CSC 411

Computer Organization (Spring 2024)
Lecture 16: Compiling/interpreting and running programs

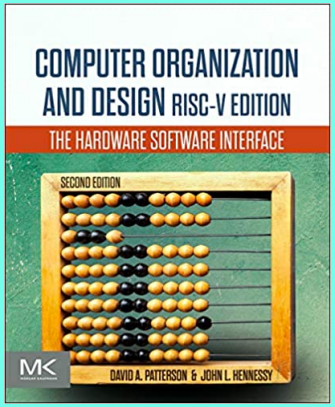
Prof. Marco Alvarez, University of Rhode Island

Disclaimer

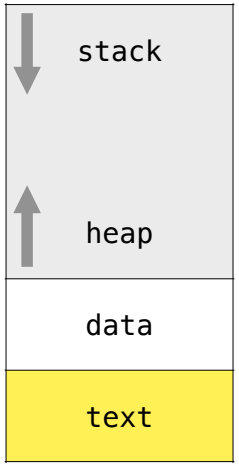
Some figures and slides are adapted from:

Computer Organization and Design (Patterson and Hennessy)

The Hardware/Software Interface



Context



High-level language program (in C)

```
swap(size_t v[], size_t k)
{
   size_t temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

Compiler

Assembly language program (for RISC-V)

```
swap:
      slli x6, x11, 3
      add  x6, x10, x6
      lw   x5, 0(x6)
      lw   x7, 4(x6)
      sw   x7, 0(x6)
      sw   x5, 4(x6)
      jalr x0, 0(x1)
```

Assembler

Binary machine language program (for RISC-V)

