Assemblers, Linkers, and Loaders

Kevin Walsh
CS 3410, Spring 2010
Computer Science
Cornell University

See: P&H Appendix B.3-4
#include <stdio.h>

int n = 100;

int main (int argc, char* argv[]) {
    int i, m = n, count = 0;
    for (i = 1; i <= m; i++) { count += i; }
    printf ("Sum 1 to %d is %d\n", n, count);
}

[csug01] mipsel-linux-gcc -S add1To100.c
Global variables in data segment
- Exist for all time, accessible to all functions

Dynamic variables in heap segment
- Exist between malloc() and free()

Function-Local variables in stack frame
- Exist solely for the duration of the stack frame

```c
int n = 100;
int main (int argc, char* argv[]) {
    int i, m = n, count = 0, *A = malloc(m*4);
    for (i = 1; i <= m; i++) { count += A[i-1] = i; }
    printf("Sum 1 to %d is %d\n", n, count);
}
```
```assembly
.data
.globl n
.align 2
n:
.word 100
.rdata
.align 2
$str0: .asciiz
    "Sum 1 to %d is %d\n"
.text
.align 2
.globl main
main: addiu $sp,$sp,-48
    sw $31,44($sp)
    sw $fp,40($sp)
    move $fp,$sp
    sw $4,48($fp)
    sw $5,52($fp)
    la $2,n
    lw $2,0($2)
    sw $2,28($fp)
    sw $0,32($fp)
    li $2,1
    sw $2,24($fp)

$L2:
    lw $2,24($fp)
    lw $3,28($fp)
    slt $2,$3,$2
    bne $2,$0,$L3
    lw $3,32($fp)
    lw $2,24($fp)
    addu $2,$3,$2
    sw $2,32($fp)
    lw $2,24($fp)
    addiu $2,$2,1
    sw $2,24($fp)
    b $L2

$L3:
    la $4,$str0
    lw $5,28($fp)
    lw $6,32($fp)
    jal printf
    move $sp,$fp
    lw $31,44($sp)
    lw $fp,40($sp)
    addiu $sp,$sp,48
    j $31
```
C Pointers can be trouble
Pointer to global variables are fine
Pointer to dynamic variables in heap segment
  int *bad()
  { s = malloc(20); ... free(s); ... return s; }
Pointer to function-local variables in stack frame
  int *trouble()
  { int a; ...; return &a; }
char *evil()
  { char s[20]; gets(s); return s; }

(Can’t do this in Java, C#, ...)

5
Assembler output is obj files

• Not executable
• May refer to external symbols
• Each object file has illusion of its own address space

Linker joins object files into one executable

Loader brings it into memory and starts execution
Assemblers and Compilers
Header
  • Size and position of pieces of file

Text Segment
  • instructions

Data Segment
  • static data (local/global vars, strings, constants)

Symbol Table
  • External (exported) references
  • Unresolved (imported) references

Debugging Information
  • line number → code address map, etc.
Global labels: External (exported) symbols
  • Can be referenced from other object files
  • Exported functions, global variables

Local labels: Internal (non-exported) symbols
  • Only used within this object file
  • static functions, static variables, loop labels, ...
```c
int pi = 3;
int e = 2;
static int randomval = 7;

extern char *username;
extern int printf(char *str, ...);

int square(int x) { ... }
static int is_prime(int x) { ... }
int pick_prime() { ... }
int pick_random() {
    return randomval;
}
```

gcc -S ... math.c
gcc -c ... math.s
objdump --disassemble math.o
objdump --syms math.o
csug01~$ mipsel-linux-objdump --disassemble math.o
math.o: file format elf32-tradlittlemips
Disassembly of section .text:

00000000 <pick_random>:
  0: 27bdfff8 addiu sp,sp,-8
  4: afbe0000 sw $s8,0(sp)
  8: 03a0f021 move $s8,sp
 c: 3c020000 lui $v0,0x0
10: 8c420008 lw $v0,8($v0)
14: 03c0e821 move $sp,$s8
18: 8fbe0000 lw $s8,0(sp)
1c: 27bd0008 addiu sp,sp,8
20: 03e00008 jr ra
24: 00000000 nop

