Computer Science Lecture Notes

Job Hernandez Lara

April 1, 2023

Contents

Abstract

This document represents lecture notes for a set of computer science textbooks that cover the core of the computer science degree curriculum at the University of Washington in Seattle. The fundamentals of the curriculum include: CSE 143 Computer Programming 2, CSE 311, 312 Discrete math, CSE 331 Software Design and Implementation, Introduction to Algorithms, and CSE 351 Hardware/Software Interface whereas the core of the University of Washington CS curriculum include about 8 upper level courses; but in these lecture notes I will cover compilers, and operating system in addition to the fundamentals.

1 Discrete Math

1.1 Constructing Direct Proofs

A conditional statement is statement written as $P \implies Q$ where P is the hypothesis and Q is the conclusion. Intuitively, $P \implies Q$ means that Q is true whenever P is true; in other words, if P is true then it follows necessarily that Q is true. Here is the truth table for conditional statements:

P	Q	$P \implies Q$
T	T	T
T	F	F
F	T	T
F	F	T

A direct proof is a type of proof in which the mathematician demonstrates that one mathematical statement follows logically from definitions and previously proven statements. To prove that a conditional statement $P \implies Q$ is true we only need to prove the Q is true whenever P is true. Why? Because $P \implies Q$ is true whenever P is false. If you take a look at the truth table for conditional statements you will notice that $P \implies Q$ is false when P is true and Q is false. So, by demonstrating that Q is true whenever P is true we can prove the statement because you are guaranteed that the statement is true. There is a technique for proving by this method called the **know-show-table.** In this technique we work forward from the hypothesis and backwards from the conclusion and we try to connect the two by building a chain of reasoning. You start with the conclusion and ask "under what conditions is the conclusion true?" and then work backwards. Suppose we are given the statement: *If x and y are odd integers, then $x \cdot y$ is an odd integer.* Here is the table for this statement:

Step	*Know*	*Reason*
P	x and$y$are odd integers	Hypothesis
$P1$	There exists integers m and n such that $x = 2m + 1$ and $y = 2n + 1$	Definition of an odd integer.
$P2$	$xy = (2m + 1)(2n + 1)$	Substitution
$P3$	$xy = 4mn + 2m + 2n + 1$	Algebra
$P4$	$xy = 2(2mn + m + n) + 1$	Algebra
$P5$	$2mn + m + n$ is an integer	Closure properties of the integers
$Q1$	There exists an integer q such that $xy = 2q + 1$.	Use $q = (2mn + m + n)$
Q	$x \cdot y$ is an odd integer.	Definition of an odd integer
Step	*Show*	*Reason*

In the above **know-show-table** we ask, by working backwards (i.e from the conclusion), we ask "How do we prove that an integer is odd?" and then we continue asking the same question until we can connect it with the hypothesis.

2 Introduction to Algorithms

3 Software Construction

4 Hardware/Software interface

4.1 Representing and Manipulating Information, chap 2

Bytes are the smallest addressable unit of memory; a byte is 8 bits. In other words the byte is the fundamental unit of information. The processor of your computer only processes bytes; in fact, consider the following quote from the book "Computer Systems - A Programmer's Perspective": "A fundamental concept of computer systems is that a program, from the perspective of the machine, is a sequence of bytes; moreover, another key idea is that bytes are the unit with which machines communicate with one another through networks. A machine level program interacts with the virtual memory which is a large array of bytes. The machine level program does not interact directly with the physical memory. Every byte of memory has an address - the set of all addresses is called the virtual address space. Interestingly, a multi-byte object is stored as a contiguous sequence of bytes; for example, in 64 bit architecture an integer in C is 4 bytes; if the address of the given int is 0x100, the given int will be stored in memory locations 0x100, 0x101, 0x102, and 0x103.

4.2 Machine Level Representation, chap 3

It is important to understand machine level representation because it will enable you to analyze how efficient your code is. By understanding how your code will be compiled to assembly code you will have a much better understanding of the performance of your program. When you are working with assembly code you are directly manipulating the 16 CPU registers; you are also interacting with the program counter. Now lets explore how different high level programming constructs are represented in assembly. Lets start with if statements. Consider this C program:

```
long lt_cnt = 0;
long ge_cnt = 0;
long absdiff_se(long x, long y)
{
        long result;
        if (x < y) {
                lt_cnt++;
                result = y - x;
        }
```

```
        else {
                ge_cnt++;
                result = x - y;
        }
        return result;
}
```

