3.4 Linking & Loading

- Static Linking & Loading
- *Shared Libraries*

Libraries

- A library is just a collection of `.o` files
  - the linker is used to create libraries
- Two types of libraries
  - *static* library
  - *dynamic (or shared)* library

Creating a Static Library

```
% cat sub1.c
void sub1() { puts("sub1"); }
% cat sub2.c
void sub2() { puts("sub2"); }
% cat sub3.c
void sub3() { puts("sub3"); }
% 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
%
```

- `puts()` is unresolved in `libpriv1.a`

Using a Static Library

```
% cat prog.c
int main() {
  sub1();
  sub2();
  sub3();
}
% gcc -o prog prog.c -L. -lpriv1
```

- Where does `puts()` come from?

```
% gcc -o prog prog.c -L. -lpriv1 -L/lib -lc
```

- The order of the libraries matter
  - will try to resolve references first in the `priv1` library (either `libpriv1.a` or `libpriv1.so`) and then in the `c` library (either `libc.a` or `libc.so`)

Substitution

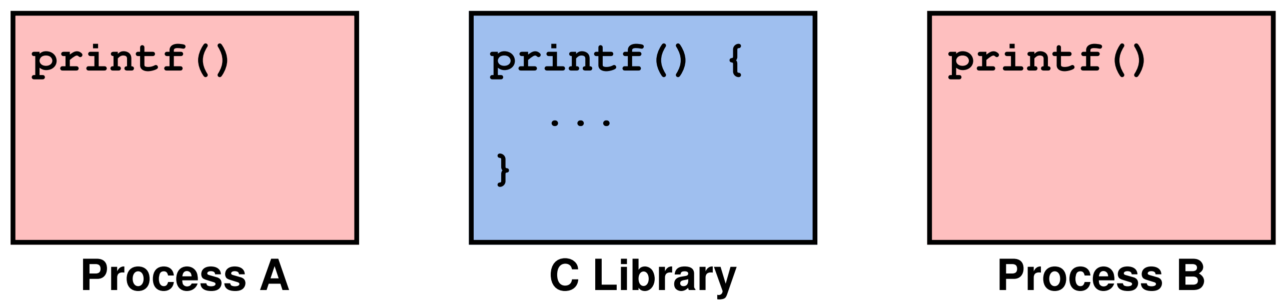
- Want to use my version of `puts()` instead of what's in the C library

```
% 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
```

  - will try to resolve `puts()` first in the `priv1` library, then in the `myputs` library, then in the `c` library

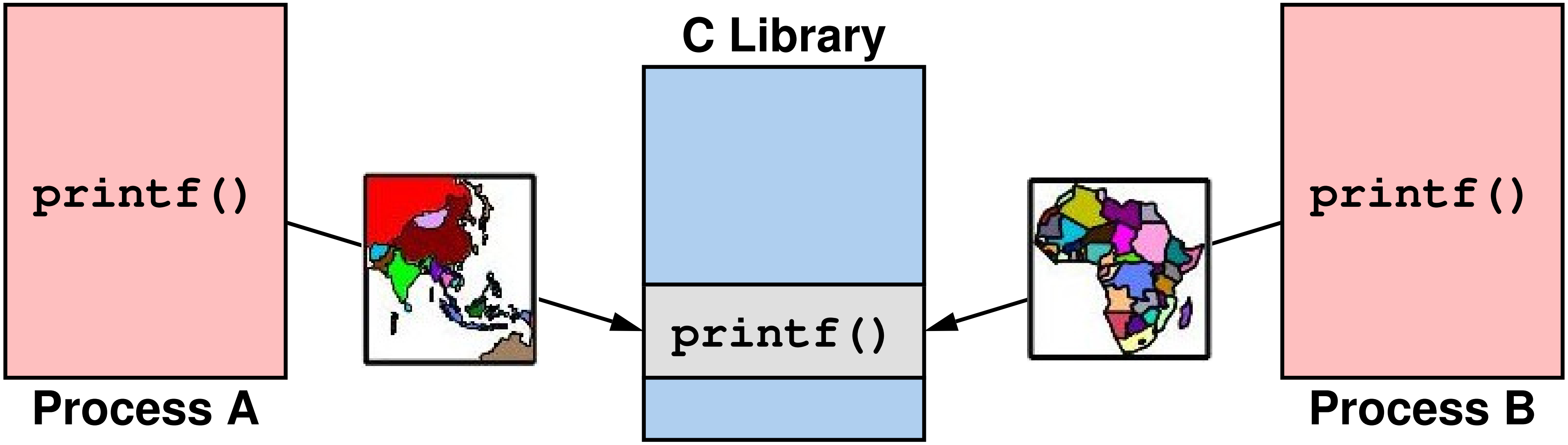
Shared Libraries

- `prog` must contain *everything* needed for execution
  - duplicate code, may be lots of duplicate code, e.g., `printf()`
    - take up *disk space*
    - take up *memory*
- Need a way to share things like `printf()`
  - on disk, and
  - in memory

