CS 33

Libraries
Libraries

- Collections of useful stuff
- Incorporate items into your program
- Replace existing items with new stuff
- 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

```c
$ cat prog.c
int main() {
    sub1();
    sub2();
    sub3();
}
$ cat sub1.c
void sub1() {
    puts("sub1");
}
```

```sh
$ gcc -o prog prog.c -L. -lpriv1
$ ./prog
sub1
sub2
sub3
Where does puts come from?
```

```sh
$ 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:
  – (assume 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

```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
A 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 libmymyputs.so myputs.o
$ gcc -o prog prog.c -L. -lpriv1 -lmyputs
$ ./prog
./prog: error while loading shared libraries: libmymyputs.so: cannot open shared object file: No such file or directory
$ ldd prog
linux-vdso.so.1 => (0x00007fff953fc000)
libmymyputs.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 -L. -lpriv1 -lmyputs -Wl,-rpath .
$ ldd prog
linux-vdso.so.1 => (0x00007fff235ff000)
libmyputs.so => ./libmyputs.so (0x00007f821370f000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
(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
Versioning

$ gcc -fPIC -c myputs.c
$ ld -shared -soname libmyputs.so.1 \ 
- o libmyputs.so.1 myputs.o
$ ln -s libmyputs.so.1 libmyputs.so
$ gcc -o prog1 prog1.c -L. -lpriv1 -lmyputs \ 
-Wl,-rpath .
$ vi myputs.c
$ ld -shared -soname libmyputs.so.2 \ 
- o libmyputs.so.2 myputs.o
$ rm -f libmyputs.so
$ ln -s libmyputs.so.2 libmyputs.so
$ gcc -o prog2 prog2.c -L. -lpriv1 -lmyputs \ 
-Wl,-rpath .
Interpositioning

prog

wrapper

puts
How To ...

```c
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) …

```c
#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
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?
Pre-Relocation

math library

- call printf

C library

- stdfiles: 1,200,600
- &stdfiles
- printf: 1,000,400

- call printf
  1000400
But …

- Ian’s library: 5,000,000
- My library: 5,500,000
But …
Quiz 1

We’ve retargeted all references in our code to Ian’s library. What option should we give to *mmap* when we map the library into our address space? (Hint: is there more work that needs to be done?)

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 register
Mapping Shared Objects

Process A

printf(

stdio

Process B

printf( )

printf( )

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

- printf text
  - page 6
  - page 7
  - page 31
  - page 32
- printf data
  - page 3
  - page 4
  - page 41
  - page 42

Process 1

- P1’s printf
  - page 2
  - page 3

Real Memory

- printf page 0
- P1’s printf page 2
- P1’s printf page 3

Disk

- page 0
- page 1
- page 2
- page 3

- printf page 1
- P2’s printf page 2
Position-Independent Code

• Processor-dependent; x86-64:
  – each dynamic executable and shared object has:
    » procedure-linkage table
      • shared, read-only executable code
      • essentially stubs for calling subroutines
    » 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
Before Calling Name1

```
.PLT0:
pushq GOT+8(%rip)
jmp  *GOT+16(%rip)
nop;  nop
nop;  nop

.PLT1:
jmp  *name1@GOTPCREL(%rip)

.PLT1next
pushq $name1RelOffset
jmp  .PLT0

.PLT2:
jmp  *name2@GOTPCREL(%rip)

.PLT2next
pushq $name2RelOffset
jmp  .PLT0

Procedure-Linkage Table
```

```
GOT:
  .quad  _DYNAMIC
  .quad  identification
  .quad  ld-linux.so

name1:
  .quad  .PLT1next
name2:
  .quad  .PLT2next

Relocation info:

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

Relocation Table
```
After Calling Name1

Procedure-Linkage Table

.GOT:  
.quad _DYNAMIC  
.quad identification  
.quad ld-linux.so

.name1:
.quad name1

.name2:
.quad .PLT2next

Relocation info:

GOT_offset(name1), symx(name1)

GOT_offset(name2), symx(name2)

Relocation Table