The above C code gets compiled to the following x86-64 assembly through compiler passes. The assembly that the above code generates is this:

```
long absdiff_se(long x, long y)
x in %rdi, y in %rsi
        absdiff_se:
                cmpq    %rsi, %rdi      Compare x:y
                jge     .L2             If >= goto x_ge_y
                addq    $1,lt_cnt(%rip) lt_cnt++
                movq    %rsi, %rax
                subq    %rdi, %rax      result = y - x
                ret                     Return
        .L2:                        x_ge_y:
                addq $1, ge_cnt(%rip)   ge_cnt++
                movq %rdi, %rax
                subq %rsi, %rax         result = x - y
                ret                     Return
```

Now, lets consider the assembly generated by writing a while loop:

```
long fact_while(long n)
{
        long result = 1;
        while (n > 1) {
                result *= n;
                n = n-1;
        }
        return result;
}
```

The above code generates the following assembly:

```
long fact_while(long n)
        n in %rdi
        fact_while:
          movl  $1, %eax        Set result = 1
          jmp   .L5             Goto test
        .L6:                loop:
          imulq %rdi, %rax       Compute result *= n
          subq  $1, %rdi        Decrement n
        .L5:                test:
          cmpq  $1, %rdi         Compare n:1
          jg    .L6             If >, goto loop
          rep; ret              Return
```

Before saying something about how procedures get compiled to assembly code, I would like to say something about the stack. The stack is dynamic – it grows as functions get called and the space that was allocated will be freed as the function returns. Stack memory behaves just like the stack data structure – i.e., last-in, first out. Here is the example in C:

```
long swap_add(long *xp, long *yp)
```

```
{
        long x = *xp;
        long y = *yp;
        *xp = y;
        *yp = x;
        return x + y;
}
long caller()
{
        long arg1 = 534;
        long arg2 = 1057;
        long sum = swap_add(&arg1, &arg2);
        long diff = arg1 - arg2;
        return sum * diff;
}
```

The above code gets compiled to the following assembly:

```
long caller()
        caller:
        subq    $16, %rsp          Allocate 16 bytes for stack frame
        movq    $534, (%rsp)       Store 534 in arg1
        movq    $1057, 8(%rsp)     Store 1057 in arg2
        leaq    8(%rsp), %rsi      Compute &arg2 as second argument
        movq    %rsp, %rdi         Compute &arg1 as first argument
        call    swap_add           Call swap_add(&arg1, &arg2)
        movq    (%rsp), %rdx       Get arg1
        subq    8(%rsp), %rdx      Compute diff = arg1 - arg2
        imulq   %rdx, %rax         Compute sum * diff
        addq    $16, %rsp          Deallocate stack frame
        ret Return
```

4.3 The Memory Hierarchy, chap 6

Memory can be seen as layers of increasingly slower memory layers; at the top of the memory hierarchy you have registers, and then cache layers, then the memory layer, and then the disk, and the disk in remote network servers. Each level k, starting from L0, uses level k+1 to retrieve data. So, interestingly the local disk retrieves from the disk in the remote servers; this was insightful for me because it connected network applications such web apps to the rest of the system. With respect to clock cycles, the processor takes 0, 4, hundreds of cycles to read from registers, cache, and memory respectively. This is why register allocation is so important in compilers and why register allocation improves the compiler performance – i.e., it is more efficient to retrieve from registers than memory. As programmers, if we want to write efficient programs we should aim at writing programs with good locality. A program has good temporal locality if the instructions in a loop execute during each iteration of the loop whereas spatial locality refers when memory is accessed sequentially. Consider the following program.

```
int sumvec(int v[N])
        {
        int i, sum = 0;

        for (i = 0; i < N; i++)
                sum += v[i];
        return sum;
}
```

'sumvec' has good temporal locality because the variable 'sum' is referenced during each iteration of the loop but since 'sum' is a scalar theres no spatial locality. On the other hand since "v" is read (fetched) sequentially during all iteration of the loop it has good spatial locality but it has poor temporal locality because each element of "v" is accessed only once. A given function is said to have good locality if the variables have either temporal locality or spatial locality. Another quality of 'sumvec' is that it has a stride-1 reference pattern. A function has a stride-1 reference pattern if each element of an array is visited sequentially. If the kth element of an array is visited then it has a stride-k reference pattern. The higher the stride the less spatial locality. Now lets consider a 2-dimensional array.

```
int sumarrayrows(int a[M][N])
{
        int i, j, sum = 0;

        for (i = 0; i < M; i++)
                for (j = 0; j < N; j++)
                        sum += a[i] [j];
       return sum;
}
```

