CS 33

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

```bash
$ 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. -lfl -lf2
$
$ gcc -o prog2 prog2.c -L. -lf2 -lfl
./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 (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
linux-vdso.so.1 => (0x00007fff953fc000)
libmyputs.so => not found
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f7389174000)
/lib64/ld-linux-x86-64.so.2 (0x00007f7389536000)
```
Creating a Shared Library (2)

$ 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

```bash
$ 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

prog

wrapper

puts
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.
$
Mmapping Libraries

available for mmap

stack

my lib

C library

dynamic

bss

data

text
Problem

• How is relocation handled?
But …
But …
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 mylibrary into our address space?

a) the MAP_SHARED option
b) the MAP_PRIVATE option
c) `mmap` can’t be used in this situation
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
Mapping Shared Objects

Process A

printf( )

stdio

printf( )

Process B

printf( )
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
Mapping printf

Process 1
- printf text
  - page 6
  - page 7
- printf data
  - page 31
  - page 32

Process 2
- printf text
  - page 3
  - page 4
- printf data
  - page 41
  - page 42

Real Memory
- printf
  - page 0
- P1’s printf
  - page 2
- P1’s printf
  - page 3
- P2’s printf
  - page 2
- printf
  - page 1

Disk
- text
  - page 0
- page 1
- page 2
- page 3
- data
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
Global-Offset Table: Data References

Global Offset Table

errno

errno address

myglob

myglob address
Procedures in Shared Objects

• Lots of them
• Many are never used
• Fix up linkages on demand
An Example

```c
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>

...
Before Calling puts

```
.PLTO:
pushq  GOTO+8(%rip)
jmp   *GOTO+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
```

```
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)
```
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
```
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*