

System Software- Part 2

CIT 595
Spring 2007

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:
 - Assembler
 - Compiler
 - Linker
 - Loader

CIT 595

14 - 2

Communicating with the Machine

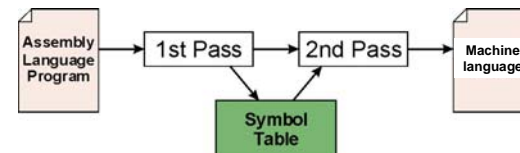
- Computers understand only ones and zeros...
 - E.g. 00000001111100
- Humans like readable form
 - E.g. A + B
- Assembler was the first step towards how people think and ,moving farther away from how the machine implements the solution

CIT 595

14 - 3

Assembler

- Assemblers translate mnemonic instructions to machine code
- ISA specific
- One assembly instruction translates to one machine instruction
- Most assemblers carry out this translation in two passes over the source code



CIT 595

14 - 4

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

CIT 595

14 - 5

Assembly Process Example: Second 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
    
```

```

0101 010 010 1 00000
0010 011 000010001
1111 0000 00100011
.
.
    
```

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

CIT 595

14 - 6

Assembler

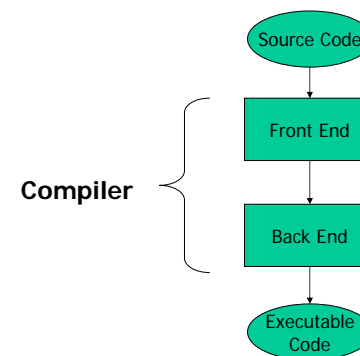
- However, assembler's job description must also include:
 - checking the illegal opcode
 - offset values
 - Register values
- Before assembler can create symbol table and output machine code it needs to
 - Extracting Keywords e.g. opcode, labels
 - Using extracted keywords to build instruction w/ its attributes
 - e.g. based on opcode, the instruction either has 0,1, 2 or 3 registers
 - Check whether attributes are within correct e.g. Register values are between 0 and 7 for LC3 ISA

CIT 595

14 - 7

Compiler

- High-level Language to Machine Code – more complicated process



CIT 595

14 - 8

Compiler – Front End

1. *Lexical analysis*

- Input character stream is split into meaningful symbols a.k.a *tokens* defined by the high-level language
 - E.g. of tokens include operators, literals, condition statements
 - For instance, an *integer* token may contain any sequence of numerical digit characters
 - "12*(3+4)" is split it into the tokens 12, *, (, 3, +, 4 and). Each of which is a meaningful symbol in the context of an arithmetic expression
- Builds the symbol table for user-defined variables and methods and with their location (local, static, dynamic) and data types
- Checks lexical errors e.g. 1myvar is illegal variable

CIT 595

14 - 9

Compiler: Front End

2. *Syntax analysis* (a.k.a Parsing)

- Checks that the tokens form an allowable expression i.e. checks whether expression or statement complies with language rules (or grammar)
 - Creates parse or syntax tree of expressions which replaces the linear sequence of tokens with a tree structure
 - Also checks illegal statement construction e.g. $A = B + C = D$
 - User-defined variables that part of syntax tree are verified using looking up symbol table

CIT 595

14 - 10

Compiler: Front End

3. *Semantic Analyzer*

- Checks data types of variables using information from symbol table in order to apply operator on them
- If language rule supports, can make data type promotions such as changing from "int" to "float"
- Associates variable and function references with their definitions a.k.a object binding

CIT 595

14 - 11

Compiler: Backend

4. *Intermediate code generation*

- Creates *pseudo-assembly from the syntax tree*
- Code produced is machine independent

5. *Optimization*

- Creates assembly code while taking into account architectural features that can make the code efficient
- E.g. Register Allocation, accounts for hazards if pipeline architecture

6. *Code generation*

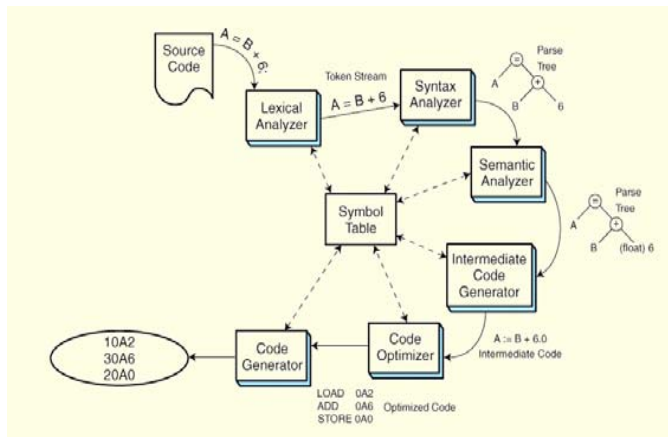
- Creates binary code from the optimized assembly code

Note: Through this modularity, compilers can be written for various platforms by rewriting only the last two phases

CIT 595

14 - 12

Compilers



CIT 595

14 - 13

Linker or Link Editor

- The machine code (a.k.a object code) produced by the compiler/assembler is outputted to a file
 - e.g. `.o` by C compiler and `.obj` by LC3 assembler
 - Each file or modules gets its own object code
i.e. `mylc3as.c ->mylc3as.o` and `parser.c ->parser.o`
- These files still have some unresolved references
 - Inclusion of library routine
 - Programmer writes his/her own modules/files
