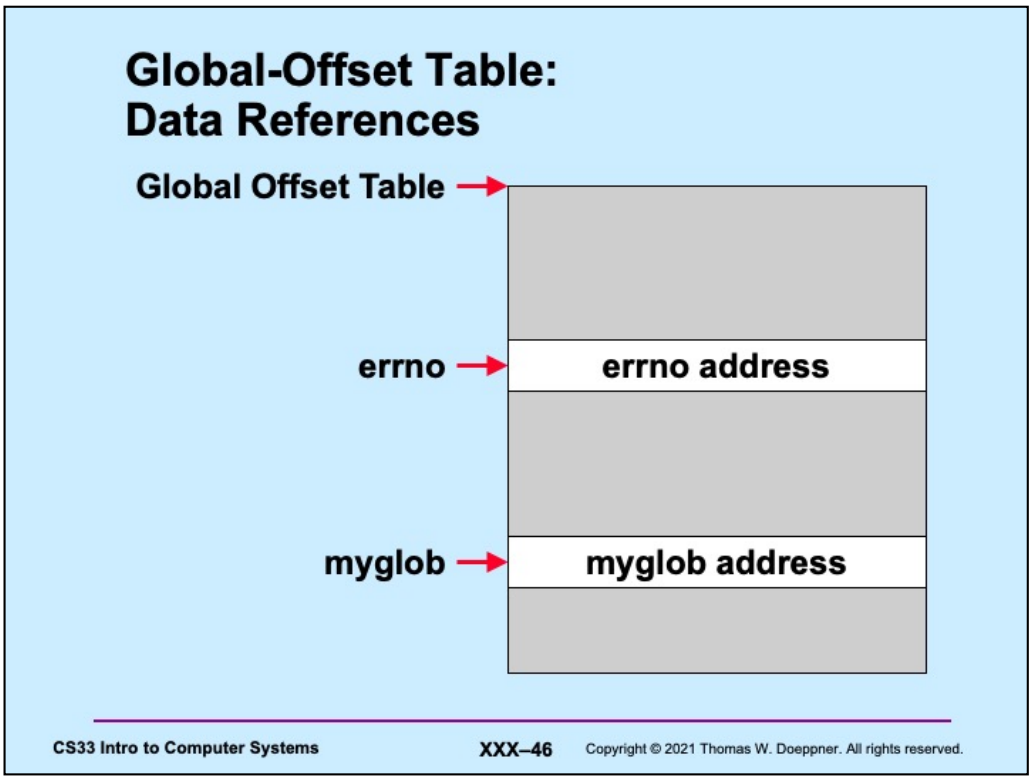
Mapping printf





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.

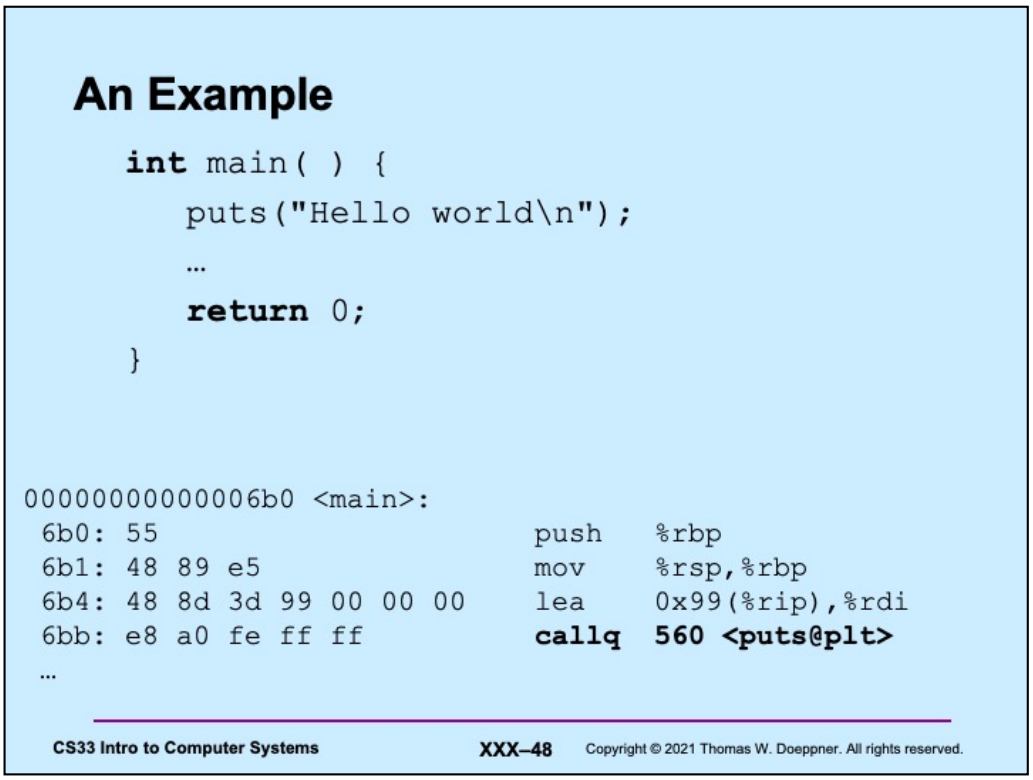
How things work is similar for other architectures, but definitely not the same.