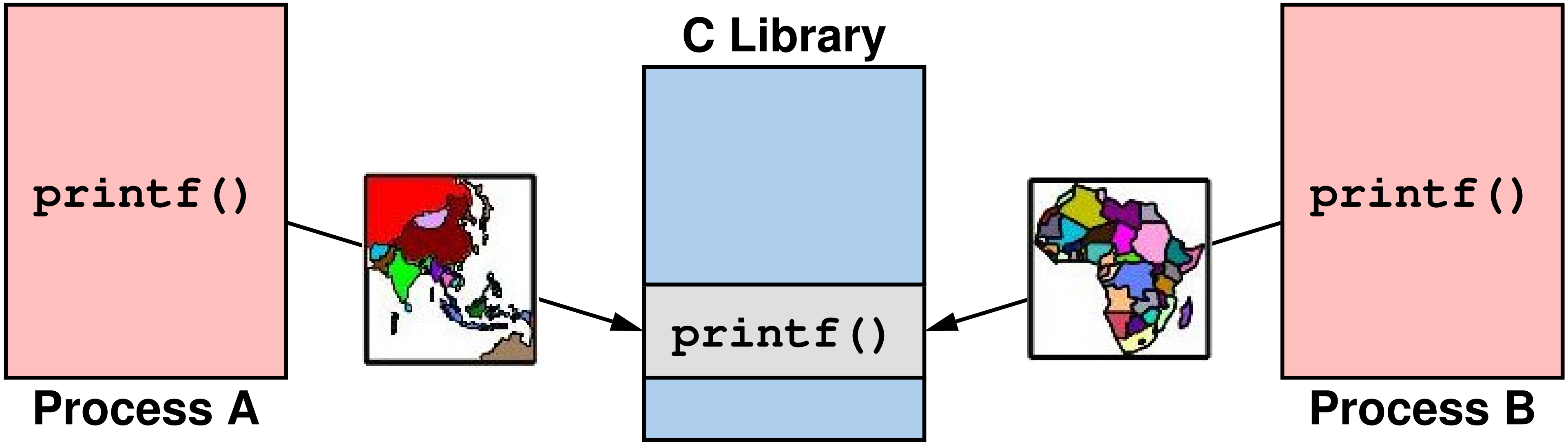


- If `printf()` required no relocation, then it's easy
  - just make sure `ld` use the right address consistently
- If `printf()` required relocation, then it's more complicated
  - problem: processes want a shared function to be at different addresses

