COMP1521 25T2

Week 7 Lecture 1

File Systems

Adapted from Angela Finlayson, Hammond Pearce,
Andrew Taylor and John Shepherd's slides

Announcements

Test 5 and Test 6 are due Thursday 9pm

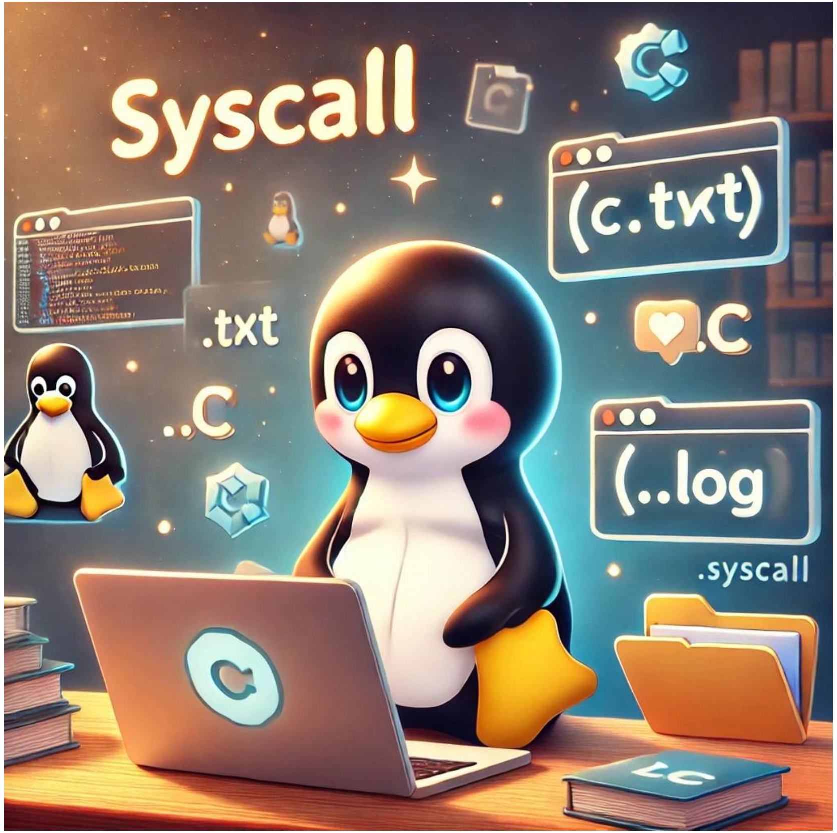
Assignment 1

- Automarking available soon
- Tutor marking ASAP

Assignment 2: coming out later this week!

Today’s Lecture

- Recap and code demos:
  - syscall, libc wrappers, stdio
- File Operations
  - open, close, read, write, seek



Recap: System Calls in Linux

syscall function

- Not usually used in practice
- Syscalls vary between operating system -- code is less portable
- Hard to understand

Libc syscall wrapper:

- More meaningful names: open(...), read(...), write(...)
- Does syscall for you and helps with error checking
- More portable than **syscall** but still not portable
    - Some work on POSIX compliant systems (e.g. Linux and MacOS)

Recap: System Calls in Linux

stdio.h provides higher-level library functions:

- fopen(...), fgets(...), fputc(...)
- Calls syscall wrapper for you
- Portable
- You have been using these to indirectly do your system calls the whole time!
- Sometimes we need lower-level non-portable functions
  - e.g. Database software needs precise control over I/O

Recap: System Calls to Manipulate Files

Important file related system calls

Id	Name	Function
0	read	read some bytes from a file **descriptor**
1	write	write some bytes to a file **descriptor**
2	open	open a file system object, returning a file **descriptor**
3	close	close a file **descriptor**
4	stat	get file system metadata for a pathname
8	lseek	move file **descriptor** to a specified offset within a file

