CS 33

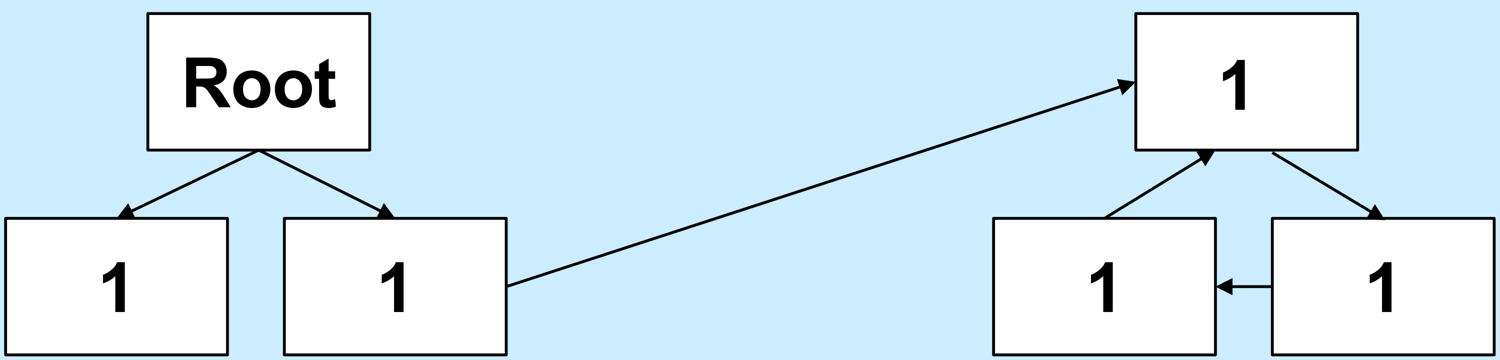
Multithreaded Programming VIII

Garbage Collection

- malloc − free
  - when a malloc'd block is no longer needed (it's *garbage*), it's (eventually) automatically returned to the free list
    - » how is this done?
    - » can it be done by one thread, while other threads are calling malloc and using the memory?

Identifying Garbage – Reference Counts

- **Assume all memory blocks are nodes in a graph, each with two links**
  - for each block, keep reference counts: how many other blocks point to it
  - if reference count is 0, then no node points to it and it's garbage
    - » certain nodes are designated as *roots*—it's ok if no nodes point to them



Quiz 1

If we can guarantee that the graph formed by memory nodes has no cycles, then reference counts form an effective means for identifying garbage.

a) **yes: a node is garbage if and only if its reference count is 0**

b) **yes: if a node's reference count is 0, it's garbage, but it might be necessary to remove some garbage nodes to find others**

c) **no: a node could have a reference count of 0 and not be garbage**

Identifying Garbage – Mark and Sweep

- Identify all nodes that lie on paths that start from a root
- All other nodes, being unreachable, are garbage

Code

```
void mark(node_t *root) {
   if (!node->visited) {
      node->visited = 1;
      if (node->left) mark(node->left);
      if (node->right) mark(node->right);
   }
}

void sweep(void) {
   for (int i=0; i<M; i++) {
      if (node[i].visited == 0)
         free(node);
      node[i].visited = 0;
   }
}
```

