CS 537
Section 2: More C

Michael Swift

Homework

- Homework 1 is on moodle
- Due next Tuesday at start of class

Project 1: Shell

- Your next assignment is to create a shell
- It is basically a loop:
  - While (! Done)
    - Read input
    - Parse input
    - Execute commands
- Extra pieces:
  - Running programs in parallel
  - Redirecting output of one program to to a file
- Assignment is up on web page, due in 2 weeks

File Descriptors

- Processes have a list of open files – a “file descriptor table” as part of the PCB
- File system calls provide an index (a file descriptor) into that table; table records whether descriptor is in use and points to a data structure representing the open file or pipe.
- On Unix, fd 0,1,2 are reserved:
  - fd 0 = standard input, can only be read
  - fd 1 = standard output, can only be written
  - fd 2 = standard error, can only be written
  - By default, stdin, stdout, stderr refer to the console/terminal

Redirection

- In the shell, “redirection” comands change where these point:
  - Basic approach: close the place stdin/stdout go, and put something else there
    - E.g. a file
- Usages:
  - Pipes: command1 | command2 means send stdout of command1 to stdin of command2 using a pipe
  - Redirecting: command1 > file means send stdout of command1 to a file
  - Redirecting: command2 < file means send the contents of a file to stdin

Redirection examples

- ls > file
  - sends the output of ls to the file named file
- sort < file
  - sorts the contents of the file and prints it to the secreen
- sort < file1 > file2
  - sorts the contents of file1 and writes it to file2

- Basic approach: close the place stdin/stdout go, and put something else there
  - E.g. a file

Implementing redirection

- To redirect output:
  - close the first file descriptor, and put another one in its place:
    - close(STDOUT_FILENO);
    - open("output_file",O_WRONLY);
      - Open uses the first empty slow in the file descriptor table
  - Or, more reliably:
    - fd = open("output_file",O_WRONLY);
    - dup2(fd, STDOUT_FILENO);
      - Dup2 closes the second file descriptor and replaces it with a copy of the first

Implementing redirection for a new process

- Goal: replace file descriptors 0 and 1 for a new process:

```
outfile = open(outfile,”r”);
pid = fork();
if (pid == 0) {
  close(FILENO_STDOUT);
  dup2(outfile, FILENO_STDOUT);
  exec(command);
}
```

Interactive vs Batch

- Interactive
  - User types commands in, hits return to invoke them
- Batch
  - shell reads from an input file
- What is the difference?
  - where the commands come from
  - Whether a prompt is printed
  - Whether the shell prints the command line
- How do you code this?
  - Change which file you read from (as in P0)
    - Read from a file instead of STDIN
    - Or, close STDIN and redirect it to a file using code on previous slide

Using System Calls for I/O

- You cannot use printf() or fprintf() for printing in this project
  - the c library buffers this output:
  - printf("hello");
  - fork();
  - printf("world\n");
- could print:
  - hello hello world
- use Linux system calls:
  - fd = open(filename, mode, permissions)
  - write(fd, buffer, size) to write

How to debug your programs

- Add print statements
  - Print things out all the time to see what is happening
  - Problem: this is hard for big input files
- Better: use a **debugger**
  - Allows you to stop your program while it is executing and see the contents of all your variables
    - You can say where to stop
  - GUI debuggers: Visual Studio
    - Shows lots of stuff in windows
  - Command line debuggers: gdb
    - you can enter command to see everything

Debugging

- Compile with debugging using "-g"
  - gcc -g -o foo.o foo.c
- Run your program with gdb

```
gdb foobar
GNU gdb 6.3
<copyright omitted>
(gdb) break main
breakpoint 1 at 0x80483b0: in file foo.c, line 5
(gdb) run
Starting program: /afs/cs.wisc.edu/…/foobar
Breakpoint 1, main (argc=1, argv=0xbfe27804) at foo.c:5
5       if (argc > 1) {
(gdb) print argc
$1 = 1
(gdb)
```

Memory Debugging

```
int main(int argc, char * argv[])
{
  char * x;
  x = malloc(10);
  strcpy(x,argv[1]);
  printf("Hello, world: %s\n",x);
  free(x);
  strcpy(x,argv[2]);
  printf("Bye, world: %s\n",x);

  return(0);
}
```

Debugging example

What happens if you run this program?

- It works correctly?
- It crashes?
- [try it]

Valgrind

```
[swift] gcc -g z.c
[swift] valgrind ./a.out hellothereworld
==858== Memcheck, a memory error detector.
==858== ERROR
==858==
==858== Invalid write of size 1
==858==    at 0x26DB0: strcpy+160 (in /usr/local/lib/
   valgrind/x86-darwin/vgpreload_memcheck.so)
==858==    by 0x1F84: main+50 (z.c:9)
==858==  Address 0x3ec35a is 0 bytes after a block of size
   10 alloc'd
==858==    at 0x22E53: malloc+99 (in /usr/local/lib/
   valgrind/x86-darwin/vgpreload_memcheck.so)
==858==    by 0x1F6A: main+24 (z.c:8)
```

