CSE13S notes
Week of 1/30/23

Assignment 4: Life
Dynamic memory allocation (DMA)

- The act of allocating memory for variables on the heap (called free store) during program run-time
- Quite different than compile time (static) allocation:
  - Compile time allocation (CTA) means memory for named variables is allocated by the compiler when it compiles
  - Dynamic memory allocation (DMA) allows for run-time allocation
  - CTA requires the exact size and type of storage to be known. DMA is calculated and allocates the exact memory it needs during run-time
- DMA is allocated from a region of memory known as the heap

Interesting thing to do

- Loop a malloc * 1000000

Why is DMA good?

- Stack space is limited (~4mb)
- We sometimes want variables to last beyond the lifetime of its current scope
  - Variables on the stack don't last beyond it's current scope
  - Variables that are DMA'd last beyond current scope
- We don't always know how much memory is needed to run a prorgam
  - Solution: dynamically allocate memory when it's needed, however much is needed
  - But how do we dynamically allocate memory?

DMA: MALLOC(), CALLOC(), REALLOC()

- There are three standard C library functions to allocate memory, all of which are defined under stdlib.h
  - malloc(), calloc(), realloc()
    - These three dynamically allocate memory on the heap and returns a pointer to the allocated memory
- Allocated memory must be freed using free()
  - Not freeing allocated memory can lead to depletion of system resources

Heap: a large region of unmanaged, anonymous memory

- Only limitation are hardware (or virtual hardware) limitations
- Slower to read from/ write to due to the need for pointers

- Variables using heap memory can be accessed globally with access to the pointer
    - A benefit of using pointer; much easier to pass around pointers for large data structures
- Possible memory fragmentation can occur over time as blocks of memory are allocated and deallocated

Malloc()
- Defined as:
    - void* malloc(size_t size)

Tracking memory usage:
- See slides

Tracking memory usage: Bitmaps
- Keep track of free / allocated memory regions with a bitmap
- One bit in the bitmap corresponds to a “page” or region of memory
- Bitmaps are a constant size
    - Chunk size determines efficiency
- Bit corresponding to page set to 1 means that there is something occupying that page of memory

Allocating memory
- Search through region list to find a large enough space
- Suppose there are several choices: which one to use?
    - First fit: the first suitable hole on the list
    - See slides
- Next > first > worst > best: in order from best to worst fits

Buddy allocation
- Allocates memory in powers of 2
    - Good for objects of different sizes
- Split larger chunks to create smaller chunks that are enough memory for the object

Freeing memory
- Allocation structures must be updated when memory is fred

Calloc()
- Defined as

    - void* calloc(size_t nmemb, size_t size)
- [nmemb] denotes the number of objects, and [size] the size of each object
- Returns a pointer to [nmemb] * [size] bytes of allocated memory on the heap, in which each byte has been initialized to zero
- Like malloc(), behavior when [nmemb] is zero is implementation defined
- Generally slower than malloc(), but the tradeoff is that contents of the allocated memory are known since it's zeroed out
- Basically, instead of malloc (size(uint32_t) * n), you can do calloc (n, sizeof(uint32_t))

A dynamic n x n matrix
- To create a multidimensional array, you need to allocated memory for each column
- Essentially, you need an array of pointers
- You need to also create a destructor function - using a bunch of frees at the end won't work out

Realloc(void ptr, size_t size)
- Reallocated [ptr] to newly point at [size] bytes of allocated memory on the heap
    - This is not strictly true; what realloc() actually does
    - If size given to free is larger, pointer returned contained original elements + unitialized memory
    - If size given is smaller, only returns elements up size given

free(void *ptr)
- If we allocate memory, we must also be able to deallocate (or free) memory
- Another standard C library function - specifically for deallocating memory allocated by malloc, calloc, an realloc
- Deallocates memory pointed to by the pointer
- Memory leaks occur when allocated memory isn't freed
- Pointers should be set to NULL after freeing to mitigate user-after-free vulnerabilities

Valgrind
- A collection of dynamic analysis tools
- Usually used for it memcheck tool, which can detect
    - Use of uninitialized memory
    - reading/writing memory after its been freed
    - reading/writing at the end of allocated blocks of memory
    - reading/writing inappropriate areas on the stack

- Memory leaks - where pointers to malloc'd blocks are lost forever
- Useful command line falgs/options for valgrind
  - --leak-check=full
  - --show-leak-kinds= full

Static vs dynamic analyzers

- Static analyzers, like infer (and scan-build), operate by analyzing the source code for a program before it's run
  - Code is compared against a set (or multiple sets) of coding rules for bugs
  - Only surface level; can't check if a function behaves completely as it's supposed to when it's executed
- Dynamic analyzers, like valgrind, operate by tracking down errors that occur during program execution
  - Goood for checking if the program executes as its supposed to
  - Can only analyze what happens during execution
  - Things that dont occur during execution are analyzed

