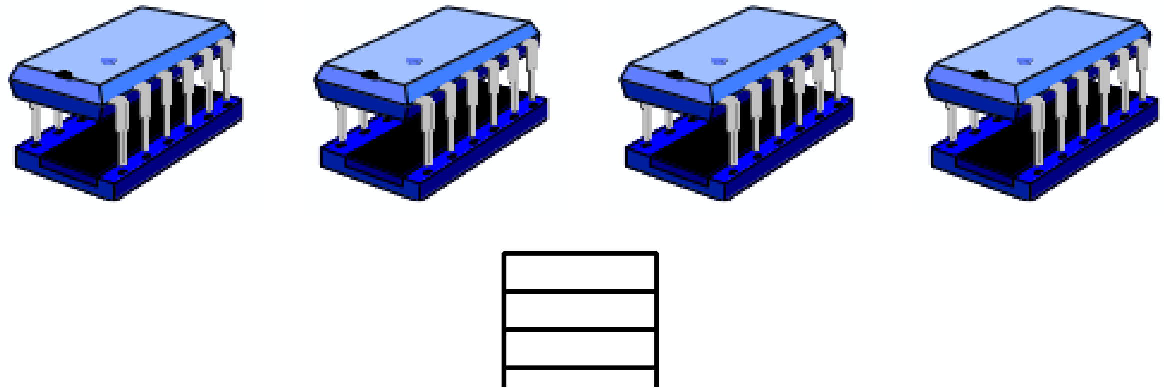
Utilizing Multiple Processors

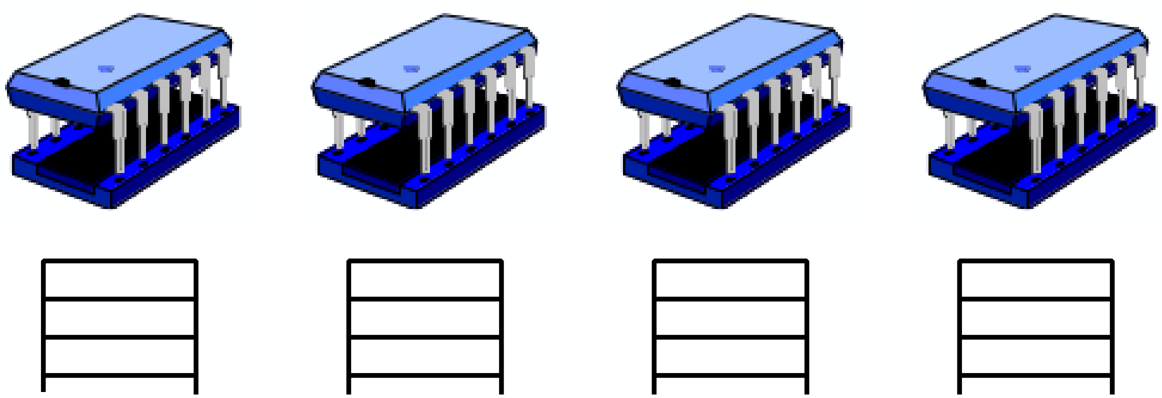
- Dedicate processors to important threads?
- Restrict interrupt handling to certain processors?
- Windows:
  - processor affinity masks
- Solaris:
  - processor sets

Cache Affinity

- After a thread has run on a particular processor, next time it runs, it would be cheaper to run it on the same processor
  - *cache affinity*

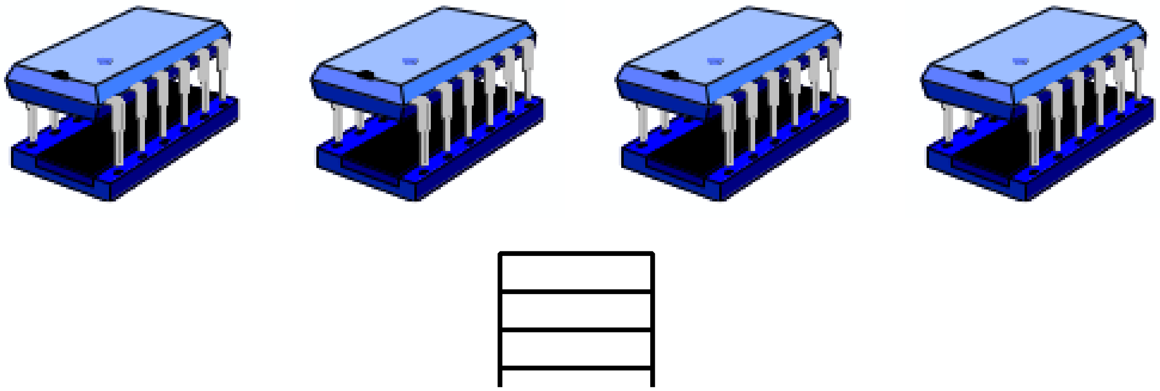


- This means that if you use a shared run queue for multiple processors, you will not be able to take advantage of cache affinity
  - therefore, you should use one run queue per processor

