CS 33

Multithreaded Programming VIII

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

Today's lecture is partly based on "On-the-Fly Garbage Collection: An Exercise in Cooperation", by E. Dijkstra, L. Lamport, A. Martin, C. Scholten, and E. Steffens: https://lamport.azurewebsites.net/pubs/garbage.pdf. The paper was published in the Communications of the ACM in November 1978.

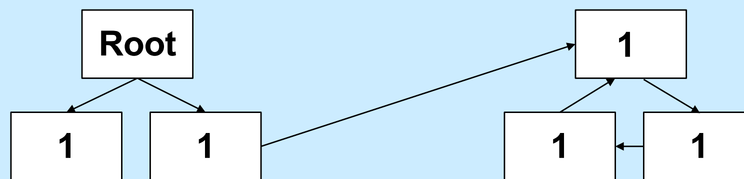
Garbage Collection

- 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?

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

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



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

If a group of nodes form a cycle, then their reference counts will always be positive.

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

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

Identifying Garbage – Mark and Sweep

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

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

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;
   }
}
```

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

In this code, we assume that each node has a left link and a right link. We also assume that there is only one root (though it could easily be modified to handle multiple roots). Any node that is not on a path from the root is garbage.

Mutator

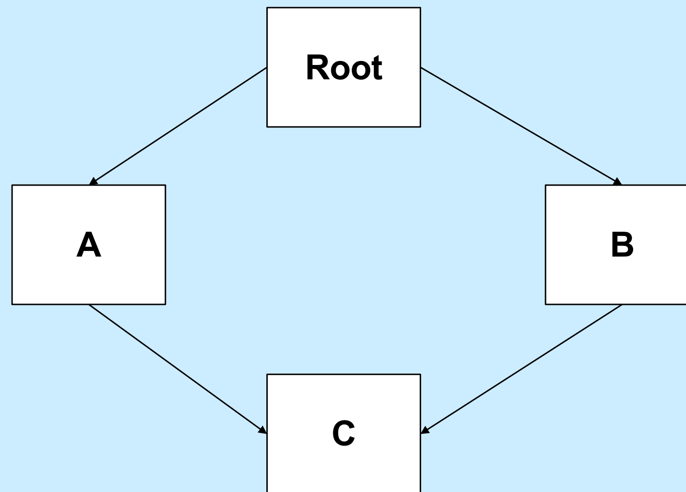
Threads that modify the graph (perhaps malloc'ing new nodes) are called *mutators*.

- 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?

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

A Problem



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

Initially, there's a path from the root to C via A (but not via B). Then after a couple mutator operations, a path through B appears, but the one through A goes away. During the mark phase of garbage collection, it might first mark what B is linked to. Then, after the link from B to C appears and the one from A to C goes away, it marks what A is connected to. Thus it misses the fact that C is reachable (though the path taken to reach it changes).

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;
   }
}
```


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

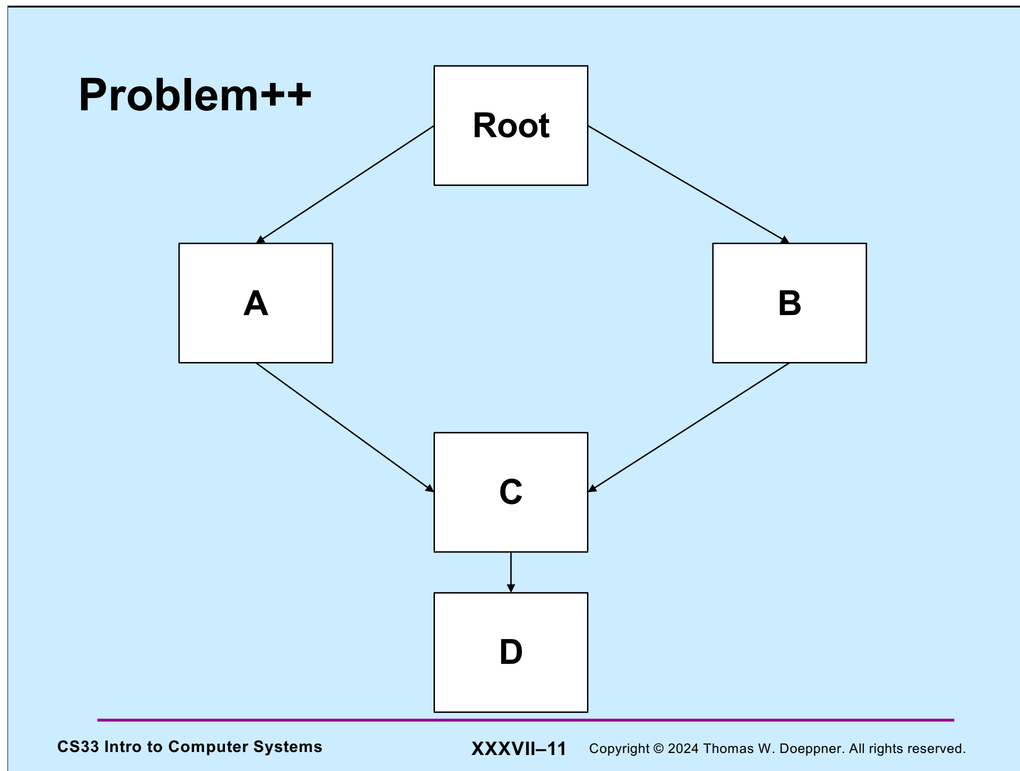
We modify the mutator so that when one of its links is modified, the old target of the link is marked as having been visited.

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

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



The problem of the earlier slide is exacerbated if another node is pointed to (only) by C.

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)

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

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
   }
}
```

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

With this new version of mark, rather than perform a depth-first search of the graph, we possibly repeatedly examine all nodes after setting the root (or roots) as visited. Because of this property, this is not a practical algorithm.

The visit function sets the visited field of the argument node to 1 if it was 0, but leaves it unchanged otherwise.

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;
   }
}
```

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

The new mutate function sets the visited field of the old target to 1 if it was zero, but leaves it unchanged otherwise.

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

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

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

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

CS 33

Linking and Libraries

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

Libraries

- Collections of useful stuff
- Allow you to:
  - incorporate items into your program
  - substitute new stuff for existing items
- Often ugly …

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

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
$
```

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

Files ending with ".a" are known as **archives** or **static libraries**.

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
```

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

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!

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

Example

```
$ cat prog2.c
int main() {
  void func1();
  func1();
  return 0;
}
$ cat func1.c
void func1() {
  void func2();
  func2();
}
$ cat func2.c
void func2() {
}
```

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

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
```

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

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
```

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

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?

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

Dynamic Linking

- Executable is not fully linked
  - contains list of needed libraries
- Linkages set up when executable is run

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

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

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

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**

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

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
```

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

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

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

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

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

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

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

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 .
```

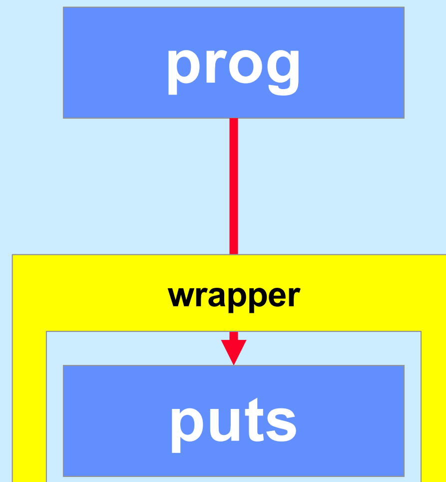
 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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



 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.