12 Introduction to Storage Allocation

Readings for this topic: The a.out (5) section of the Unix manual (type *man 5 a.out*).

Information stored in memory is used in many different ways. Some possible classifications are:

- Role in Programming Language:
  - Instructions (specify the operations to be performed and the operands to use in the operations).
  - Variables (the information that changes as the program runs: locals, owns, globals, parameters, dynamic storage).
  - Constants (information that is used as operands, but that never changes: pi for example).
- Changeability:
  - Read-only (code, constants).
  - Read & write (variables).

  Why is this important? (no need for disk write-back, sharing).
- Addresses vs. Data. Why is this important? (must modify addresses if memory is re-arranged).
- Binding time (when is its space allocated):
  - Static : arrangement determined once and for all, before the program starts running. May happen at *compile-time*, *link-time*, or *load-time*.
  - Dynamic : arrangement cannot be determined until runtime, and may change.

  Note that the classifications overlap: variables may be static or dynamic, code may be read-only or read&write, etc.

When a process is running, what does its memory look like? It's divided up into areas called *segments*. In Unix, each process has three segments:

- Code (called "text" in Unix terminology)
- Data
- Stack

Why distinguish between different segments of memory? Suppose each process only had one segment: couldn't separate read-only code from read-write data.

In some systems, can have many different kinds of segments.

Division of responsibility between various portions of system:

- Compiler : generates one *object file* for each source code file containing information for that file. Information is incomplete, since each source file generally references some things defined in other source files.
- Linker : combines all of the object files for one program into a single object file, which is complete and self-sufficient.
- Operating system : loads object files into memory, allows several different processes to share memory at once, provides facilities for processes to get more memory after they've started running.
- Run-time library : works together with OS to provide dynamic allocation routines, such as *calloc* and *free* in C.

Linkers (or Linkage Editors, "ld" in Unix): tie together many separate pieces of a program, re-organize storage allocation. Often considered part of OS. In Unix, ld is hidden by cc.

Three functions of a linker:

- Collect all the pieces of a program.
- Figure out a new memory organization so that all the pieces fit together (combine like segments).
- Touch up addresses so that the program can run under the new memory organization.

The result is a runnable program stored in a new object file.

Problems linker must solve:

- Compiler doesn't know where the things it's compiling will go in memory. It will just assume that things start at zero, and let linker re-arrange. Compiler puts info in object file to tell linker how to re-arrange safely. What makes re-arrangement tricky? Addresses. This stuff is called *relocation information.*

- Compiler doesn't know where everything is when compiling files separately. E.g. where is *printf* routine? Where it doesn't know, compiler just puts zero in the object file and leaves an additional note in the object file telling the linker to fix things up. These things are called *cross-references.* There are two parts to a cross-reference:

  – The *global definition*: one file provides a variable or procedure that can be used by other files.

  – The *external reference*: a file accesses a variable or procedure that isn't in that file.

Each object file consists of:

- Two segments: code and data (the OS creates an empty stack segment when it loads the process).

- For each segment, the object file gives the size of the segment, the address where that segment should begin when loaded, and initial data if there is any.

- Symbol Table (global definitions) : information about stuff defined in this module that may be used in other modules. Used to get from name of thing to the thing itself.

- Relocation Information: information about addresses that the linker should fix up:

  – External references : Never knew to begin with.

  – Internals : Knew once, but if the linker re-arranges the segments then this will change.

- Additional information for the use of a debugger.
- Type *man 5 a.out* on UNIX for a complete description of UNIX object files.

Linker can shuffle segments around at will, but cannot rearrange information within a segment.

Example link job:

- Main program:

```
main()
{
    static float x, val;

    extern float sin();
    extern printf(), scanf();

    printf("Type number: ");
    scanf("%f", &x);
    val = sin(x);
    printf("Sine is %f", val);
}
```

- Math module:

```
float sin(x)
float x;
{
    static float tmp1, tmp2, result;
    – compute sin(x) –
    return result;
}
```

- C library:

printf(....)
....
scanf(....)
....

Linker runs in several passes (can't do final touchup until all files have been read):

- Pass 1: read in all of the symbol table information, decide how memory will be arranged. Must also read relocation information to see what additional stuff has to be gotten from libraries.
- Pass 2: read in segment and relocation information, modify addresses, write out new module containing symbols, segments, and relocation.

While linker is running it keeps around information about program, in *symbol table*.

- Segments – name, size, old location, new location.
- Symbols – name, input segment containing it, offset within input segment.

Typically, some kind of hash-coding scheme is used to make name accesses fast.

Pass 1 just assigns input segment locations in order to fill up the output segments. No information needs to be loaded into memory at this point except symbol information.

Do main/sin example, with the following segment sizes:

- Main: code 420, data 42.
- Math: code 1600, data 12.
- Library: code 1230, data 148.

Pass 2: read data and relocation information from files, fix up addresses, write out new object file.

Relocation Information:

- Address and size of value to be relocated.
- Symbol (or segment) that determines the amount of relocation.
- How to relocate:
  - 1. "Place final address of symbol here."
  - 2. "Add final address of symbol to contents of this location."
  - 3. "Add the difference between the final and original address of segment to the contents of this location."

Try a few example instructions to show how all of this fits together:

- MOVE R1,@#X (X is external).
- Record assignment: x = y.q, with y external.
- CALL SIN (SIN is external).
- JMP @#X (X in same original segment).
  BR X (X in same original segment, short displacement).