Runtime environments

Handout written by Maggie Johnson and revised by me.

Architecture vocabulary

Let's review a few relevant hardware definitions:

register: a storage location directly on the CPU, used for temporary storage of small amounts of data during processing.

memory: array of randomly accessible memory bytes each identified by a unique address. Flat memory models, segmented memory models, and hybrid models exist.

instruction set: the set of instructions that are interpreted directly in the hardware by the CPU. These instructions are encoded as bit strings in memory and are fetched and executed one by one by the processor. They perform primitive operations such as "add 2 to register B1", "store contents of D1 into memory location 16", etc. Instructions consist of an operation code (opcode) e.g., load, store, add, etc., and one or more operand addresses.

CISC: Complex instruction set computer. Most older processors fit into the CISC family, which means they have a large and fancy instruction set. In addition to a set of common operations, the instruction set may also have many special purpose instructions that are designed for limited situations. CISC processors tend to have a slower clock cycle, but accomplish more in each cycle because of the fancy instructions. It is usually more difficult to write an effective back-end for a CISC processor.

RISC: Reduced instruction set computer. RISC processors are distinguished by a relatively lean instruction set, containing just simple and general-purpose instructions. Even basics like multiply and divide often aren't present (you have to do repeated addition/subtraction/shift/ etc.) But the clock cycle can be cranked to unbelievable speeds allowing comparable or better performance than CISC chips overall. (RISC of the many innovations we can attribute to our current university president).

Memory hierarchy

All computers need storage of some sort to store and work on data during processing. Ideally, this storage would be quick to access and large in size. However, these two goals conflict. Fast memories are expensive and small; and slow memories are cheaper and large. To provide the illusion of both fast and large, the memory system of modern computers is organized in a hierarchical way. The very top of the hierarchy is CPU registers, between the CPU and main memory, a fast cache memory is added. Via virtual memory, the hard disk can expand the capacity of main memory. A typical memory hierarchy might have 5 levels:

registers: Your average modern processor has 16 to 128 on-board registers, each register can hold a word of memory (typically 32-bits). The total register space is thus just a couple hundred bytes total. Time to access registers is perhaps a few nanoseconds. Registers are the most expensive. You cannot expand space by adding registers to a chip, they are fixed as part of its design.

L1 cache: An level-1 cache is on the same chip as the processor and usually cannot be expanded. Typical L1 caches are 32-64 Kb. Time to access the L1 cache is

usually 10s of nanoseconds. L1 caches are pricey, but not as expensive as registers.

L2 cache: A level-2 cache is usually a separate chip from the processor and a typical size is 1-2 MB. Access time is usually in the range of 10s of nanoseconds and cost is around $100 per megabtye.

main memory: Main memory is what you think of when you say your computer has 128MB of memory. These are DRAM chips that can usually be expanded. Today's computers often have 64-256 MB of main memory. Access time is in the 50-100 ns range and cost is around a few dollars per megabyte.

virtual memory: When using your hard disk as back-up, you can further extend memory in the many gigabyte size. Disks are cheap, just dimes per megabyte, but slow, access times are measured in 10s or 100s of milliseconds.

Summary: you get what you pay for…

Processor architectures

A processor's architecture refers to the way in which its memory and control is organized. Most early computers were designed as *accumulator machines*. The processor has a single register called the accumulator where arithmetic, logic and comparison operations occur. All other values and variables are stored in memory and transferred to the accumulator register when needed. These machines are also referred to as one-address machines since the instruction set uses an opcode and a single operand address. Accumulator machines are rarely seen anymore; they reached their heyday in the era of the PDP-8, Zilog Z-80, and Intel 8080.

```
load, MEM(y)          ;; load value of y into accum
add, MEM(z)           ;; add value of z into accum x = y + z
store, MEM(x)         ;; store value of accum in x (thus x = y + z)
```