'sumarrayrows' enjoys good spatial locality because it references the array in row-major order; that is, it references one row at a time which is how C arrays are laid out in memory. 'sumarrayrows' has a stride-1 reference pattern. You probably have heard about loop interchange, a compiler optimization. Here is an example of how interchanging gives rise to worst performance.

```
int sumarraycols(int a[M][N])
{
        int i, j, sum = 0 ;

        for (j = 0; j < N; j++) {
          for (i = 0; i < M; i++) {
                sum += a[i] [j];
        }
    }
    return sum;
}
```

This function above has poor spatial locality because it references the array column by column so it has a stride-N reference pattern. As to what cache hits and cache misses mean, a cache hit is when a program looks for an item d in level k+1 but finds d in level k. Conversely, a cache miss is when d is not found in level k but instead is found in level k+1. Recall that higher levels in the memory hierarchy are faster. Here is a quote from "Computer Systems, A Programmer's Perspective": "Programs with better locality will tend to have lower miss rates, and programs with lower miss rates will tend to run faster than programs with higher miss rates. Thus, good programmers should always try to write code that is cache friendly, in the sense that it has good locality." With respect to how compilers work, any respectable optimizing compiler will cache local variables with good temporal locality in the register file. "In general, if a cache has a block size of B bytes, then a stride-k reference pattern (where k is expressed in words) results in an average of min (1, (word size $\times$ k)/B) misses per loop iteration. This is minimized for k = 1, so the stride-1 references to v are indeed cache friendly."

4.4 Exceptional Control Flow, chap 8

I was familiar with this chapter because I have basic OS knowledge but a few things stood out. Firstly, the processor has control flow. From the time the computer gets turned on to the time it shuts down the processor fetches instructions and executes them one at a time. This sequence is the control flow. The control flow can be altered by the program state - i.e., jumps, branches, call, return. But other changes to the control flow react to system changes such as data arriving from disk or network adapter. This type

of change in the control flow due to system changes is known as exceptional control flow; there is two mechanisms for exceptional control flow: exceptions and higher level mechanisms such as context switch. Exceptions happen in response to a significant change in the processors state – e.g., virtual memory page fault occurs, an arithmetic overflow occurs, or an instruction attempts a divide by zero. Control flow passes from one process to another via context switch.

5 Compilers

6 Operating Systems

6.1 Processes and Threads

A process is an abstraction for a program in execution; each process has it own address space and its memory is layed out as follows: text section, data section, heap and stack. The text section consists of the executable code, the data section consists of global variables, the stack consists of data storage associated with invocations of functions such as parameters, return address, and the heap is associated with dynamically memory storage such as when you allocate memory in C with malloc. A thread is the execution control center of a given process. There could be multiple threads per process. A thread has its own stack but shares the same address space with other threads. This makes sense because threads need to be aware of the same program.

6.2 Interprocess Communication

Often process needs to communicate with other process because they may need to access main memory. This can lead to race conditions. A race condition happens when two or more processes are sharing data (from main memory for instance) and the final result depends on the order of the processes. If one process runs before another process it may change the result because the order is off. How do we avoid race conditions? The answer is mutual exclusion; mutual exclusion dictates that when two or more processes share memory the processes that are not using the shared data are excluded from accessing the data. Mutual exclusion is achieved by semaphores, mutexes. A semaphore is initialized to the number of resources; when a process wants to use a resource it calls the wait() function thereby decrementing the count. When the process releases a resource it performs a signal() thereby incrementing the count. When the count is 0 processes are blocked until it increments again. When a process modifies a semaphore value another process cannot access the same semaphore value – i.e., cannot access the same resource.

6.3 Memory Management

The job of the memory manager is to keep track of the parts of memory that are being used, to allocate memory to processes and to deallocate memory when the processes are done with it. How does the memory manager accomplish this? It does this through an abstraction mechanism known as virtual memory. As discussed above a process is an abstraction for a program; likewise, an address space is abstraction of main memory – it gives the illusion that each process has its own main memory. The set of addresses that a process can use to address memory is called the address space. A further extension of this is called virtual memory whereby a computer can work with programs that have more memory than what the computer is capable; the basic idea behind main memory is that a process has its own address space which is layed out as pages where a page is a contiguous range of memory addresses. How does the memory manager manage free memory? Free memory is managed by a bitmap; in a bitmap memory is divided by allocation units where each allocation unit is either free or being used. A free unit is denoted by a bit 0 whereas a unit being used is denoted bit 1. When a k unit process asks for memory the memory manager needs to allocate consecutive k 0 bits. A further question to ask now is how is this implemented in C. One thing to keep in mind is the programs refer to memory addresses which programming in assembly makes explicit with the statement ‘mov 1000, reg‘. So, programs generate addresses which are called virtual addresses if a system has virtual memory. Virtual address then get translated to physical memory addresses.