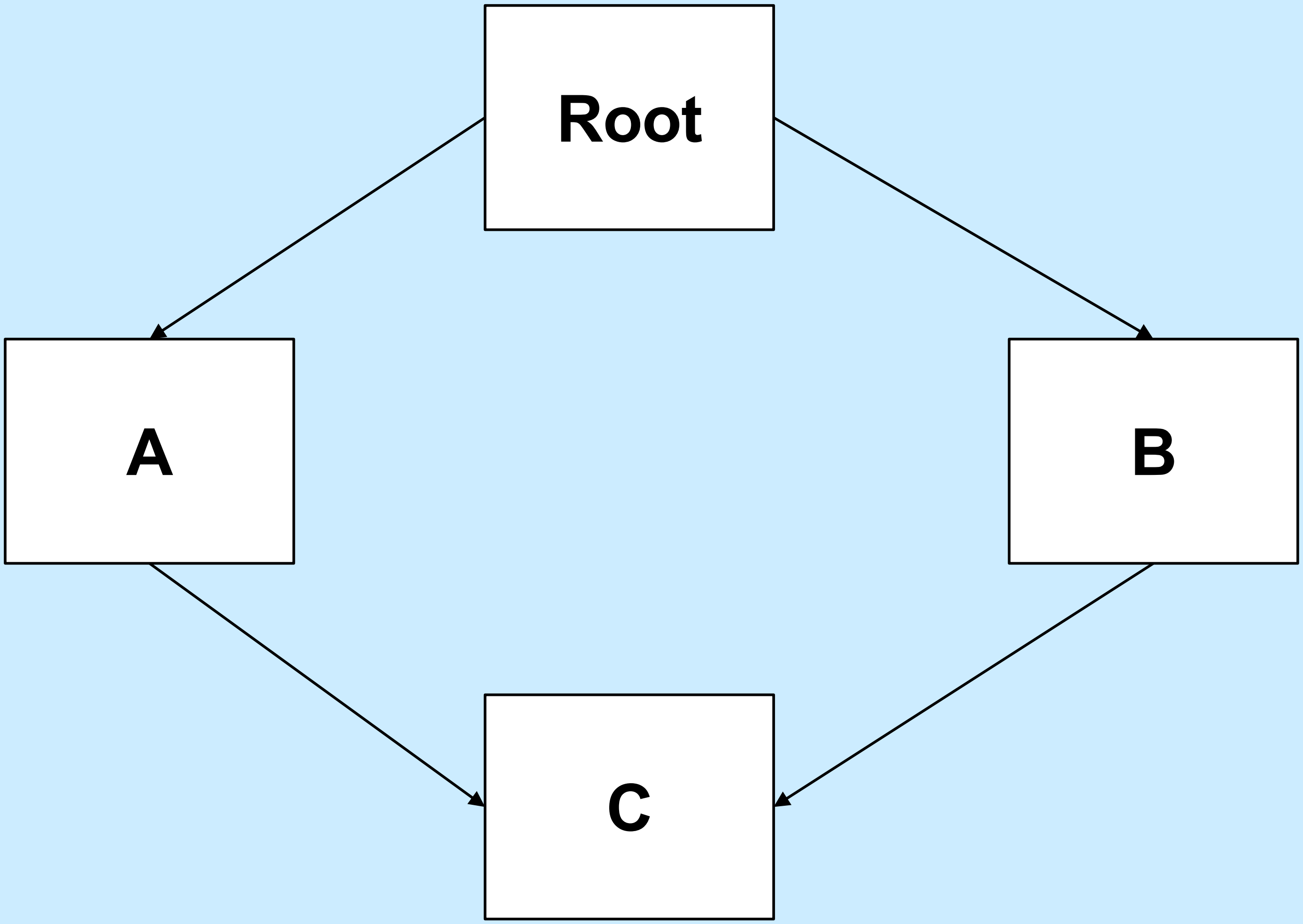

Mutator

Threads that modify the graph (perhaps malloc’ing new nodes) are called *mutators*.

- Mutators perform mutate operations on individual nodes. They might
  - change either the left or right link of a node to point to a non-garbage node, possibly resulting in the old target becoming garbage
  - cause a node to point to a newly allocated node
  - cause a link to be NULL

Can the mutator and the garbage collector run in parallel as separate threads?

A Problem



A Fix

```
void mutate(node_t *node, int dir, node_t *new_target) {
   if (dir == LEFT) {
      node->left.visited = 1;
      node->left = new_target;
   } else { // dir == RIGHT
      node->right.visited = 1;
      node->right = new_target;
   }
}
```