00000028 <square>:
  28: 27bdfff8 addiu sp,sp,-8
  2c: afbe0000 sw $s8,0(sp)
  30: 03a0f021 move $s8,sp
  34: afc40008 sw $a0,8($s8)
  ...
<table>
<thead>
<tr>
<th>Symbol Address</th>
<th>Type</th>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>l</td>
<td>df <em>ABS</em></td>
<td>00000000 math.c</td>
</tr>
<tr>
<td>00000000</td>
<td>d</td>
<td>.text</td>
<td>00000000 .text</td>
</tr>
<tr>
<td>00000000</td>
<td>d</td>
<td>.data</td>
<td>00000000 .data</td>
</tr>
<tr>
<td>00000000</td>
<td>d</td>
<td>.bss</td>
<td>00000000 .bss</td>
</tr>
<tr>
<td>00000000</td>
<td>d</td>
<td>.mdebug.abi32</td>
<td>00000000 .mdebug.abi32</td>
</tr>
<tr>
<td>00000008</td>
<td>l</td>
<td>O .data</td>
<td>00000004 randomval</td>
</tr>
<tr>
<td>00000060</td>
<td>l</td>
<td>F .text</td>
<td>00000028 is_prime</td>
</tr>
<tr>
<td>00000000</td>
<td>d</td>
<td>.rodata</td>
<td>00000000 .rodata</td>
</tr>
<tr>
<td>00000000</td>
<td>d</td>
<td>.comment</td>
<td>00000000 .comment</td>
</tr>
<tr>
<td>00000000</td>
<td>g</td>
<td>O .data</td>
<td>00000004 pi</td>
</tr>
<tr>
<td>00000004</td>
<td>g</td>
<td>O .data</td>
<td>00000004 e</td>
</tr>
<tr>
<td>00000000</td>
<td>g</td>
<td>F .text</td>
<td>00000028 pick_random</td>
</tr>
<tr>
<td>00000028</td>
<td>g</td>
<td>F .text</td>
<td>00000038 square</td>
</tr>
<tr>
<td>00000088</td>
<td>g</td>
<td>F .text</td>
<td>0000004c pick_prime</td>
</tr>
<tr>
<td>00000000</td>
<td><em>UND</em></td>
<td></td>
<td>0000000000 username</td>
</tr>
<tr>
<td>00000000</td>
<td><em>UND</em></td>
<td></td>
<td>0000000000 printf</td>
</tr>
</tbody>
</table>
Q: Why separate compile/assemble and linking steps?
A: Can recompile one object, then just relink.
Linkers
Linker combines object files into an executable file

- Relocate each object’s text and data segments
- Resolve as-yet-unresolved symbols
- Record top-level entry point in executable file

End result: a program on disk, ready to execute
<table>
<thead>
<tr>
<th>main.o</th>
<th>math.o</th>
<th>calc.exe</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ 0C000000</td>
<td>→ 0C000000</td>
<td>→ 21032040</td>
</tr>
<tr>
<td>21035000</td>
<td>→ 1b301402</td>
<td>0C40023C</td>
</tr>
<tr>
<td>1b80050C</td>
<td>→ 3C040000</td>
<td>1b301402</td>
</tr>
<tr>
<td>→ 4C040000</td>
<td>→ 34040000</td>
<td>3C041000</td>
</tr>
<tr>
<td>21047002</td>
<td></td>
<td>34040004</td>
</tr>
<tr>
<td>→ 0C000000</td>
<td></td>
<td>0C40023C</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>21035000</td>
</tr>
<tr>
<td>00 T main</td>
<td>00 D pi</td>
<td>1b80050c</td>
</tr>
<tr>
<td>00 D uname</td>
<td><em>UND</em> printf</td>
<td>4C048004</td>
</tr>
<tr>
<td><em>UND</em> printf</td>
<td><em>UND</em> uname</td>
<td>21047002</td>
</tr>
<tr>
<td><em>UND</em> pi</td>
<td>28, JL, printf</td>
<td>0C400020</td>
</tr>
<tr>
<td>40, JL, printf</td>
<td>30, LUI, uname</td>
<td>...</td>
</tr>
<tr>
<td>4C, LW/gp, pi</td>
<td>34, LA, uname</td>
<td>10201000</td>
</tr>
<tr>
<td>54, JL, square</td>
<td></td>
<td>21040330</td>
</tr>
<tr>
<td></td>
<td></td>
<td>22500102</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0000000003</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0077616B</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>entry:400100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>text: 40000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>data:10000000</td>
</tr>
<tr>
<td>printf.o</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3C T printf</td>
<td></td>
</tr>
</tbody>
</table>
Header
  • location of main entry point (if any)

Text Segment
  • instructions

Data Segment
  • static data (local/global vars, strings, constants)

Relocation Information
  • Instructions and data that depend on actual addresses
  • Linker patches these bits after relocating segments

Symbol Table
  • Exported and imported references

Debugging Information
Unix
  • a.out
  • COFF: Common Object File Format
  • ELF: Executable and Linking Format
  • ...

