CS 111 week 1

Tianxiang Li

About myself

- Email: tianxiang@cs.ucla.edu
- Office hours: 4-6Pm PDT, Monday, Zoom
  - Feel free to email me if I didn't see you, or if you don't see me in the room
  - FIFO
- Discussion Section 1B, Friday 10AM-12PM PDT
  - Discussions will be recorded and uploaded to CCLE
  - Slides will be uploaded to CCLE
- Questions
  - Pizza posts
  - Project grading questions, please send a private post

What will be on the discussion section

- Mainly focus on projects
  - Clarify the projects
  - Discuss background knowledge for projects
  - Introduce APIs used for the projects and show code examples
- Q&A
  - Feel free to speak up when you have question, or type in chatbox
  - Feel free to answer any questions
  - Feel free to correct me when there’s a mistake

Project logistics

- Code will be graded on lnxsrv09.seas.ucla.edu
  - Create a Seasnet account if you don't have one
    - Username 8 characters, password 8 characters: at least one number, one lowercase, and one uppercase (no special characters)
  - Forgot you account?
    - help@seas.ucla.edu
  - *ssh user_name@lnxsrv09.seas.ucla.edu*
- Make sure the gcc version is correct (/usr/local/cs/gcc-9.3.0/bin/)
  - Command to check version:
    which gcc
  - If version is incorrect:

    export PATH= /usr/local/cs/gcc-9.3.0/bin/:$PATH

    Add the line PATH=/usr/local/cs/gcc-9.3.0/bin:$PATH in ~/.profile or ~/.bash_profile
  - Safe way to compile

