Programming Tools

CIT 595
Spring 2008

System Software: Programming Tools

- Programming tools carry out the mechanics of software creation within the confines of the operating system and hardware environment
- These include:
  - Compiler & Assembler
    - Translate source program to machine language
    - Still not suitable for execution because of unresolved references to external and library routines
  - Linker
    - Bring together the binaries of separately compiled modules
    - Search libraries and resolve eternal references
  - Loader
    - Bring an object program into memory for execution by allocating memory to program (now becomes a process)
    - Might have to fix up some address

Assembly Process Example: First Pass

```
             .ORIG   x3000
x3000        AND     R2,R2,#0
x3001        LD      R3,PTR
x3002        TRAP    x23
x3003        LDR     R1,R3,#0
x3004        ADD     R4,R1,#-4
x3005 TEST   BRz     OUTPUT
x3006        NOT     R1,R1
x3007        ADD     R1,R1,#1
x3008        ADD     R1,R1,R0
x3009        BRnp    GETCHAR
x300A        ADD     R2,R2,#1
x300B GETCHAR ADD    R3,R3,#1
x300C        LDR     R1,R3,#0
x300D        BRnzp   TEST
x300E OUTPUT LD      R0,ASCII
x300F        ADD     R0,R0,R2
x3010        TRAP    x21
x3011        TRAP    x25
x3012 ASCII  .FILL   x0030
x3013 PTR    .FILL   x4000
             .END
```

Symbol	Address
TEST	x3005
GETCHAR	x300B
OUTPUT	x300E
ASCII	x3012
PTR	x3013

Assembly Process Example: Second Pass

```
             .ORIG   x3000
x3000        AND     R2,R2,#0    -> 0101 010 010 1 00000
x3001        LD      R3,PTR      -> 0010 011 000010001
x3002        TRAP    x23         -> 1111 0000 00100011
x3003        LDR     R1,R3,#0       .
x3004        ADD     R4,R1,#-4      .
x3005 TEST   BRz     OUTPUT
x3006        NOT     R1,R1
x3007        ADD     R1,R1,#1
x3008        ADD     R1,R1,R0
x3009        BRnp    GETCHAR
x300A        ADD     R2,R2,#1
x300B GETCHAR ADD    R3,R3,#1
x300C        LDR     R1,R3,#0
x300D        BRnzp   TEST
x300E OUTPUT LD      R0,ASCII
x300F        ADD     R0,R0,R2
x3010        TRAP    x21
x3011        TRAP    x25
x3012 ASCII  .FILL   x0030
x3013 PTR    .FILL   x4000
             .END
```

Symbol	Address
TEST	x3005
GETCHAR	x300B
OUTPUT	x300E
ASCII	x3012
PTR	x3013

Object File

- The partial machine code (a.k.a object code) produced by the compiler/assembler is outputted to a file
  - e.g. gcc –c filename.c produces file called filename.*o*
  - Each file or modules gets it own object code
    i.e. proc1.c ->proc1.o and proc2.c ->proc.o
- Format allows other programs Linkers and Loader to extract information
- Depending on the Operating System, object formats may vary
  - E.g. Unix uses a file format called ELF (Executable and Linking Format)

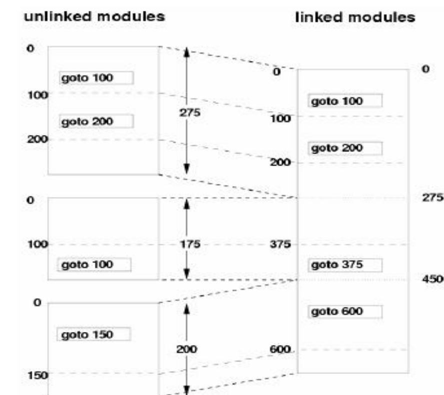
Object File Contents

- Header: Overall Information about the file and its contents
- Object code and data
- Relocation information
  - Information to help fix up the object code during linking
  - Which addresses are to be relocated is provided in this section
- Symbol Table
  - Information about symbols defined in the module (file) and symbols to be imported from other modules
- Debugging Information (optional)