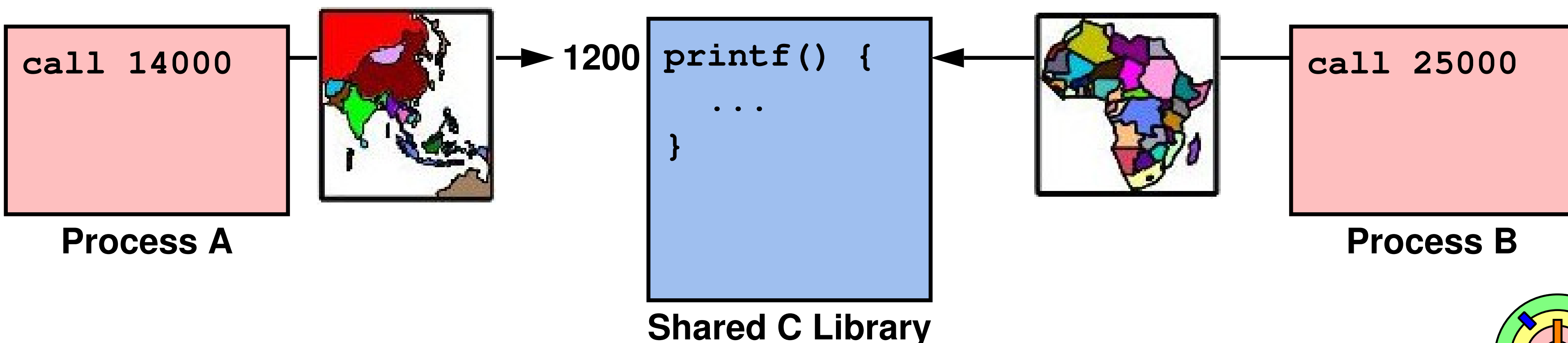
Sharing



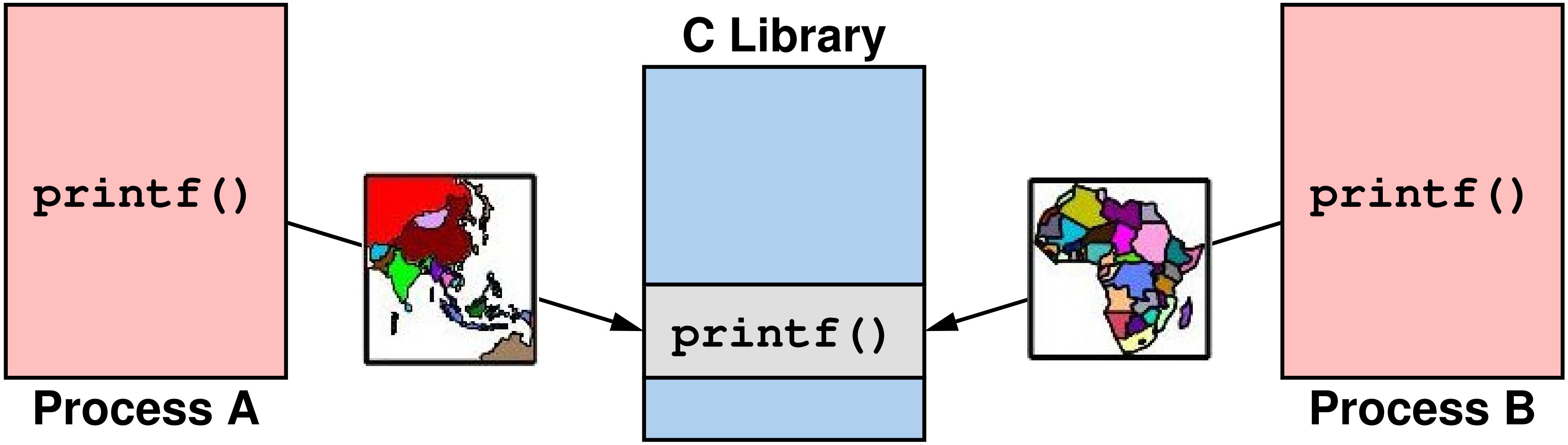
Sharing



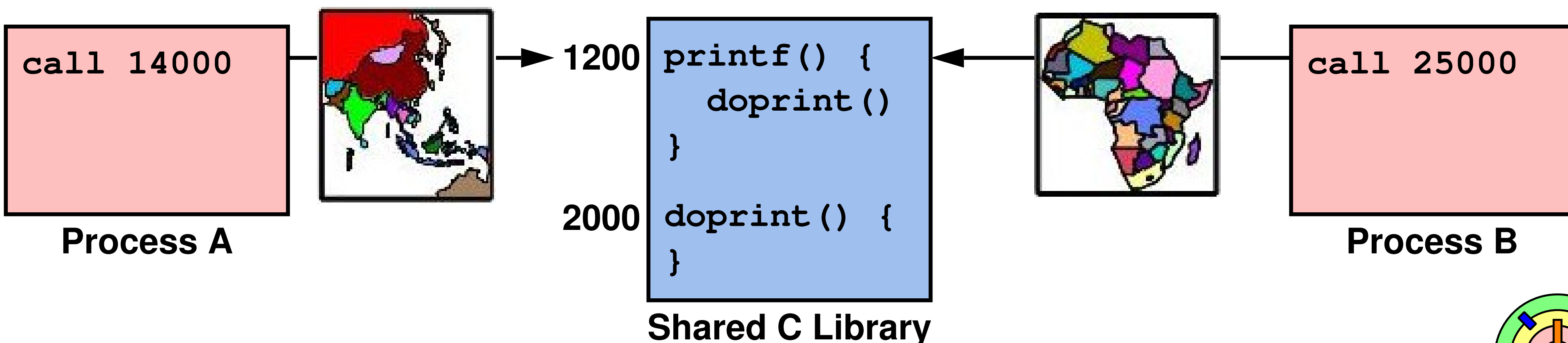
- Looks like it can work



Sharing



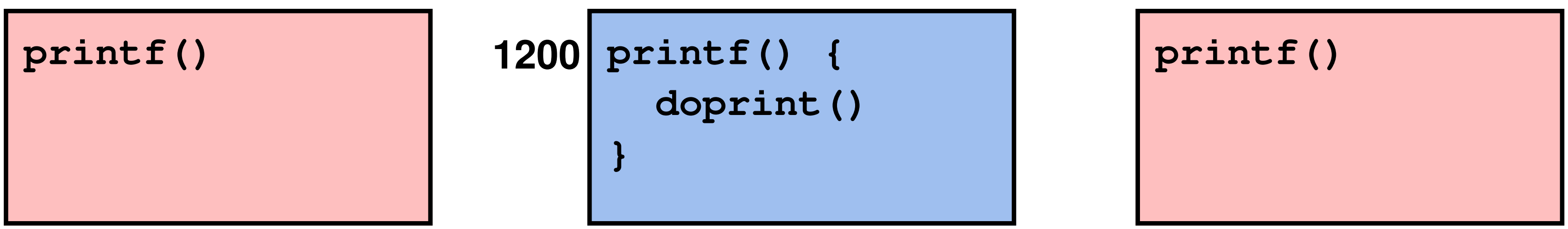
What about this?