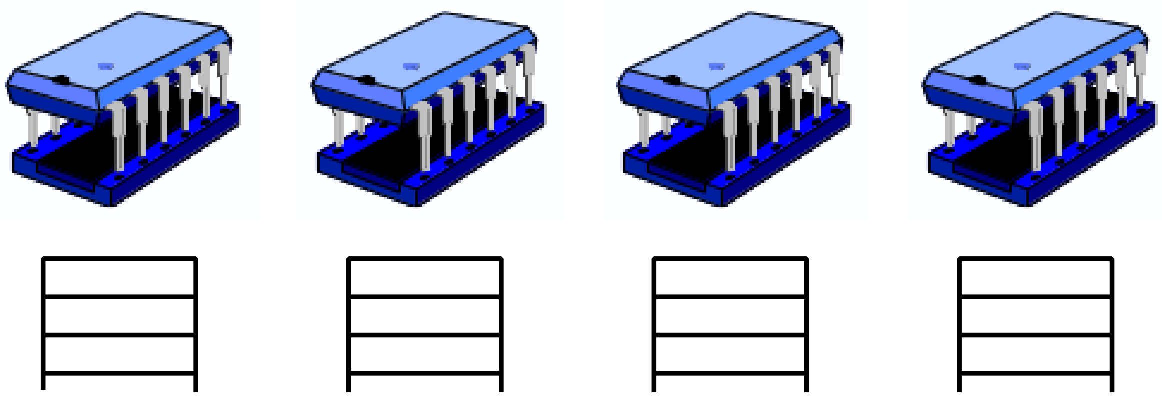


Cache Affinity

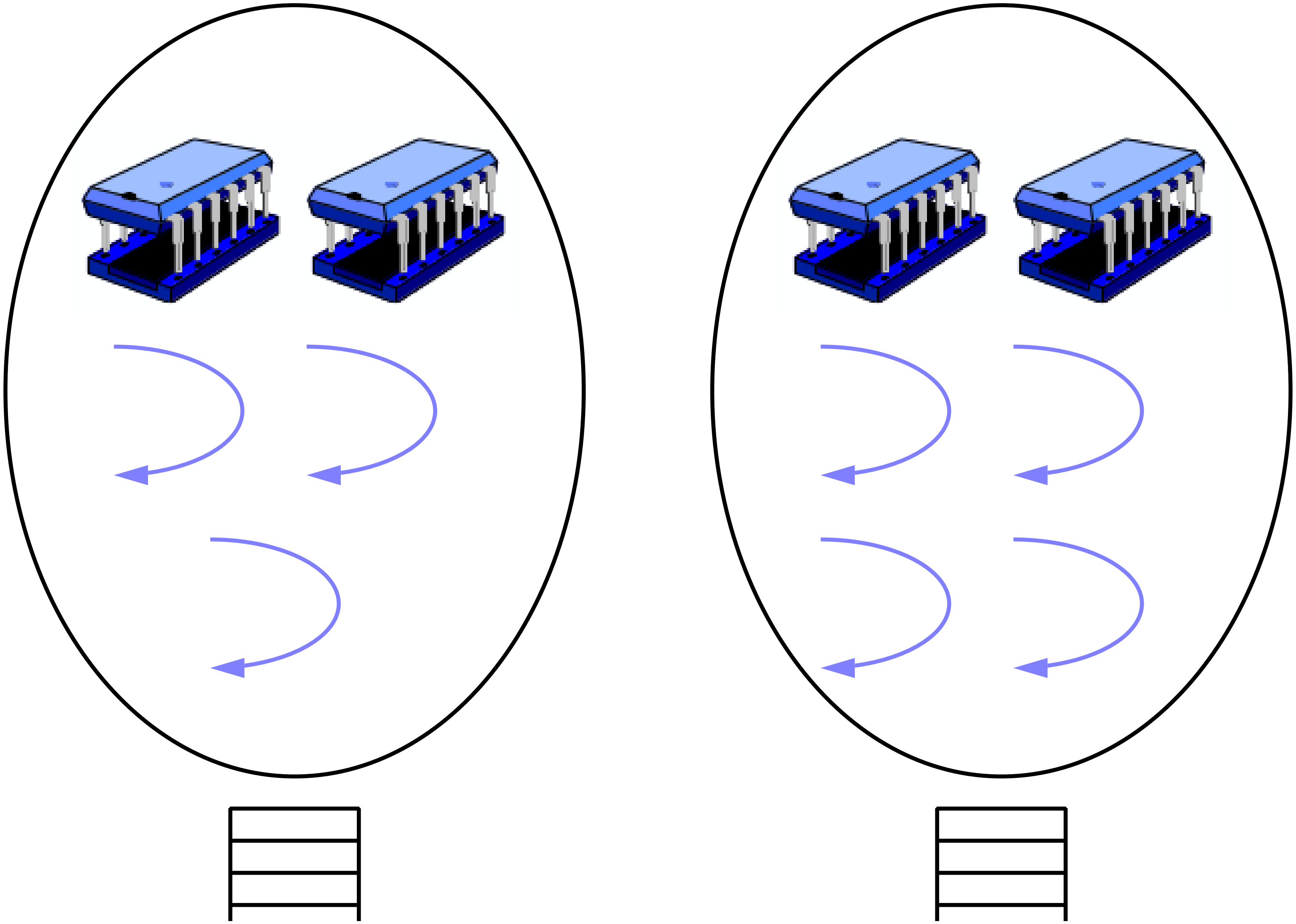
- After a thread has run on a particular processor, next time it runs, it would be cheaper to run it on the same processor
  - *cache affinity*



