Chapter 5: **Programs as Data**
- Programs that manipulate programs
- Assemblers, compilers
- Linking, name binding
- Relocation

**Olav Beckmann**
Huxley 449
http://www.doc.ic.ac.uk/~ob3

Acknowledgements: There are lots. See end of Chapter 1.

**Home Page for the course:**

This is only up-to-date after I have issued printed version of the notes, tutorials, solutions etc.
Chapter 5: Programs as Data

The purpose of this chapter:

• Assemblers and the assembly language representation of machine code
• Compiling high-level language to machine code
• How names (such as variable names) are bound to concrete values
• Linking separate program components; name binding
• What has to happen when a program is loaded
• Textbook: Nutt page 120-121 and 419-429
Programming the NARC

• You program a computer by putting just the right numbers into its memory
• This means that early programmers had to be incredibly clever
• Programming has been de-skilled
• Some simple tools mean that anyone can do it 😊
• Key tools:
  • Assembler
  • Compiler
  • Linker
Compiler, Assembler, Linker...

• The funny thing is that these three words all seem to mean the same thing

• In computing, all definitions are flexible, but they have come to refer to the following structure:

  • **Assembler**: translates human-readable versions of machine instructions into the machine encoding, ready for direct interpretation by the processor
  
  • **Compiler**: translates a high-level language (C, C++, etc) into machine instructions
  
  • **Linker**: combines chunks of machine instructions (e.g. separately purchased software) together
Assembler Source

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>.word 23</td>
</tr>
<tr>
<td>B</td>
<td>.word 45</td>
</tr>
<tr>
<td>C</td>
<td>.word 0</td>
</tr>
<tr>
<td>main</td>
<td>loadm A</td>
</tr>
<tr>
<td></td>
<td>jmpz end</td>
</tr>
<tr>
<td></td>
<td>loadm C</td>
</tr>
<tr>
<td></td>
<td>addm B</td>
</tr>
<tr>
<td></td>
<td>storem C</td>
</tr>
<tr>
<td></td>
<td>loadm A</td>
</tr>
<tr>
<td></td>
<td>subc 1</td>
</tr>
<tr>
<td></td>
<td>storem A</td>
</tr>
<tr>
<td>end</td>
<td>jmp main</td>
</tr>
</tbody>
</table>

Object Code

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 2 9 ? 2 2 5 1 3 2 2 0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2 0 0 6 1 3 0 8 3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3 1 2 3 0 4 4 5 5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4 2 6 7 8 9 10 11 12</td>
<td></td>
</tr>
</tbody>
</table>

Textual representation, one line per instruction

Instructions and data, encoded in binary, ready to be operated upon directly by the processor
As each line of the assembler source is translated, it is assigned an address - so we know the numeric value of ‘A’, ’B’, ’C’ - but not ‘end’.

Labels in the assembler source - ‘A’, ‘B’, ‘C’, ‘main’, ‘end’ - represent numbers which have to refer to the right address wherever it might turn out to be.
Two Pass Assembler

Pass 1 - build up Symbol Table

Symbol Table

<table>
<thead>
<tr>
<th>Label</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>main</td>
<td>3</td>
</tr>
<tr>
<td>end</td>
<td>12</td>
</tr>
</tbody>
</table>

Pass 1

A .word 23
B .word 45
C .word 0
main loadm A
    jmpz end
    loadm C
    addm B
    storem C
    loadm A
    subc 1
    storem A
    jmp main
end
Two Pass Assembler

Pass 2 - output Object Code

Symbol Table

<table>
<thead>
<tr>
<th>Label</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>main</td>
<td>3</td>
</tr>
<tr>
<td>end</td>
<td>12</td>
</tr>
</tbody>
</table>

Pass 1

A .word 23
B .word 45
C .word 0
main loadm A
jmpz end
loadm C
addm B
storem C
loadm A
subc 1
storem A
jmp main
end
Assembler - Summary

- Assembler input language: textual representation of each machine instruction, one line per instruction
- Assembler language includes “directives” to tell assembler to assign symbolic names (“labels”)
- Also directives to name and set aside working storage (variables)
- Assembler typically operates in two passes:
  - pass 1: calculate space occupied, build symbol table
  - pass 2: reprocess source using symbol table to fill in symbol values
Binding: resolving names

- Two pass assembler:
  - pass 1: calculate space occupied, build symbol table
  - pass 2: reprocess source using symbol table to fill in symbol values

- Two functions:
  - translate human-readable representation to machine encoding
  - resolve references to names

- This issue of naming and “binding” of names to values is a recurring theme in operating systems
The Job of the Linker

• Suppose we want our new program to use the functionality provided by another program
• E.g. newly-purchased word processor needs to access software driver for new keyboard
• Simpler working example: ‘main’ program uses separately-provided ‘incr’ procedure (next slide)
• Two issues:
  • **relocation**: concatenate the binary code - and adjust symbol references according to new addresses
  • **name binding**: resolve names used but undefined in ‘main’ with names defined in ‘incr’
The Linker

1: Concatenate

2: Relocate

3: Resolve names

T means “Text”, referring to program text, rather than data.
Suppose a program has been assembled starting at address 0. Suppose it is now loaded into store starting at address 100.

