CSCB09

Brian Chen, Albion Fung
github.com/conanap

The UNIX Filesystem

Brian Chen, Albion Fung
github.com/conanap

Common UNIX Commands

cd - change directory

pwd - print working directory

cat - read a file

mkdir - make a directory

ls - list directory

cp - copies file

Useful Commands

grep - globally search regex and print — grep "word" *

chmod - change access permissions (r,w,x) — chmod 644 text

who - displays users logged in right now

sort - sorts a collection of data — sort -k4 text

wc - word count — wc -l

Piping in UNIX

`cmd1 | cmd2`

This runs both commands, with the standard output of cmd1 connected into the pipe, and the standard input of cmd2 connected so as to come out of the pipe.

UNIX Commands

Write a series of UNIX commands that would scan the following file and print the lines with 'on' in it, sorted by second column:

users(.txt) ----------

uma x online utsc.utoronto.ca

suna a online utoronto.ca

falc m offline york.ca

mali y online uoit.ca

UNIX Commands

```
cat users | grep 'on' | sort -k2
```

In/Out Redirection

Input and output redirection is important.

`cmd > filename` puts the standard out into filename

`cmd < filename` makes the standard input from filename

In/Out Redirection

Given a file, users, find out how many lines are in the file that contain the name 'sav' and output it to another file called usernum

In/Out Redirection

```
grep 'sav' < users | wc -l > usernum
```

The C Language

Brian Chen, Albion Fung
github.com/conanap

Primitives

```
int x;

char c;

long x;

double x;
```

Printf and Scanf

printf("something, %val %val", val1, val2...);

%d = integer

%f = float

%s = string

So to print Hello, 1, 2, 3.0 we can do
(x=1,y=2,z=3.0)

printf("%s, %d, %d, %f", "Hello", x, y, z);

Forward Declaration

The C language needs to be forwardly declared. In Java, we can call a function that we havent defined yet, but in C we must specify that it exists with the **extern** keyword.

```c
#include <stdio.h>

int main(){
    int i;
    extern int gcd(int x, int y);

    for (i = 0; i < 20; i++)
        printf("gcd of 12 and %d is %d\n", i, gcd(12, i));
    return(0);
}
```

```c
int gcd(int x, int y){
    int t;

    while (y) {
        t = x;
        x = y;
        y = t % y;
    }
    return(x);
}
```

Memory in C

C is a low level language that has manual memory allocation

Memory is a vast array of bytes. Each byte has an address.

Bytes are collected into words. These days a byte is often 8 bits and a word often 4 bytes or 32 bits.

Pointers in C

Pointers in C are a high level version of an address

```
int i;

int *p;     -> declare p to be of type pointer-to-int

i = 3;

p = &i;    -> assign p to point to i

printf("%d\n", *p);    -> "dereference" -- follow a
pointer

i = 4;

printf("%d\n", *p);
```

Pointer Arithmetic

Same idea as memory and byte spaces.

If we have *p pointing to a[0], and we do p+3, which is 3 ints later, we basically have *(p.address() + 12), or a[3]

In this sense, if we have x, y, our x[y] can be

*((x) + (y))

Arrays in C

```
#include <stdio.h>

int main()
{
    int a[10];
    extern void setsquares( ... what goes here? ... );
    setsquares(a);
    printf("three squared is %d\n", a[3]);
    return(0);
}

void setsquares( ... what goes here? ... );
{
    int i;
    for (i = 0; i < 10; i++)
        a[i] = i * i;
}
```

C Strings

For the most part, we imagine Strings in C as arrays of char, which makes sense.

We terminate the array of char with a '\0' character so C knows that this is the end of the string

C String Example

```
#include <stdio.h>

#include <string.h>



int main()
{
    char a[20];

    strcpy(a, "Hello");

    printf("%s, world\n", a);

    return(0);
}
```

C String Functions

strcpy(a, b) -- copy string b to a

strcat(c, d) -- concatenate string d onto the end of string c (modifying c)

strlen(e) -- length of string e (not counting the terminating \0)

strcmp(f, g) -- difference between strings f and g (zero for equal)

strchr(h, x) -- find first occurrence of character x in string h (NULL for not found)

strstr(i, j) -- find first occurrence of STRING j in string i

strtok -- break up a string into tokens (words). Limited applicability and strange interface, but very useful when it happens to be suitable.