Linker Function 1: Fixing Addresses

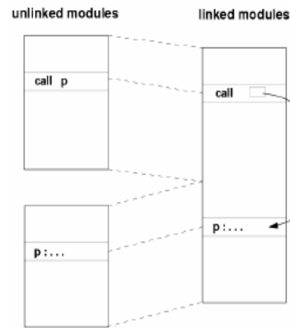
- Address in an object file are usually relative to start of the code or data segment in that file
  - Because compiler seldom knows where an object will reside, it often assumes a fixed base location (for example, zero).
    - Known as relocatable code
- When different object files are combined:
  - Similar segments (e.g. text section) from different object files get merged
  - Address have to be fixed up to account for this merging
    - Esp. absolute jumps, loads and stores instructions
  - The relocation information in the object file helps linker in the fixing of addresses

Fixing Addresses Example



Linker Function 2 : Symbol Resolution

- Example:
  - Module Y defines a symbol *p*
  - Module X refers to symbol *p*
- Task of the linker:
  - Determine the location of p in the linked module
  - Modify reference to *p*



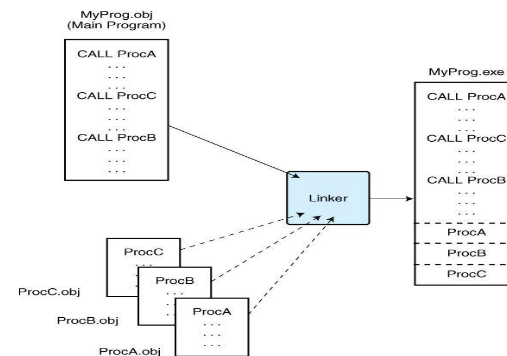
For Symbol Table Resolution

- Each module contains the following information in the symbol table
  - Symbols defined in the module
  - Symbols referenced but not defined in the module
    - Called externals (as they external to the current module)
  - Segment names (e.g. text, data ..)
    - Symbols defined to be at the beginning of the segment

Linking Process

- Construct a table of all the object modules and their lengths
- Based on this table, assigns a starting address to each module
- Copy the object module in the order of their load addresses
- Address relocation:
  - Find each instruction that contains memory reference and add to each a relocation constant equal to starting address of its module
- Symbol external resolution:
  - Create a symbol table containing symbol names of external modules and their address
  - For each instruction that references an external object module, insert the actual address for that object module based on table

Linking (contd..)



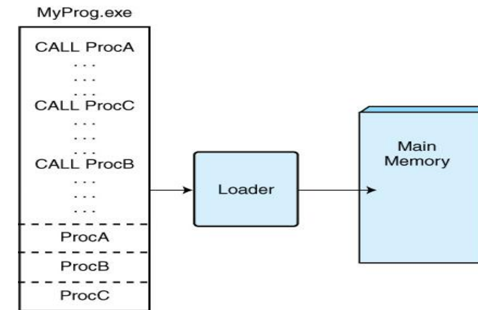
- Object modules have machine code + information for the linker
- Single binary file is called as *executable*

Loader

- *Loading* is the process of copying an executable image into memory i.e. at the actual physical location

Steps:
- Determine how much address space is need from the object file header (for e.g. text and data section)
- Allocate the address space
- Read the program segments in the address space
- Create a stack segment
- Tell the OS the value of PC (program counter) to start executing program (OS puts this info in PCB)

Loader (contd..)



Binding

- Regardless of relocatable or absolute code
  - Program's instruction and data are bound to physical address
  - This known as address *binding* a process
- Binding can be done at:
  - *Compile-Time*: executable already indicates where exactly it should be loaded i.e. absolute code
  - Loader learns whether the object file is absolute or relocatable depending on the system:
    - Have distinct file extension e.g. .COM (absolute) vs. .EXE in MS-DOS
    - Encode some magic number (some code)

Binding

- Load-Time
  - The executable output by the linker needs another relocation pass when it is finally loaded into memory
    - Adds starting address to each reference in the object module
  - The loaded executable a.k.a *process image* cannot be moved during execution as starting address must remain same (code now is absolute)
- Run-time or execution time Binding
  - Relocation step is omitted on hardware offering virtual memory
  - Every program is put into its own address space, so there is no conflict even if all programs start at the same base address
  - Allows for process image to be moved

Dynamic Linking

- ***Dynamic Linking*** defers much of the linking process till either load or runtime
- At load time, program and it's unlinked modules are linked and loaded by the loader
  - Causes start up delays
- The unlinked module can be linked upon invocation of that module while the program that requires it is running (run-time)
  - Static linker links each external reference in the program to a dummy module whose sole purpose is to call the ***dynamic linker*** or ***linking loader*** & pass the external symbol to it
  - The dummy module code to call the dynamic linker is known as *stub*