**Need to *relocate* absolute address operands**

Relocation
- add 100 to absolute addresses
- some operand fields have to change (e.g. 1 to 101)

Note: NARC uses only absolute addressing
Relative addresses (e.g. PC relative jumps) do not have to be relocated.
Which locations will need to be adjusted? Object code file must record this information.

<table>
<thead>
<tr>
<th>A</th>
<th>.word 23</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>.word 45</td>
</tr>
<tr>
<td>C</td>
<td>.word 0</td>
</tr>
<tr>
<td>main</td>
<td>loadm A</td>
</tr>
<tr>
<td>jmpz</td>
<td>end</td>
</tr>
<tr>
<td>loadm</td>
<td>C</td>
</tr>
<tr>
<td>addm</td>
<td>B</td>
</tr>
<tr>
<td>storem</td>
<td>C</td>
</tr>
<tr>
<td>loadm</td>
<td>A</td>
</tr>
<tr>
<td>subc</td>
<td>1</td>
</tr>
<tr>
<td>storem</td>
<td>A</td>
</tr>
<tr>
<td>jmp</td>
<td>main</td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Loc</th>
<th>Value</th>
<th>Relocate?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>23</td>
<td>n</td>
</tr>
<tr>
<td>1</td>
<td>45</td>
<td>n</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>n</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© Department of Computing, Imperial College London. Operating Systems Concepts (MSc Computing / JMC2) 2005/06 – Chapter 5
## Name Binding: External Symbol Table

### T = Code Label

<table>
<thead>
<tr>
<th>Loc</th>
<th>Value</th>
<th>Rel?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>addc</td>
<td>n</td>
</tr>
<tr>
<td>1</td>
<td>ret</td>
<td>n</td>
</tr>
</tbody>
</table>

### D = Data Label

<table>
<thead>
<tr>
<th>Loc</th>
<th>Value</th>
<th>Rel?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>n</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>n</td>
</tr>
<tr>
<td>2</td>
<td>loadc</td>
<td>n</td>
</tr>
<tr>
<td>3</td>
<td>call</td>
<td>y</td>
</tr>
<tr>
<td>4</td>
<td>storem</td>
<td>y</td>
</tr>
<tr>
<td>5</td>
<td>loadc</td>
<td>n</td>
</tr>
<tr>
<td>6</td>
<td>call</td>
<td>y</td>
</tr>
<tr>
<td>7</td>
<td>storem</td>
<td>y</td>
</tr>
<tr>
<td>8</td>
<td>ret</td>
<td>n</td>
</tr>
</tbody>
</table>

### U = Undefined

<table>
<thead>
<tr>
<th>Loc</th>
<th>Value</th>
<th>Rel?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>incr</td>
<td>3</td>
</tr>
</tbody>
</table>

The symbol table entry for an undefined label points to a linked list of entries which use that label.

© Department of Computing, Imperial College London. Operating Systems Concepts (MSc Computing / IM02) 2005/06 – Chapter 5
### External Symbol Table

<table>
<thead>
<tr>
<th>Loc</th>
<th>Value</th>
<th>Rel?</th>
</tr>
</thead>
<tbody>
<tr>
<td>incr</td>
<td>addc</td>
<td>1</td>
</tr>
<tr>
<td>incr</td>
<td>ret</td>
<td>n</td>
</tr>
</tbody>
</table>

### Linker

#### incr.o
- **Loc**: incr
- **Value**: T
- **Rel?**: 0

#### main.o
- **Loc**: main
- **Value**: T
- **Rel?**: 2

- **Loc**: incr
- **Value**: T
- **Rel?**: 3

### prog

<table>
<thead>
<tr>
<th>Loc</th>
<th>Value</th>
<th>Rel?</th>
</tr>
</thead>
<tbody>
<tr>
<td>incr</td>
<td>addc</td>
<td>1</td>
</tr>
<tr>
<td>incr</td>
<td>ret</td>
<td>n</td>
</tr>
<tr>
<td>main</td>
<td>0</td>
<td>n</td>
</tr>
<tr>
<td>main</td>
<td>0</td>
<td>n</td>
</tr>
<tr>
<td>main</td>
<td>loadc</td>
<td>23</td>
</tr>
<tr>
<td>main</td>
<td>call</td>
<td>0</td>
</tr>
<tr>
<td>main</td>
<td>storem</td>
<td>2</td>
</tr>
<tr>
<td>main</td>
<td>loadc</td>
<td>45</td>
</tr>
<tr>
<td>main</td>
<td>call</td>
<td>0</td>
</tr>
<tr>
<td>main</td>
<td>storem</td>
<td>3</td>
</tr>
<tr>
<td>main</td>
<td>ret</td>
<td>n</td>
</tr>
</tbody>
</table>

### External Symbol Table

| main | T | 4 |
| incr | T | 0 |
File Format for Machine Code

• Programs and program components are stored in files
• We need a general-purpose “object code file” format which can represent them
• Need to be able to include:
  - machine code instructions
  - values for initialised data
  - details of uninitialised data space required
  - names defined
  - names used but undefined
  - relocation information
  - entry point

