Systems I

Linking II

Topics
- Relocation
- Static libraries
- Loading
- Dynamic linking of shared libraries
Relocating Symbols and Resolving External References

- **Symbols** are lexical entities that name functions and variables.
- Each symbol has a **value** (typically a memory address).
- Code consists of symbol **definitions** and **references**.
- References can be either **local** or **external**.

```c
int e=7;
int main() {
    int r = a();
    exit(0);
}
```

```c
extern int e;
int *ep=&e;
int x=15;
int y;
int a() {
    return *ep+x+y;
}
```

- **Def of local symbol e**
- **Ref to external symbol exit (defined in libc.so)**
- **Ref to external symbol ep**
- **Def of local symbol a**
- **Ref to external symbol e**
- **Defs of local symbols x and y**
- **Refs of local symbols ep, x, y**
m.o Relocation Info

Disassembly of section .text:

Disassembly of section .data:

source: objdump
extern int e;
int *ep=&e;
int x=15;
int y;
int a() {
  return *ep+x+y;
}

Disassembly of section .text:

00000000 <a>:
  0:  55 pushl %ebp
  1:  8b 15 00 00 00 movl 0x0,%edx
  6:  00

  3: R_386_32 ep

  7: a1 00 00 00 00 movl 0x0,%eax

  8: R_386_32 x

  c:  89 e5 movl %esp,%ebp
  e:  03 02 addl (%edx),%eax
  10: 89 ec movl %ebp,%esp
  12: 03 05 00 00 00 addl 0x0,%eax
  17: 00

  14: R_386_32 y

  18: 5d popl %ebp
  19: c3 ret
a.c

extern int e;
int *ep=&e;
int x=15;
int y;

int a() {
    return *ep+x+y;
}

Disassembly of section .data:
00000000 <ep>:
  0:  00 00 00 00

00000004 <x>:
  4:  0f 00 00 00
Executable After Relocation and External Reference Resolution (.text)

08048530 <main>:
  8048530:   55  pushl  %ebp
  8048531:   89 e5  movl  %esp,%ebp
  8048533:   e8 08 00 00 00  call  8048540 <a>
  8048538:   6a 00  pushl  $0x0
  804853a:   e8 35 ff ff ff  call  8048474 <_init+0x94>
  804853f:   90  nop

08048540 <a>:
  8048540:   55  pushl  %ebp
  8048541:   8b 15 1c a0 04  movl  0x804a01c,%edx
  8048546:   08
  8048547:   a1 20 a0 04 08  movl  0x804a020,%eax
  804854c:   89 e5  movl  %esp,%ebp
  804854e:   03 02  addl  (%edx),%eax
  8048550:   89 ec  movl  %ebp,%esp
  8048552:   03 05 d0 a3 04  addl  0x804a3d0,%eax
  8048557:   08
  8048558:   5d  popl  %ebp
  8048559:   c3  ret
Executable After Relocation and External Reference Resolution (.data)

m.c

int e=7;
int main() {
    int r = a();
    exit(0);
}

a.c

extern int e;
int *ep=&e;
int x=15;
int y;

int a() {
    return *ep+x+y;
}

Disassembly of section .data:

0804a018 <e>:
    804a018: 07 00 00 00

0804a01c <ep>:
    804a01c: 18 a0 04 08

0804a020 <x>:
    804a020: 0f 00 00 00
Strong and Weak Symbols

Program symbols are either strong or weak

- **strong**: procedures and initialized globals
- **weak**: uninitialized globals

```c
int foo = 5;
p1() {
}
```

```c
int foo;
p2() {
}
```
Linker’s Symbol Rules

Rule 1. A strong symbol can only appear once.

Rule 2. A weak symbol can be overridden by a strong symbol of the same name.
- references to the weak symbol resolve to the strong symbol.

Rule 3. If there are multiple weak symbols, the linker can pick an arbitrary one.
Linker Puzzles

```
int x;
p1() {}
```

```
int x;
p2() {}
```

```
int x;
int y;
p1() {}
p2() {}
```

```
int x=7;
int y=5;
p1() {}
p2() {}
```

```
int x=7;
p1() {}
```

```
int x;  
p2() {}
```
Linker Puzzles

Link time error: two strong symbols (p1)

References to x will refer to the same uninitialized int. Is this what you really want?

Writes to x in p2 might overwrite y! Evil!

Writes to x in p2 will overwrite y! Nasty!

References to x will refer to the same initialized variable.

Nightmare scenario: two identical weak structs, compiled by different compilers with different alignment rules.
Packaging Commonly Used Functions

How to package functions commonly used by programmers?
- Math, I/O, memory management, string manipulation, etc.