- This means that if you use a shared run queue for multiple processors, you will not be able to take advantage of cache affinity
  - therefore, you should use one run queue per processor
  - scheduler may do *load balancing* occasionally



Solaris: Processor Sets



- Somewhere between the two extremes
  - reducing the frequency of requiring load balancing

5.3 Scheduling

- Goals
- Scheduling Algorithms
- Implementation Issues
- *Case Studies*

Linux Scheduling

- Policies mandated by POSIX
  - SCHED_FIFO (highest)
    - "real time"
    - *infinite time quantum*
  - SCHED_RR
    - "real time"
    - *adjustable time quantum*
  - SCHED_OTHER
    - "normal" scheduler
    - parameterized allocation of processor time

Linux Scheduler Evolution

- **Old scheduler**
  - very simple
  - poor scaling
- **O(1) scheduler**
  - introduced in 2.5
  - less simple
  - better scaling
- **Completely fair scheduler *(CFS)***
  - even better
  - simpler in concept
  - much less so in implementation
  - based on *stride* scheduling

Old Scheduler

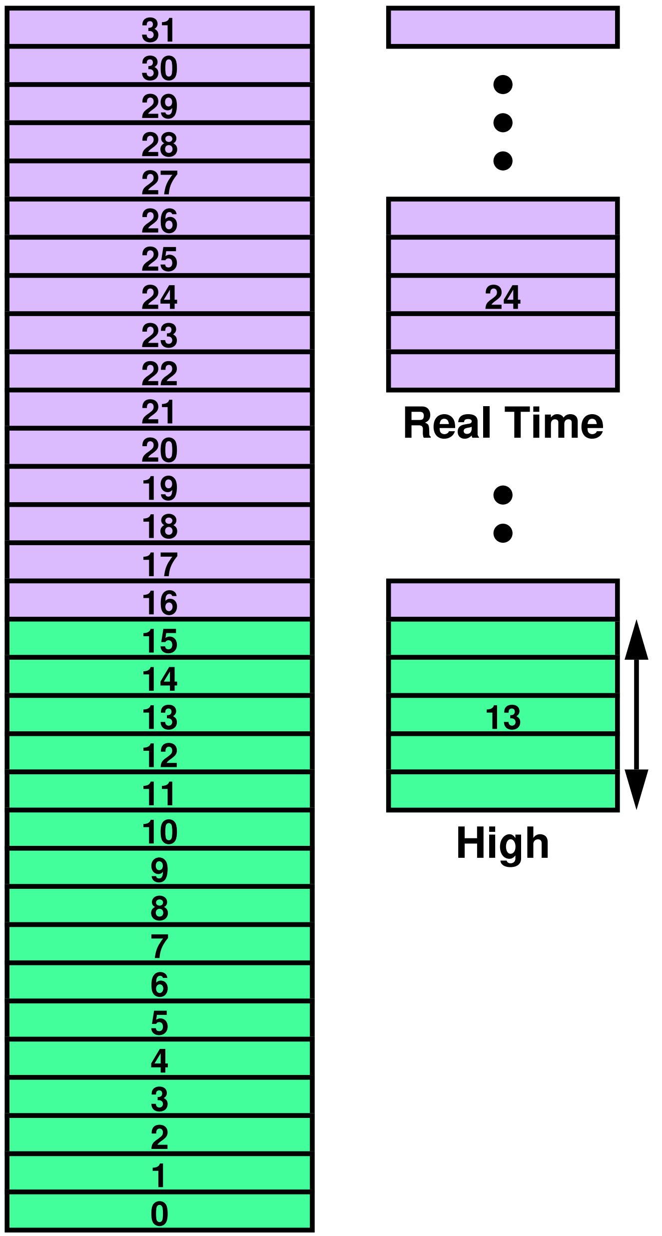
- Four per-process scheduling variables
  - *policy:* which one
  - *rt_priority:* real-time priority
    - 0 for SCHED_OTHER
    - 1 - 99 for others
  - *priority:* time-slice parameter ("nice" value)
  - *counter:* records processor consumption

