Storage Allocation: Linkers (Topic 4-1)

홍 성 수

서울대학교 공과대학 전기 공학부
Real-Time Operating Systems Laboratory

Storage Allocation (1)

- Readings for the topic:
  - The a.out(4) section of the Unix manual.
    Type *man a.out*
- Information stored in memory is used in many different ways.
  - Some possible classifications are:
    (1) Role in programming Language:
    - Instructions
      Specify the operations to be performed.

Storage Allocation (2)

- Variables
  The information that changes as the program runs.
- Constants
  Information used a operands, but that never changes.

(2) Changeability:
- Read-only
  Examples: Code, constants.
- Read & write
  Example: Variables
  Important in disk write-back and sharing.

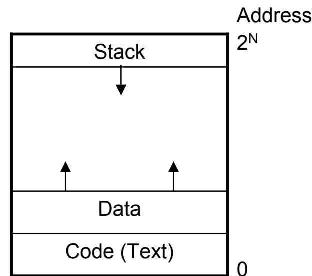
Storage Allocation (3)

(3) Addresses vs. Data.
- Must modify addresses if memory is rearranged.
- Example: Relocation and garbage collection.

(4) Binding time or When is memory allocated for the object ?
- Static: Location determined before program starts.
  Possibilities: Compile-time, link-time, or load-time.
- Dynamic: Location is determined at runtime and may change.

Process Memory Allocation (1)

- What does a process memory look like ?
  - In Unix it's divided up into areas called segments.



Process Memory Allocation (2)

- Why have different segments?
  - Separate read-only code from read-write data.
- Division of responsibility between various portions of system:
  - Compiler: Generates one object file for each source file.

    Information is incomplete, since source file reference some thing defined in other source files.
  - Linker: Combines all of the object files for one program into a single object file.

Process Memory Allocation (3)

  - Operation system: Loads object files into memory.

    Allows several different processes to share memory.

    Provides facilities for a process to get more memory.
- Linkers (or Linkage Editors, "ld" in Unix)
  - Tie together many separate pieces of a program.
  - Re-organize storage allocation.
  - Must interface with the operating system.

    (In Unix, ld is hidden by cc and g++).

Linkers (1)

- 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 segments of the same type).
  - 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.

Linkers (2)

- 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.
    - The linker must relocation things.
    - Compiler puts info in object file to tell linker how to re-arrange safely. This info is called relocation information.

Linkers (3)

- Compiler doesn't know where everything is when compiling files separately.

  Example: where is printf or is it a new routine?

  Where it doesn't know, compiler just puts zero in the object file and leaves additional note in the object file telling the linger to fix things up.

  These notes are called *cross-references*.

Linkers (4)

- 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.

Linkers (5)

- In Unix, each object file consists of:
  - Two segments: Code and data (the OS creates and 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.

    Initial data if there is any.

Linkers (6)

- 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.
  - Example: Printf is at 0x234
- Relocation Information: Information about addresses that the linker should fix up:
  - External references: Never know 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.
  - Example: Variable “foobar” is and integer located at 0x123.

Linkers (7)

- Type “man –s 4 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: Linker doesn’t rearrange instructions in a routine.

Linkers (8)

- Example link job:

Main program :	Math module :

main ()
{
static float x, val;
 	
extern float sin();
extern printf(), scanf();
 	
printf(“Type number:”)
scanf(“%f”, &x)
printf(“Sine is %f”, val);
}
	float sin (x)
	float x;
	{
	static float tmp1, tmp2;
	static float result;
	
	-- compute sin(x) –
	
	retrun result;
	}

C library : printf(....)	scanf(....) ....

Linkers (9)

- Starting object files:

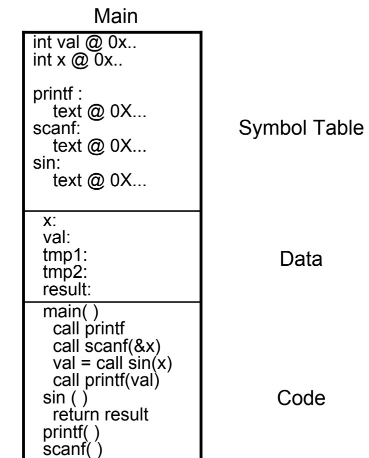
	Main	Math module
Symbol	main @ 0x.. int val @ 0x.. float x @ 0x.. ext printf : text @ 0X... ext scanf: text @ 0X... ext sin: text @ 0X	sin @ 0x... int tmp1 @ 0x... int tmp2 @ 0x... int result @ 0x... tmp1: tmp2: result: sin ( ) return result
Data	x: val:	libc printf @ 0x.. scanf @ 0x..
Txt	main( ) call printf call scanf(&x) val = call sin(x)	printf( ) scanf( )

Linkers (10)

- Three general functions:
  - Collect together segments of the same type from different files.
  - Compute new memory organization.
  - Relocate addresses

Linkers (11)

- Linker reorg:



- Note that addresses must be updated.

Linkers (12)

- Linker run in several passes:

Why: Can't do final touchup until all files have.

  - Pass 1: Read in all of the symbol table information.
    - Decide how memory will be arranged.
    - Read external references 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.

Linkers (13)

- While linker is running it keeps around information about program, in a symbol table.
  - Segments: Name, size, old location, new location.
  - Symbols: Name, input segment containing it, offset within input segment.
- Symbol names typically implemented using hashing.
  - Segments: Name, size, old location, new location.
  - Fast lookups.

Linkers (14)

- Pass 1 just assigns input segment location in order to fill up the output segments. No information needs to be loaded into memory at this point except symbol information.
  - From main/sin example:
    - Segments : Name, size, old location, new location.
    - Main: (Code, 420, ,) (data, 42, ,).
    - Math: (Code, 1600, ,) (data, 12, ,).
    - Library: (Code, 1230, ,) (data, 148, ,).

Linkers (15)

- Pass 2:
  - Read data and relocation information from files.
  - Fix up addresses.
  - Write out new object file.

Linkers (16)

- 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."

     Case 1: Extern int X; X=1; move #1, _X
  2) "Add final address of symbol to contents of this location."

     Case 2: Record/struct offsets.
  3) "Add the difference between the final and original address of segment to the contents of this location."

     Case 3: static int X; X=1;

Linkers (17)

- Example instruction relocation:
  - CALL SIN
  - BR X