Quiz 2

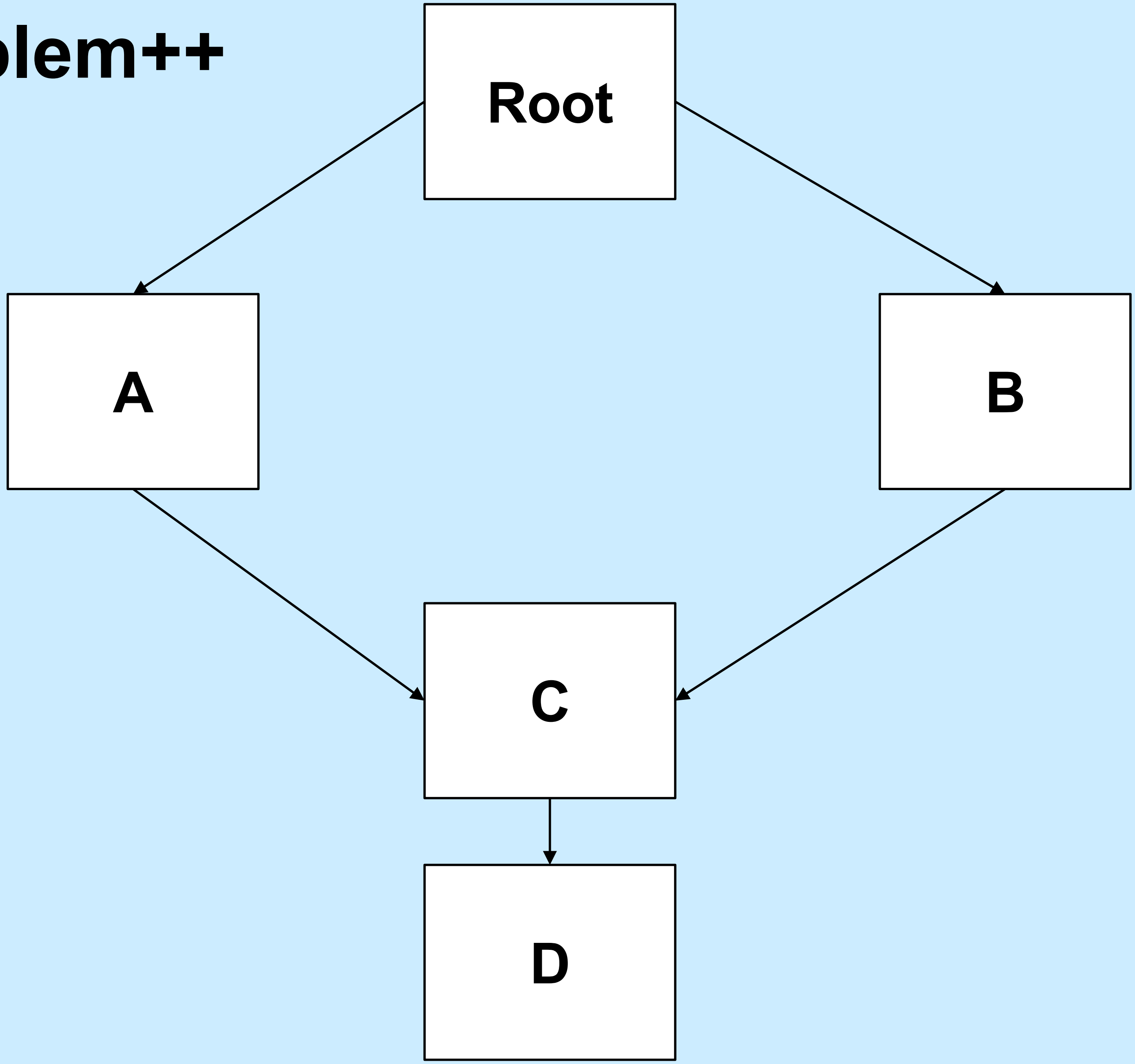
The fix:

a) allows the mutator and GC to correctly run concurrently

b) solves the specific problem of two slides ago, but leaves other problems unsolved

c) solves nothing

Problem++



Coping

- When a node is marked “visited”, we must mark all nodes reachable from it
- It’s necessary to distinguish nodes that have been visited and whose direct descendants have also been marked visited from those that have only been visited
  - visited = 0: not visited
  - visited = 1: visited, but status of direct descendants is unknown
  - visited = 2: visited and direct descendants are marked visited (1 or 2)
- Ultimately, all nodes will have visited values of 0 (meaning garbage) or 2 (nongarbage)

New Mark Function

```
void mark(void) {
   root->visited = 1;
   i=0;
   k=M; // total number of nodes in memory
   while (k>0) {
      if (node[i].visited == 1) {
         k = M; // reset k so all nodes are reexamined
         visit(node[i].left);
         visit(node[i].right);
         node[i].visited = 2;
      } else
         k--; // the node's visited value was 0 or 2
      i = i++ mod M; // not legal C syntax
   }
}
```