    /usr/local/cs/gcc-9.3.0/bin /gcc –o lab0 lab0.c

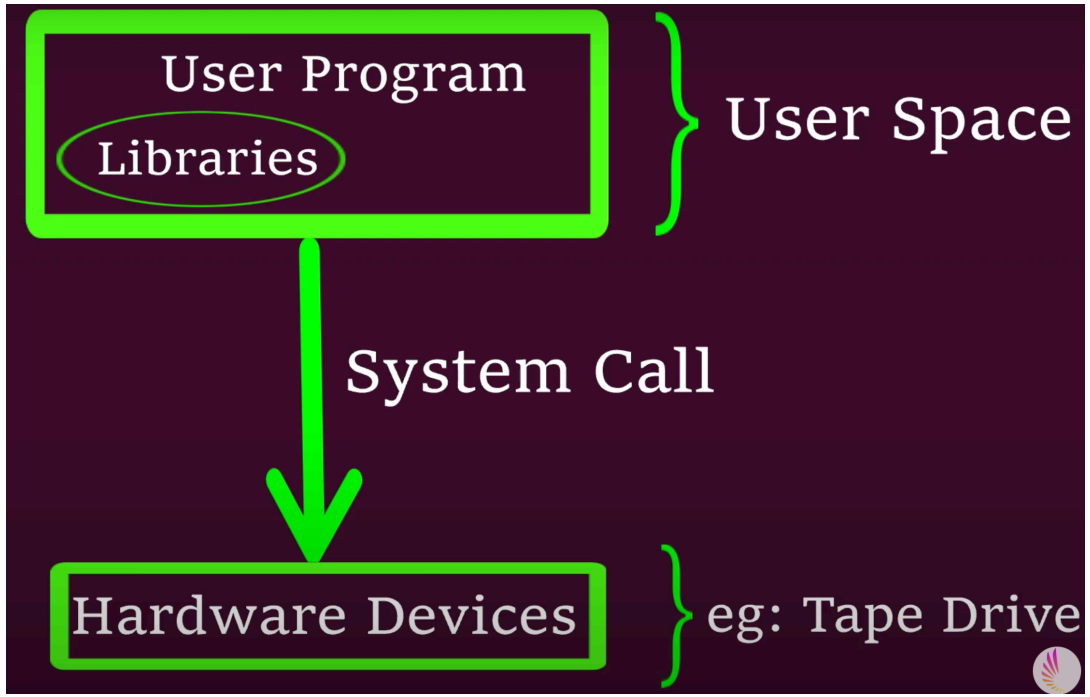
Project 0: Warm Up

Project Requirements

- First thing: Write a program (lab0.c)
  - **Copies** its standard **input** to its standard **output** by:
    - **read**(2)-ing from file descriptor 0 (until encountering an end of file) and,
    - **write**(2)-ing to file descriptor 1. If no errors (other than EOF) are encountered,
    - your program should **exit**(2) with a return code of 0.
  - Support **optional** command line **arguments**
    - **--input =** ***filename*** *u*se the specified file as standard input
    - **--output =** ***filename*** create the specified file and use it as standard output
    - **--segfault** force segmentation fault
    - **--catch** catch segmentation fault, log error, exit

System call for file operation: quick review

- Functions provided by Linux to create an interface with the OS
  - open(2), read(2), write(2), close(2)
  - #include <unistd.h>



System call for file operation: quick review

- open(2), read(2), write(2), close(2)
- int open(const char **path*, int *oflags*);
  - e.g. fd = open("/var/tmp/a.out", O_RDONLY)

ReadOnly: O_RDONLY
WriteOnly: O_WRONLY
ReadWrite: O_RDWR

- ssize_t read(int ***fd***, void ****buf***, size_t ***count***);
- ssize_t write(int ***fd***, void ****buf***, size_t ***count***);

  fd: file descriptor created by open (read from it/write to it)

  buf: pointer to the buffer (store read results/write using its content)

  count: size of the buffer

  return value: Number of bytes read/written, -1 if error occurred

- int close(int fd);

  close fd.

Code example: copy one file to the other

```
#include <fcntl.h>
// error handling code omitted for briefty
#define BUFFERSIZE 1024
char buffer[BUFFERSIZE];
int main(argc, char * argv[])
{
     src_fd = open(argv[1], O_RDONLY);
    des_fd = open(argv[2], O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR | S_IRGRP);
     while ((ret = read(src_fd, buffer, BUFFERSIZE)) > 0)
          write(des_fd, buffer, ret); //FIXME: not handling actual bytes written < ret.
      close(src_fd);
      close(des_fd);
      return 0;
}
```

cp /var/tmp/a.txt /home/user/file.txt

File Descriptor Review

- File Descriptor: Abstract representation of a file
- 0 → reading from keyboard
- 1 → writing to screen
- 2 → error → screen

Project overview

- Write a program that **reads** from **stdin** and **writes** to **stdout**, essentially a simplified cat command
- Special file descriptors: 0: stdin, 1: stdout, 2: stderr
- Pseudo-code:

```
while (1)
{
    read(0, buffer);
    if error or reach EOF
        break;
    write(1, buffer);
    if error or reach EOF
        break;
}
```

- File Descriptor: Abstract representation of a file
- 0 → reading from keyboard
- 1 → writing to screen
- 2 → error → screen

IO re-direction

- Support two options:
  - --input=filename ... use the specified file as standard input
  - --output=filename ... create the specified file and use it as standard output
- One possible way to do it:

```
if (input option is set)
        in_fd = open(input_filename);
else
        in_fd = 0;
while (1)
{
        read(in_fd, buffer);
        write(stdout, buffer);
}
```

IO re-direction (cont)

- Support two options:
  - --input=filename ... use the specified file as standard input
  - --output=filename ... create the specified file and use it as standard output
- The other way of doing it: dup(2) system calls:

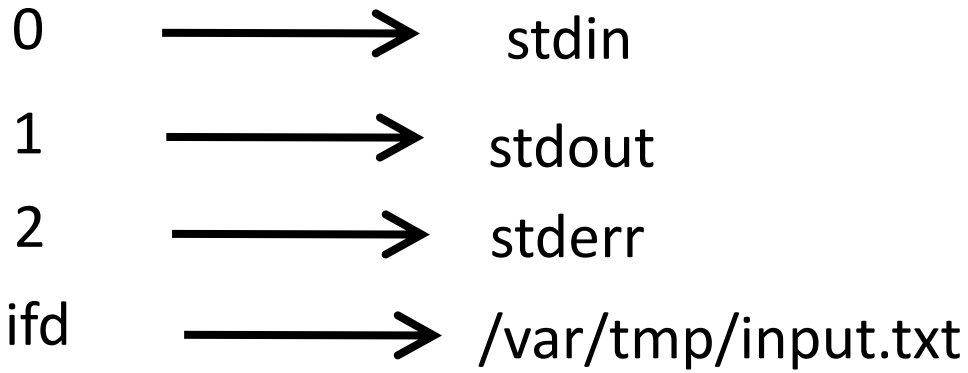
  prototype: int dup(int fd);
  Make the lowest available file descriptor to point to the same file as fd.
- Read redirection example:

```
int ifd = open("/var/tmp/input.txt", O_RDONLY);
if (ifd >= 0) {
    close(0);
    dup(ifd);
    close(ifd);
}
```

IO-redirection with dup(2)

- Prototype: int dup(int fd);
  Make the lowest available file descriptor to point to the same file as fd.
- Read redirection example:

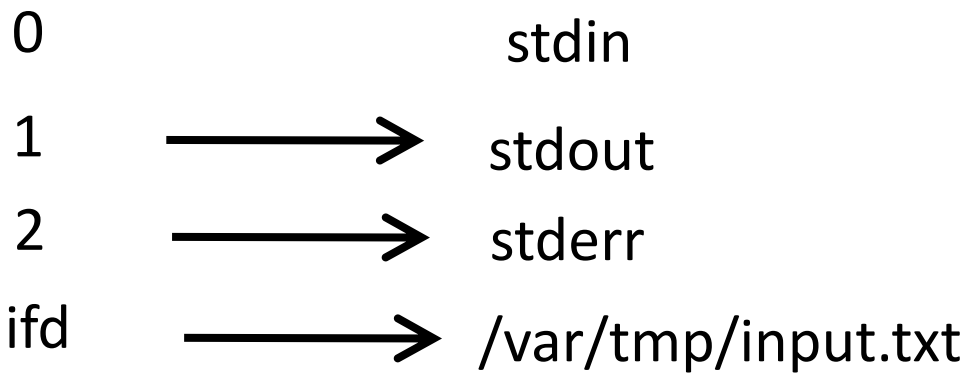
```
int ifd = open("/var/tmp/input.txt", O_RDONLY);
if (ifd >= 0) {
    close(0);
    dup(ifd);
    close(ifd);
}
```



IO-redirection with dup(2)

- Prototype: int dup(int fd);
  Make the lowest available file descriptor to point to the same file as fd.
- Read redirection example:

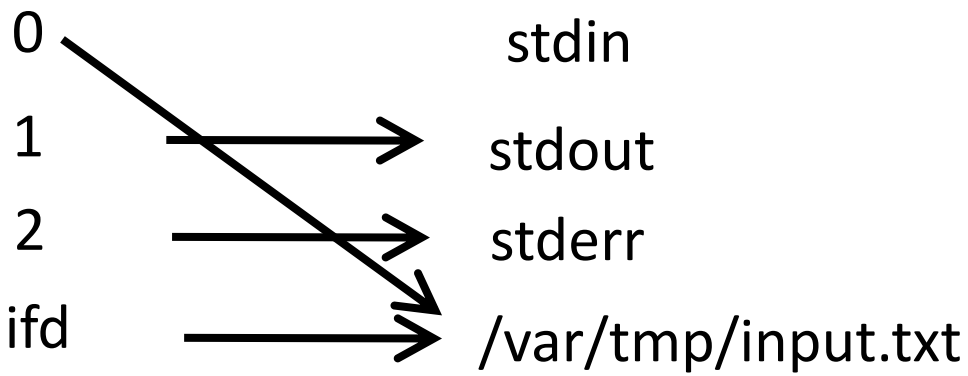
```
int ifd = open("/var/tmp/input.txt", O_RDONLY);
if (ifd >= 0) {
    close(0);
    dup(ifd);
    close(ifd);
}
```



