Assemblers

Sandhya Dwarkadas

Assembler

Translation takes 2 steps:

1. associate memory locations with labels
2. translate each statement by combining the numeric representation of
   - the opcode,
   - the register specifiers, and
   - the label/immediate values

Why? - Forward References

The label is used before it is defined

```
        bne     $4, $8, L1
...
L1:     addu    $4, $4, 5
```

When the assembler sees `bne` the first time, it doesn't know `L1`'s location.

Two solutions — time versus space trade-off

1. 2 step translation
2. *backpatching* — maintain a table of forward references; update each reference (instruction) when the label is defined.

Step 1

Record the name and position of each label in the *symbol table*.

To determine the position, the assembler must determine how many words each instruction or data declaration occupies.

- position is relative from the start of text, data, or bss.
- variable length instructions or data declarations complicate the calculation.
  - SPARC/MIPS instructions are easy, but data declarations such as `.ascii` are harder.

Symbol Table

Often a hash table

Each entry contains

- the string representing the symbol
- the segment (and a bit for private versus global)
  - one of undefined, absolute, text, data, or bss
- the offset from the start of the segment

Step 2

Produce machine code using an *opcode table*.

The opcode table describes how to combine

- the opcode,
- the register specifiers, and
- the label/immediate values

into the representation for the instruction or data declaration.

References (to external symbols in another file) are unresolved.

The symbol table is appended to the object file for use by the linker and debugger.

Opcode Table

For a simple assembler

```
int size_align();
int size_ascii();
int size_1();
int size_4();

void output_align();
void output_ascii();
void output_byte();
void output_instr_3opnd();

struct  Opcode  opcodes[] = {
  { ".align", 0x00000000, size_align, output_align },
  { ".ascii", 0x00000000, size_ascii, output_ascii },
  { ".byte",  0x00000000, size_1,     output_byte },
  { "save",   0x81E00000, size_4,     output_instr_3opnd },
  { "sub",    0x80200000, size_4,     output_instr_3opnd },
```

Object File Format

Six distinct sections

1. *object file header* – describes the size and position of the other pieces of the file
2. *text segment*
3. *data segment*
4. *relocation information* – identifies instructions and data words that depend on absolute addresses
5. *symbol table* – see `/usr/include/nlist.h`
6. *debugging information*

Object File Header

The first 32 bytes of every object file

`/usr/include/sys/exechdr.h`

```
/*
 * format of the exec header
 * known by kernel and by user programs
 */
struct exec {
   unsigned long a_magic;  /* magic number */
   unsigned long a_text;   /* size of text segment */
   unsigned long a_data;   /* size of initialized data */
   unsigned long a_bss;    /* size of uninitialized data */
   unsigned long a_syms;   /* size of symbol table */
   unsigned long a_entry;  /* entry point */
   unsigned long a_trsize; /* size of text relocation */
   unsigned long a_drsize; /* size of data relocation */
};
```

Linker/Separate Compilation

Search libraries to find undefined labels

Determine memory locations for each module and relocate instructions by adjusting absolute addresses

Resolve references among files - external reference to unresolved reference with the same name

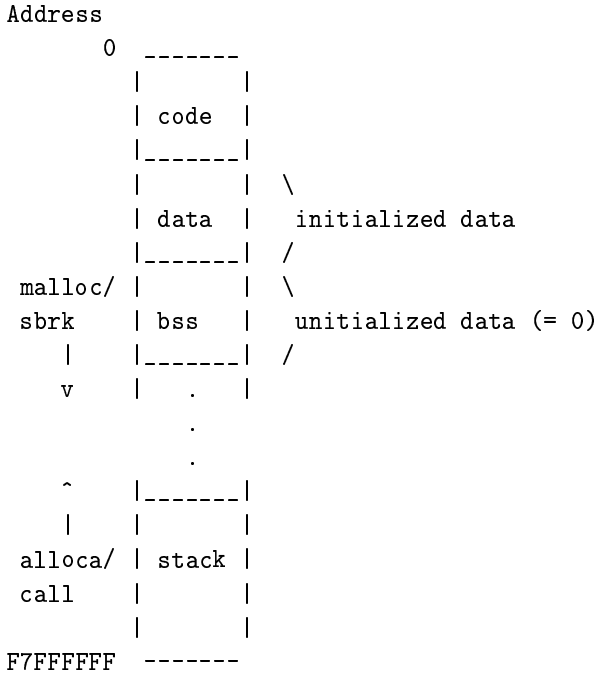
Loader

Load file from disk/secondary storage

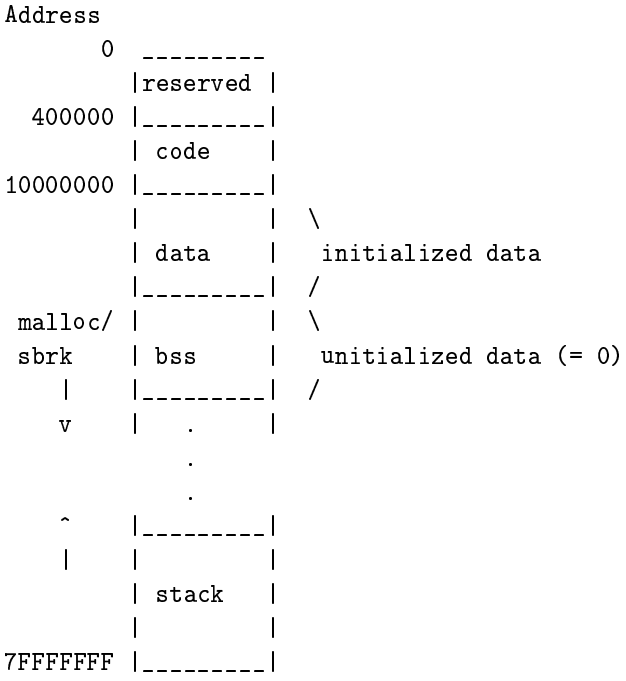
- Read header for size of text and data segments
- Create new address space - text, data, stack
- Copy instrs/data from file into new address space (memory)
- Copy arguments to program onto stack
- Initialize machine registers/stack pointer
- Jump to startup routine
  - copy program arguments from stack to registers
  - call program's main routine
  - on return, terminate program with exit system call

Rules not imposed by hardware but by software for interoperability

SPARC/Unix Memory Layout



MIPS Memory Layout



Tools

There are a variety of tools for interpreting object files.

- `od` — displays the contents of *any* file
- `nm` — displays the symbol table information appended to an object file
- `int nlist(char *filename, struct nlist *nl)`

Example

```
example% nm -n water.o
         U _CNSTNT
000002dc T _main
000004b0 t _twin_alloc
00000620 B _TLC
000009e8 D _NFRST
```