Strings

- Strings in C are arrays of bytes:
  - char str[100];
- Or pointers to memory
  - char * str;
  - str = **(char *)** malloc(100);
- They are null terminated – so you need to make space for it
  - str[0] = '\0';
  - strlen(str) = 0;
- There are a bunch of functions for working with them:
  - strlen, strcpy, strcat

Question

- What was wrong with the code to reverse the characters on a line?

```
char * reverse(char * line) {
    char *tmp;
    tmp = (char *) malloc(strlen(line));
    for (int j=0, i = strlen(line); i >= 0; i--)
        tmp[j++] = line[i];
    return(tmp)
```

String Example

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
  char s[100];
  strcpy(s,"hello");
  strcat(s,", world");
  printf("S = %s\n",s);
}
```

parsing with strtok

```
include <string.h>
char *strtok(char str, char * sep);
```

- the strtok() function tokenizes a string into words
  - **str** is the string to tokenize
  - **sep** are the characters that separate tokens, e.g., space, tab, new line
  - strtok remembers the strong after the first call
- Example:

```
tmp = strtok(buffer,” \t\n");
while (tmp) {
  cmds[num_cmds] = tmp;
  num_cmds++;
  tmp = strtok(NULL,"\n \t");
}
```

strtok Things to Remember

- strtok() modifies the string
  - It replaces the separator with a null character
  - strtok() returns NULL when you get the last token of a string
  - As long as the buffer you parse remains allocated, you can store the pointers returned from strtok()
- You can use strtok() again on the strings returned from strtok to parse with different separators
    - e.g., separate a strong into commands, and then a command into arguments

Strtok hints

- No nested loops on one buffer:

```
tmpcmd = strtok(buffer,"+\n");
while (tmpcmd) {
  tmp = strtok(tmpcmd," \t");
   while (tmp) {
     cmds[num_cmds] = tmp;
     num_cmds++;
     tmp = strtok(NULL,"\n \t");
   }
  tmpcmd = strtok(NULL,"\n \t");
}
```

- Why?
  - inner strtok ovewrites outer strtok
- How do you do this?
  - copy the the string

Strtok hints

- Fixed version

```
char * lasts;
int num_cmds = 0;
tmpcmd = strtok_r(buffer,"+\n",&lasts);
 while (tmpcmd) {
   char * tmp;
   char tmpbuf[MAX_LEN];
   strcpy(tmpbuf,tmpcmd);
   tmp = strtok(tmpbuf," \t");
   while (tmp) {
     cmds[num_cmds] = strdup(tmp);
       num_cmds++;
     tmp = strtok(NULL,"\n \t");
   }
   tmpcmd = strtok_r(NULL,"+\n",&lasts);
 }
```

Pointers in C

- Pointers are addresses
  - `char * c = malloc(10 * sizeof(char));`
  - c now contains the **address** of some memory
- '*' operator returns what is at an address
  - *c returns the character at address c
- p[n] operator returns what is at address:
  - p + n * sizeof(*p) – the size of the type p points to
- two dimensional arrays:
  - int **p;
  - p[n] = what is at p+n*sizeof(int *); call this q: a pointer to integers
  - q[m] = what is at q + m*sizeof(int); an integer

Memory Allocation

- Malloc allocates memory in blocks
  - writing to locations off the end of your block may not cause a bug
    - if your block is not a multiple of 16 (or so bytes)
  - But it may cause later calls to malloc/free/realloc to fail
    - malloc may store a header on a block, and overwriting can corrupt that header

Rules for allocating memory

- Call malloc() for any array or structure returned from a function with a pointer:

```
char * reverse(char * line) {
    char tmp[100];
    for (int j=0, i = strlen(line); i >= 0; i--)
        tmp[j++] = line[i];
    return(tmp)
```

- Better:

```
char * reverse(char * line) {
    char *tmp;
    tmp = (char *) malloc(strlen(line));
    for (int j=0, i = strlen(line); i >= 0; i--)
        tmp[j++] = line[i];
    return(tmp)
```

Rules for allocating memory

- Call malloc() for any array or structure returned from a function with a pointer:

```
char * reverse(char * line) {
    char *tmp;
    tmp = (char *) malloc(strlen(line));
    for (int j=0, i = strlen(line); i >= 0; i--)
        tmp[j++] = line[i];
    return(tmp)
```

- Call free() when you are done:

```
char * output;
fgets(line,MAX_LEN,file);
output = reverse(line);
free(output);
```

Separate Compilation

- Larger programs may have multiple files
  - gcc file1.c file2.c file3.c
- You can keep things simpler by compiling each one separately
  - gcc **–c** file1.c
  - gcc **–c** file2.c
  - gcc **–c** file3.c
  - gcc –o program file1.o file2.o file3.o
- This gets cumbersome, so you can create scripts to do all the pieces for you, called **Makefiles**

Makefiles

- Specify the commands to compile code
  - in a file named "Makefile"
- Example:

```
foo.o: foo.c
	gcc -c -O -Wall foo.c
bar.o: bar.c
	gcc -c -O -Wall bar.c
foobar: foo.o bar.o
	gcc -o foobar foo.o bar.o
default: foobar
```

- General format:

```
target: prereq1 prereq2
<tab>	command1
<tab>	command2
```