Unix

Unix: a set of standards and tools commonly used in software development.

The **command-line** is a text-based interface (i.e., terminal interface) to navigate a computer, instead of a Graphical User Interface (GUI).

Unix Commands

cd – change directories (..)

ls – list directory contents (-l, -a: hidden files)

mkdir – make directory

emacs – open text editor

rm – remove file or folder (-rf)

rmdir - remove empty dir

man – view manual pages

tree cs107 -F (show files and directories in tree)

pwd - output absolute path to current location

cp source dest - copy (-r to copy directory)

mv - move (rename)

cat file1 (file2 file3) print file(s one after another)

grep "binky(.*)" program.c - search text in files (. any char, * zero or more repeats of left char, ^ beginning of line, $ end of line)

find assign1 -name "*.c" - search the assign1 folder for all .c files

diff hello.c hello2.c - find the diff of two files

./hello > outputFile.txt - save output to file

>> - append the output to an existing file

diff file1.c file2.c | grep "#include" | wc -l - pipe, find # of diff lines that contain #include for two files

./addTwoNumbers < twoNumbers.txt - read user input from file

Bits and Bytes

Two's Complement: binary digits inverted, plus 1

Overflow: Exceed max val-->overflow back to smallest; below min val-->overflow back to largest

SCHAR_MIN (-128), UCHAR_MAX (255), SHRT_MIN, INT_MAX (2147483647), UINT_MAX, ULONG_MAX

Casting: Replicate bit, interpreted differently (*int v = -1; unsigned int uv = v;| (unsigned int) v|* -12345U)

C will implicitly cast the signed argument to unsigned when comparing

Max is 0 followed by all 1s, min is 1 followed by all 0s in signed

Expanding bit representation: zero (unsigned) / sign extension (signed); promote to larger type for comparison

Truncating bit representation: discard more significant bits

bitwise operators: &, |, ~, ^, <<, >>

^ with 1 to flip, with 0 to let a bit go through

^ flip isolated bits, ~ flip all bits

num & (num - 1): clears the lowest 1

Right shift fills with sign bit (signed, arithmetic right shift); fills with 0s (unsigned, logical right shift)

long num = 1L << 32;, CHAR_BIT = 8

int sign = value >> (sizeof(int) * CHAR_BIT - 1); return (value ^ sign) - sign;

Characters and C Strings

char: single character / "glyph" ('\\', '\n', 'A' (65)), represented as int (ASCII), lowercase 32 more than upper

isalpha(ch) (alphabet), islower, isupper, isspace (space, \t, \n...), isdigit, toupper, tolower (return char, not modify existing)

C Strings: array of chars with '\0', null-terminating character, pass char* as param (add. of 1st char), str == &str[0]

Characters and C Strings (cont)

int foo(char *str) == int foo(char str[]), str-pointer (char** argv == char* argv[], double pointer)

Pointers and Arrays

Pointer: A variable that stores a memory address

Memory: A big array of bytes; each byte unique numeric index (generally written in hex)

*: declaration-pointer, operation-dereference/value at address

Pass value as param, C passes a copy of the value; take add (ptr) as a param, go to add when need val

char* could also ptr to **single char**

create strings as char[], pass them around as char *

Avoid &str when str is char[]! str/&str[0]

&arr does nothing on arrays, but &ptr on pointers gets its address

sizeof(arr) gets the size of an array in bytes, but sizeof(ptr) is always 8

An array variable refers to an entire block of memory. **Cannot** reassign an existing array to be equal to a new array.

Pass an array as param, C makes copy of add. of 1st element and pass a ptr to function (No sizeof with param!!)

Stack Memory and Heap Memory

The **stack** is the place where all local variables and parameters live for each function. Goes downwards when func called and shrinks upwards when func finished

The **heap** is a part of memory below the stack. Only goes away when free. Grows upward. Dynamic memory during program runtime.