  - see "extra slides" at the end

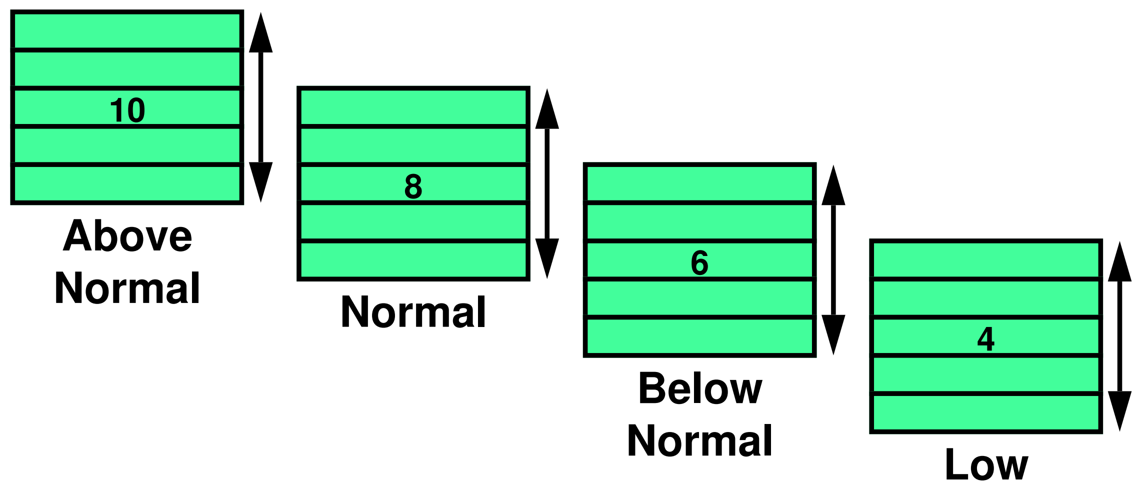
Scheduling in Windows

- Handling "normal" interactive and compute-bound threads
- Real-time threads
- Multiple processors

Windows Priority Classes and Levels



- these are *user* thread priority levels
  - *not* related to interrupt priority levels
- longer time slice for foreground threads (i.e., belonging to the *"active"* window)
- a *real-time* thread gets a *fixed priority*
- not exactly the same rules for multi-level feedback queues than what we have talked about for non-real-time threads



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`)

- Running it

```
% ./prog
sub1sub2sub3%
```

Substitution

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

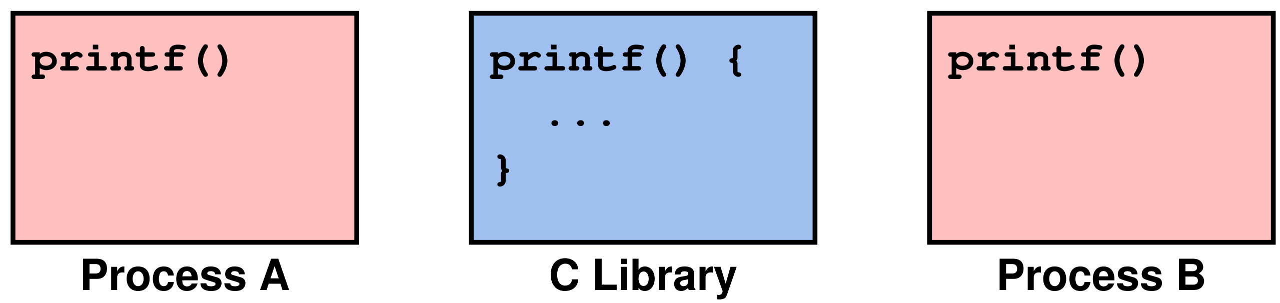
```
% cat myputs.c
int puts(char *s) {
  write(1, "My puts: ", 9);
  write(1, s, strlen(s));
  write(1, "\n", 1);
  return 1;
}
% gcc -c myputs.c
% ar cr libmyputs.a myputs.o
% gcc -o prog prog.c -L. -lpriv1 -lmyputs
```

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

```
% ./prog
My puts: sub1
My puts: sub2
My puts: sub3
```