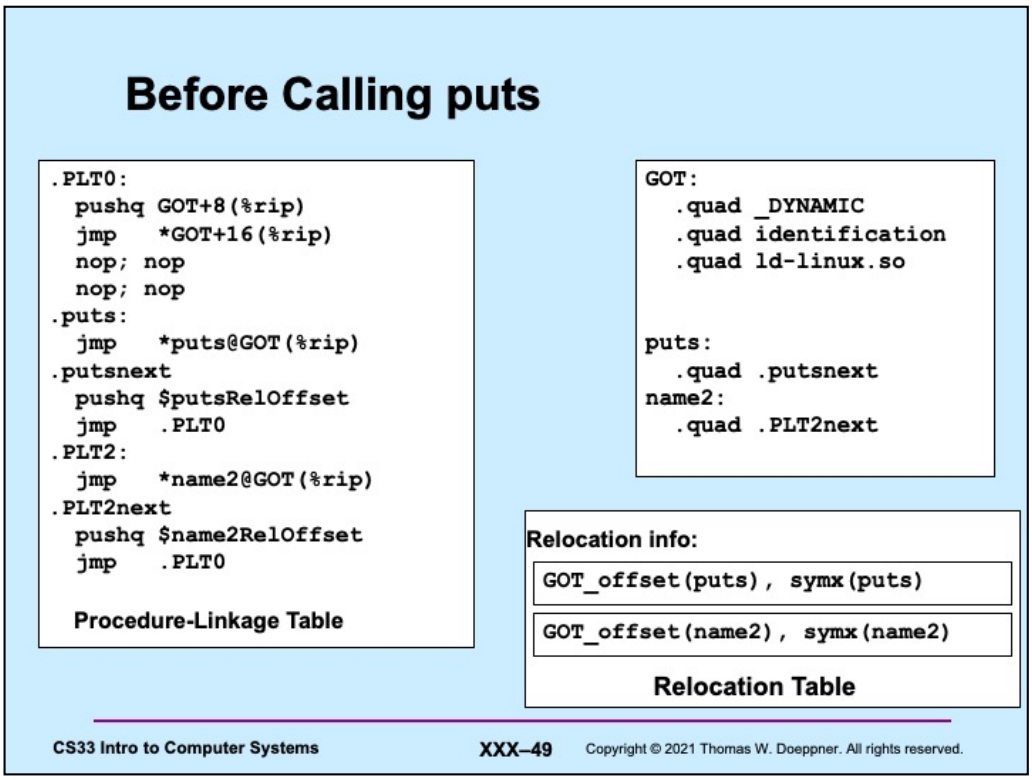
To establish position-independent references to global variables, the compiler produces, for each module, a **global-offset table**. Modules refer to global variables indirectly by looking up their addresses in the table, using PC-relative addressing. The item needed is at some fixed offset from the 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

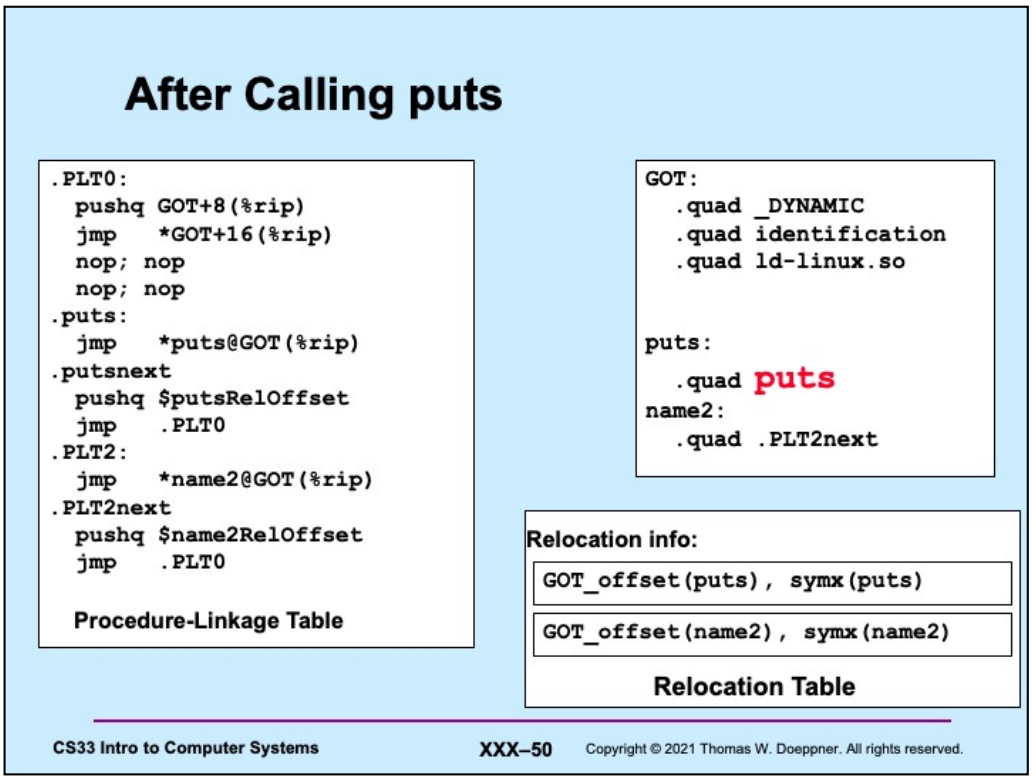


The top half of the slide contains an excerpt from a C program. For the bottom half, we've compiled the program and have printed what "objdump –d" produces for main. Note that the call to puts is actually a call to "puts@plt", which is a reference to the procedure linkage table.



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.

Not a Quiz!

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*