Relocation and Shared Libraries

Approaches

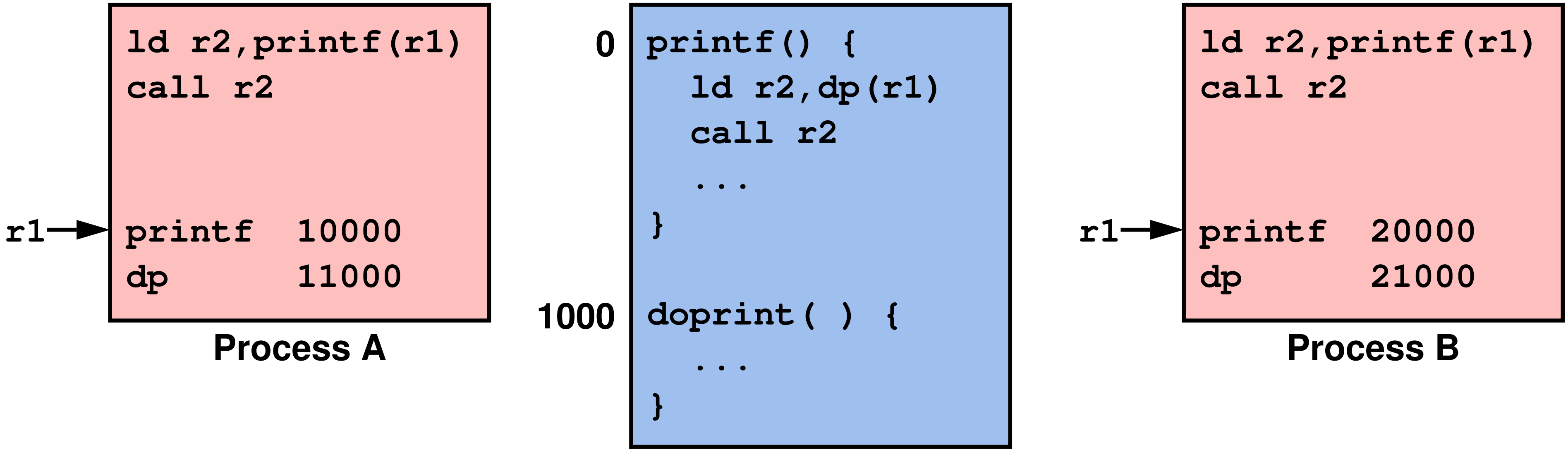
- Limited sharing: relocate separately for each process
  - have a single copy of `printf()` on disk
  - as `printf()` gets copied into memory, perform relocation
  - this would work, but still end up with too many copies of `printf()` in memory
- *Prerelocation:* relocate libraries ahead of time
  - difficult to prerelocate all shared functions
    - may need to preform rerelocation



Process A   C Library   Process B

- *Position-Independent Code:* no need for relocation
  - producing code that can be placed anywhere in memory without requiring modification
  - need *indirection*

Position-Independent Code



- each process maintains a private table, pointed to by register `r1`
  - table contains addresses of shared routines
- don't call functions directly
  - make a position-independent call (i.e., an indirect call)
  - i.e., call the function located at a ***fixed index*** into the table pointed by `r1`
    - implemented as two instructions in the above example

⇨ **Please note that `ld` is not the same as `mov` in x86 CPU**

Position-Independent Code Details

- Processor-dependent; x86 32-bit version:
- ELF requires 3 data structures for each dynamic executable and shared object
  - the *procedure linkage table (PLT)*
    - read-only executable code, *shared* by all processes
    - essentially stubs for calling subroutines
  - the *global offset table (GOT)*
    - read-write data, private (to each process)
    - relocated dynamically for each process
  - the *dynamic structure*
    - read-only data, *shared* by all processes
    - contains relocation info and symbol table

Shared Libraries In Practice

- ***Shared libraries*** are used extensively in many modern systems
  - often implemented with either prerelocation or position-independent code
  - in Windows, they are known as ***Dynamic-Link Libraries (DLLs)***
  - in Unix, they are known as ***shared objects*** (`.so` files)
    - vs. ***static libraries*** (`.a` files)
  - they need not be loaded when a program starts up
    - can be loaded when needed, i.e., ***on-demand***
    - this way, the startup time of a program may be reduced

- Disadvantages of DLLs and shared objects
  - they can have ***dependencies***
  - different ***versions*** of the same library

Linking and Loading on Linux with ELF

- x86 ELF (Excutable and Linking Format)
  - used in Unix/Linux systems
    - not used in either MacOS X or Windows
- Creating and using a shared library
- Substitution
- Shared library details
- Versioning
- Dynamic linking
- Interpositioning

