Week 7 Monday | Computer Science 50
November 03, 2008 | Fall 2008
Scribe Notes | Anjuli Kannan

Announcements (0:00 – 5:00)

A Simple Voting Program (5:00 - 8:00)

- David's e-voting program asks you repeatedly to enter 1 for Obama or 9 for McCain, then prints the tally at the end
- Note the design decision to make the numbers for the two candidates far apart, to address the fat finger error
- Also creates a paper trail by printing votes to a file as they are entered
- After voting is done, this file can be piped through grep (finds a word) and wc (counts words) to doublecheck the tally

Grades (8:00 – 12:45)

Valgrind (12:45 – 18:45)

- A useful tool to find memory-related bugs
- In memory.c, we see two bugs in the function f()
    - Never free memory that is malloc-ed
    - Indexing outside of the boundaries of the array
- But when we run this program, we don't get a seg fault
- Seg faults do not always happen when you go out of bounds
- As for the memory leak, these usually will not become evident unless the program is running for a very long time to the point that there is a noticeable lag
- We can use valgrind to check for memory leaks by running the following command:

    valgrind –v –leak-check=full memory
- The relevant part of the print out is where it says "invalid write of size 4"
- The print out also tells us the problematic line and function, so we can locate and remove the error
- We also see under "LEAK SUMMARY" that we've lost 40 bytes in one block
- This refers to the array that we malloc-ed and did not free

Hex (18:45 – 24:30)

- Just as decimal is base 10 and binary is base 2, hexadecimal is base 16
- This means we have 16 counting digits: 0, 1, …, 9, a, b, c, d, e, f
- This is useful because we can take 4 bits and represent them with a single hex digit
- For instance, 0001 in binary is 0x1 in hex, and 1111 in binary is 0xf in binary (15 in decimal)
- (0x is notation meaning "here comes a hex number")
- We can quickly convert between hex and binary by expanding each hex digit to four binary digits: 0xff = 1111 1111 (255 in decimal) since 0xf = 1111

- Why bother using hex at all?
  - Convenient to represent a byte with just 2 hex digits
  - This means we can represent a 32 bit address with an 8-character string
  - Simpler mapping between hex and binary than between decimal and binary

Endian-ness (24:30 – 42:30)

- If a 4 byte value is laid out from left to right (highest order bits are at lowest memory address), the processor is big-endian.
- If a 4 byte value is laid out right to left (lowest order bits are stored at the lowest memory address), the processor is little-endian. See slide 3.
- Your computer probably is little-endian.
- This can lead to some subtle bugs.
- Look at endian.c.
- This program, given a .bmp file, should look at the header of the file and print out its size, which is contained in the header
- fseek is used to move to a particular point in the file
- fread reads from that location into the variable bfsize
- bfsize is then printed
- Next, we "rewind" to the beginning and read in 14 raw bytes
- We want to print starting from 2 bytes after buffer, so add 2 to buffer
- Then, we want to print out 4 bytes at once, so cast the pointer from a char * to a long *. Then, dereference the long * to go to the location and read the 4 bytes
- Next, we print the bytes individually using buffer[2], … , buffer[5], in decimal, hex, and finally binary
- From the output we see that if we read in the 4 bytes as a long, and print it out, we are given 58
- But if we print out the four bytes individually, we see that they are, from right to left: 58 0 0 0, or 0x3a 0x00 0x00 0x00, or 0011 1010 0000 0000
- As this demonstrates, the bytes that compose value 58, which would be 0000 0000 1010 0011 in binary, are laid out right to left as 0011 1010 0000 0000
- If you didn't know that it was little-endian, you would read it as 11101000000000
- But fread has apparently accounted for this and known to switch the order of the bytes before converting to decimal

Bitwise Operators (42:30 – 57:30)

- Bitwise operators compute the result of performing an operation bit by bit on inputs
- We will see logical operators and (&), or (|), not (~), xor (^)
- Refer to truth tables if you are unfamiliar with any of these operators
- Can also do left shift (<<) and right shift (>>)

- These shift the bits by the number specified, and pad the "empty space" remaining with zeroes
- Left shift is an efficient way to double a number
- Left shift also good for making a mask when you only want to look at some particular bits
- Referring back to endian.c, look at inner loop. Here we desire to print out each bit. To do this, we "bitwise and" the full byte with each of 8 masks: 10000000, 01000000, 00100000, etc. (The masks are computed by repeatedly doing a left shift on 1)
- To perform a bitwise "and" on, for example, 11111111 and 10000000, we go from left to right and-ing the bits in the same column to get the result for that column. So 11111111 & 10000000 = 10000000. It has ones only where there is a one in both inputs.
- Each time, if we get a nonzero result from mask & buffer[i], we know there was a 1 in buffer[i] at the location where there is a one in the mask.
- Home exercise: How can you swap two variables using bitwise operators only?

Hash Tables (57:30 – 77:00)

- What was frustrating about arrays? Growth.
- Growth became easier with linked lists. What was a disadvantage of linked lists? Waste memory on metadata, give up random access.
- In general, we want to address the problems of linear time lookup, insertion, and deletion.
- We can get closer to constant time operations using hash tables.
- Suppose we want to store students who are identified by ID numbers.
- In hash tables, we fix the size of our array at a size n. Suppose n = 6.
- As we get students, we want to put them into the array based on their ID numbers.
- We can put student 1 in the first slot, and student 2 in the second slot, but what about when student 7 comes along?
- One option is just to put him in the next available slot (the third slot).
    - Then our algorithm is, given a student, put him in the next available slot.
    - In the worst case, this is O(n) insertion. But we can be smart and keep track of the next available slot, resulting in constant time.
    - Lookup, however, remains linear because, when we go to look for student 7, we have no idea where he'll be.
- Another option is to put student X in slot X mod n.
    - This will map every integer to a number between 0 and 6, which corresponds perfectly to the slots of the array.
    - This is also deterministic, so it should make finding a student more efficient than it was with the previous algorithm.
    - In this case, we take student 7, and find that 7 mod 6 = 1. But there is already a number in slot 1. What to do?
        - One option is two put it in the next available slot after slot 1.

- This is called linear probing.
- The algorithm for insertion is: take X, map to h(X) using some hash function h(.), then put in next available slot after slot h(X)
- The algorithm for lookup is: take X, map to h(X) and start a linear search from slot h(X) until an empty slot is hit
- Both of these are O(n), but in practice, are much closer to constant time.

▪ How much closer to constant time? This depends on a few things, including the function and the size of the array. It also depends on the number of collisions we expect while putting elements in the array.

▪ The problem of collisions (in a group of m values, what is the probability that 2 or more map to the same of n hash values?) is analogous to the birthday problem (in a group of m people, what is the probability that 2 have the same birthday?)
  - If you are not familiar with the birthday problem see its entry on Wikipedia.
  - You may be surprised to learn that you need only 23 people to have a >50% probability that 2 or more people have the same birthday
  - We can use similar math to figure out the probability of collisions for a particular hash table.

▪ Another option is to do linear probing, but to leave a pointer in the place where 7 originally mapped to the place where 7 ended up
  - This is called coalesced chaining

▪ A third option is to put a linked list in each slot. Then when 7 maps to slot 1, just hook him on to the growing linked list there.
  - This is called separate chaining.
  - If the hash function is good, the list in each slot will be roughly the same size, n/m.
  - Then insertion, deletion, and lookup are O(n/m)