IO-redirection with dup(2)

- Prototype: int dup(int fd);
  Make the lowest available file descriptor to point to the same file as fd.
- Read redirection example:

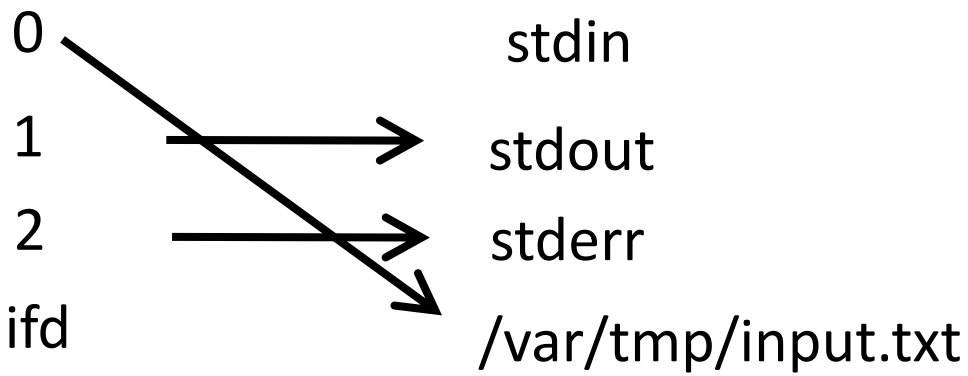
```
int ifd = open("/var/tmp/input.txt", O_RDONLY);
if (ifd >= 0) {
      close(0);
      dup(ifd);
      close(ifd);
}
```



IO-redirection with dup(2)

- Prototype: int dup(int fd);
  Make the lowest available file descriptor to point to the same file as fd.
- Read redirection example:

```
int ifd = open("/var/tmp/input.txt", O_RDONLY);
if (ifd >= 0) {
      close(0);
      dup(ifd);
      close(ifd);
}
```



SIGSEGV signal

- Support two options:
  - --segfault: force a segmentation fault
  - --catch: use signal(2) to register a SIGSEGV handler that catches the segmentation fault
- What truly happens upon a segfault:

```
int main(void)
{
        char * ptr = NULL;
        (*ptr) = 0;
}
```

NULL pointer dereference
when a program attempts to read or write to memory with a NULL pointer.

→ cause an illegal memory access exception

→ CPU traps to the illegal memory access exception handler in the OS

→ OS sends a SIGSEGV signal to the process

→ return from OS to process, process execute the default SIGSEGV handler that kills the process (via exit)

signal(2)

- Support two options:
  - **--segfault**: force a segmentation fault
  - **--catch**: use signal(2) to register a SIGSEGV handler that catches the segmentation fault

- signal(2): register a signal handler to replace the default signal handler

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

signal() sets the disposition of the signal *signum* to *handler*, which

SIGINT is a signal generated when a user presses Control-C. This will terminate the program from the terminal.
SIGSTOP tells LINUX to pause a process to be resumed later.
SIGSEGV is sent to a process when it has a segmentation fault.
SIGKILL is sent to a process to cause it to terminate at once.
…

signal(2) example

```
void sigsegv_handler(int sig)
{
        fprintf(…)
        exit(….)
}
int main(void)
{
        char * ptr = NULL;
        signal(SIGSEGV, sigsegv_handler);
        (*ptr) = 0; // Jump to execute code in sigsegv_handler function
}
```

Error handling: errno and strerror

- When you print out an error message, your message should include enough information for user to understand its cause
- **errno**: a *global integer variable* which is set by system calls and some library functions in the event of an error to indicate what went wrong.
  - *errno is a global variable defined in <errno.h>, the systems sets it whenever a system call error occurs*

ret = read(src_fd, buffer, BUFFERSIZE) → ret = -1 → an error occurred

errno is set to tell you what specifically went wrong:

errno = 9 (EBADF): src_fd is an invalid file descriptor

errno = 14 (EFAULT): buffer points to an invalid memory address

Error handling: strerror and code example

- When you print out an error message, your message should include enough information for user to understand its cause