Memory Allocation

What about dynamically sized arrays?

We need to allocate memory using **malloc**

malloc's single parameter specifies a number of bytes of memory to allocate, and it returns a pointer to the beginning of the allocated data area, or NULL if there is not enough memory available.

Memory Allocation

```
int x[10]; <- ok

int y[z]; <- not ok in C89 if 'z' is
a variable
```

instead:

```
int *y;

y = malloc(z * sizeof(int));
```

Using memory

```
int a[17];
```

Java

```
int arraySize = a.length;
```

C++

```
int n = sizeof(a) / sizeof(int);
```

Free

You want to free things after we're done using it.
So after we malloc and we finish using whatever, just free(item) it

e.g.

```
x = ...;

use x;

free(x);
```

Structs

```
struct bar {
    int time;
    int beats;
    char *clef;
};
struct bar bar1, bar2;
```

```
struct bar {
    int time;
    int beats;

    char *clef;
} bar1, bar2;
```

```
typedef struct {
   int data;
   double decimal;
} coolint;

cooling *a;

a->data = 2;
```

Create a struct for a binary tree, data is integer

```
struct binarynode{
   int data;
   binarynode *left;
   binarynode *right;
};
//typedef struct binarynode bnode;
```

input

```
char str1[100]; // or size
int rar;

scanf(“%s”, str1);

scanf(“%d”, rar);
```

```
int len;

char *str1 = NULL;

size_t size = 0;

len = getline( &str1, &size,
stdin); // or a file descriptor
```

- sscanf
- fscanf (for files)
- fgets (for files)

```
int main (int argc, char **argv) {

…};
```

// argv are arguments passed, separated at spaces

// argc = number of arguments aka len(argv)

Output

```
printf(“Hi %s %d\n”, “wot”, 12);

perror(…); // string representation of the error

// e.g.: perror(“ls”);
```

- sprintf
- fprintf

Create a software that takes in a string of 5 characters. If its first character is an A, print “Albion is awesome”. Otherwise, print “Brian > Albion”

```
char in[6]; // don’t forget
terminating byte

scanf(“type something of 5
characters:\n%s”, in);

if(in[0] == ‘A’)

  printf(“Albion is awesome”);

else

  printf(“Brian > Albion”);
```

Error

- use perror(…);
- can pass file names or in general, a string and it will figure it out for you
- Errors are represented by numbers
- utilize error number constants
- General rule of thumb: 0 or 1 is success of some kind, -1 is error, other positive numbers depend on function
- main method returns 0 if no error

MakeFile

- make

– with no options looks for a file called Makefile, and evaluates the first rule

- make myprogram

– looks for a file called Makefile and looks for a rule with the target myprogram and evaluates it.

MakeFile

myprogram : file1.c

    gcc -Wall –o myprogram file1.c

somethingelse : requisitefiles.o/.c/.h

    do something...

MakeFile Example

```
dinner: pizza salad

pizza: cheese topping.peppers
        echo making pizza
        cat cheese topping.peppers > pizza

cheese:
        echo gooey cheese > cheese

topping.%:
        echo yummy $@ > $@

salad:
        echo salad is healthy
```

Files

FILE *fp = fopen(“~/Documents/chocolates.txt”, “w”); // or “r”, “a” for write, read, append

// write will overwrite anything there already

// append adds to end of file

// figure out what read means

// fgets or any of the print / input functions here

// notable writing tool: fputs(char *, FILE *);

fclose(fp); // close it! don’t forget!

