Lecture #11

Red Flag – Inconsistent View

- Changing from 2nd to 3rd person
  - “Limit your disk storage to 100 Mb. The user can submit a request for more storage space to the system administrator.”
  - “Limit your disk storage to 100 Mb. For more storage space, you can submit a request to the system administrator.”

Red Flag – Passive Voice

- The verb expresses what is *done to* the subject (by someone or something)
- Occasional use is OK (and even unavoidable in many technical documents)
- But excessive use weakens your writing
  - “This error is used by the parser to indicate…”
  - “The parser issues this error to indicate…”

Red Flag - Wordiness

- In the final analysis, the end result of a wordy document is increased cost in terms of pages of paper, bytes on a disk, and inefficient use of the reader’s time and the writer’s effort
- Words cost money. It is cheaper to print a short book than a long one.

Red Flag – Faulty Parallelism

- The use of different grammatical constructs in a parallel structure
- Consider the list:
  - Preparing for installation
  - How to configure
  - Do you want the advanced options?

Red Flag – Dangling Modifier

- The use of a verbal phrase that does not connect (or modify) anything else in the sentence
  - “After typing enter, the system will continue with the second pass over the program”
  - “After you type enter, the system will continue with the second pass over the program”

Red Flag – Ending a Sentence With a Preposition

- Example:
  - "Before using the software, you must set it up"
  - "Before you can use it, you must set up the software"
- Winston Churchill:
  - "That is criticism up with which I will not put."

Red Flag – Splitting a Verb

- Dogma: Avoid splitting an infinitive
  - Infinitive is "to" + verb, or "will" + verb
    - eg "to think", "to breathe", "will dance"
- Avoid putting an adverb inside
  - Patients should try to if possible avoid going up and down the stairs
  - If possible, patients should try to avoid…
- Famous counter-example: Captain Kirk
  - "Its five-year mission, to boldly go…"
- Famous counter-example: Chief Justice Roberts
  - "…will execute the office of the president of the United States faithfully…"

Red Flag – Provincial and Sexist Language

- Unless you are sure your readership is homogeneous, be sensitive to and inclusive of many cultures and both genders
- Example a list of names:
  - "Bill White, Ken Williams, and Bob Smith"
  - "Chris Amini, Lea Sanchez, and Rei Chi Lee"

Red Flag – "Which" vs. "That"

- Simple rule: if "that" sounds OK, use it!
- Use "which" with nonrestrictive clauses
- Use "that" with restrictive clauses
- Example:
  - "Ed's country house, which is located on five acres, had bats in the attic."
  - "The house that sat on the top of the hill had bats in the attic."

Red Flag - Utilize

- Why would anyone utilize the word "utilize" when the word "use" works just as well?

Things to Check – Appropriate Style

- Amount of detail
  - Verbose vs. terse
- Formality
  - Formal vs. informal
  - Use of contractions, informal language, slang
- Tone
  - Distant, warm, familiar, intrusive
- For whatever style is chosen, *consistency* is very important

Examples from Design Review

The component type which contains the math definition of the component is stored in the file called Type.h which provides the client with the general component description.

The component type, which contains the math definition of the component, is stored in a file called `Type.h`. This file also provides the client with a general description of the component.

If users input a file has invalid header record which contains more than 13 characters, the program will pop up an error message: “Parse Error: Invalid header record”, then it terminates.

If the input file given by the user does not begin with a valid header record, the following error message is displayed: “*Parse Error: Invalid header record*”.

The overall is contained in Machine_Program.cpp

The main program is contained in the file `Machine_Program.cpp`.

If the user wishes to quit, then a “q” will be entered.

To quit, the user must enter “q”.

The function will return 0 if successful, and an integer value if an error of any type is recorded.

The function returns an integer. A return value of 0 indicates successful completion, while a non-zero return value indicates an error.

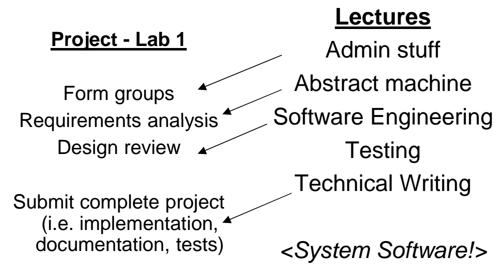
So now your wondering how these classes will work together? Well it’s pretty simple these classes will extend the functionality of Score.java and Error.java. So, what does that mean? Well, it means that…

Please give me an E.

Bottom Line

- Technical writing requires work, practice, skill, technique, time; *not talent*.
- The first draft is *always* bad writing. Allow time (and energy) for revisions.
- There is no substitute for having something to say. You can't bluff it.