Windows
  • PE: Portable Executable

All support both executable and object files
Loaders and Libraries
Loader reads executable from disk into memory

- Initializes registers, stack, arguments to first function
- Jumps to entry-point

Part of the Operating System (OS)
**Static Library:** Collection of object files
(think: like a zip archive)

Q: But every program contains entire library!
A: Linker picks only object files needed to resolve undefined references at link time

e.g. libc.a contains many objects:
- `printf.o`, `fprintf.o`, `vprintf.o`, `sprintf.o`, `snprintf.o`, ...
- `read.o`, `write.o`, `open.o`, `close.o`, `mkdir.o`, `readdir.o`, ...
- `rand.o`, `exit.o`, `sleep.o`, `time.o`, ....
Q: But every program still contains part of library!

A: shared libraries

• executable files all point to single *shared library* on disk
• final linking (and relocations) done by the loader

Optimizations:

• Library compiled at fixed address
  (makes linking trivial: few relocations needed)
• Jump table in each program
• Can even patch jumps on-the-fly
Direct call:

00400010 <main>:
  ...
  jal 0x00400330
  ...
  jal 0x00400620
  ...
  jal 0x00400330
  ...
00400330 <printf>:
  ...
00400620 <gets>:
  ...

Drawbacks:
Linker or loader must edit every use of a symbol (call site, global var use, ...)

Idea:
Put all symbols in a single "global offset table"
Code does lookup as needed
Direct call:

00400010 <main>:
  ...
  jal
  ...
  jal
  ...
  ...
  00400330 <printf>:
    ...
  00400620 <gets>:
    ...

GOT: global offset table
Indirect call:
00400010  <main>:
  ...
  lw  t9,  -32708(gp)
  jalr  t9
  ...
  lw  t9,  -32704(gp)
  jalr  t9
  ...
00400330  <printf>:
  ...
00400620  <gets>:
  ...

# data segment
  ...
  ...

# global offset table
# at -32712(gp)
.got
  .word  00400010
  .word  00400330
  .word  00400620
  ...

Indirect call with on-demand dynamic linking:

00400010 <main>:

... ...

lw t9, -32708(gp)
jalr t9
...

.got

.word 00400888
.word 00400888
.word 00400888
.word 00400888

00400888 <dlresolve>:
Indirect call with on-demand dynamic linking:

00400010 <main>:
  ...
  # load address of prints
  # from .got[1]
  lw t9, -32708(gp)
  # also load the index 1
  li t8, 1
  # now call it
  jalr t9
  ...

.got
  .word 00400888 # open
  .word 00400888 # prints
  .word 00400888 # gets
  .word 00400888 # foo

... 00400888 <dlresolve>:
  # t9 = 0x400888
  # t8 = index of func that
  # needs to be loaded
  ...
  # load that func
  ... # t7 = loadfromdisk(t8)
  # save func’s address so
  # so next call goes direct
  ...
  # got[t8] = t7
  # also jump to func
  jr t7
  # it will return directly
  # to main, not here
Windows: dynamically loaded library (DLL)
  • PE format

Unix: dynamic shared object (DSO)
  • ELF format

Unix also supports Position Independent Code (PIC)
  – Program determines its current address whenever needed (no absolute jumps!)
  – Local data: access via offset from current PC, etc.
  – External data: indirection through Global Offset Table (GOT)
  – ... which in turn is accessed via offset from current PC
Static linking

- Big executable files (all/most of needed libraries inside)
- Don’t benefit from updates to library
- No load-time linking

Dynamic linking

- Small executable files (just point to shared library)
- Library update benefits all programs that use it
- Load-time cost to do final linking
  - But dll code is probably already in memory
  - And can do the linking incrementally, on-demand