Awkward, given the linker framework so far:
- Option 1: Put all functions in a single source file
  - Programmers link big object file into their programs
  - Space and time inefficient
- Option 2: Put each function in a separate source file
  - Programmers explicitly link appropriate binaries into their programs
  - More efficient, but burdensome on the programmer

Solution: *static libraries* (\*.a archive files)
- Concatenate related relocatable object files into a single file with an index (called an archive).
- Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives.
- If an archive member file resolves reference, link into executable.
Static Libraries (archives)

Further improves modularity and efficiency by packaging commonly used functions [e.g., C standard library (\texttt{libc}), math library (\texttt{libm})]

Linker selects only the .\texttt{0} files in the archive that are actually needed by the program.
Creating Static Libraries

Archiver allows incremental updates:
- Recompile function that changes and replace .o file in archive.
Commonly Used Libraries

\texttt{libc.a} \textit{(the C standard library)}

- 8 MB archive of 900 object files.
- I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math

\texttt{libm.a} \textit{(the C math library)}

- 1 MB archive of 226 object files.
- Floating point math (\texttt{sin}, \texttt{cos}, \texttt{tan}, \texttt{log}, \texttt{exp}, \texttt{sqrt}, ...)

\begin{Verbatim}
\texttt{\% ar -t /usr/lib/libc.a | sort}
\end{Verbatim}

\begin{Verbatim}
... 
fork.o 
...
fprintf.o
fputc.o
fopen.o
fputc.o
fscanf.o
fseek.o
fstab.o
...
\end{Verbatim}

\begin{Verbatim}
\texttt{\% ar -t /usr/lib/libm.a | sort}
\end{Verbatim}

\begin{Verbatim}
... 
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
e_acoshl.o
e_acosl.o
e_asin.o
e_asinf.o
e_asinl.o
...
\end{Verbatim}
Using Static Libraries

Linker’s algorithm for resolving external references:

- Scan .o files and .a files in the command line order.
- During the scan, keep a list of the current unresolved references.
- As each new .o or .a file obj is encountered, try to resolve each unresolved reference in the list against the symbols in obj.
- If any entries in the unresolved list at end of scan, then error.

Problem:

- Command line order matters!
- Moral: put libraries at the end of the command line.

```
bass> gcc -L. libtest.o -lmine
bass> gcc -L. -lmine libtest.o
libtest.o: In function `main':
libtest.o(.text+0x4): undefined reference to `libfun'
```
Loading Executable Binaries

Executable object file for example program p

<table>
<thead>
<tr>
<th>Section</th>
<th>Virtual addr</th>
<th>Process image</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELF header</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program header table</td>
<td></td>
<td>init and shared lib</td>
</tr>
<tr>
<td>(required for executables)</td>
<td></td>
<td>segments</td>
</tr>
<tr>
<td>.text section</td>
<td>0x08048494</td>
<td>.text segment (r/o)</td>
</tr>
<tr>
<td>.data section</td>
<td>0x080483e0</td>
<td>.data segment (initialized r/w)</td>
</tr>
<tr>
<td>.bss section</td>
<td>0x0804a010</td>
<td>.bss segment (uninitialized r/w)</td>
</tr>
<tr>
<td>.symtab</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.rel.text</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.rel.data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.debug</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Section header table</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(required for relocatables)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Shared Libraries

Static libraries have the following disadvantages:

- Potential for duplicating lots of common code in the executable files on a filesystem.
  - e.g., every C program needs the standard C library
- Potential for duplicating lots of code in the virtual memory space of many processes.
- Minor bug fixes of system libraries require each application to explicitly relink

Solution:

- **Shared libraries** (dynamic link libraries, DLLs) whose members are dynamically loaded into memory and linked into an application at run-time.
  - Dynamic linking can occur when executable is first loaded and run.
    - Common case for Linux, handled automatically by `ld-linux.so`.
  - Dynamic linking can also occur after program has begun.
    - In Linux, this is done explicitly by user with `dlopen()`.
    - Basis for High-Performance Web Servers.
  - Shared library routines can be shared by multiple processes.
Dynamically Linked Shared Libraries

Translators (cc1, as)

m.c

m.o

Translators (cc1, as)

a.c

a.o

Linker (ld)

Libc.so

Loader/Dynamic Linker (ld-linux.so)

Partially linked executable p
(on disk)

Fully linked executable p’ (in memory)

Shared library of dynamically relocatable object files

_libc.so functions called by m.c and a.c are loaded, linked, and (potentially) shared among processes.
The Complete Picture

Translator

m.o

Translator

a.o

libwhatever.a

Static Linker (ld)

p

libc.so libm.so

Loader/Dynamic Linker
(ld-linux.so)

p'