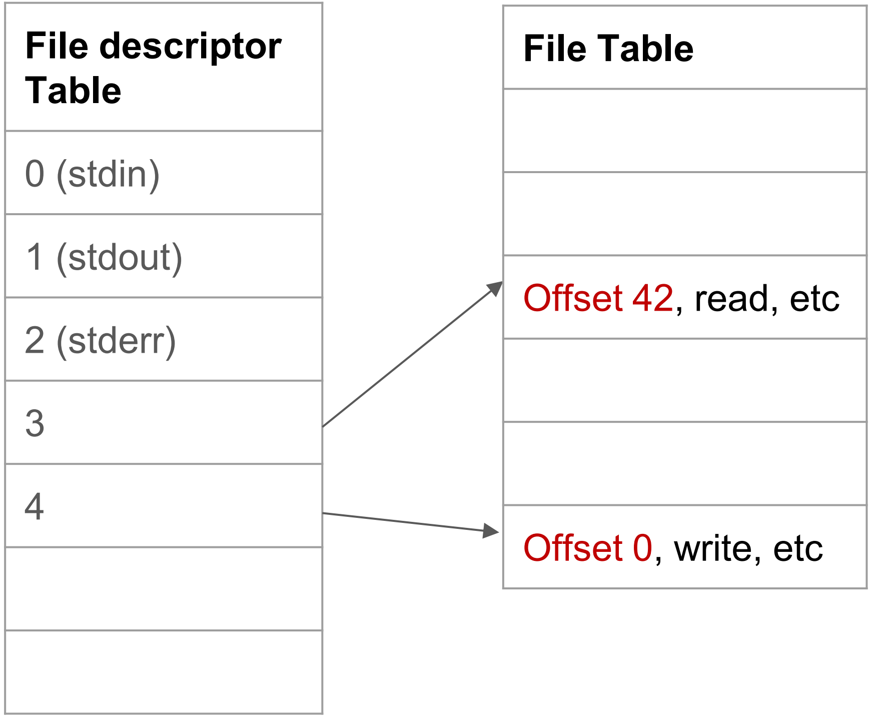
Reca: File Descriptors

Every process starts with the 3 standard streams, 0, 1, 2.

When a file is opened a new file descriptor is added to the table.

When a file is closed the file descriptor is removed

When a file is read to or written from, the offset is updated



File descriptor Table

0 (stdin)

1 (stdout)

2 (stderr)

3

4

File Table

Offset 42, read, etc

Offset 0, write, etc

System call to print a message to stdout

`syscall` : make a system call without writing assembler code

- not usually used by programmers
- use to experiment and learn

```
char bytes[13] = "Hello, Zac!\n";

// argument 1 to syscall is the system call number, 1 is write
// remaining arguments are specific to each system call

// write system call takes 3 arguments:
//   1) file descriptor, 1 == stdout
//   2) memory address of first byte to write
//   3) number of bytes to write

syscall(1, 1, bytes, 12); // prints Hello, Zac! on stdout
```

Source code for hello_syscalls.c

Libc wrapper to print message to stdout

```
char bytes[13] = "Hello, Zac!\n";

// write takes 3 arguments:
//   1) file descriptor, 1 == stdout
//   2) memory address of first byte to write
//   3) number of bytes to write
write(1, bytes, 12); // prints Hello, Zac! on stdout
```

Source code for hello_libc.c

Recap: errno

- C library has an interesting way of returning error information
  - functions typically return **-1** to indicate error
  - and set errno to integer value indicating reason for error
  - you can think of **errno** as a global integer variable

- These integer values are **#define**-d in **errno.h**
  - see man errno for more information
  - **perror()** looks at **errno** and prints message with reason
  - **strerror()** converts **errno** to string describing reason for error

- To see all error codes type **errno -l** on command line

Exercise

Implement linux **cp** command

1. byte at a time stdio.h
2. using fgets and fprintf/fputs - what is the problem with this approach?

We also have implementations using syscall and libc

Which is the best approach?

libc Code Demo

open()

read()

write()

close()

IO Performance libc

```
$ clang -O3 cp_x.c -o cp_x
$ dd bs=1M count=10 < /dev/urandom > random_file
10485760 bytes (10 MB, 10 MiB) copied, 0.183075 s, 57.3 MB/s
$ time ./cp_x random_file random_file_copy
```

stdio Code Demo

fopen()

fgetc(), fread()

fputc(), fwrite()

fclose()

IO Performance & Buffering libc vs stdio

Let’s compare our implementations of cp!

```
$ clang -O3 cp_x.c -o cp_x
$ dd bs=1M count=10 < /dev/urandom > random_file
10485760 bytes (10 MB, 10 MiB) copied, 0.183075 s, 57.3 MB/s
$ time ./cp_x random_file random_file_copy
```

Can we get any insights from strace?

```
$strace ./cp_x random_file random_file_copy
```

Compare:

Linux cp command, cp_fgetc_one_byte.c, cp_libc_one_byte.c, cp_libc.c

stdio.h buffering for efficiency

- **Goal:** reduce number of system calls (expensive)
- **Reading:**
  - Uses a **read** system call to fill whole buffer
  - subsequent reads get bytes from the buffer
  - does not do another **read** system call till it runs out of data in the buffer
- **Writing**:
  - Delays calls to **write** system call by storing data in buffer (array) instead
  - calls **write** system call only when
    - buffer is full,
    - file is closed,
    - fflush is called
    - a newline is encountered for output to terminal

fflush stdio buffers

You can manually flush stdio buffers by using:

```
int fflush(FILE *stream);
```

For example

- this would force a write system call to stdout and empty the output buffer
  ```
  fflush(stdout);
  ```
- Can also be used for files that have been opened for writing.
- Should not be used for stdin or files opened for read only.

Demo: fgetc return type bug

- To make a buggy version:
    - Use char instead of int for fgetc (this creates bugs with getchar too)

- Reminder: getchar and fgetc return int
    - Legal values they can return -1..255. (257 possible values)
    - This can’t fit in signed char or unsigned char!