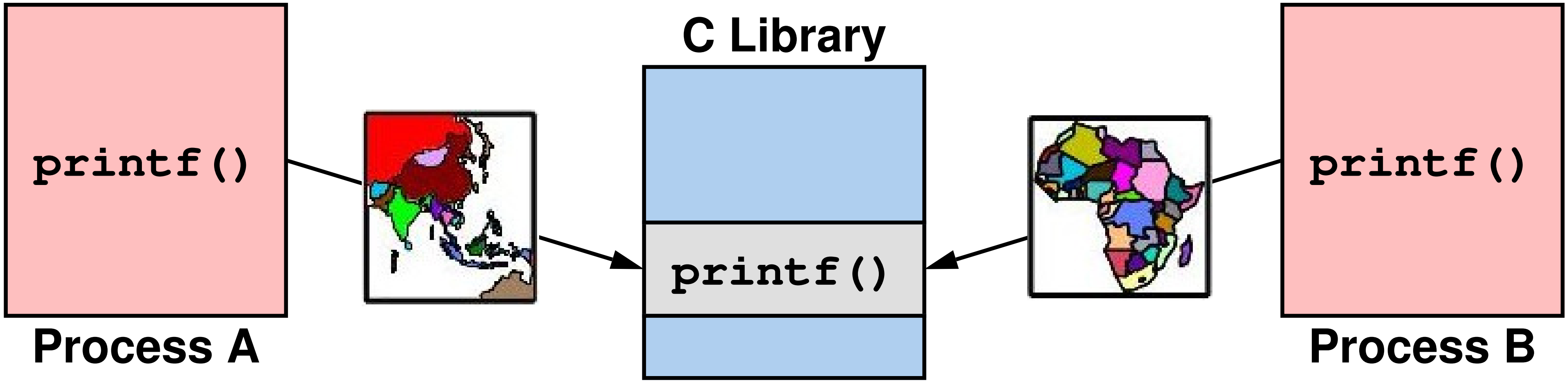
Shared Libraries

- `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

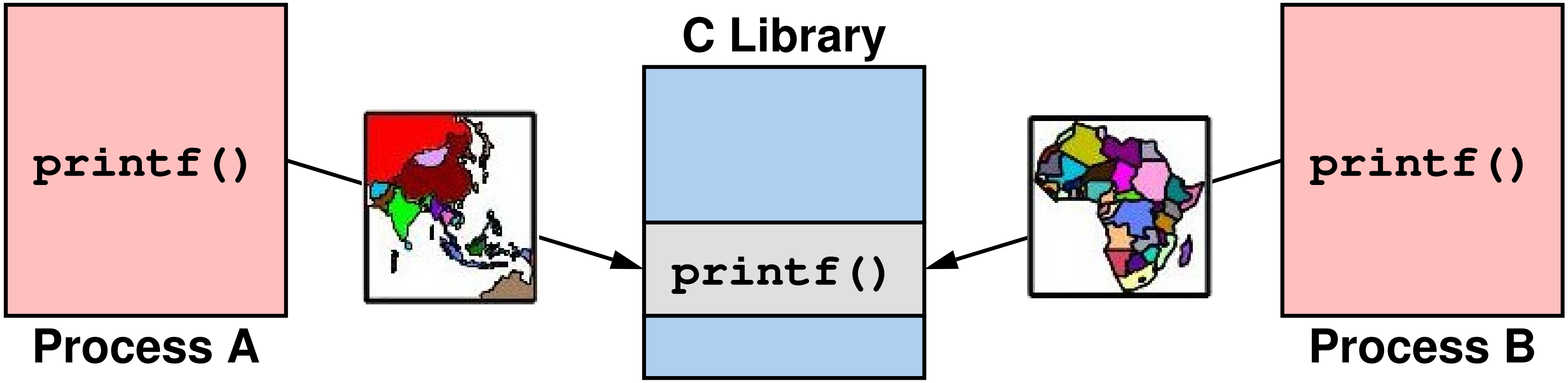


- If `printf()` required no relocation, then it's easy
  - just make sure `ld` use the right address consistently
- If `printf()` required relocation, then it's more complicated
  - problem: processes want a shared function to be at different addresses