New Mutate Function

```
void mutate(node_t *node, int dir, node_t *new_target) {
   if (dir == LEFT) {
      visit(node->left);
      node->left = new_target;
   } else { // dir == RIGHT
      visit(node->right);
      node->right = new_target;
   }
}
```


Quiz 3

Assume that each line of code is executed atomically. Suppose, during the execution of *mark* by the GC thread, a mutator thread causes a node (that previously was not garbage) to become garbage. That node's *visited* field will become 0 (causing it to be treated as garbage)

a) during the current execution of *mark*

b) during the upcoming execution of *sweep*

c) during the next execution of *mark*

d) never

Visit and Sweep Functions

```
void visit(node_t *node) {
   if (node->visited == 0)
      node->visited = 1;
}

void sweep(void) {
   for (int i=0; i<M; i++) {
      if (node[i].visited == 0)
         free(node);
      node[i].visited = 0;
   }
}
```

Quiz 4

When sweep is being executed, will it encounter any nodes for which *visited* is 1?

a) yes

b) no

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
$
```


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
```


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

Shared Objects: Unix's Dynamic Linking

1 **Compile program**

2 **Track down references 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 first to complete the linking and relocation steps, if necessary**

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
```


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

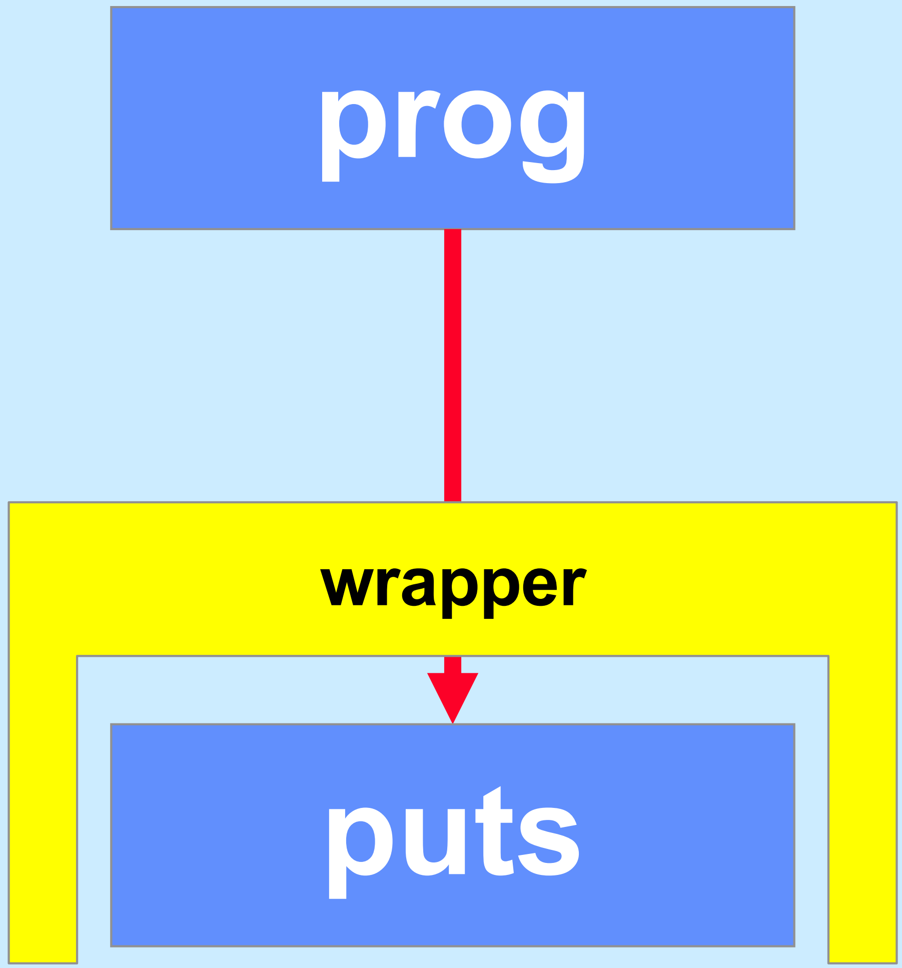
A Solution

- 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 .
```

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

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.
$
```