Road Map



Lecture #12

System Software - Overview

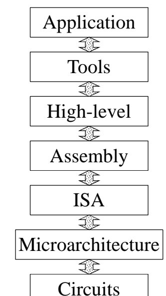
- What *is* "system software" anyway?
  - "programs that support the operation of a computer"
  - often closely related to the architecture
  - allow us to focus on *application* without knowing details of *machine*

Overview (II)

- Examples:
  - operating systems
  - software used to create other software!
    - compiler
    - linker / loader
    - assembler
    - debugger
- Driving force: people more expensive than machines

Layers of Abstraction - Architecture

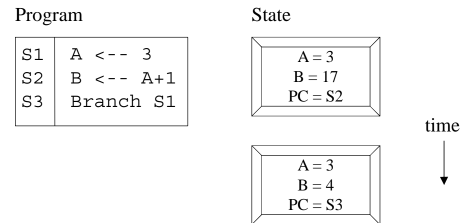
- Computer can be viewed at different levels of *abstraction*
- Each layer is a "virtual machine" (VM)
- This helps bridge the human / machine gap
- Each VM corresponds to a *language*



Program vs Process

- Program: a collection of actions
- Process: a program in execution
  - contains *state*:
    - i) values of variables,
    - ii) location in program,
    - iii) pending I/O, etc. …
  - state changes over *time*

Program vs Process (Example)



- Executing an instruction *changes* the state

Two Important Kinds of Program

Interpreter

- a program that advances the state of another process
- quicker debugging and prototyping, portability

Compiler

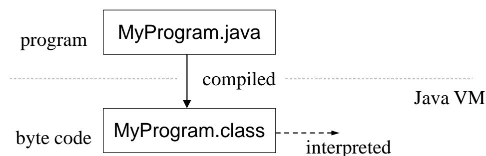
- a program that translates a program from one VM level to another (lower) one
- faster executable

Compiling vs. Interpreting

- Some languages are usually interpreted
  - Javascript, PHP, Lisp, Ruby, Smalltalk
- Some languages are usually compiled
  - C, C++, Fortran, COBOL, Lisp (!)
- Some languages are usually *both*
  - first compiled to "byte-code"
  - then interpreted by some virtual machine
  - Java, C#, Perl, Python

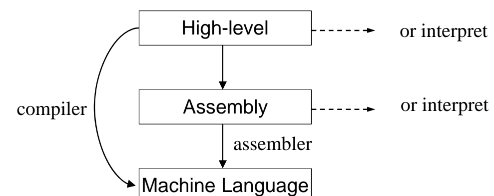
Compiling vs. Interpreting

- Consider Java:



Compiling vs. Interpreting

- Compilation / interpretation isn't fundamental to a language definition
- Can occur at all levels



Operating Systems - Introduction

- When does a program become a process?
  - when it is assigned certain system resources
  - e.g., processor, memory, I/O, registers, …
- At any instant, there are many processes
  - multiple, concurrent users
  - mix of batch and interactive jobs
  - OS tasks (e.g. spooling)
- But only a fixed number of resources...

OS - Introduction (II)

- Resources must be *managed*
- This is the job of the *operating system* (OS)

Challenges in OS

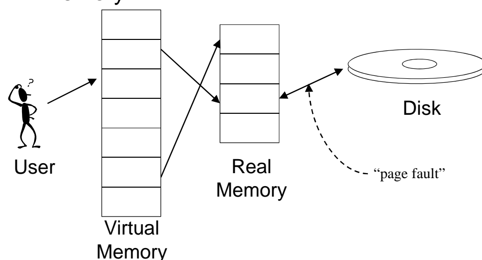
- Concurrency is fundamental
- Concurrency is hard
  - Example: sharing a bridge
    - Long tunnel that only fits 1 lane of traffic
    - What policy do you use to control traffic?
  - Example:
    - Process A is using X and needs Y,
      " B " " Y " " X
    - OS must avoid *deadlock* and *starvation*

Responsibilities of OS

- Handles interrupts
  - may be generated by I/O or by programs
- Manages real memory
  - loading of segments
- Manages virtual memory
  - "virtual": appears to user to have different characteristics than it has in actuality
  - virtual memory: a large block of contiguous memory space

Responsibilities of OS (II)

- virtual memory may be *larger* than real memory!



Responsibilities of OS (III)

- File management
  - keeps file handles and position marks
- CPU
  - schedules processes (running / ready / waiting)
- Security
  - prevents one user / process from damaging another's data
  - prevents user from damaging operating system

Lecture #13

Introduction to (and Review of) Assembly

Definition

- Recall: translation
  - source program translated into target program
  - source and target define _______________
  - source is *not* directly executed
  - target ("object file") is executed or translated later

Definition II

- When the source is a symbolic representation of machine language:
  - source language = __________________
  - translator – __________________
- (When the source is higher-level, the translator is usually called a _____________)

Advantages of Assembly

- Over machine code (lower level)
  - easier to remember mnemonic operations than actual opcodes
    - e.g., ADD, SUB, MUL, DIV, ...
vs, 24576, 57344, 28672, 61440, …
  - similarly for addresses in program
    - e.g., BR LOOP1
vs, BR 46554

Advantages of Assembly II

- Over higher-level languages
  - access to full capabilities of the machine
    - e.g., testing overflow flag, test-and-set instruction,…
    - how would you do that in Java?
  - speed ...