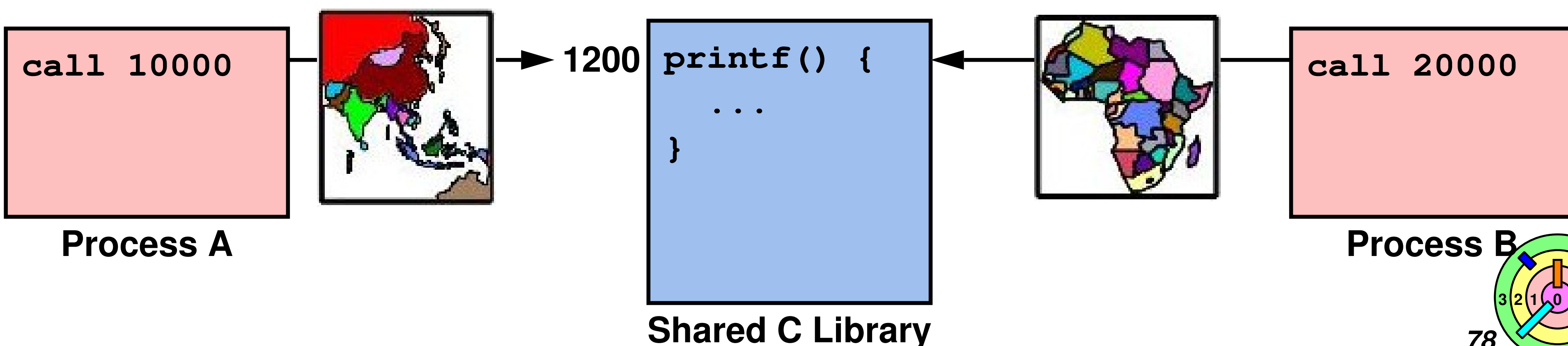
Sharing



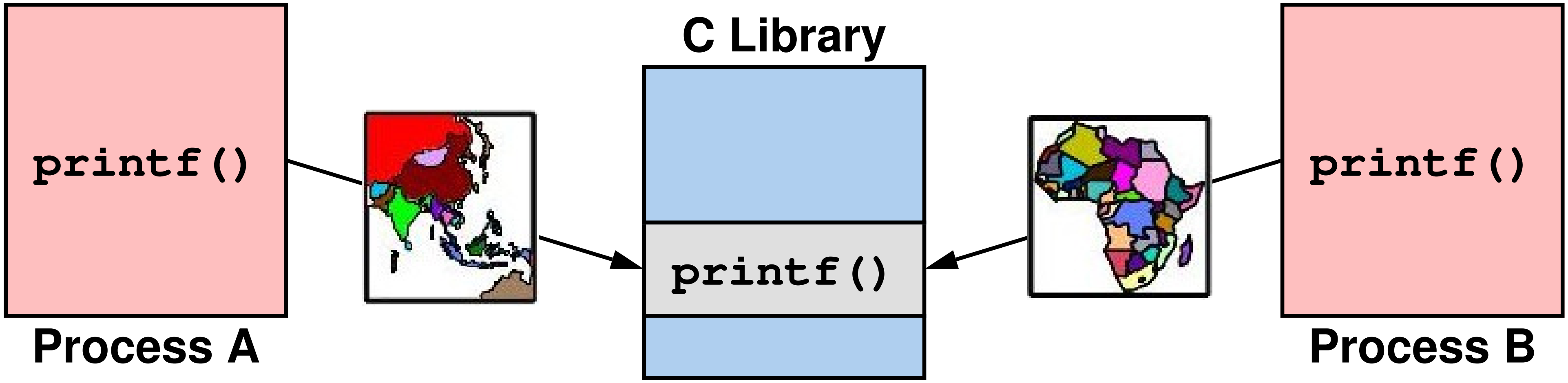
Sharing



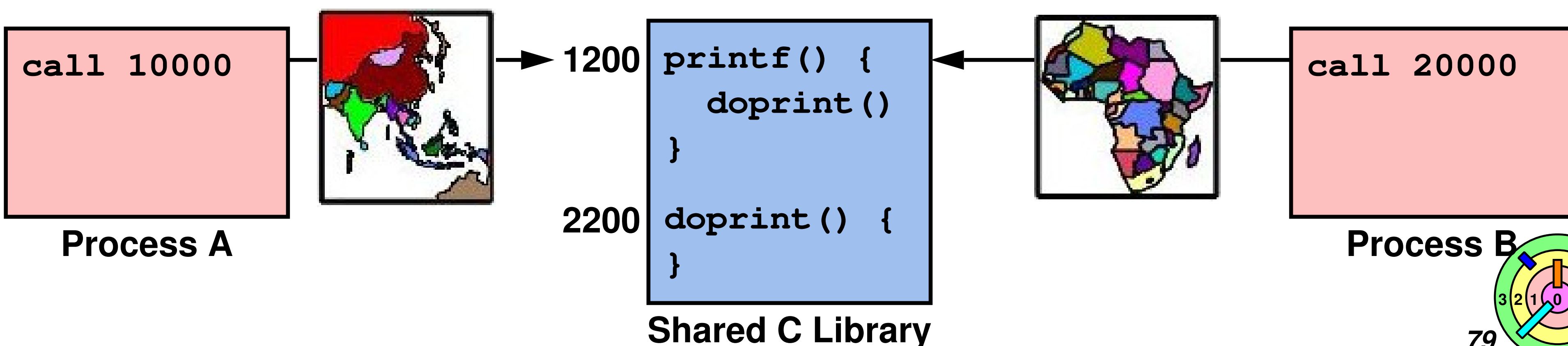
- Looks like it can work



Sharing

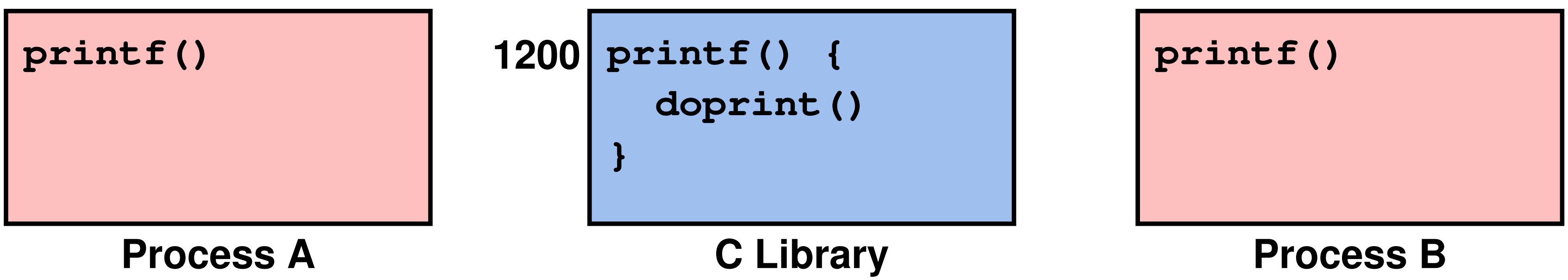


- What about this?
  - with static linking, should `doprint()` be relocated to `11000` or `21000` (we must use virtual address)?



