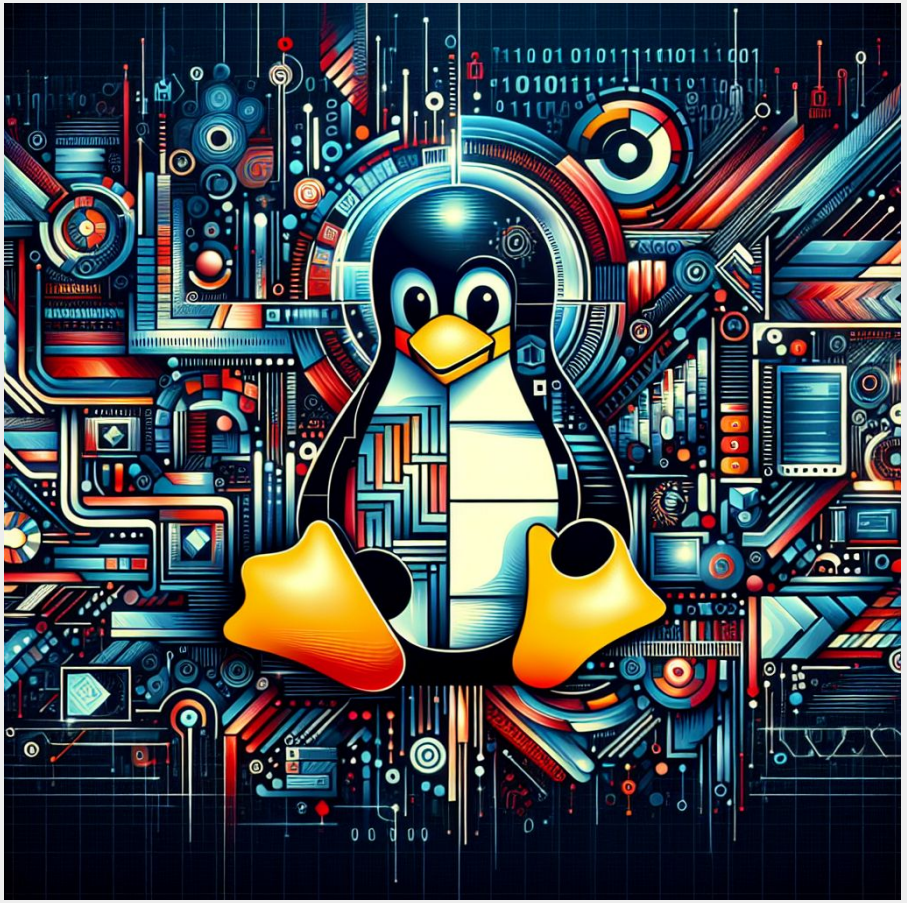
UNSW SYDNEY

COMP1521 24T2 Lec09/10

Operating Systems
File Systems



2024

Angela Finlayson

Format from Hammond Pearce, material from COMP1521

Aside: Linux Manual

The linux manual (**`man`**) is divided into sections.

Important sections for this course include:

1. Executable programs eg. ls, cp
2. System calls
   - we will be looking at many of these today and in the coming weeks
3. Library calls eg. strcpy, scanf

And other sections that you can find out about by using the command **`man man`**
Advice: **`man`** will be available in the exam. Get used to using it!

Operating Systems

This course is a great way to see different areas in computing to

- See what electives you might be interested in!!
- See what area you might want to work in!!

Question 1: What is YOUR favourite operating system?

Question 2: What kind of things do they do for us and what would it be like using a computer without an operating system?

Operating Systems

Operating system (OS) sits between the user and the hardware

The OS effectively provides a virtual machine to each user.

- much easier for user to write code and use machine
- difficult (bug-prone) code implemented by operating system
- coordinates access to resources e.g. file systems, multiple processes

The virtual machine interface can stay the same across different hardware.