The “Best” of Both Worlds

- Systems programming is often done in a language like C
  - syntax of a higher-level (problem-oriented) language
  - but gives access to low-level machine, like assembly

An Old Myth

- “If a program will be used a lot, it should (for efficiency) be written in assembly.”
- No longer true
  - good compilers
  - fast machines
  - hard to write
    - 10 lines of code / day, independent of language
  - hard to read
    - high cost of maintenance
      - can be 2/3 of total
      - 15% programmer turnover

Modern Approach

- Write in high-level language
- Analyze to find where time spent
- Invariably, it’s a *small* part of the code
- Tune that tiny part for high performance
  - *perhaps* by writing in assembly

Modern Approach II

- Higher level can be a *performance win too!*
  - problem-oriented language gives problem-level insights
  - huge performance gains are in algorithmic insights
    - e.g., $O(n^3)$ vs $O(n \lg n)$
  - assembly programmer immersed in bit-twiddling
    - saves small amounts all over, but misses big picture
    - “penny wise, pound foolish”

Modern Approach III

- Conclusion:
  - assembly use is often a holdover from when machines were expensive, a*nd people* were *cheap*

So Why Do We Learn This Stuff?

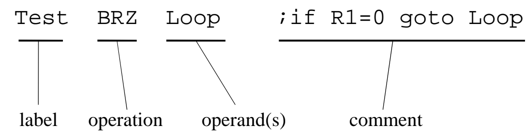
- You may still need to write that tiny, critical part in assembly
- Concepts, techniques, algorithms similar to those used for compilers (but simpler)
- Pedagogical use
  - good vehicle for understanding architecture
  - enriches understanding of higher-level languages
- Legacy code with large parts in assembly
- **This world still needs assemblers**!
  - target for translation and compiler optimization

Assembly Instructions

- We've seen the basic structure in memory:

- Four parts to an instruction in assembly:
  1. Label
  2. Operation
  3. Operands
  4. Comments

Example Instruction



Label Field

- Symbolic name for the instruction *address*
- Clarifies *branching* to a particular instruction
  – e.g., `BRNP Loop1`
- Also allows symbolic access to *data*
  – e.g., `LD R3,Sum`
- Often severely limited in length

Operation Field

- Mnemonic for an instruction
  – e.g., ADD, SUB, BRZ
- Mnemonic for a "pseudo-operation"
  – e.g., .FILL
  – we'll see what these mean later...

Operand Field

- Addresses and registers used by instruction
  – recall: "arguments to the function"
- What to add, where to branch, where to store, …
- Operands for pseudo-operations
  – used to give information to the assembler
  – e.g., program name, how much space to save,…

Comment Field

- No effect on assembler
  – no semantic impact on program
- But huge impact on legibility!
  – clarify the program
  – strictly for human consumption

Pseudo-Operations

- Recall: “operation” field can be either:
  - instruction (BRZ, NOT, TRAP …)
  - pseudo-op
- Unlike instructions, pseudo-ops do *not* have a corresponding machine instruction (opcode) in the ISA
- Give information to the *assembler* itself
  - used to control various aspects of the object file that is generated by the assembler
  - “assembler directives”

Pseudo-Ops: Uses

- Four principal uses:
  a) segment definition
  b) symbol definition
  c) memory initialization
  d) storage allocation

(A) Segment Definition

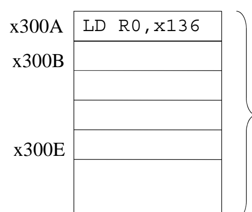
- Recall information in header & end records:
  - segment name
  - segment load address
  - segment length
  - initial execution address
- All this information comes directly from pseudo-ops
  - (all except _______________)

Segment Definition II

- Two important pseudo-ops:
  - .ORIG ("origin")
  - .END ("end")

```
MainP     .ORIG     x300A
          LD        R0,x136
. . .
          .END      x300E
```

- What is the header record of the resulting object file?
  - ie what is the footprint of the defined segment?



Header record: `HMainP_300A????`
End record: `E300E`

(B) Symbol Definition

- A label creates a symbol
- Symbol is *implicitly* defined to be the address of that instruction

```
Hello     .ORIG     x300A
          LD        R0,x136
Test      BRZ       x147
      . . .
```

- What is the value of "Test"?