Many of today's processors are *general register machines*, which replace the single accumulator with a small set of general registers in which arithmetic, logic and comparison operations occur. Motorola 68K, Intel Pentium, PowerPC, etc. are all general register processors. Frequently-used data can be stored in registers and accessed more quickly. CISC machines commonly have 8 to 16 general registers, a RISC machine might have two or three times that. Some general register machines are two-address machines since the instruction set uses an opcode and at most two operand addresses. Others use three-address code where the result and two operands can be specified in the instructions.

```
load R1, MEM(y)       ;; load value of y into register #1
load R2, MEM(z)       ;; load value of z into register #2
add R1, R2            ;; add reg #1 to reg #2, store result in reg#1
store R1, MEM(x)      ;; store result in z (thus x = y + z)
```

A variant on the general register machine is the *register window machine*, which has a facility for saving and restoring groups of registers to memory, allowing the machine to appear as though it had many more registers that are physically available on the chip, in a sort of "virtual memory" like fashion. Sun Sparc and Knuth's MMIX are examples of register window architectures.

A *stack machine* has no accumulator or registers and instead uses only a stack pointer that points to the top of a stack in memory. All operations except data movement are performed on operands at the top of the stack. Thus, these are referred to as zero-address machines. The Java virtual machine is an example of a stack machine, so are many of the HP calculators.

```
push MEM(y)           ;; load value of y onto top of stack
```

```
push MEM(z)          ;; load value of z onto top of stack
add                  ;; pop top 2 operands, add, and push result
store MEM(x)         ;; store top in z (thus x = y + z)
pop                  ;; remove result from stack
```

Which of the architectures do you think is the fastest? Which is in most common usage?

Addressing modes

The *addressing mode* refers to the way an operand is specified. In other words, "How does the add instruction identify which numbers to add?" There are dozens of possible addressing modes, but they all generally fall into one of the categories below. Under what circumstances would each of these be useful?

direct: a register contains the operand value

indirect: a register contains an address that specifies a storage location of a direct address or another indirect address

indexed: an address that is modified by the contents of a register or a memory value

Some of the more exotic ones might include register indirect with offset and update (which is available on the PowerPC).

The compiler back end: code generation

In order to understand the back-end of a compiler, we need to review assembly language since this is our target language. We will do this by defining a simple general assembly language for an accumulator machine. Our mythical machine has a single ALU register R with three condition codes (GT, EQ, LT) used for comparison results. The instructions are encoded using a 4-bit opcode, and we have exactly 16 different instructions in our set:

op code	op code mnemonic	meaning
0000	load x	R = Mem(x)
0001	store x	Mem(x) = R
0010	clear x	Mem(x) = 0
0011	add x	R = R + Mem(x)
0100	increment x	Mem(x) = Mem(x) + 1
0101	subtract x	R = R - Mem(x)
0110	decrement x	Mem(x) = Mem(x) + 1
0111	compare x	Sets condition code of ALU if Mem(x) > R then GT = 1 if Mem(x) = R then EQ = 1 if Mem(x) < R then LT = 1
1000	jump x	set PC (program counter) to location x
1001	jumpgt x	set PC to x if GT = 1
1010	jumpeq x	set PC to x if EQ = 1
1011	jumplt x	set PC to x if LT = 1
1100	jumpneq x	set PC to x if EQ = 0
1101	in x	read standard input device, store in Mem(x)
1110	out x	output Mem(x) to standard output device
1111	halt	stop program execution

A *pseudo-op* is an entry that does not generate a machine language instruction but invokes a special service of the assembler. Pseudo-ops are usually preceded by period.

`.data` builds signed integers

```
FIVE: .data -5
```
generate binary representation for -5
put it in next available memory location
make label "FIVE" equivalent to that memory address

Data is usually placed after all program instructions, so the format of an executable program is:

```
.begin
    assembly instructions
halt
    data generation pseudo-ops
.end
```

Example assembly programs

Some simple arithmetic:

Source program

```
B = 4
C = 10
A = B + C - 7
```

Generated assembly

```
        .begin
        load FOUR
        store B
        load TEN
        store C
        load B
        add C
        subtract SEVEN
        store A
        halt
A:      .data   0
B:      .data   0
C:      .data   0
FOUR:   .data   4
TEN:    .data   10
SEVEN:  .data   7
        .end
```

Routing control through an if/else, using input and output:

Source program

```
Read x
Read y
if x >= y then
      Print x
else
      Print y
```

Generated assembly

```
        .begin
        in x
        in y
        load y
        compare x
        jumplt PRINTY
        out x
        jump DONE
PRINTY: out y
DONE:   halt
x:      .data   0
y:      .data   0
        .end
```

A simple loop:

Source program

```
Set sum to 0
```

Generated assembly

```
.begin
```

```
Read N                              clear SUM
repeat until N < 0                  in N
      add N to sum            AGAIN:  load ZERO
      Read N                          compare N
end loop                              jumplt NEG
Print sum                             load SUM
                                      add N
                                      store SUM
                                      in N
                                      jump AGAIN
                              NEG:    out SUM
                                      halt
                              SUM:    .data   0
                              N:      .data   0
                              ZERO:   .data   0
                                      .end
```

The assembler

Before we can execute the program, we must convert the assembly to machine code. The assembly code is already machine-specific, so it doesn't need much conversion, but it is expressed using symbolic names (load, store, add, etc.) and labels. These need to be translated to actual memory locations and the instructions need to be binary encoded in the proper format. The *assembler* is the piece that does this translation. The assember's tasks are:

1) convert symbolic opcodes to binary (just a lookup of name to find matching code)
2) convert symbolic addresses to binary: each symbol is defined by appearing in label field of instruction or as part of .data pseudo-op. In an assembler, two passes over the code are usually made; the first pass is for assigning addresses to each instruction and for building a symbol table where it stores each symbol name and its address. During the second pass, the source program is translated into machine language: the opcode table is used to translate the operations, and the symbol table is used to translate the symbolic addresses.

```
                .begin
LOOP:           in x                0 (assigned location)
                in y                2
                load x              4
                compare y           6
                jumpgt DONE         8
                out x               10
                jump LOOP           12
DONE:           out y               14
                Halt                16
X:              .data  0            18
Y:              .data  0            20

symbol table:
                LOOP 0
                DONE 14
                X 18
                Y 20
```

If instruction format is 4-bit opcode followed by 12-bit address, the first instruction is encoded as:

```
in x        1101  0000 0001 0010
            opcode  operand address
```

3) perform assembler service requested by pseudo-ops

4) output translated instructions to a file. During the second pass, the assembler converts .data constants to binary and writes each instruction to a file. This file is called an object file.

The linker

The linker is a tool that takes the output file(s) from the assembler and builds the executable. The linker is responsible for merging in any library routines and resolving cross-module references (i.e when a function or global variable defined in one translation unit is accessed from another). An executable is said to be *statically* linked if all external references are resolved at link-time and the executable contains all necessary code itself. For executables that need large libraries (e.g. Xwindow routines), it is common to *dynamically* link the executable to those libraries so that the executable does not contain the contents of the library, but it instead references to it will be resolved at runtime on an as-needed basis.

The loader

The *loader* is the part of the operating system that sets up a new program for execution. It reads instructions from the object file and places them in memory. An *absolute* loader places each line from the object file at the address specified by the assembler (usually just a sequence from 0 upwards). (You might ask yourself: how might a relocatable loader work?) When this is completed, it places the address of the first instruction in the program counter register. The loader also usually sets up the runtime stack and initializes the registers. At that point the processor takes over beginning the "fetch/decode/execute" cycle.

Data representation

Simple variables: are represented by sufficient memory locations to hold them:

char: 1 byte
integers: 2 or 4 bytes
floats: 4 to 16 bytes
booleans: 1 bit (but usually 1 byte)

If the variable is global, it will be stored in the data segment (just following the code segment in memory). If they are local, they are located in the activation record of the runtime stack.

Pointers: are usually represented as unsigned integers. The size of the pointer depends on the range of addresses on the machine. Currently almost all machines use 4 bytes to store an address, creating a 4GB addressable range. There is actually very little distinction between a pointer and a 4 byte unsigned integer. They both just store integers— the difference is in whether the number is interpreted as a number or as an address.