Shared Library Details

- When a program is invoked via the *exec* system call
  - the code that is first given control is `ld.so`, the *run-time linker*
  - the job of `ld.so` is to complete the linking and relocation steps, if necessary
    - it does some initial set up of linkages
    - then calls the actual program code
    - it may be called upon later to do some further loading and linking

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
 libmyputs.so => not found
 libc.so.6 => /lib/tls/i686/cmoc/libc.so.6
 /lib/ld-linux.so.2 => /lib/ld-linux.so.2
```

Creating a Shared Library (2)

```
% gcc -o prog prog.c -L. -lpriv1 -lmyputs -Wl,-rpath .
% ldd prog
  libmyputs.so => ./libmyputs.so
  libc.so.6 => /lib/tls/i686/cmoc/libc.so.6
  /lib/ld-linux.so.2 => /lib/ld-linux.so.2
% ./prog
My puts: sub1
My puts: sub2
My puts: sub3
```

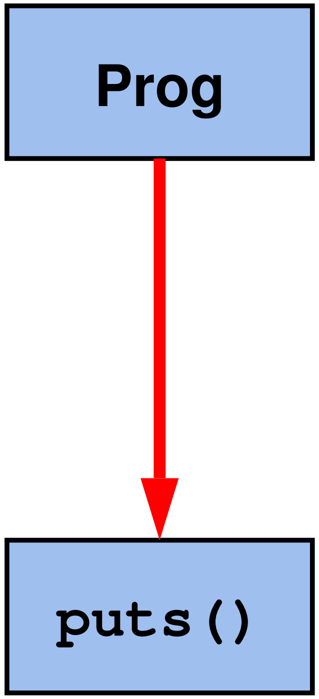
- "`-Wl,-rpath .`" means that what comes after `-Wl` are linker options (i.e., pass them to the linker)
  - in this example, the linker will be invoked with "`-rpath .`"
- also try "`-Wl,-rpath,.`" if the space character is giving you trouble

Versioning

```
% gcc -fPIC -c myputs.c
% ld -shared -soname libmyputs.so.1 \
      -o libmyputs.so.1.0 myputs.o
% ldconfig -v -n .
% ln -s libmyputs.so.1 libmyputs.so
% gcc -o prog1 prog1.c -L. -lpriv1 -lmyputs \
      -Wl,-rpath .
% vi myputs.c
% gcc -fPIC -c myputs.c
% ld -shared -soname libmyputs.so.2 \
      -o libmyputs.so.2 myputs.o
% rm -f libmyputs.so
% ldconfig -v -n .
% ln -s libmyputs.so.2 libmyputs.so
% gcc -o prog2 prog2.c -L. -lpriv1 -lmyputs \
      -Wl,-rpath .
```

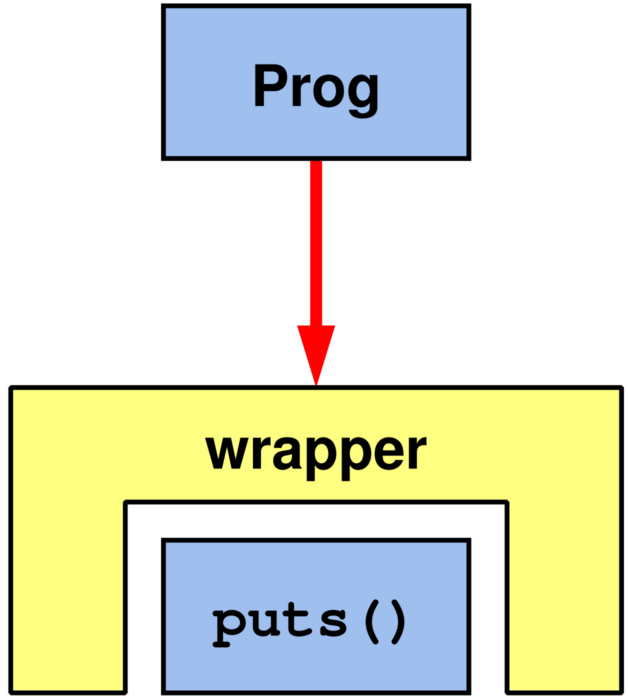
- "`libmyputs.so.1`" is the ***soname***
- "`libmyputs.so.1.0`" is the ***real name***
- "`libmyputs.so`" is the ***linker name***

Interpositioning



- `prog` thinks it's calling `puts()`

Interpositioning



- `prog` thinks it's calling `puts()`
- interpose your `puts()`
  - you can call the original `puts()` that `prog` thought it was calling
  - security problem?!
    - "DLL injection" if you pick up a DLL unknowningly

How To ...