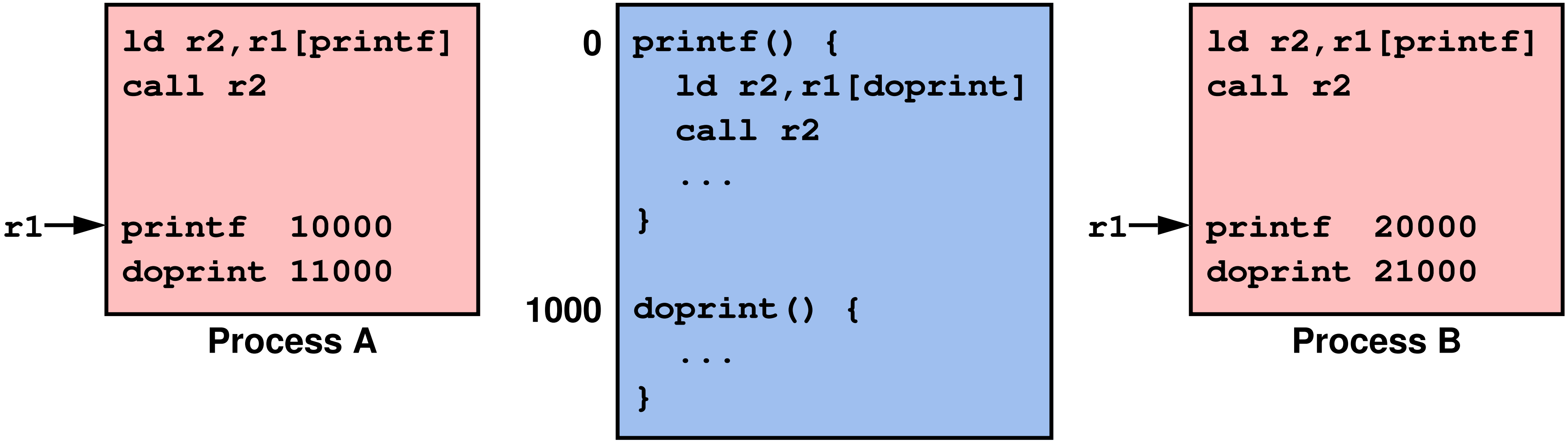
Relocation and Shared Libraries

- **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



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

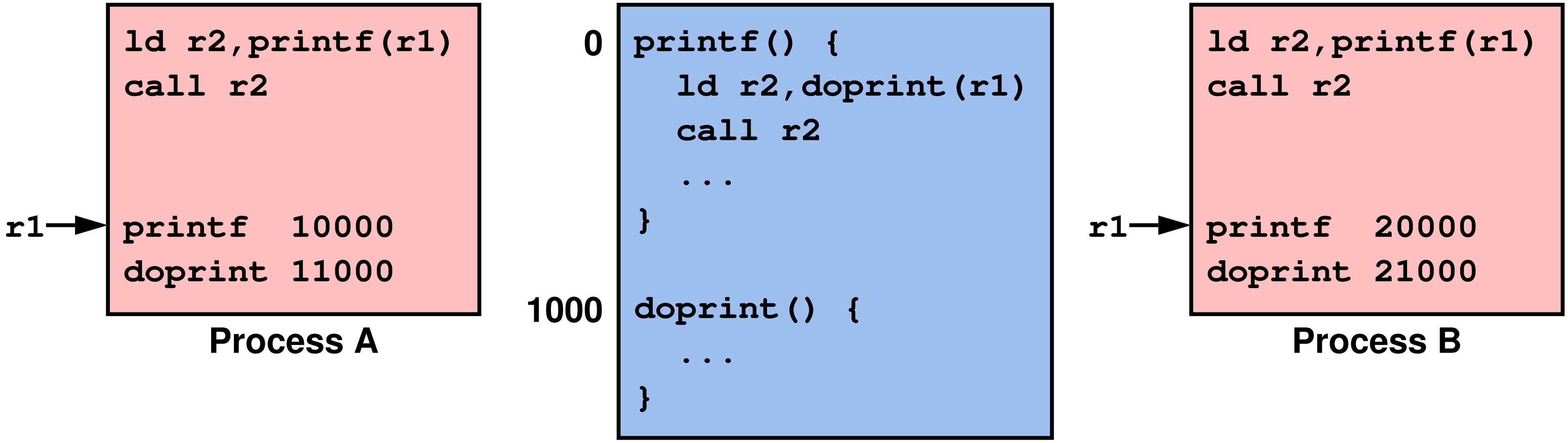
Position-Independent Code



- 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., a *register-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