- char * strerror(int errno);

The strerror() function returns a pointer to a string that describes the error code passed in the argument errno.

```
int infd = open(infile, O_RDONLY);
if (infd < 0)
{
    fprintf(stderr, "%s: %s\n", infile, strerror(errno));
 }
```

Handle input arguments

Suppose we have a program called test, which supports two arguments:
--verbose #output the debugging logging message
--input file # specify a file as the input

```
./test --input /home/user/file.txt
./test --verbose
./test --input /home/user/file.txt --verbose
./test --input /home/user/file.txt
./test --verbose --input #error
…
```

Imagine:
(1) parse command line arguments for programs with tens/hundreds of options.
(2) parse command line arguments for every program you wrote
→ Standard APIs: (e.g. getopt_long) to handle command line arguments

```
struct option args[] = {
 /*          option          has_arg       flag     return value */
            {"input",           1,          NULL,    'i'},        /* input file          */
            {“verbose",         0,          NULL,    ‘v'},        /* verbose             */
            { 0, 0, 0, 0 }
};

int main(int argc, char * argv[])  {
            int verbose = 0;
            char* infile;
            while( (i = getopt_long(argc, argv, “", args, NULL) ) != -1)  {
            switch(i)
           {
                        case 'i':
                                    infile = optarg; //optarg is a predefined variable, points to the argument of the option
                                    break;
                        case ‘v':
                                    verbose = 1;
                                    break;
                        default:
                                    error_handling
                                    break;
              }
            }
}
```

The getopt_long() function works like getopt() except that it also accepts long options, started with two dashes.

e.g. --arg=param or --arg param.

Makefile

create a Makefile that supports the following targets:

(default): build the lab0 executable
check: runs a quick smoke-test on whether or not the program seems to work
clean: delete all files created by the Makefile
dist: build the distribution tarball.

Makefile review

- Makefile consists of multiple rules, for each rule:
- Target: depend0, depend1, depend2…
  [TAB] command1
  [TAB] command2

  ….

  if (1) target does not exist or (2) the modification time of dependencies is newer than target, then execute the commands. Dependencies can be omitted;

  myprog: myprog.c

  ```
      gcc –o myprog myprog.c
  ```

  clean:

  ```
      rm –f myprog
  ```

  $make #execute the first target myprog

  $make clean

Makefile skeleton for project 0

```
lab0: lab0.c:
        gcc …

clean:
        rm –f …

dist: lab0.c and some more files
        tar …

check: lab0 and some more files
        …
```

GDB

Task0:

run your program (with the **--segfault** argument) under gdb(1)

take the fault

get a stack **backtrace**

take a screen snapshot (to be included with your submission)

Task1:

run your program (with the **--segfault** argument) under gdb(1)

set a **break-point** at the bad assignment

run the program up to the breakpoint

inspect the pointer to confirm that it is indeed NULL

take a screen snapshot (to be included with your submission)

GDB review

1. Compile the program with –g option

   gcc –g –o myprog myprog.c

2. gdb ./myprog

3. (gdb) run options
   e.g. run --input=/var/tmp/input.txt

4. Use various gdb commands to help you debug the program

   backtrace (bt). → a summary of how your program got where it is

   breaktpoint (b) *line#* → makes your program stop at a certain point

   print(p) *variable_name.* → Prints the value of a given expression

GDB commands

1. (gdb) bt #show the stacktrace

   (gdb) bt

   #0 0x080486a2 in func () at test.c:128

   #1 0x0804879b in main (argc=2, argv=0xbffff3c4) at test.c:72

2. (gdb) b test.c:14 #set a breakpoint at line 14 of test.c

3. (gdb) print var #print the value of variable var

Usage Statement

Usage statement usually consists of the correct command line usage for the program and includes a list of the correct command-line arguments or options

usage: myprog

usage: myprog arg1 arg2

usage: lab0 [--input=file] (example: *lab0 --input=file1.txt*)