```
% cat myputs.c
#include <dlfcn.h>

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

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

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

- `dlsym()` returns a *function pointer* for the named function
- `RTLD_NEXT` asks for the next occurrence of the named function
  - `RTLD_DEFAULT` will get you the first occurrence of the named function using the default library search order

Compiling/Linking It

```
% gcc -fPIC -c myputs.c -D_GNU_SOURCE
% ld -shared -soname libmyputs.so.1 \
      -o libmyputs.so.1.0 myputs.o -ldl
% ldconfig -v -n .
% cat tputs.c
int main() {
  puts("This is a boring message.");
  return 0;
}
% gcc -o tputs tputs.c ./libmyputs.so.1 -Wl,-rpath .
% ./tputs
calling myputs: This is a boring message.
%
```

- `-D_GNU_SOURCE` is needed or won't recognize `RTLD_NEXT`
- `ldconfig` may be in `/sbin`

What's Going On ...

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

Delayed Wrapping

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

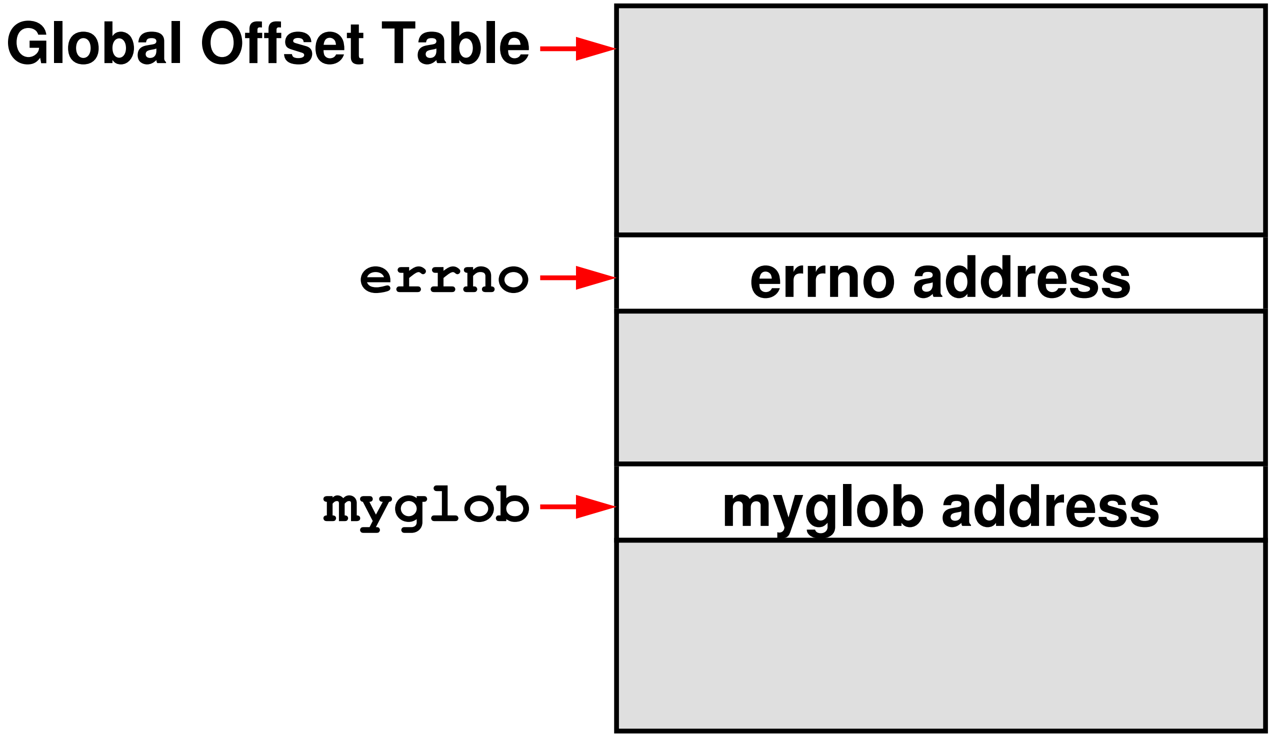
```
% gcc -o tputs tputs.c
% ./tputs
This is a boring message.
% setenv LD_PRELOAD ./libmyputs.so.1
% ./tputs
calling myputs: This is a boring message.
%
```

Extra Slides

Position-Independent Code Details

- Processor-dependent; x86 32-bit version:
- ELF requires 3 data structures for each dynamic executable and shared object
  - the *procedure linkage table (PLT)*
    - read-only executable code, *shared* by all processes
    - essentially stubs for calling subroutines
  - the *global offset table (GOT)*
    - read-write data, private (to each process)
    - relocated dynamically for each process
  - the *dynamic structure*
    - read-only data, *shared* by all processes
    - contains relocation info and symbol table