- Linker's task is to match *external symbols* of a program with all symbols from other files and produce a *single binary file* with no unresolved external symbols

CIT 595

14 - 14

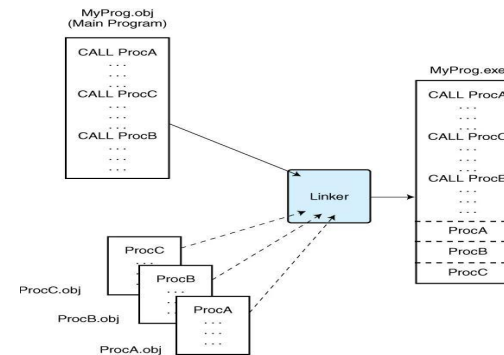
Object File Formats

- operating systems embed a "magic number" in executable files to assure that only allowable file formats get executed
 - Solaris uses the Executable and Linking Format (ELF) file format identifiable by the "magic number" at its beginning:
 - hex 7f followed by ASCII chars E L F: **7F 45 4C 46**
- Format allows other programs Linkers and Loader (discussed shortly) to extract information
- File format contains header section that defines start of each section (code and data) and how long that section is noted in bytes
- Also contains symbol tables for variables and functions

CIT 595

14 - 15

Linker (contd..)



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

CIT 595

14 - 16

Absolute Code

- *Absolute* code requires that executable be loaded at particular location in memory
- All user programs in LC3 required program to loaded from address x3000
 - The loader program is notified by encoding this address in the .obj file
 - hex t1.obj: 0000 **30 00** 12 43
 - if you forget, explanation also provided in hw5 document!!
- Usually required for specific purpose e.g. involve control of attached devices e.g. driver software for disk

CIT 595

14 - 17

Relocatable Object Code

- In real systems the executable can be loaded anywhere in user-space section in memory to allow multiprogramming
- The generated executable code then contains relative addressing for offsets with address starting at location 0
 - The code then produced is known as *relocatable code*
- A *Loader* program now chooses which location in memory executable will be loaded
 - Depending on activity in the system i.e. which chunks of memory are free and how much memory does the executable require

CIT 595

14 - 18

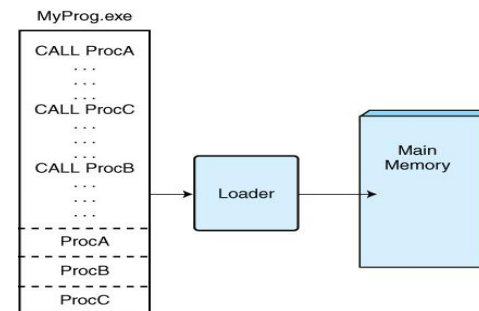
Loader

- *Loading* is the process of copying an executable image into memory i.e. at the actual physical location
- How does the Loader distinguish between relocatable and absolute code?
 - Depends from system to system
 - Example 1: Pretending binary executable code with prefix or preamble – uncommon in real systems
 - Example 2: MS-DOS operating system uses different file formats
 - .COM extension for absolute (nonrelocatable) code
 - .EXE extension for relocatable code

CIT 595

14 - 19

Loader (contd..)



CIT 595

14 - 20

Loader (cont..)

- 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
 - Load-Time
 - Adds starting address to each reference in the binary module
 - The loaded executable a.k.a *process image* cannot be moved during execution as starting address must remain same (code now is absolute)
 - The loader tells OS the value of PC (program counter) to start executing program (OS puts this info in PCB)

CIT 595

14 - 21

Loader (contd..)

- Run-time or execution time Binding
 - Delays binding till the time process is actually running
 - User program generates logical address or virtual address
 - Need to map this address to physical address
 - Possible due to Virtual Memory

CIT 595

14 - 22

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 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
 - Dynamically linked functions are populated with "stubs"
 - A stub is code to call the dynamic linker a.k.a *linking loader* at runtime

CIT 595

14 - 23

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
 - Shared object modules are called *Dynamic Linked Libraries* (DLL)
 - 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 relinked
- Can you think of a disadvantage ?

CIT 595

14 - 24

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
 - addresses are *offsets* in the TEXT or DATA segments at this point

CIT 595

14 - 25

GCC compile time (contd..)

4. compiler invokes the static linker (*ld*)

- input = .o file, output = executable file (no extension)
- **static** library functions are combined into the executable at this time
- *gcc* tells *ld* which versions of which libraries to link
- all intermediate files (.o) are deleted unless otherwise specified

CIT 595

14 - 26

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)

- maps dynamic library functions into process image as needed

CIT 595

14 - 27