- 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., a *register-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

  - see "extra slides" at the end

Shared Libraries In Practice

- ***Shared libraries*** are used extensively in many modern systems
  - often implemented with 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 (details in "extra slides")
    - then calls the actual program code
    - it may be called upon later to do some further dynamic loading and linking

```
ld r2,printf(r1)
call r2


printf  10000
doprint 11000
```

r1 → printf

Process A

```
0     printf() {
        ld r2,doprint(r1)
        call r2
        ...
      }

1000  doprint() {
        ...
      }
```

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
```

- `ldd` prints shared library dependencies

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
```

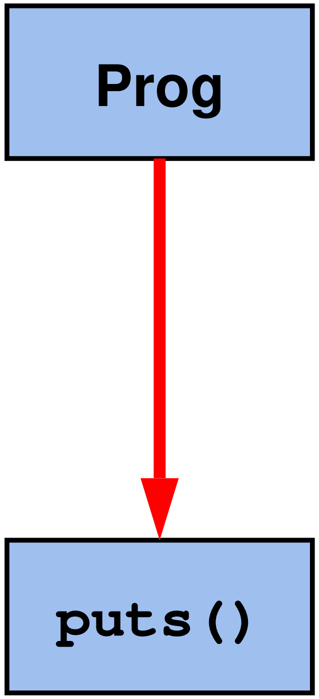
- "`-Wl,-rpath .`" means that what comes after `-Wl` are linker options (i.e., pass them to the linker)
  - in this example, the linker will be invoked with "`-rpath .`"
- also try "`-Wl,-rpath,.`" if the space character is giving you trouble

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 .
```

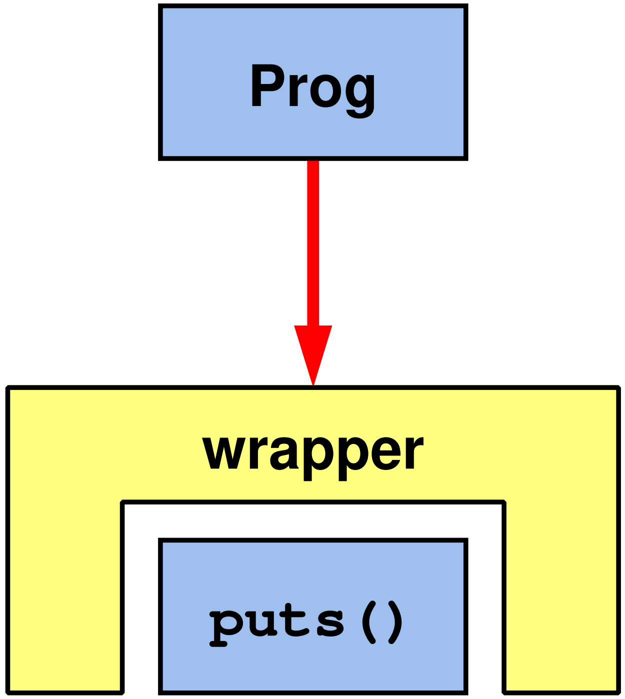
- "`libmyputs.so.1`" is the ***soname***
- "`libmyputs.so.1.0`" is the ***real name***
- "`libmyputs.so`" is the ***linker name***

Dynamic Substitution: Interpositioning



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

Dynamic Substitution: 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, "intercepted by myputs: ", 23);
  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
intercepted by myputs: This is a boring message.
%
```

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

Delayed Wrapping

- Some environment variables are checked by `ld.so`
  - potential security problem
- LD_PRELOAD
  - 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
intercepted by myputs: This is a boring message.
%
```

- LD_LIBRARY_PATH
  - list of directories to search for dynamic libraries

```
% setenv LD_LIBRARY_PATH /usr/lib:/lib:.
```

Summary

- `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` on Linux) to *finish linking*
  - maps in shared objects
  - does relocation and procedure linking as required
  - `dlsym()` invokes `ld.so` to do *more linking*

Wrap Up

- **We are done for the semester!**

Wrap Up

- The final exam coverage will be posted on the class web site
  - the coverage does *not* overlap the midterm coverage
  - but since the 2nd half of the semester depends on the 1st half, I cannot say that I will not ask anything that was covered in the midterm coverage
    - it would be more appropriate to say that the final exam will *focus* on the material *not covered by the midterm*
- No office hours after this week
- I apologize again for going so fast
  - but as you can see, there is no time left!
  - the only way I can slow down is to cut lecture materials
    - but there is really nothing I can think of to cut