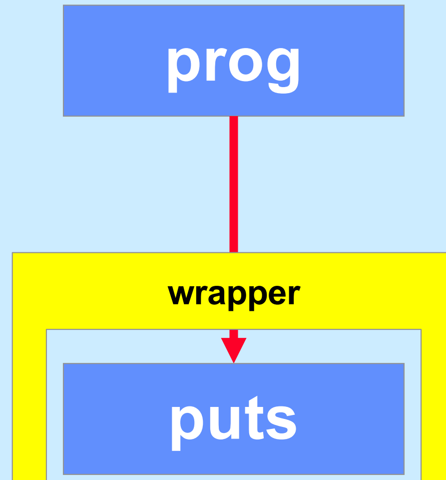
# CS 33

## Linking and Libraries (2)

 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

# Interpositioning




 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

The idea expressed in the slide is that when **prog** calls **puts**, control first goes to the **wrapper**, which then calls **puts**.

Thus references to **puts** from within prog actually refer to **wrapper**. But if we do this uniformly, replacing all references to **puts** with **wrapper**, how does **wrapper** call **puts**?

# How To …

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

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

 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

**__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.
$
```

 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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);
}
```

 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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

 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

# Delayed Wrapping

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

 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

# 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

 Copyright © 2024 Thomas W. Doeppner. All rights reserved.
    ```

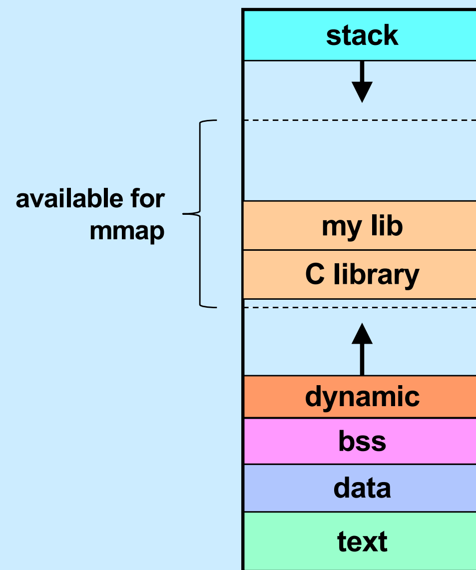
## Example

```
$ 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.
$
```

 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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



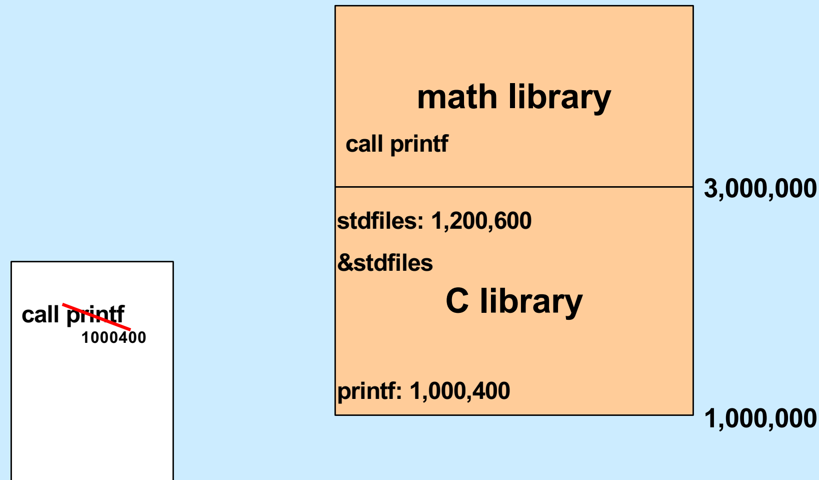


Copyright © 2024 Thomas W. Doeppner. All rights reserved.