Dynamic Linker or Linking Loader

- Loads the unlinked module in memory
  - May just call the loader program to do this
- The unlinked module has some relocation and symbol table information that will help the dynamic linker to perform the linking process
  - i.e. relocates any references in executable to symbols defined by unlinked modules
- The application/program is given back control after the linking process is completed

Advantage of Dynamic Linking

- Many programs can *share* the same object module (library code), hence only one copy of the module needs to be present in memory
  - With prior linking (static linking) there will be n copies of same routine as n programs might require it
- Dynamically linked shared libraries are easier to *update* than static linked shared libraries
  - Change is visible to all programs that use it
  - Static libraries would have to be re-linked

Shared Libraries in Unix

- ***.so*** stands for shared library
- To create shared library and setting up dynamic linking
  - Create: **gcc –shared –fPIC –o lib*name.so* f1.o f2.o**
    - Tells gcc to create shared library with Position Independent Code(PIC)
    - PIC allows library code to be loaded and executed at any address
  - Usage: assume f3.c actually uses dynamic library
    gcc f3.c ./libname.so
  - Here a.out does not contain the object modules f1.o and f2.o
    - Still a partial executable
    - Contains some relocation and symbol table information that allow references to code and data in *libname.so* to be resolved at run time

Shared Libraries in Unix

- In general, libraries are stored in location /usr/lib
  - E.g. Math Library
    gcc prog.c –lm or
    gcc prog.c /usr/lib/libm.so
- If the library is in different directory then shell environment search path would need to include the that directory
  - Usually ***make*** files can be great help here
- By default, the compiler uses the *dynamic* version of a library whenever possible
  - This also applies to standard C libraries
    - E.g. functions in stdio.c and string.c
  - All standard C functions are linked to library called libc.so
  - We do not explicitly mention library, gcc compiler by default passes it to linker

Static Libraries in Unix

- Static libraries are stored in files *.a* (archive) extension
  - Is an archive relocatable object modules (.o files) concatenated together
  - It contains size and location of each object module
- To create static library
  - ar c libvector.a addvec.o multvec.o
  - ar - archive program & c - flag creates new archive
- To use it with your program
  - gcc –static prog.c ./libvector.a
  - static argument tells the linker should fully link executable object file that can be loaded into memory and run without further linking
    - Linker will only link the .o that is needed by prog.c
      - e.g. prog.c has #include <addvec.h> then addvec.o module will be linked
    - Linker will also copy any standard C libraries needed from static archive libc.a
      - E.g. prog,c has #include<stdio.h> then stdio.o module will be linked

Example: GCC program

- At *compile time*, *gcc* directs the *building* of executable files in 4 phases
- 1. compiler invokes the C preprocessor (*cpp*)
  - #includes and #defines cause string substitutions
- 2. compiler translates C to assembler (*gcc*)
  - input = preprocessed C, output = assembler (.s file)
- 3. compiler invokes the assembler (*gas*)
  - input = .s file, output = object file (.o file)
    - *partial* machine instructions are present in the .o file

GCC compile time (contd..)

4. compiler invokes the static linker (*ld*)
   - input = .o file, output = executable file (a.out)
   - **Static** library functions are combined into the executable at this time
     - *gcc* tells *ld* which versions of which libraries to link
   - **Dynamic** library functions are not combined executable contains some relocation and symbol table information for references
   - All intermediate files (.o) are deleted unless otherwise specified

GCC programs at Run-time

- 1. loader (*ldd*)
  - input is an executable file
  - output is a process image on disk
  - result is a PCB (process control block) placed in the "ready to run" queue of the operating system's scheduler
  - *ldd* is also called the "dynamic linker" or the "linking loader"
- 2. scheduler (part of OS)
  - all populated parts of the process image exist (either in main memory or on disk)
  - PCB is either 1) running, 2) in "ready to run" queue, 3) in "blocked" queue
- 3. dynamic linker (accessed via "stub" functions)

Disadvantage of Dynamic Linking

- No direct control over the contents of the dynamic linked library routine
- Execution of the program is vulnerable
  - If the library is moved, renamed, deleted or if an incompatible version of the DLL is copied to a place that is earlier in the search
  - Basically program will fail