```
00000000001101011001001100010011
00000000011001010000001100110011
00000000000000011001100101010011
00000000100000110011001110000011
00000000111100110011000000100011
00000000101001100110100001000011
00000000000000001000000001100111
```

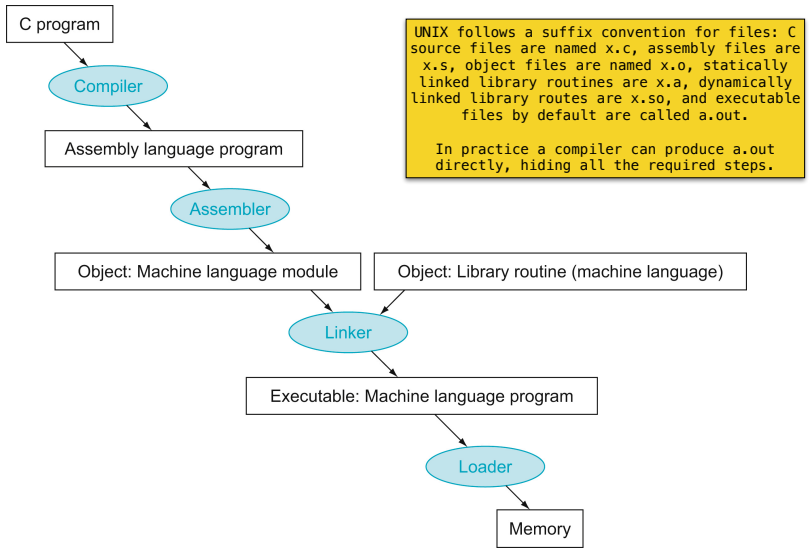
Compiling vs Interpreting

- Compiler
  - a program that acts as a translator, converting a program from a high level language into another (lower level language)
  - preferred for increasing performance
  - e.g. C/C++ compiler
- Interpreter
  - a program that executes a program in the source language
  - e.g. Python, Java Virtual Machine
    - any language can be interpreted (including C or assembly)
  - preferred when performance is not critical

Compiling vs Interpreting

- Compiled code “hides” the actual source code from others
  - may be preferred in certain contexts (commercialization)

Translating / running programs



Compiler

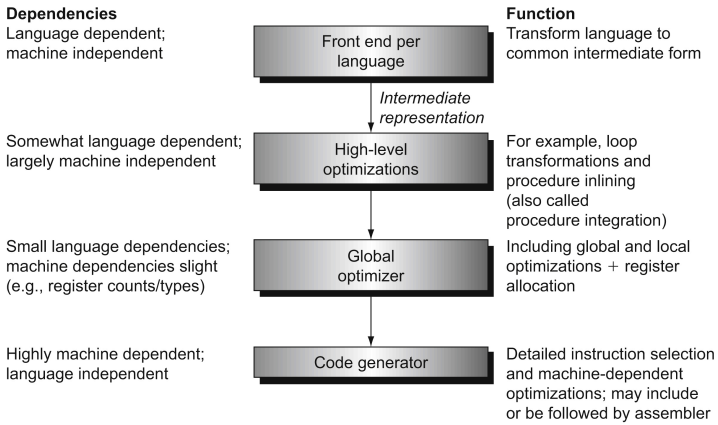
- Translates high-level language into assembly
  - symbolic form of machine language
  - in the 70s many operating systems were written in assembly language
- Optimizes generated code
  - compilers can generate assembly code sometimes better than human experts

Compiler

- Major types of optimizations

Optimization name	Explanation	gcc level
High level	***At or near the source level; processor independent***	
Procedure integration	Replace procedure call by procedure body	03
Local	***Within straight-line code***	
Common subexpression elimination	Replace two instances of the same computation by single copy	01
Constant propagation	Replace all instances of a variable that is assigned a constant with the constant	01
Stack height reduction	Rearrange expression tree to minimize resources needed for expression evaluation	01
Global	***Across a branch***	
Global common subexpression elimination	Same as local, but this version crosses branches	02
Copy propagation	Replace all instances of a variable ***A*** that has been assigned ***X*** (i.e., ***A*** = ***X***) with ***X***	02
Code motion	Remove code from a loop that computes the same value each iteration of the loop	02
Induction variable elimination	Simplify/eliminate array addressing calculations within loops	02
Processor dependent	***Depends on processor knowledge***	
Strength reduction	Many examples; replace multiply by a constant with shifts	01
Pipeline scheduling	Reorder instructions to improve pipeline performance	01