# Problem

- **How is relocation handled?**

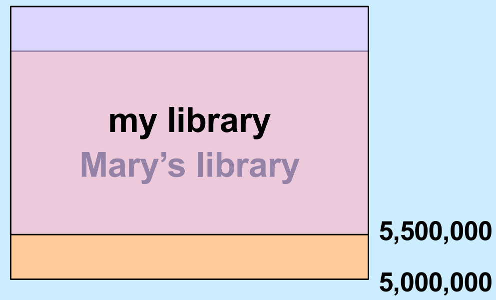

Copyright © 2024 Thomas W. Doeppner. All rights reserved.




One simple approach to relocation is to avoid it: everything is pre-assigned a location in memory–this is known as **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 identifier, such as stdfiles. Its address would also be known.

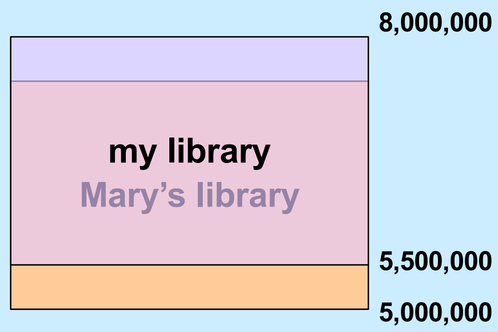
# But …




 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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 …




 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

# 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

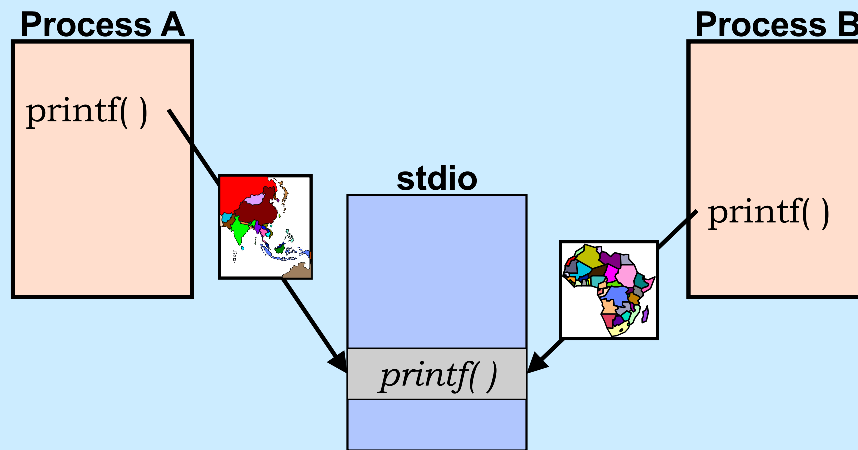
 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

# 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

 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

## Mapping Shared Objects




 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

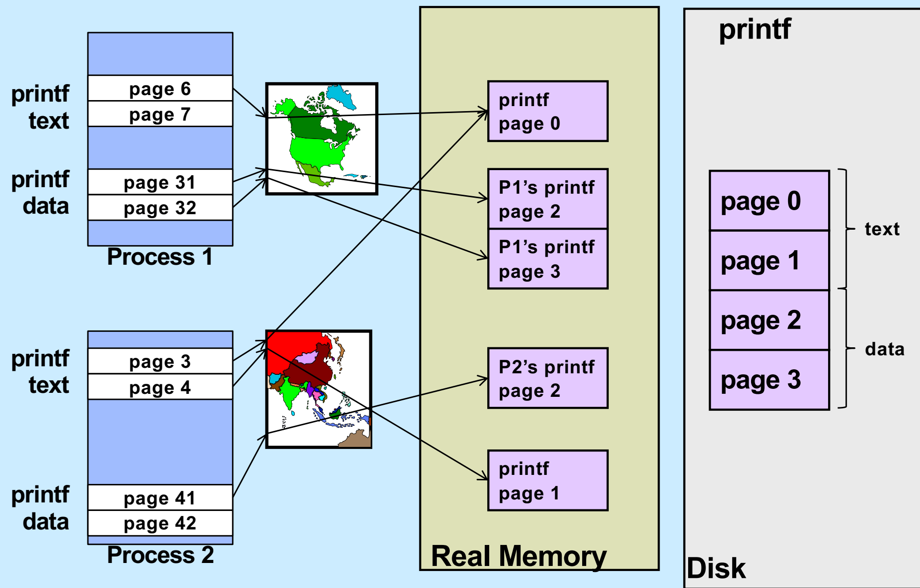
The C library (and other libraries) can be mapped into different locations in different processes' address spaces.