Security & Cryptography

- Next assignment - Smitz-moah cryptography
- Goal: render a message incomprehensible to all except the intended recipient
- Requirement: use a well-known algorithm to encrypt
  - But why?
  - Relying upon the secrecy of the algorithm is a very bad idea:
    - German Enigma: cracked
    - Japanese Purple: cracked
- Algorithm has two inputs: data & key
  - Key is known only to authorized users
- Strongest encryption algorithms are the ones that have been repeatedly attacked over the years

Cryptography basics

- Algorithms (E, D) are publicly known
- Keys (K_e, K_d) represent a shared secret
- The cipher text is shared to everyone (publicly known) but only people with the private key will be able to decrypt the ciphertext
- History: Julius caesars caesar cipher: all letters shited the right 3 digits
  - But caesar is very easily breakable

Unbreakable codes

- There is such a thing as an unbreakable code: it's called the one time pad
    - Use a truly random key that is as long as the message to be encoded
        - XOR the message with the key a bit at a time
- Code is unbreakable because:
    - Key could be any string of bits
    - The message could be anny message given the appropriate key
- Difficulties:
    - Distributing the key is a hassle
        - (may be easier because of timing?)
    - Generating truly random bits (what is truly random?)

Secure encryption algorithms (modern)

- DES: 56 bit encryption
    - Same key to encrypt and decrypt
    - 2^{55} keys required to guess, on average, but modern computers can do that
- AES: 128 bit encryption
    - Adding one bit to the key makes it twice as hard to guess
    - Requires 2^{127} keys on average
    - Modern algorithms usually aren't broken by brute force
- Attacking AES
    - Even with quantum computers, the amount of energy required to flip a bit is constant, so energy required exceeds the suns energy for 1000 years

Public-key cryptography

- Instead of using a single shared secret, keys come in pairs, that are not equivalent
- The keys are typically (but not always) inverses of one another
- Encrpytion and decryption are the same algorithm
- See slides

Diffie-Hellman Key exchange

- Works off simply math, see slides for the math

RSA cryptography

- Choose two large primes p and q:
    - These are secret, and temporary - you can throw them away
- Let n = p x q

Attacking RSA

- Requires factoring n in order to find:
    - $f(n) = (p-1)(q-1)$
    - Public key is comprised of two integers: n (public modulus) and e (public exponent)
        - Choose two large primes p and q
        - Let $n = p \times q$

Testing primality
- How can we figure out if a number is prime
    - We could try to factor it. But that requires O(sqrt(n)) trial divisions
    - We can use the sieve of

GNU multiprecision integer project
- Mpz_t: GMP integer object
    - Must be initialized before use
- Everything is an array, is pass by reference

RECURSION (RECURSION(RECURSION))
- A recursive function is defined in terms of itself
- Function calls are expensive (in terms of efficiency/speed)
- Recursion uses stack space, be careful when using it with large spaces

So, are recursive functions/algorithms inherently inefficient? (Since it uses Stack space)
- The short answer is no…
- The real answer is more subtle:
- A function call requires creating a stack frame, and that takes time and space
- All tail recursive functions can be rewritten as iteration (often by the compiler)

When to use recursion?
- The answer: only when it *makes sense*
- Use recursion when it is natural for you to express your algorithm recursively
- Do not use it when it does not make the code clearer!!!!
    - Check slides
- Binary Search!
    - We can search an ordered array in O(log(n)) time
    - If the list is empty, then it is not there. Look in the middle, if key is smaller, look in the left half, if larger check right half
- Trees

String table
- To efficiently store strings so that only one copy kept (set of strings)
- Why? Compilers have to keep track of a lot strings, such as variable names

- Check slides:
- You need to understand what you implement, so implement whatever makes sense the most - wether recursion or iteration

Recursion is natural for search
- We use recursion for searching by dividing the space up
  - Places we *have* searched and places we *have yet* to search
  - If we get stuck, we go back and try a different path (think maze pathfinding)
- Example in slides: can a knight (in chess) visit a cell in chess exactly once?
  - It can!
- Example: 8 queens (different team, hypothetically) in one chess board with none threatening the other

Recursion? (what did we learn)
- Recursion is natural
- It is good for search problems
- It is not inherently inefficient
- Like all tools and techniques, use it where it makes sense

Bit Vector part 2 electric boogaloo
- Need more than 64 bits? Use an array of units
- Quiz!!: question might be on logical shifting/rotation (C does not have rotations)

Getting a “high order” nibble
- A high order nibble is the most significant 4 bits
- Bit shift right 4 (on a 8 bit integer) times so that the high order nibble takes the place of the low order nibble
- AND with 0x0F