Branch offset optimization	Choose the shortest branch displacement that reaches target	01

Modern compiler



The structure of a modern optimizing compiler consists of a number of passes or phases. Logically, each pass can be thought of as running to completion before the next occurs. In practice, some passes may handle one procedure at a time, essentially interleaving with another pass.

Assembler

- Converts pseudo-instructions into actual instructions
- Converts assembly instructions into machine instructions
  - must determine memory addresses and offsets for all labels (requires two passes over program)
- Generates an object file
  - combination of machine instructions, data, and information needed to load program properly in memory

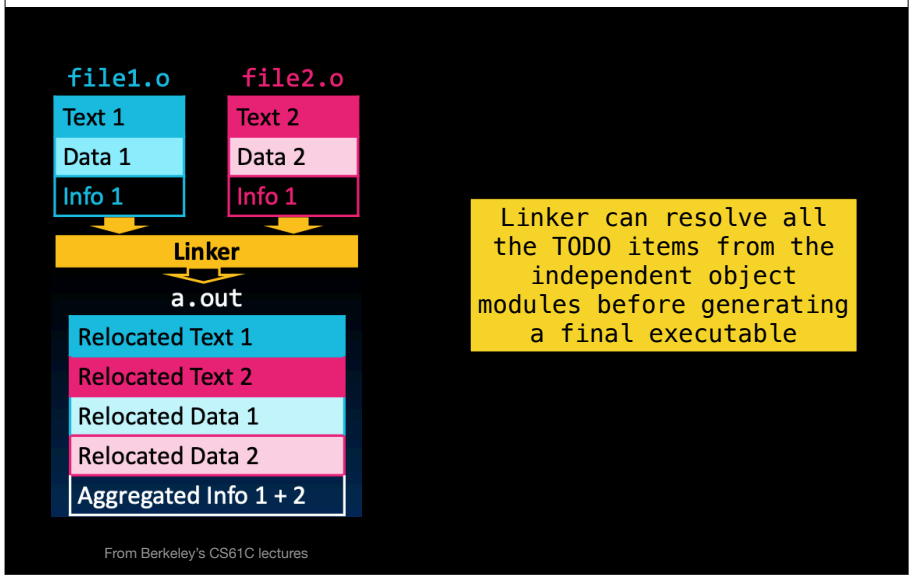
Producing an "object"

- Assembler translates program into machine instructions
- Provides information for building a complete program from the pieces
  - header: describes contents (size/position) of other pieces of the object
  - text segment: translated instructions
  - static data segment: data allocated for the life of the program
  - relocation info: for contents that depend on absolute location (fixed by linker)
  - symbol table: global definitions and external refs that can be used by other objects
  - debug info: for associating with source code

Linker

- Merges all object modules, producing an executable
  - merges segments
  - resolves labels (determine their addresses)
  - patch location-dependent and external refs
- Could leave location dependencies for fixing by a relocating loader
  - with virtual memory, no need to do this
  - program can be loaded into absolute location in virtual memory space

Linker



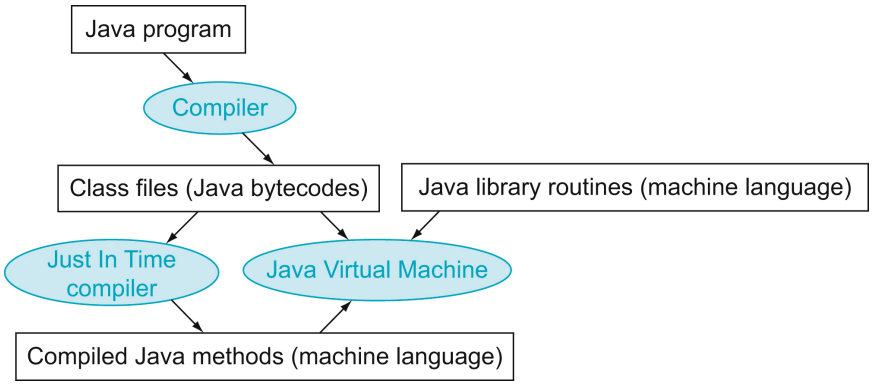
Loader (OS task)

- 1 read header to determine segment sizes
- 2 create virtual address space
- 3 copy text and initialized data into memory
  - or set page table entries so they can be faulted in
- 4 set up command line arguments on stack
- 5 initialize registers (including sp, fp, gp)
- 6 jump to startup routine
  - copies arguments to x10 and x11 and calls main
  - when main returns, terminate program with exit syscall

Dynamic linking

- Only link/load library procedure when it is called
  - requires procedure code to be relocatable
  - avoids image bloat caused by static linking of all (transitively) referenced libraries
  - automatically picks up new library versions

Java applications



Impacts on performance

Sort example in C