Read from file “chknnuggets.txt”, write what’s in there to “waterdose.txt”

```
FILE *ckngt = fopen(“./chknnuggets.txt”,
“r”);

FILE *water = fopen(“./waterdose.txt”,
“a”); // it will create file for you

char * read …. // read it

fupts(read, water); // or another write
function is fine too

fclose(ckngt);

fclose(water);
```

File system

```
#include <sys/stat.h>
#include <sys/types.h>

#include <dirent.h>

// order is very, very important for these
includes
```

```
struct DIR *dir =
opendir(“directory”);

struct dirent *dirp;

while ((dirp = readdir(dir))) { //
extra brackets so while is testing for
null

  printf(“%s\n”, dirp->d_name); //
prints ea dir and file name

}
```

```
char *path = …;
struct stat lstatbuf, statbuf;
if(lstat(path, &lstatbuf)) {
   perror(path);
   exit(2);
}
if(stat(path, &statbuf) {
   perror(path);
   exit(2);
}
```

lstat vs stat

- lstat does not follow symlink, instead tells you the file is a symlink
- stat follows the link and tells you what it is linked to
- eg: symlink to directory /etc/bin at ./sl
- lstat ./sl will show as link
- stat ./sl will show directory

```
struct stat lstatbuf, statbuf; // assume opened from prev
if(S_ISREG(lstatbuf.st_mode))
  printf(“regular file\n”);
else if(S_ISDIR(lstatbuf.st_mode))
  printf(“directory\n”);
else if(S_ISLNK(lstatbuf.st_mode)) {// else is okay, but eg
  printf(“symlink to ”);
  if(S_ISREG(statbuf.st_mode))
    printf(“regular file\n”);
  else if((S_ISDIR(lstatbuf.st_mode))
    printf(“directory\n”); // again, else is fine
} // remember to add perror(dirp->d_name); at relevant places
```

closedir(dir); // don’t forget this

Read current directory, find a file (not directory) called “wattup” in current directory that is not symlinked

Challenge: also find it in its sub directory

```
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <dirent.h>
int main() {
  struct stat lstatbuf;
  DIR *dir = opendir(“.”);
  struct dirent *dirp;
  const char tgt[] = “wattup”;
  while((dirp = readdir(dir)) {
    if(lstat(dirp->d_name, &lstatbuf)) { perror(dirp->d_name);
return 2;} // stat(…) also okay
    if(!strcmp(tgt, dirp->d_name) && S_ISREG(lstatbuf.st_mode)) {
      printf(“found\n”);
      return 0; }
    }
  printf(“can’t find it\n”); return 1;}
```

challenge solution was my assignment from summer - solution is on GitHub

UNIX commands in C

```
#include <unistd.h>

extern char **environ; //env vars

int out = execl(“/bin/ls”, “ls”, “-al”, (char
*)0); // char * = end of args

char *cmd[] = {“ls”, “-al”, (char *)0};

out = execve(“/bin/ls”, cmd, environ);

// execVariation(pathToProgram, commandToRun,
endOfArgs);
```

- execs should not return to original process, ie no return code since it calls exit()
- So if you get a return code -> an error has occurred
- This makes error checking easy

Using C, have the program print out the contents of “hi.txt”

```
#include <unistd.h>

int main() {

  if((execl(“/bin/cat”, “cat”, “hi.txt”,
(char *)0)) {
    perror(“cat”);
    return 1;
}
```

fork, wait

- fork is like duplicating the current process
- How new processes are started
- Processi?
- fork duplicates pointer, memory and variables for the new process

```
#include <sys/type.h>
switch(fork()) {
  case -1: // error
    perror(“fork”); exit(2);
  case 0: // child
    …
  default: // parent
    …
}
```

```
pid_t pid = getpid();
```

In parent process

```
pid_t pid;

int status;

pid = wait(&status);

// pid = child pid, status = child return status
```

Wait

- Waits until child completes execution
- Return status can be helpful in deciding what to do next

Create a program that forks to write some words to a file using unix commands; then when it's done writing, print it on the console

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/type.h>
int main() {
  int status;
  switch(fork()) {
    case -1: … //perror
    case 0:
      execl(“/bin/echo”, “echo”, “We’re going on a trip”,
“>newfile”, (char *)0);
      perror(“echo”);
      exit(-1);
    default:
      wait(&status);
      // read file, print, etc
}
```