Lecture 18. Elementary Systems Programming

- ***Software tools*** **are programs that** ***manipulate programs*****, each in potentially different** ***languages***

	Name	Input	Output
Macro preprocessor	`cpp`	C	C
Compiler	`rcc`	C	assembly code
Assembler	`as`	assembly code	object code
Linker	`ld -r`	object code, libraries	object code
Loader	`ld`	object code	executable code
Operating system	UNIX	executable code	

- **'Driver' programs, like `lcc`, hide many of these steps**

```
% lcc -v hello.c
/usr/local/lcc/lib/cpp ... hello.c hello.i
/usr/local/lcc/lib/rcc -target=sparc-solaris hello.i hello.s
/bin/as -o hello.o hello.s
/bin/ld -o a.out ... hello.o -lm -lc
% a.out
Hello world!
```

Compilation Pipeline

```
% cat hello.c
/* Everyone's first
    C program. */
#include <stdio.h>

int main(void) {
    printf("Hello world!\n");
    return 0;
}
```

- **The macro *preprocessor* strips comments, expands macro definitions, processes conditional compilation directives, and injects include files**

```
% lcc -E hello.c >hello.i; cat hello.i
#line 1 "hello.c"
...
#line 1 "/usr/local/lib/lcc/include/stdio.h"
...
extern int printf(const char *, ...);
...
#line 4 "hello.c"

int main(void) {
    printf("Hello world!\n");
    return 0;
}
```

See Chap. 13 in Deitel and Deitel for details

Compilation and Assembly

- **The *compiler* translates C to symbolic assembly language, alà symbolic TOY instructions**

```
% lcc -S hello.i; cat hello.s
...
_main:
save %sp,-96,%sp
set L2,%o0
call _printf; nop
mov %g0,%i0
L1:
ret; restore
L2: ...
```

- **The *assembler* translates symbolic assembly language to relocatable object code, alà TOY instruction encodings**

```
% lcc -c hello.s; dis hello.o
0:  9d e3 bf a0      save    %sp, -96, %sp
4:  11 00 00 00      sethi   %hi(printf), %o0
8:  90 12 20 00      or      %o0, printf, %o0
c:  40 00 00 00      call    0xc
10: 01 00 00 00      nop
14: b0 10 00 00      clr     %i0
18: 81 c7 e0 08      ret
1c: 81 e8 00 00      restore
```

Linking

- **The *linker* combines object code files and libraries in a new object code file**

```
% ld -r -o foo.o hello.o -lc; dis foo.o
main     0:  9d e3 bf a0      save    %sp, -96, %sp
         4:  11 00 00 00      sethi   %hi(.L350), %o0
         8:  90 12 20 00      or      %o0, .L350, %o0
         c:  40 00 00 00      call    (.L350+12)
         10: 01 00 00 00      nop
         14: b0 10 00 00      clr     %i0
         18: 81 c7 e0 08      ret
         1c: 81 e8 00 00      restore
printf   20: 9d e3 bf a0      save    %sp, -96, %sp
         24: 15 00 00 00      sethi   %hi(0x0), %o2
         28: f2 27 a0 48      st      %i1, [%fp + 72]
         2c: d4 02 a0 00      ld      [%o2], %o2
         30: 11 00 00 00      sethi   %hi(0x0), %o0
         34: f4 27 a0 4c      st      %i2, [%fp + 76]
...
```

Loading

- **The *loader* translates object code to executable code**

```
% ld foo.o; dis a.out
...
15148:   9d e3 bf a0      save    %sp, -96, %sp
1514c:   11 00 00 8a      sethi   %hi(0x22800), %o0
15150:   90 12 22 2c      or      %o0, 0x22c, %o0
15154:   40 00 00 05      call    0x15168
15158:   01 00 00 00      nop
1515c:   b0 10 00 00      clr     %i0
15160:   81 c7 e0 08      ret
15164:   81 e8 00 00      restore
15168:   9d e3 bf a0      save    %sp, -96, %sp
1516c:   13 00 00 e8      sethi   %hi(0x3a000), %o1
15170:   f2 27 a0 48      st      %i1, [%fp + 72]
15174:   d2 0a 62 b4      ldub    [%o1 + 692], %o1
15178:   f4 27 a0 4c      st      %i2, [%fp + 76]
...
```

- **The operating system loads the executable code into memory and jumps to it**

```
% a.out
Hello world!
```