## 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

 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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


Mapping printf
printf
text
page 6
page 7
printf
data
page 31
page 32
Process 1
printf
text
page 3
page 4
printf
data
page 41
page 42
Process 2
printf
page 0
P1's printf
page 2
P1's printf
page 3
P2's printf
page 2
printf
page 1
Real Memory
printf
page 0
page 1
page 2
page 3
text
data
Disk
CS33 Intro to Computer Systems
XXXVIII–19
Copyright © 2024 Thomas W. Doeppner. All rights reserved.

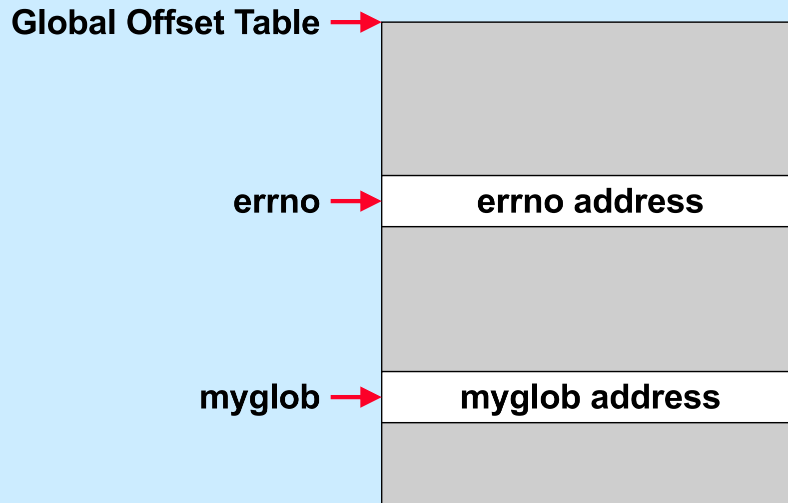
## Position-Independent Code

- 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

 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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

 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

## 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>
 …
```

 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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

 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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 (which contains a reference to the name, "puts", as well as the offset within the global-offset-table of where the actual address of puts will be written). 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

 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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*

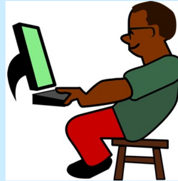
c) control still goes to ld-linux.so, but it now transfers control directly to *puts*

 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

# You'll Soon Finish CS 33 …

- You might
  - celebrate

  - take another systems course
    - » 320
    - » 1380
    - » 1660
    - » 1670
    - » 1680



  - become a 300 TA

 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

## Systems Courses Next Semester

- CS 320 (Intro to Software Engineering)
  - you've mastered low-level systems programming
  - now do things at a higher level
  - learn software-engineering techniques using Java, XML, etc.
- CS 1380 (Distributed Systems)
  - you now know how things work on one computer
  - what if you've got lots of computers?
  - some may have crashed, others may have been taken over by your worst (and smartest) enemy
- CS 1660/1620/2660 (Computer Systems Security)
  - liked buffer?
  - you'll really like 1660
- CS 1670/1690/2670 (Operating Systems)
  - still mystified about what the OS does?
  - write your own!

 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

2660 is for graduate students only and combines 1660 and 1620.

2670 is for graduate students only and combines 1670 and 1690.

# The End

**Well, not quite …**
**Database is due on 12/13**

## Happy Coding and Happy Holidays!

 Copyright © 2024 Thomas W. Doeppner. All rights reserved.