Symbol Definition II

x300A	LD R0,x136
x300B	BRZ x147

- So “Test” has value: _________

Explicit Symbol Definition

- Symbols can also be defined *explicitly*
- Pseudo-op:
  - .EQU ("equate")
- Example:

```
STEP  .EQU  #2   ;step size = 2
```

- Symbols are used as program constants

Use of Symbols

- Example 1:

```
ADD R1,R1,STEP
```

  - means:
  - i.e.:
- Example 2:

```
HALT  .EQU   x25
      TRAP   HALT
```

  - means:
  - i.e.:

⋮	Memory	Binary	Instruction/Data		
30B0	0 0 0 4	0000000000000100	BR	x4	;#4
30B1	2 2 B 0	0010001010110000	LD	r1,xB0	
30B2	E 0 B 7	1110000010111001	LEA	r0,xB7	
30B3	F 0 2 2	1111000000100010	TRAP	x22	
30B4	1 2 7 F	0001001001111111	ADD	r1,r1,#-1	
30B5	0 2 B 3	0000001010110011	BRP	xB3	
30B6	F 0 2 5	1111000000100101	TRAP	x25	
30B7	0 0 6 8	0000000001101000	BR	x68	;`h`
30B8	0 0 6 9	0000000001101001	BR	x69	;`i`
30B9	0 0 2 0	0000000000100000	BR	x21	;` `
30BA	0 0 0 0	0000000000000000	BR	x0	;\0
⋮					

(C) Memory Initialization

- Recall “hi ” example
- Sometimes want to load “*data*” into memory
  - could use corresponding instruction (since machine doesn’t care!)
  - but that is inconvenient, and may not even exist
- Two pseudo-ops (for 2 kinds of data):
  - .FILL, for filling a cell with a numeric value
  - .STRZ, for a null-terminated string

.FILL and .STRZ

- Format for these pseudo-ops:
  - Numerical data:
    - decimal or hex
      - Note: different ranges for decimal (+ve & -ve) and hex (+ve only)
    - label
  - Character data: string in quotes
- Example:

```
Count    .FILL   #10007
Mask     .FILL   xF
Array    .FILL   Count
Text     .STRZ   "Yo!"
```

.FILL and .STRZ (II)

- These pseudo-ops often have *labels*
  - (but not required)

Address	Contents		
x406E	2 7 1 7		(ie #10007)
x406F	0 0 0 F		(ie xF)
x4070	4 0 6 E		(ie Count)
x4071	0 0	'Y'	
x4072	0 0	'o'	
x4073	0 0	'!'	
	0 0	0 0	
		...	

Count = ________
Mask = ________
Array = ________
Text = ________

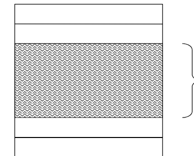
Lecture #14

(D) Storage Allocation

- Set aside a block of memory
  - not initialized (i.e. don't care)
- Pseudo-op:
  - .BLKW ("block of words")
  - operand is:
    - constant (i.e., hexadecimal or decimal integer), or
    - previously defined symbol
- Example:

```
X     .FILL  x1
Buffer.BLKW  x64
Y     .FILL  x64
```

- Yields:



- Q: How would we "typically" address values in this buffer?

Using Blocks of Storage

- A:
- Example:

```
    LEA  R3,Buffer
  . . .
    STR  R0,R3,#1
```

- Note: some pseudo-ops affect the "location counter" (storage) others don't
  - do:
  - don't:

Symbols - Shortcomings

- We've seen a lot of utility for symbols
  - mnemonics for data constants & memory addresses
- But they are sometimes inconvenient:
  - consider incrementing a register by #16
    - i.e. R2 <-- R2 + #16
  - can't just use `ADD R2,R2,#16`
    - why not?
  - must initialize another register with the value
  - must explicitly *name* and *allocate* a constant!

```
SIXTN  .FILL  #16
       LD     R3,SIXTN   ;R3 <- #16
       ADD    R2,R2,R3
```

Symbols - Shortcomings II

- Problems with this approach:
  - ___________________
  - ___________________
  - ____________________
  - ____________________

Alternative: Literals

- *Implicit* allocation & initialization of memory
- Allows us to put the value itself right in the instruction
- Prefice with “=“
- Example:

```
LD   R3,=#16
```

Literals

- This means:
  - allocate storage at end of program
  - initialize this storage with the value #16
  - use this *address* in the instruction
- So it is equivalent to:

```
       LD   R3,SIXTN
    . . .
SIXTN  .FILL   #16
```

Literals - Restrictions

- Value must be in the range $-2^{15} \ldots 2^{15}-1$
  - i.e. can be represented by 1 word
- Can only replace the “pgoffset9” field
- Cannot be used with indirect addressing
- Cannot use with:
  - branch, store
- Each restriction has a motivation