Assembly Language

- An ***assembly language*** is a ***symbolic representation*** for machine language

  Mnemonic names for opcodes and registers; usually terse

  Symbolic names for addresses — data locations and jump 'targets'

  Easy to delete, insert, and rearrange instructions

- TAL: *T*OY *A*ssembly *L*anguage

HALT		halt	
ADD	R,R_1,R_2	add	R ← R_1 + R_2
SUB	R,R_1,R_2	subtract	R ← R_1 − R_2
MUL	R,R_1,R_2	multiply	R ← R_1 × R_2
XOR	R,R_1,R_2	exclusive OR	R ← R_1 ^ R_2
AND	R,R_1,R_2	logical AND	R ← R_1 & R_2
SHL	R,R_1,R_2	shift left	R ← R_1 << R_2
SHR	R,R_1,R_2	shift right	R ← R_1 >> R_2
LI	R,*const8*	load immediate	R ← *const8* (8-bit constant)
LD	R,(R_1 + *const8*)	load	R ← M[R_1 + *const8*]
ST	R,(R_1 + *const8*)	store	M[R_1 + *const8*] ← R
SYS	R,*const8*	system call	system call *const8*, arg in R
J	*label*	jump	PC ← *label*
JLT	R,*label*	jump if less	PC ← *label* if R < 0
JI	(R)	jump indirect	PC ← R
JAL	R,*label*	jump and link	R ← PC, PC ← *label*

Programming in TAL

`power.t`: (see page 9-6)

```
POWER   LI   R4,1          initialize R4
        LI   R3,1          initialize z
LOOP    SUB  R2,R2,R4      decrement exponent
        JLT  R2,DONE       quit when done
        MUL  R3,R3,R1      set z to z*x
        J    LOOP          do it again
DONE    JI   (R5)          return to caller
```

`main.t`: computes $A^4 + B^5$ (see page 9-7)

```
MAIN    LI   R0,0          initialize R0
        LI   R1,A          load A into R1
        LD   R1,(R1+0)
        LI   R2,4          want A^4
        JAL  R5,POWER      call POWER
        ADD  R6,R3,R0      copy A^4 to R6
        LI   R1,B          do it again for B^5
        LD   (R1+0)
        LI   R2,5
        JAL  R5,POWER
        ADD  R6,R6,R3      R6 now holds A^4 + B^5
        SYS  R6,2          print it
        HALT
A       3
B       2
```

Object Code

- **The assembler reads TAL and emits *relocatable object code***

  `power.o`:

```
00:     B401    =POWER          initialize R4
01:     B301                    initialize z
02:     2224                    decrement exponent
03:     6206    +start address  quit when done
04:     3331                    set z to z*x
05:     5002    +start address  do it again
06:     7500                    return to caller
```

- **Relocation information tells the *linker***

  The definitions of symbols

  How to adjust jump targets relative to the ultimate *starting address* of the module

  Which symbols are defined in other *separately compiled modules*

- **Object code is usually a compact, binary format, not text as suggested above**

Object Code, cont'd

`main.o`:

```
00:     B100    =MAIN           initialize R0
01:     B100    +A              load A into R1
02:     9110
03:     B204                    want A^4
04:     8500    +POWER          call POWER
05:     1630                    copy A^4 to R6
06:     B100    +B              do it again for B^5
07:     9110
08:     B205
09:     8500    +POWER          call POWER
0A:     1663                    R6 now holds A^4 + B^5
0B:     4602                    print it
0C:     0000
0D:     0003    =A
0E:     0002    =B
```

- **Assemblers maintain symbol tables: Sets of (symbol,value) pairs used to map**

  Mnemonics to values **`LI`** $\rightarrow B_{16}$, **`R6`** $\rightarrow 6$, …
  Labels to offsets **`LOOP`** $\rightarrow 2_{16}$, **`DONE`** $\rightarrow 6_{16}$, **`POWER`** $\rightarrow 0$, **`A`** $\rightarrow 0D_{16}$, …

  `power.o` symbol table: (**`POWER`**, 0)

  `main.o` symbol table: (**`MAIN`**, 0) (A, $0D_{16}$) (**`POWER`**, ?) (B, $0E_{16}$)