Allocate with **malloc/realloc/strdup/calloc**, e.g. int *arr = malloc(sizeof(int)*len)); assert(arr != NULL); free(arr);

Stack Memory and Heap Memory (cont)

int *scores = calloc(n_elem, sizeof(int)); (zeros out memory); char* str = strdup("Hello"); malloc + strcpy

CANNOT free part of previous alloc, MUST free add received in alloc

A **memory leak** is when you do not free memory you previously allocated.

char *str = strdup("Hello"); str = realloc(str, new_len + 1); (Must be ptrs returned by malloc, etc.), automatic free of prev smaller one

Generics

void*: any pointer, No dereferencing/Pointer Arithmetic (cast to char* to do pointer arithmetic)

memcpy is a function that copies a specified amount of bytes at one address to another address (returns dest).

memmove handles overlapping memory figures (returns dest)

Function pointers: [return type] (*[name])([parameters]) ("callback" function, function writer vs function caller)

qsort: sort arr of any type; bsearch: binary search to search for a key in arr any type; lfind: linear search to search for key (return NULL not found); lsearch: linear search, add key if not found

GDB

GDB: p/x num (hex), p/d num (digit), p/t num (binary), p/c num (char), p/u (unsigned decimal); p nums[1]@2 (start at nums[1] print 2)

gdb myprogram; b main; r 82 (run with arts); n, s, continue (next,step into, continue); info (args, locals)

ctrl-c + **backtrace** - display the current call stack, meaning what functions are currently executing.

Optimizationn

Optimization: task of making program faster/more efficient with space or time

gcc -O0 (mostly literal translation), O2 (enable nearly all reasonable optimizations), O3 (more aggressive, trade size for speed), Os (optimize for size), -Ofast (disregard standards compliance)

Target: static instruction count, dynamic, cycle count/execution time

Constant Folding pre-calculates constants at compile-time where possible.

Common Sub-Expression Elimination prevents the recalculation of the same thing many times by doing it once and saving the result.

Dead code elimination removes code that doesn't serve a purpose (empty for loop, if/else same operation)

Strength reduction changes divide to multiply, multiply to add/shift, and mod to AND to avoid using instructions that cost many cycles (multiply and divide)

Code motion moves code outside of a loop if possible.

Tail recursion is an example of where GCC can identify recursive patterns that can be more efficiently implemented iteratively.

Loop unrolling: Do n loop iterations' worth of work per actual loop iteration, so we save ourselves from doing the loop overhead (test and jump) every time, and instead incur overhead only every n-th time.

Heap Allocator

A heap allocator is a suite of functions that cooperatively fulfill requests for dynamically allocated memory.

When initialized, a heap allocator tracks the base addr and size of a large contiguous block of memory: heap.

Heap Allocator (cont)

Throughput: # requests completed per unit time (minimizing avg time to satisfy a request) vs Utilization: how efficiently we make use of the limited heap memory to satisfy requests.

Utilization: largest addr used as low as possible

Internal Fragmentation: allocated block larger than what's needed, external fragmentation; no single block large enough to satisfy allocation request, even though enough aggregate free memory available

Implicit free list allocator: 8 byte (or larger) header, by storing header info, implicitly maintaining a list of free blocks (malloc linear in total number of blocks)

Explicit free list allocator: stores ptrs to next and previous free block inside each free block's payload (look just the free blocks on linked list for malloc, linear in # free blocks, update linked list when free), throughput increase, costs: design and internal fragmentation

Assembly: Control Flow & Function Call

%rip stores addr of next instruction to execute (%rip += size of bytes of curr instruction)

direct jump: jum Label, *indirect jump*: jmp *%rax (jump to instruction at addr in %rax)

Condition code regs store info about most recent arithmetic/logical operation (**lea** NOT update; logical like xor set CF & OF to 0; shift set CF to last bit shifted out and OF to 0; inc and dec set OF and ZF, leave CF unchanged)

CF: unsigned overflow, OF: two's-complement overflow/underflow

test and **cmp** just set condition codes (not store result)

static instruction count: # of written instructions; **dynamic instruction count**: # of executed instructions when program is run

Assembly: Control Flow & Function Call (cont)