Global-Offset Table: Data References



- Modules refer to ***global variables*** indirectly by looking up their addresses in this table
  - a ***registger*** contains the address of the table
  - modules refer to entries via their ***offsets***
- When a module is loaded into memory
  - `ld.so` puts the actual addresses into the GOT

Procedure References

- More complicated than data references
- Lots of them
- Many are never used
- Fix them up on demand

Before Calling `name1`

```
.PLT0:
  pushl  4(%ebx)
  jmp    8(%ebx)
  nop;  nop
  nop;  nop
.PLT1:
  jmp    name1(%ebx)
.PLT1next:
  pushl  $name1RelOffset
  jmp    .PLT0
.PLT2:
  jmp    name2(%ebx)
.PLT2next
  pushl  $name2RelOffset
  jmp    .PLT0
```

Procedure-Linkage Table

ebx →

```
_GLOBAL_OFFSET_TABLE:
   .long _DYNAMIC
   .long identification
   .long ld.so

name1:
   .long .PLT1next
name2:
   .long .PLT2next
```

Relocation info:

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

_DYNAMIC

- Before the first call to `name1`
  - the actual address of the `name1` procedure is *not* in the GOT
  - first call to `name1` invokes `ld.so` with indication of the above fact
  - `ld.so` finds `name1` and update the GOT

Before Calling name1

```
.PLT0:
  pushl  4(%ebx)
  jmp    8(%ebx)
  nop; nop
  nop; nop
.PLT1:
  jmp    name1(%ebx)
.PLT1next:
  pushl  $name1RelOffset
  jmp    .PLT0
.PLT2:
  jmp    name2(%ebx)
.PLT2next
  pushl  $name2RelOffset
  jmp    .PLT0
```

Procedure-Linkage Table

ebx →

```
_GLOBAL_OFFSET_TABLE:
   .long _DYNAMIC
   .long identification
   .long ld.so

name1:
   .long .PLT1next
name2:
   .long .PLT2next
```

Relocation info:

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

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

_DYNAMIC

- **How did this work?**
  - references from module to `name1` are statically linke to entry `.PLT1` in the procedure-linkage table
    - the index into the symbol table in `_DYNAMIC` tells use where to start (in this example, it's 0 for `name1`)

Before Calling `name1`

```
.PLT0:
  pushl  4(%ebx)
  jmp    8(%ebx)
  nop; nop
  nop; nop
.PLT1:
  jmp    name1(%ebx)
.PLT1next:
  pushl  $name1RelOffset
  jmp    .PLT0
.PLT2:
  jmp    name2(%ebx)
.PLT2next
  pushl  $name2RelOffset
  jmp    .PLT0
```

Procedure-Linkage Table

ebx →

```
_GLOBAL_OFFSET_TABLE:
   .long _DYNAMIC
   .long identification
   .long ld.so

name1:
   .long .PLT1next
name2:
   .long .PLT2next
```

Relocation info:

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

_DYNAMIC

- unconditional jump to the address contained in the `name1` offset of the GOT (pointed to by the register `ebx`)

Before Calling `name1`

```
.PLT0:
  pushl  4(%ebx)
  jmp    8(%ebx)
  nop; nop
  nop; nop
.PLT1:
  jmp    name1(%ebx)
.PLT1next:
  pushl  $name1RelOffset
  jmp    .PLT0
.PLT2:
  jmp    name2(%ebx)
.PLT2next
  pushl  $name2RelOffset
  jmp    .PLT0
```

Procedure-Linkage Table

ebx

```
_GLOBAL_OFFSET_TABLE:
   .long _DYNAMIC
   .long identification
   .long ld.so

name1:
   .long .PLT1next
name2:
   .long .PLT2next
```

Relocation info:

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

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

_DYNAMIC

- unconditional jump to the address contained in the `name1` offset of the GOT (pointed to by the register `ebx`)
  - initially, this address is of the instruction following the jump instruction

Before Calling `name1`

```
.PLT0:
  pushl  4(%ebx)
  jmp    8(%ebx)
  nop; nop
  nop; nop
.PLT1:
  jmp    name1(%ebx)
.PLT1next:
  pushl  $name1RelOffset
  jmp    .PLT0
.PLT2:
  jmp    name2(%ebx)
.PLT2next
  pushl  $name2RelOffset
  jmp    .PLT0
```

Procedure-Linkage Table

ebx

```
_GLOBAL_OFFSET_TABLE:
   .long _DYNAMIC
   .long identification
   .long ld.so

name1:
   .long .PLT1next
name2:
   .long .PLT2next
```

Relocation info:

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

_DYNAMIC

- pushd onto the stack the offset of the `name1` entry in the relocation table
  - in the above example, this is 12 (if `name1` is the 4th entry in the GOT)