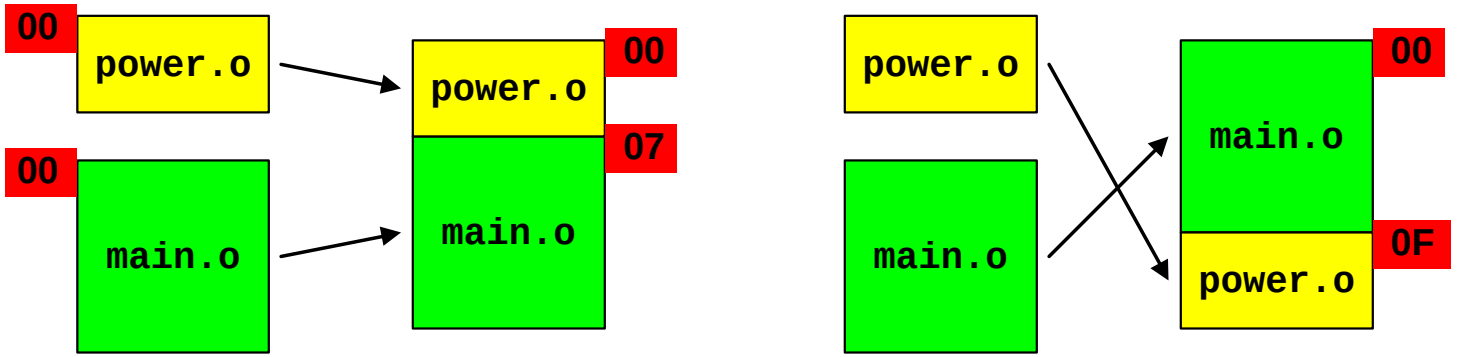
  Can implement symbol tables with binary trees

Linking

- The linker reads several object files and emits ***one relocatable object code*** file

  Concatenates object code from input files

  Relocates symbol definitions and instructions based on starting addresses in output object code



  Resolves references to undefined symbols

  Merges symbol tables

Linking, cont'd

- **Linking `main.o` and `power.o` resolves references to `POWER`, `A`, `B`, and adjusts offsets to jump targets**

```
00:     B100    =MAIN               initialize R0
01:     B10D    +start address      load A into R1
02:     9110
03:     B204                        want A^4
04:     850F    +start address      call POWER
05:     1630                        copy A^4 to R6
06:     B10E    +start address      do it again for B^5
07:     9110
08:     B205
09:     850F    +start address      call POWER
0A:     1663                        R6 now holds A^4 + B^5
0B:     4602                        print it
0C:     0000
0D:     0003    =A
0E:     0002    =B

0F:     B401    =POWER              initialize R4
10:     B301                        initialize z
11:     2224                        decrement exponent
12:     6215    +start address      quit when done
13:     3331                        set z to z*x
14:     5011    +start address      do it again
15:     7500                        return to caller
```

Output includes relocation information for additional linking

Loading

- **The loader reads a *starting address* and object code with *no undefined symbols* and emits executable code, adding in the starting address where necessary**

```
          20
20:       B100          initialize R0
21:       B12D          load A into R1
22:       9110
23:       B204          want A4
24:       852F          call POWER
25:       1630          copy A^4 to R6
26:       B12E          do it again for B^5
27:       9110
28:       B205
29:       852F          call POWER
2A:       1663          R6 now holds A^4 + B^5
2B:       4602          print it
2C:       0000
2D:       0003
2E:       0002

2F:       B401          initialize R4
30:       B301          initialize z
31:       2224          decrement exponent
32:       6235          quit when done
33:       3331          set z to z*x
34:       5031          do it again
35:       7500          return to caller
```

Separate Compilation

- A ***program*** is made up of many small ***modules***

  A 'few' application-specific modules

  'Many' general-purpose modules, e.g., standard I/O functions like `printf`

- Compile general-purpose modules ***separately***, collect their object code in ***libraries***
- Compile application-specific modules separately, keep their object code
- To build a program

  1. Link together the application-specific object code modules
  2. Search the libraries for the general-purpose modules used by (1)

- Advantages

  Avoid recompiling infrequently changed modules

  Share libraries of well-tested general-purpose modules — don't reinvent, reuse

- Designing and implementing general-purpose modules sounds easy, but it's ***not***

  Take COS 217, Introduction to Programming Systems

  Read D. R. Hanson, *C Interfaces and Implementations: Techniques for Creating Reusable Software*, Addison-Wesley, 1997 (used in COS 217)