%rsp: stores addr of "top" of stack, must point to same place before func called and after returned

push: R[%rsp]<-R[%rsp] - 8; pop+8

call: push next value of %rip onto stack, set %rip point to beginning of specified function's instructions

ret: pops instruction addr from stack and stores it in %rip

stored %rip: **return address**, addr of instruction where execution would have continued had flow not been interrupted by function call

nop: no-op, do nothing (make functions align); mov %ebx,%ebx, zeros out top 32 bits; xor %ebx,%ebx, set to 0, optmizes for performances & code size

Suppose %rcx stores arr[1] addr, to get arr[0] value: p *((int*)$rcx-1)

Assembly: Arithmetic and Logic

Machine code 1s and 0s, human-readable form assembly (GCC compiler)

Sequential instructions sequential in memory

Instruction operation name "opcode" (mov, add, etc.), "operands" (arguments, max 2)

$[number] constant value, "immediate"; %[name] register

Register: fast read/write memory slot right on CPU that can hold variable values (not in memory, 64-bit space inside processor, total 16)

mov: $ only src, % both, memory location at least one (copy value at addr)

Indirect(): dereferencing, (%rbx) copy value at addr stored in %rbx

%rip: addr of next instruction to execute

%rsp: addr of current top of stack

movl writing to reg also set high order 4 bytes to 0

movabsq 64-bit immediate, movq only 32-bit. 64-bit imm src, only reg as dest

Assembly: Arithmetic and Logic (cont)

movz, movs, smaller src larger dst, src: memory/reg, dest: reg

cltq: sign-extend %eax to %rax

parentheses require regs in par. be 64-bit

mov copies data **at** addr, **lea** copies value of src (addr) **itself** (only lea not dereferencing)

inc D D<-D + 1, **dec D** D <- D-1

shift k, D, k only %cl (w bits data, looks at lower-order **log2(w)** bits of %cl to know how much to shift) or imm

imul: two operands, multiplies and truncates to fit in the second; one operand, multiplies by %rax, higher-order 64 bits in %rdx, lower in %rax

idivq: divide 128-bit by 64-bit, higher-order 64 bit of dividend stored in %rdx, lower order %rax, only list divisor as operand (quotient %rax, remainder %rdx, **cqto** sign-extends 64-bit dividend)

C Program Example

```
#define CONSTANT 0x8
int main(int argc, char *argv[])
{
       char *prefix = " CS";
       int number = 107;
       // %s (string), %d
(integer), %f (double)
        pri ntf ("You are in
%s%d\n ", prefix, number);
        return 0;
}
```

Assignment 0

Assignment 0 (cont)

```
// void error(int status, int
errnum, const char *format,
...);
                          err -
or(1, 0, "out of range");
               }
       }
        pri nt_ tri ang le( -
nle vels);
        return 0;
}
```

Assignment 1

```
/* Unix
ls sample s/s erv er_ fil es/ -
home/ >> home_d ir.txt
diff sample s/s erv er_ fil -
es/ use rs.list home_d ir.txt
grep " sud o" sample s/s erv -
er_ fil es/ hom e/m att v/.b -
as h_h istory */
int main(int argc, char *argv[])
{
       int nlevels = DEFAUL -
T_L EVELS;
       if (argc > 1) {
                nlevels =
atoi(a rgv [1]);
               if (nlevels < 0
|| nlevels > 8) {
```

```
long signed_max(int bitwidth) {
        return ~signe d_m in( -
bit width);
}
long signed _mi n(int bitwidth)
{
        return -1L << (bitwidth
- 1);
}
long sat_ad d(long operand1,
long operand2, int bitwidth) {
       if (!((op erand1 >>
(bitwidth - 1)) & 1L) &&
        !(( ope rand2 >>
(bitwidth - 1)) & 1L) &&
        ((( ope rand1 +
operand2) >> (bitwidth - 1)) &
1L)) {
                return signed -
_ma x(b itw idth);
       }
       if (((ope rand1 >>
(bitwidth - 1)) & 1L) &&
        ((o perand2 >> (bitwidth
- 1)) & 1L) &&
        !(( (op erand1 +
operand2) >> (bitwidth - 1)) &
1L)) {
                return signed -
_mi n(b itw idth);
       }
        return operand1 +
operand2;
}
int to_utf 8(u nsigned short
code_p oint, unsigned char
utf8_b ytes[]) {
       if (code_ point <= 0x7f)
{
                utf 8_b ytes[0]
= code_p oint;
                return 1;
       } else if (code_ point <=
0x7ff) {
                utf 8_b ytes[0]
= 0xc0; // represents 11000000.
```