```
void swap(int v[], size_t k) {
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

```
swap:
  slli x6, x11, 2  # reg x6 = k * 4
  add  x6, x10, x6 # reg x6 = v + (k * 4)
  lw   x5, 0(x6)   # reg x5 (temp) = v[k]
  lw   x7, 4(x6)   # reg x7 = v[k + 1]
  sw   x7, 0(x6)   # v[k] = reg x7
  sw   x5, 4(x6)   # v[k+1] = reg x5 (temp)
  jalr x0, 0(x1)   # return to caller
```

```
void sort (int v[], size_t n) {
    size_t i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1; j >= 0 && v[j] > v[j + 1] ; j -= 1) {
            swap(v, j);
        }
    }
}
```

Saving registers

```
sort:   addi sp, sp, -20        # make room on stack for 5 registers
        sw x1, 16(sp)           # save return address on stack
        sw x22, 12(sp)          # save x22 on stack
        sw x21, 8(sp)           # save x21 on stack
        sw x20, 4(sp)           # save x20 on stack
        sw x19, 0(sp)           # save x19 on stack
```

Procedure body

Move parameters	`addi x21, x10, 0`	`# copy parameter x10 into x21`
	`addi x22, x11, 0`	`# copy parameter x11 into x22`
Outer loop	`addi x19,x0, 0`	`# i = 0`
	`for1tst:bge x19, x22, exit1`	`# go to exit1 if i >= n`
Inner loop	`addi x20, x19, -1`	`# j = i - 1`
	`for2tst:blt x20, x0, exit2`	`# go to exit2 if j < 0`
	`slli x5, x20, 2`	`# x5 = j * 4`
	`add x5, x21, x5`	`# x5 = v + (j * 4`
	`lw x6, 0(x5)`	`# x6 = v[j]`
	`lw x7, 4(x5)`	`# x7 = v[j + 1]`
	`ble x6, x7, exit2`	`# go to exit2 if x6 < x7`
Pass parameters and call	`addi x10, x21, 0`	`# first swap parameter is v`
	`addi x11, x20, 0`	`# second swap parameter is j`
	`jal x1, swap`	`# call swap`
Inner loop	`addi x20, x20, -1`	`j for2tst`
	`jal, x0 for2tst`	`# go to for2tst`
Outer loop	`exit2: addi x19, x19, 1`	`# i += 1`
	`jal, x0 for1tst`	`# go to for1tst`

Restoring registers

```
exit1:  lw x19, 0(sp)           # restore x19 from stack
        lw x20, 4(sp)           # restore x20 from stack
        lw x21, 8(sp)           # restore x21 from stack
        lw x22, 12(sp)          # restore x22 from stack
        lw x1, 16(sp)           # restore return address from stack
        addi sp, sp, 20         # restore stack pointer
```

Procedure return

```
        jalr x0, 0(x1)          # return to calling routine
```

Compiler optimizations

gcc optimization	Relative performance	Clock cycles (millions)	Instruction count (millions)	CPI
None	1.00	158,615	114,938	1.38
01 (medium)	2.37	66,990	37,470	1.79
02 (full)	2.38	66,521	39,993	1.66
03 (procedure integration)	2.41	65,747	44,993	1.46

The programs sorted 100,000 32-bit words with the array initialized to random values. These programs were run on a Pentium 4 with a clock rate of 3.06 GHz and a 533 MHz system bus with 2 GB of PC2100 DDR SDRAM. It used Linux version 2.4.20.

Comparing sorting algorithms

- Compiler optimizations are sensitive to the algorithm
- Java/JIT significantly faster than JVM interpreted
- Nothing can fix slow algorithms

Language	Execution method	Optimization	Bubble Sort relative performance	Quicksort relative performance	Speedup Quicksort vs. Bubble Sort
C	Compiler	None	1.00	1.00	2468
	Compiler	01	2.37	1.50	1562
	Compiler	02	2.38	1.50	1555
	Compiler	03	2.41	1.91	1955
Java	Interpreter	–	0.12	0.05	1050
	JIT compiler	–	2.13	0.29	338

Arrays vs pointers

- Array indexing involves:
  - multiplying index by element size
  - adding to array base address
- Pointers correspond directly to memory addresses
  - can avoid indexing complexity

Example: clearing an array

```
void clear1(int array[], int size) {
    int i;
    for (i = 0 ; i < size ; i += 1)
        array[i] = 0;
}
```

```
void clear2(int *array, int size) {
    int *p = array;
    int *end = p + size;
    for ( ; p < end ; p++)
        *p = 0;
}
```

```
    li   x5, 0       # i = 0
l1:
    slli x6, x5, 2   # x6 = i * 4
    add  x7, x10, x6 # x7 = &array[i]
    sw   x0, 0(x7)   # array[i] = 0
    addi x5, x5, 1   # i = i + 1
    blt  x5, x11, l1 # if (i<size) go to l1
```

```
    add x5, x0, x10 # p = addr of array[0]
    slli x6, x11, 2 # x6 = size * 4
    add x7, x10, x6 # x7 = addr of array[size]
l2:
    sw x0, 0(x5)    # *p = 0
    addi x5, x5, 4  # p = p + 4
    bltu x5, x7, l2 # if (p<end) go to l2
```

Fallacies

- Powerful instructions ⇒ higher performance
  - fewer instructions required but complex instructions are hard to implement
  - may slow down all instructions, including simple ones
  - compilers are good at making fast code from simple instructions
- Use assembly code for high performance
  - modern compilers are better at dealing with modern processors
  - more lines of code ⇒ more errors and less productivity

Fallacies

- Backward compatibility ⇒ instruction set “doesn’t change”
  - but, adding more instructions

Growth of the x86 instruction set over time