- signed char (or char on our system) can store -1 and detect EOF,
    - but valid byte value 0xFF gets mistaken for EOF

- unsigned char can’t store -1 and can’t detect EOF

stdio.h reading and writing text only

```
char *fputs(char *s, FILE *stream);                // write a string
char *fgets(char *s, int size, FILE *stream);  // read a line

//formatted input/output
int fscanf(FILE *stream, const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
```

stdio.h reading and writing text only

```
char *fputs(char *s, FILE *stream);              // write a string
char *fgets(char *s, int size, FILE *stream);  // read a line
```

```
//formatted input/output
int fscanf(FILE *stream, const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
```

These functions can not be used for binary data as they may contain 0x00 bytes

- can use to read text (ASCII/Unicode)
- can **not** use to read a *jpg* for example

Demo: cp using fgets and fprintf

- Using fgets and fprintf to copy a file

Demo: cp using fgets and fprintf

- Using fgets and fprintf to copy a file
- Seems to work fine when copying text files BUT
    - Breaks for binary files with 0x00 bytes
    - They are interpreted as end of string ‘\0’ character

Reminder: only use fgets, fprintf, fscanf, or fputs for text

Recap: stdio.h convenience functions

To read/write to stdin/stdout

```
int getchar(void);                 // fgetc(stdin)
int putchar(int c);                // fputc(c, stdout)
int puts(char *s);                 // fputs(s, stdout)
int scanf(char *format, ...);      // fscanf(stdin, format, ...)
int printf(char *format, ...);     // fprintf(stdout, format, ...)
```

These should never be used: security vulnerability, buffer overflow

```
char *gets(char *s);                 // Ok in general.
scanf("%s", array);                  // Don't use with %s
```

stdio.h - IO to strings

`stdio.h` provides useful functions which operate on strings

```
// like scanf, but input comes from char array str
int sscanf(const char *str, const char *format, ...);
```

```
// like printf, but output goes to char array str
// handy for creating strings passed to other functions
// size contains size of str
// Do not use similar function sprintf as it is a security vulnerability
int snprintf(char *str, size_t size, const char *format, ...);
```

seeking

- So I can now read and write files sequentially... But
  - How do I know which position in the file I am at?
  - How can I skip to the end of the file?
  - How can I go back and read earlier data again?

Seeking with libc system call wrapper

```
off_t lseek(int fd, off_t offset, int whence);
```

- change the **current position** in given stream
- **offset** is in bytes, and can be negative
- **whence** can be one of
    - SEEK_SET : set **offset** from start of file
    - SEEK_CUR: set file **offset** from current position
    - SEEK_END: set file **offset** from end of file

- seeking beyond end of file leaves a gap which reads as 0's
- seeking back beyond start of file sets position to start of file

Seeking with stdio.h

```
int fseek(FILE *stream, long offset, int whence);
```

- is stdio equivalent to **`lseek()`** except:
    - requires a FILE * input instead of int file descriptor
    - influences stdio buffers
    - returns 0 or -1 for error

```
fseek(stream, 42, SEEK_SET); // move to after 42nd byte
fseek(stream, 58, SEEK_CUR); // 58 bytes forward from current position
fseek(stream, -7, SEEK_CUR);  // 7 bytes backward from current position
fseek(stream, -1, SEEK_END);  // move to before last byte in file
```

```
long ftell(FILE *stream);  //return current file position
```

Demo code fseek.c and fuzz.c and advanced example: create_gigantic_file.c

What we learnt today

- System calls relate to files:
  - open, close, read, write, lseek
- Equivalent stdio portable functions:
  - fopen, fclose, fgetc, fputc etc. fseek

Next Lecture

- File Systems:
  - File metadata
    - Permissions
    - system call stat
  - Hard Links and Symbolic Links
  - Working with directories

Reach Out

Content Related Questions:

Forum

Admin related Questions email:

cs1521@cse.unsw.edu.au

Student Support | I Need Help With…

My Feelings and Mental Health Managing Low Mood, Unusual Feelings & Depression	**Mental Health Connect**	student.unsw.edu.au/**counselling** Telehealth	**In Australia Call Afterhours UNSW Mental Health Support Line**	1300 787 026 5pm-9am
	Mind HUB	student.unsw.edu.au/**mind-hub** Online Self-Help Resources	**Outside Australia Afterhours 24-hour Medibank Hotline**	+61 (2) 8905 0307
Uni and Life Pressures Stress, Financial, Visas, Accommodation & More	**Student Support Indigenous Student Support**	— student.unsw.edu.au/**advisors**		
Reporting Sexual Assault/Harassment	**Equity Diversity and Inclusion (EDI)**	— edi.unsw.edu.au/**sexual-misconduct**		
Educational Adjustments To Manage my Studies and Disability / Health Condition	**Equitable Learning Service (ELS)**	— student.unsw.edu.au/**els**		
Academic and Study Skills	**Academic Language Skills**	— student.unsw.edu.au/**skills**		
Special Consideration Because Life Impacts our Studies and Exams	Special Consideration	— student.unsw.edu.au/**special-consideration**		