Assignment 1 (cont)

```
                utf 8_b ytes[1]
= 0x80; // represents 10000000.
                utf 8_b ytes[0]
|= (code_ point & 0x7c0) >> 6;
// 0x7c0 provides the bit mask
11100000.
                utf 8_b ytes[1]
|= code_point & 0x3f; // 0x3f
provides the bit mask 00111111.
                return 2;
        } else {
                utf 8_b ytes[0]
= 0xe0; // represents 11100000.
                utf 8_b ytes[1]
= 0x80; // represents 10000000.
                utf 8_b ytes[2]
= 0x80; // represents 10000000.
                utf 8_b ytes[0]
|= (code_ point & 0xf000) >> 12;
// 0xf000 provides the bit mask
111100 000 000 0000.
                utf 8_b ytes[1]
|= (code_ point & 0xfc0) >> 6;
// 0xfc0 provides the bit mask
000011 111 100 0000.
                utf 8_b ytes[2]
|= code_point & 0x3f; // 0x3f
provides the bit mask 000000 -
000 011 1111.
                return 3;
        }
}
#define BIT_MASK_3 7L
unsigned long advanc e(u nsigned
long cur_gen, unsigned char
ruleset) {
        uns igned long next_gen
= 0;
        uns igned long neighb -
orhood = 0;
        nei ghb orhood =
(cur_gen << 1) & BIT_MA SK_3;
        nex t_gen |= (ruleset >>
neighb orhood) & 1L;
       for (int i = 0; i <=
sizeof (long) * CHAR_BIT - 2;
++i) {
                nei ghb orhood =
(cur_gen >> i) & BIT_MA SK_3;
```

Assignment 1 (cont)

```
                        pri -
ntf (LI VE_ STR);
                } else {
                        pri -
ntf (EM PTY _STR);
                }
        }
        pri ntf ("\n ");
}
```

Assignment 2

Assignment 2 (cont)

```
        buf [ma xlen] = '\0';
        *p_ input = begin +
maxlen;
        return true;
}
int main(int argc, char argv[],
const char envp[]) {
        const char *searc hpath
= get_en v_v alu e(envp, " MYP -
ATH ");
       if (searc hpath == NULL)
{
                sea rchpath =
get_en v_v alu e(envp, " PAT -
H");
       }
       if (argc == 1) {
               char dir[PA TH_ -
MAX];
                const char
*remaining = search path;
                pri ntf ("Di -
rec tories in search path: \n");
                while (scan_ -
tok en( &r ema ining, " :", dir,
sizeof (dir))) {
                        pri -
ntf ("%s \n", dir);
               }
       } else {
               for (size_t i =
1; i < argc; ++i) {
                        const
char *execu table = argv[i];
                       char
dir[PA TH_ MAX];
                        const
char *remaining = search path;
                        while
(scan_ tok en( &r ema ining,
" :", dir, sizeof (dir))) {
                                -
str cat (dir, " /");
                                -
str cat (dir, execut able);
                                i
f (acces s(dir, R_OK | X_OK) ==
0) {
                                -
```

```
                nex t_gen |=
((ruleset >> neighb orhood) &
1L) << (i + 1);
       }
        return next_gen;
}
void draw_g ene rat ion (un -
signed long gen) {
       for (int i = sizeof -
(long) * CHAR_BIT - 1; i >= 0; -
-i) {
               if ((gen >> i) &
1L) {
```

```
const char *get_env_value(const
char envp[], const char key) {
       int lenKey = strlen -
(key);
       for (int i = 0; envp[i]
!= NULL; ++i) {
               char* match =
strstr (en vp[i], key);
               if (match ==
envp[i] && *(match + lenKey) ==
'=') {
                       return
match + lenKey + 1;
               }
       }
        return NULL;
}
bool scan_t oke n(const char
**p_input,
const char *delim iters, char
buf[], size_t buflen) {
        const char begin =
p_input;
        begin += strspn (begin,
delimi ters);
        const char* end = begin
+ strcsp n(b egin, delimi ters);

       int maxlen = 0;
       if (end - begin <= buflen
- 1) {
               maxlen = end -
begin;
       } else {
               maxlen = buflen
- 1;
       }
       if (maxlen <= 0) {
               *p_ input =
begin;
               return false;
       }
        str ncp y(buf, begin,
maxlen);
```

```
        pri ntf ("%s \n", dir);
                                -
        break;
                                }
                        }
                }
        }
        return 0;
}
```