One dimensional arrays: are a contiguous block of elements. The size of an array is at least equal to the size of each element multiplied by the number of elements. The elements are laid out consecutively starting with the first element and working from low-memory to high. Given the base-address of the array, the compiler can generate code to compute the address of any element as an offset from the base address:

```
&arr[i] = arr + 4 * i
```
(4-byte integers; arr = base address)

Global static arrays are stored in the data segment. Local arrays may be on stack if the language supports such a feature. What about dynamic arrays?

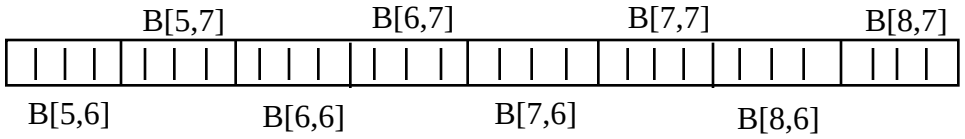
Multi-dimensional arrays: one of two approaches, either a true multi-dimensioned array, or an array of arrays strategy.

1) As one contiguous block. In a row-major configuration (C, Pascal), the rows are stored one after another. In a column-major (Fortran), the colums are stored one after another. For a row-major array of m rows x n columns integer array, we compute the address of an element as `&arr[i,j] = arr + 4 * (m * i + j)` or in a more general form for the array:

   `array[`L_1`..`U_1`, `L_2`..`U_2`]` (in Pascal form)

   `&arr[i, j] = arr + (sizeof T) *((i -` L_1`)*(`U_2`-`L_2`+1) + (j-` L_2`))`

   For example, the array `B[5..8, 6..7]` of integers would look like this:



   (4 bytes per integer)

   L_1 = 5; U_2 = 7, L_2 = 6

   &B[8,6] = B + 4 * ((8-5)*(7-6+1) + (6-6)) = B + 24

2) As a array of arrays, i.e., we have a vector with one pointer element for each row. Each element of this vector holds the address of a vector for the corresponding row. We use the following formula to find the location of the address of the ith row: B = A + 4 * i. The jth element in this vector is at byte address: B + 4 * j. How can this method be extended to arrays of more than two dimensions?

Structs: are laid out by allocating the fields sequentially in a contiguous block, working from low-memory to high. The size of a record is equal to at least the sum of the field sizes. In the example below, if integers require 4 bytes, chars 1 and doubles 4 bytes, 8 bytes, we would need at least 13 bytes with the individual fields at offsets 0, 4 and 5 bytes from the base address. Why, then, on most machines, will it reserve 16 bytes for this struct?

```
struct example {
   int index;
   char type;
   double length;
};
```

Objects: are very similar to structs, where the fields are the instance variables for the class. Methods are not stored in the object itself, but usually there is a hidden extra pointer with each object that references shared class-wide information (often called the *vtable*) that provides information about the class methods.

Instructions: themselves are also encoded using bit patterns, usually in native word size. The different bits in the instruction encoding indicate things such as what type of instruction it is (load, store, multiply, etc.) and the registers involved to read or write from.

Storage classes

The *scope* of a variable defines the lexical areas of a program in which the variable may be referenced. The *extent* or *lifetime* refers to the different periods of time for which variable remain in existence.

global: exist throughout lifetime of entire program and can be referenced anywhere.

static: exist throughout lifetime of entire program but can only be referenced in the routine(or module) in which they are declared.

local: (also called *automatic)* exist only for the duration of a call to the routine in which they are declared; a new variable is created each time the routine is entered (and destroyed on exit). They may be referenced only in the routine in which they are declared. Locals may have even smaller scope inside an inner block within a routine (although they might "exist" for the entirety of the routine for convenience of the compiler, they can only be referenced within their block).

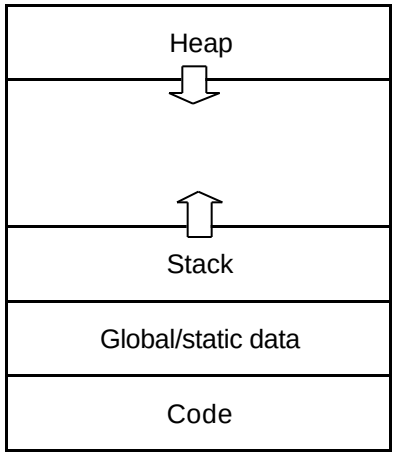
dynamic: variables that are created during program execution; usually represented by an address held in a variable of one of the above classes. Extent is the lifetime of the program (unless explicitly returned to the system); scope is the scope of the variable used to hold its address.