• An incomplete program doesn’t have an entry point
• A complete program has no names used but undefined
• May include further information to aid in debugging
Example - COFF

• Common Object File Format (COFF)
• Used in some Unix systems, basis for Windows Portable Executable format [http://msdn.microsoft.com/library/specs/msdn_pecoff.htm](http://msdn.microsoft.com/library/specs/msdn_pecoff.htm), widely used for embedded systems
• Linux uses ELF (Executable and Linking Format) instead
• To decode an ELF file try using the ‘objdump’ or ‘readelf’ command (eg “readelf -a /bin/echo”)
Object file format - variations

- When all object files which form a program have been linked, all external references will be resolved and the global symbol table can be discarded.

- However, relocation information must be kept if the program is to be loaded at a different address to that for which it was linked.

- If relocation information is discarded (i.e. loader does not relocate) the program object file is an exact binary image of its representation in main store.

- The object file usually records the program entry point for use by the OS when it starts execution of the program - in C/C++ this is the “main” function.
Loading a program

• A key function of an OS is to load a user’s program and run it
• The program is delivered as an object file, e.g. in COFF format
• Where should the newly-loaded program be put - at which range of memory addresses?
Program Loading - Memory Management

When machine is turned on, PC=0 so it starts execution here

Address of first free word:

Operating system’s instructions and data structures

Free

Memory address: 0

Memory address: max
Program Loading - Memory Management

When machine is turned on, PC=0 so it starts execution here

Memory address: 0

Start address of game:

Address of first free word:

Memory address: max

Operating system’s instructions and data structures

DeathCopter video game

Free

Once game has been loaded, OS jumps to its start address
Program Loading - Memory Management

When machine is turned on, PC=0 so it starts execution here

Memory address: 0

Start address of game:

Start address of player:

Operating system’s instructions and data structures

DeathCopter video game

MP3 player

Free

What if the MP3 player had been loaded first?

Address of first free word:

Memory address: max
### Program Loading - Memory Management

<table>
<thead>
<tr>
<th>Memory address: 0</th>
<th>Operating system’s instructions and data structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start address of game:</td>
<td>MP3 player</td>
</tr>
<tr>
<td>Start address of player:</td>
<td>DeathCopter video game</td>
</tr>
<tr>
<td>Address of first free word:</td>
<td>Free</td>
</tr>
</tbody>
</table>

If the order of program loading had been different, the start addresses would be different.
The figure illustrates a relocating loader, which is a mechanism used in operating systems to load programs into memory at arbitrary addresses. The figure consists of two main parts:

1. **prog** (Program): This part shows the program code and its layout in memory. The program contains instructions for loading and storing values. The table displays the location (Loc), value (Value), and whether the location is relocatable (Rel?). For example, at location 0, the instruction is `addc`, and it is not relocatable (n). The entry point of the program is set at location 4.

2. **memory**: This part shows the memory layout with addresses ranging from 0 to 110. Each row represents a memory location with the instruction type, the instruction address, and its value. For instance, at address 100, the instruction is `addc` with a value of 1.

The relocating loader allows the program to be loaded into memory at any desired location, which is particularly useful for relocating programs in the memory when there are multiple programs running simultaneously.
Note two stages of relocation:
- at link time and load time

```
main.s main.o prog
```

Assembler  Linker  Loader

```
B .word 0
storem B

0

1

storem 1

0

3

storem 3

0

100

103

storem 103
```
Linking, Loading and Relocation - Summary

• Assembler translates human-readable representation of instructions into object file:
  – object file includes list of external names defined, and names used but not defined
  – object file also includes relocation information

• Linker combines object files:
  – concatenate, relocate so internal name references OK
  – resolve name binding

• Loader finds sufficiently-large free memory region, interprets object file format, loads object file instructions and initialised data into memory
  – relocates so that internal name references are right
Library files - constructed by Archiver system utility

- The library file is constructed from a set of object files.
- The Linker only extracts those object files necessary to resolve external references for `main`.
- Typically libraries contain i/o and maths routines.
Using Textbooks

- Nutt and Stallings cover assemblers, compilers and linkers only very briefly
- Eg Nutt page 120-121 and 419-429
- Terminology:
  - Linker = linkage editor
  - External symbol table = external reference table + external definition table
  - In unix/linux, the assembler is “as” and the linker is “ld”
In Real Life...

- This chapter has presented a simplified view
- With **dynamic linking** (Windows DLLs, Linux shared libraries) an object file is loaded during the program’s execution
  - Relies on position-independent code
  - Name binding via a table which maps each external reference to its run-time address
- With **just-in-time compilation** (eg Java JIT) object files are represented as machine-independent “bytecode”, which is translated to machine code as it is loaded
In real life... relocation by address translation

- Relocation is not needed if code is “position-independent”
- Relocation is also not needed if the processor has an address translation mechanism
- We will cover this in detail later in this course
- The basic idea is that there is a hardware lookup table that intercepts and translates addresses issued by the processor
- This has to be inactive when executing operating system code - which has to be able to set it up