Assignment 3

```
char *read_line(FILE
*file_pointer) {
        char* buffer = malloc -
(MI NIM UM_ SIZE);
        ass ert (buffer !=
NULL);
        size_t curSize =
MINIMU M_SIZE;
        char* curPtr = fgets( -
buffer, curSize, file_p oin -
ter);
       if (curPtr == NULL) {
                fre e(b uffer);
                return NULL;
       }
        size_t strLen = strlen -
(bu ffer);
        while (*(buffer + strLen
- 1) != '\n') {
                curSize *= 2;
                buffer =
reallo c(b uffer, curSize);
                ass ert (buffer
!= NULL);
                curPtr = buffer
+ strLen;
                char* newPtr =
fgets( curPtr, curSize - strLen,
file_p oin ter);
               if (newPtr ==
NULL) {
                        *curPtr
= '\0';
                        break;
                } else {
                        curPtr =
newPtr;
                }
                strLen +=
strlen (cu rPtr);
       }
       if (*(buffer + strLen -
1) == '\n') {
                *(b uffer +
strLen - 1) = '\0';
       }
        return buffer;
}
void print_ las t_n (FILE
```

Assignment 3 (cont)

```
                idx = (idx + 1) %
n;
                ++c nt_ read;
       }
       if (cnt_read < n) {
                idx = 0;
       } else {
                cnt _read = n;
       }
       line = lines[ idx];
        size_t cnt_print = 0;
        while (cnt_print <
cnt_read) {
                pri ntf ("%s -
\n", line);
                fre e(l ine);
                idx = (idx + 1) %
n;
                line = lines[ -
idx];
                ++c nt_ print;
       }
}
struct Element {
        char* str;
       int cnt;
};
void print_ uni q_l ine s(FILE
*file_ poi nter) {
        size_t curSize =
ESTIMATE;
        struct Element* arr =
malloc (si zeo f(s truct
Element) curSize);
        ass ert(arr != NULL);
        size_t cntElement = 0;
        char* line = NULL;
        while ((line = read_l -
ine (fi le_ poi nter)) != NULL)
{
                bool found =
false;
                for (size_t i =
0; i < cntEle ment; ++i) {
                        if
(strcm p(line, arr[i].str) == 0)
{
                                -
++a rr[ i].cnt;
```

Assignment 3 (cont)

```
                }
                if (!found) {
                        arr -
[cn tEl eme nt].str = line;
                        arr -
[cn tEl eme nt].cnt = 1;
                        ++c -
ntE lement;
                       if
(cntEl ement == curSize) {
                                -
curSize += ESTIMATE;
                                a
rr = reallo c(arr, sizeof -
(struct Element) * curSize);
                                -
ass ert(arr != NULL);
                        }
                }
       }
       for (size_t i = 0; i <
cntEle ment; ++i) {
                pri ntf ("%7d
%s\n", arr[i].cnt, arr[i].str);
                fre e(a rr[ -
i].s tr);
       }
        fre e(arr);
}
```

```
*file_ poi nter, int n) {
        char* lines[n];
        char* line = NULL;
       int idx = 0;
        size_t cnt_read = 0;
        while ((line = read_l -
ine (fi le_ poi nter)) != NULL)
{
                lin es[idx] =
line;
                                -
found = true;
                                -
fre e(l ine);
                                -
break;
                        }
```