Before Calling `name1`

```
.PLT0:
  pushl  4(%ebx)
  jmp    8(%ebx)
  nop; nop
  nop; nop
.PLT1:
  jmp    name1(%ebx)
.PLT1next:
  pushl  $name1RelOffset
  jmp    .PLT0
.PLT2:
  jmp    name2(%ebx)
.PLT2next
  pushl  $name2RelOffset
  jmp    .PLT0
```

Procedure-Linkage Table

ebx

offset 0
offset 4
offset 8
offset 12
offset 16

```
_GLOBAL_OFFSET_TABLE:
   .long _DYNAMIC
   .long identification
   .long ld.so

name1:
   .long .PLT1next
name2:
   .long .PLT2next
```

Relocation info:

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

_DYNAMIC

- pushd onto the stack the offset of the `name1` entry in the relocation table
  - in the above example, this would be 12 (if `name1` is the 4th entry in the GOT)

Before Calling `name1`

```
.PLT0:
  pushl  4(%ebx)
  jmp    8(%ebx)
  nop; nop
  nop; nop
.PLT1:
  jmp    name1(%ebx)
.PLT1next:
  pushl  $name1RelOffset
  jmp    .PLT0
.PLT2:
  jmp    name2(%ebx)
.PLT2next
  pushl  $name2RelOffset
  jmp    .PLT0
```

Procedure-Linkage Table

ebx →

```
_GLOBAL_OFFSET_TABLE:
   .long _DYNAMIC
   .long identification
   .long ld.so

name1:
   .long .PLT1next
name2:
   .long .PLT2next
```

Relocation info:

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

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

_DYNAMIC

- unconditional jump to the beginning of the procedure-linkage table, `.PLT0`

Before Calling `name1`

```
.PLT0:
  pushl  4(%ebx)
  jmp    8(%ebx)
  nop; nop
  nop; nop
.PLT1:
  jmp    name1(%ebx)
.PLT1next:
  pushl  $name1RelOffset
  jmp    .PLT0
.PLT2:
  jmp    name2(%ebx)
.PLT2next
  pushl  $name2RelOffset
  jmp    .PLT0
```

Procedure-Linkage Table

ebx

offset 4

```
_GLOBAL_OFFSET_TABLE:
   .long _DYNAMIC
   .long identification
   .long ld.so

name1:
   .long .PLT1next
name2:
   .long .PLT2next
```

Relocation info:

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

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

_DYNAMIC

- pushd the "identification" of the current executing module on to the stack

Before Calling `name1`

```
.PLT0:
  pushl  4(%ebx)
  jmp    8(%ebx)
  nop; nop
  nop; nop
.PLT1:
  jmp    name1(%ebx)
.PLT1next:
  pushl  $name1RelOffset
  jmp    .PLT0
.PLT2:
  jmp    name2(%ebx)
.PLT2next
  pushl  $name2RelOffset
  jmp    .PLT0
```

Procedure-Linkage Table

ebx

offset 8

```
_GLOBAL_OFFSET_TABLE:
   .long _DYNAMIC
   .long identification
   .long ld.so

name1:
   .long .PLT1next
name2:
   .long .PLT2next
```

Relocation info:

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

_DYNAMIC

- unconditional jump to `ld.so`
  - `ld.so` can figure out who was making the request (TOS)
  - TOS-4 contains the GOT offset (12 in our example) of where to write the result into
  - TOS-8 contains index into symbol table in `_DYNAMIC`

Before Calling `name1`

```
.PLT0:
  pushl  4(%ebx)
  jmp    8(%ebx)
  nop; nop
  nop; nop
.PLT1:
  jmp    name1(%ebx)
.PLT1next:
  pushl  $name1RelOffset
  jmp    .PLT0
.PLT2:
  jmp    name2(%ebx)
.PLT2next
  pushl  $name2RelOffset
  jmp    .PLT0
```

Procedure-Linkage Table

ebx →

```
_GLOBAL_OFFSET_TABLE:
   .long _DYNAMIC
   .long identification
   .long ld.so

name1:
   .long name1
name2:
   .long .PLT2next
```

Relocation info:

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

_DYNAMIC

- `ld.so` writes the actual address of the `name1` procedure into the `name1` entry of the GOT
  - unwinds the stack a bit and then passes control to `name1`

After Calling `name1`

```
.PLT0:
  pushl  4(%ebx)
  jmp    8(%ebx)
  nop; nop
  nop; nop
.PLT1:
  jmp    name1(%ebx)
.PLT1next:
  pushl  $name1RelOffset
  jmp    .PLT0
.PLT2:
  jmp    name2(%ebx)
.PLT2next
  pushl  $name2RelOffset
  jmp    .PLT0
```

Procedure-Linkage Table

ebx →

```
_GLOBAL_OFFSET_TABLE:
   .long _DYNAMIC
   .long identification
   .long ld.so

name1:
   .long name1
name2:
   .long .PLT2next
```

Relocation info:

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

_DYNAMIC

- Subsequent call to `name1` invokes more directly