Both global and static variables have a single instance that persists throughout life of program and the two classes vary only in scope. The usual implementation for these is a collection of memory locations in the global/static data segment of the executable. These locations are fixed at the end of the compilation process.

Local variables only come into existence on entry to a routine and persist until its exit. To handle these we use a runtime stack that holds the values of locals. The area of memory used to hold all the locals of a routine is called the stack frame. The stack frame for the routine currently executing will be on top of the stack. The base address of the current stack frame is usually held in a machine register and is called the stack pointer.

Dynamic allocation of further storage during the run of a program is done by calling library functions (e.g., malloc()). This storage is obtained from memory in a different segment than the program code, global/static, or stack. Such memory is called the heap.

Here's a map depicting the *address space* of an executing program:



Runtime stack

Each active function call has its own unique stack frame. In a stack frame (activation record) we hold the following information:

1) dynamic link: pointer value of the previous stack frame so we can reset the top of stack when we exit this routine. This is also sometimes called the frame pointer.
2) static link: in languages (like Pascal but not C) that allow nested routine declarations, then a routine may be able to access the variables of the routine(s) within which it is declared. In the static link, we hold the pointer value of the stack frame in which the current pointer was declared.
3) return address: point in the code to which we return at the end of execution of the current routine.
4) values of arguments passed to the routine and locals declared in the routine.

Here is what typically happens when we call a routine (every machine and language is slightly different, but the same basic steps need to be done):

Before a function call, the calling routine:

1) saves any necessary registers
2) pushes the arguments onto the stack for the target routine
3) set up the static link (if appropriate)
4) pushes the return address onto the stack
5) jumps to the target routine

During a function call, the target routine:

1) saves any necessary registers
2) sets up the new frame pointer (dynamic link)
3) makes space for any local variables
4) does its work
5) tears down its frame
6) restores the previous static and dynamic link
7) restores any saved registers
8) jumps to saved return address

After a function call, the calling routine:

1) removes return address and parameters from the stack
2) restores any saved registers
3) continues executing

The above description represents parameter passing by value (the values of variables are passed and therefore cannot be modified in the subroutine). How would you implement parameter passing by reference?

Memory management

Typically, the program requests memory from the operating system, which usually returns it in pages. A page is a fairly large chunk anywhere from four to eight KB chucks, and sometimes even more. The program then divides up this memory on its own. There is quite a bit of research on what makes a fast allocator that minimizes fragmentation on a page. Your allocator quite likely is very smart, and keeps track of how much memory is allocated per pointer and so on.

```
a = malloc(12);   // give me 12 bytes
a = b;            // oops, lost it!
```

By now everyone has heard that Java has garbage collection, where the programmer doesn't have to free dynamically allocated memory herself. The run-time environment detects that an allocated chunk of memory can no longer be referenced by the program and disposes of the memory for her. There are several methods for implementing garbage collection. The two most common are reference counting and mark and sweep

Automatic storage management is is very convenient for the programmer, but sometimes causes problems. Why isn't garbage collection a panacea for memory management? What semantic rules are needed to for a garbage-collected language?

Bibliography

A. Aho, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques, and Tools, Reading, MA: Addison-Wesley, 1986.

J.P. Bennett, Introduction to Compiling Techniques. Berkshire, England: McGraw-Hill, 1990.

J.P. Hayes, Computer Architecture and Organization. New York, NY: McGraw-Hill, 1988.

A. Pyster, Compiler Design and Construction. New York, NY: Van Nostrand Reinhold, 1988.

G.M.Schneider, J. Gersting, An Invitation to Computer Science, New York: West, 1995.

J. Wakerly, Microcomputer Architecture and Programming. New York, NY: Wiley, 1989.