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);
}
```
.data
.globl  n
.align 2
.word  100
.rdata
.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)

.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#, ...)

The diagram illustrates the process of compiling and linking source code to produce an executable program.

1. **Compilation Stage**:
   - `calc.c` is compiled into an object file `calc.o`.
   - `math.c` is compiled into `math.o`.
   - `io.s` is compiled into `io.o`.
   - `libc.o` and `libm.o` are compiled into their respective object files.

2. **Linking Stage**:
   - The object files `calc.o`, `math.o`, `io.o`, `libc.o`, and `libm.o` are linked together by the linker to produce the executable file `calc.exe`.

3. **Loading Stage**:
   - The loader loads the `calc.exe` into memory, making it executable.

The diagram shows the flow of compilation and linking processes, with arrows indicating the progression from source files to object files to the final executable.
Assembler output is obj files
  • Not executable
  • May refer to external symbols
  • Each object file has illusion of its own address space

(\textbf{Linker joins object files into one executable})

(\textbf{Loader brings it into memory and starts execution})
Assemblers and Compilers
Object File

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, ...
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;
}
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)

...
csug01 ~$ mipsel-linux-objdump --syms math.o
math.o:  file format elf32-tradlittlemips

SYMBOL TABLE:
00000000 1 df *ABS* 00000000  math.c
00000000 1 d  .text 00000000  .text
00000000 1 d  .data 00000000  .data
00000000 1 d  .bss 00000000  .bss
00000000 1 d  .mdebug.abi32 00000000  .mdebug.abi32
00000008 1 O  .data 00000004  randomval
00000060 1 F  .text 00000028  is_prime
00000000 1 d  .rodata 00000000  .rodata
00000000 1 d  .comment 00000000  .comment
00000000 4 g  O  .data 00000004  pi
00000004 4 g  O  .data 00000004  e
00000000 4 g  F  .text 00000028  pick_random
00000028 4 g  F  .text 00000038  square
00000088 4 g  F  .text 0000004c  pick_prime
00000000  *UND*
00000000  *UND*
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
main.o

math.o

calc.exe

printf.o

---

entry: 400100
text: 400000
data: 1000000

---
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  
  ...  
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 t9

...
jal...
jal...

00400330 <printf>:

...

00400620 <gets>:

...
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

... 00400888 <dlresolve>:

  t9 is entry
  - xlate to name
  - load (name)
  fix got w/ new name.
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
```

... <dlresolve>:

```
... # 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