- easier for user to write portable code

Operating Systems: Privileged Mode

Needs hardware to provide a **privileged** mode

- code running in privileged mode can access all hardware and memory

Needs hardware to provide a **non-privileged** mode which:

- code running in non-privileged mode can not access hardware directly
- code running in non-privileged mode has limited access to memory
- provides mechanism to make requests to operating system

Operating Systems: Privileged Mode

OS (kernel) code runs in **privileged** mode

OS runs user code in **non-privileged** mode

- user code can only use memory allocated to it

User code can make requests to the OS called **system calls**

- a system call transfers execution to OS code in privileged mode

System Calls

System calls allow programs to request hardware operations

System calls transfer execution to OS code in **privileged** mode

- includes arguments specifying details of request being made
- OS checks operation is valid & permitted
- OS carries out operation
- transfers execution back to user code in **non-privileged** mode

System Calls

Different operating system have different system calls

- e.g Linux system calls are very different Windows system calls

Linux provides 400+ system calls

Examples of operations that might be provided by system call:

- read or write bytes to a file
- create a process (run a program) or terminate a process
- send information over the network (there are some great networks courses at cse)

Mipsy System Calls

`mipsy` provides a virtual machine which can execute MIPS programs

`mipsy` also provides a tiny operating system

`mipsy` system calls

- **`syscall`** instruction
- small number of very specific system calls
- designed for students writing small programs with no library functions

MIPS programs running on real hardware and real OS also use **`syscall`**

Experimenting with Linux System Calls

Linux system calls also have a number

- e.g system call **1** is **`write`** bytes to a file

Linux provides 400+ system calls

```
$ cat /usr/include/x86_64-linux-gnu/asm/unistd_64.h
...
#define __NR_read 0
#define __NR_write 1
#define __NR_open 2
#define __NR_close 3
...
#define __NR_set_mempolicy_home_node 450
```

system calls in linux

syscall command: Not usually used in practice

- Not portable
- Hard to understand

Libc syscall wrapper: Useful sometimes

- does syscall for you and helps with error checking
- More portable than syscall but not portable

Higher level library functions like stdio.h: Useful most of the time

- calls syscall wrapper for you
- portable
- does other cool stuff to make thing easier!

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

Unix Files and standard streams

On Unix-like systems a **file** is sequence/stream of zero or more bytes

- file metadata doesn't record that it is e.g. ASCII, MP4, JPG, …
- file extensions are just hints

Standard streams are treated like files in linux

- stdin, stdout, stderr

Demo: text files vs binary files

Demo: stdout vs stderr

File Descriptors

file descriptors are small integers

- Uniquely identify a stream/file that is open within a process
- Are indexes into a per process OS file descriptor table

OS stores info for each file descriptor such as:

- File offset: current position in the file
- File status: read-only, write-only etc
- Information to locate the actual bytes related to the file/stream

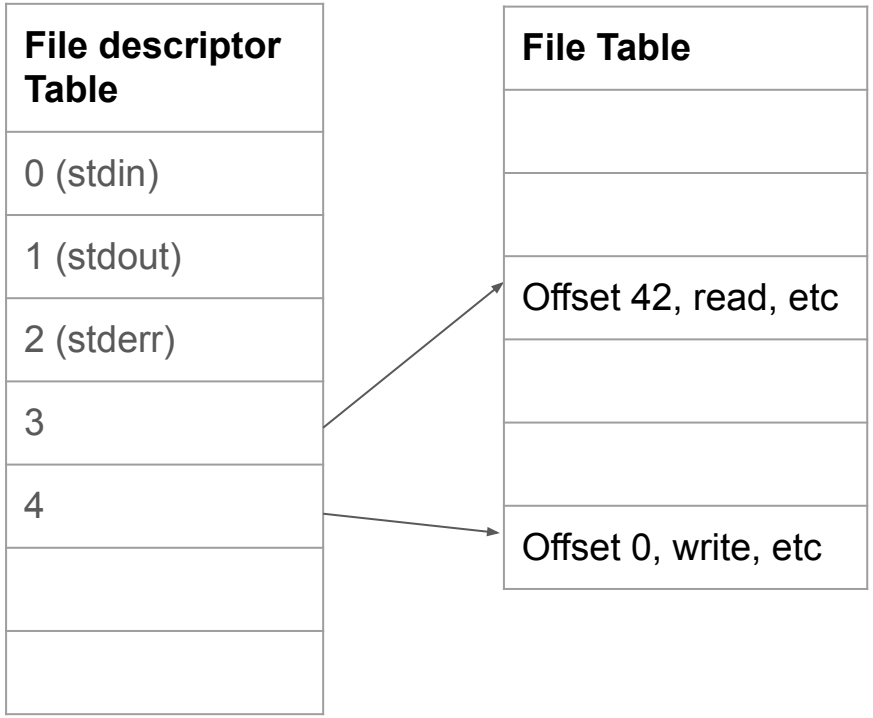
File Descriptors

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 written from, the offset is updated



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

```c
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

Unix C Library Wrappers for System Calls

Unix-like systems have C library functions corresponding to most system calls

- e.g. **open**, **read**, **write**, **close**
- not portable
- some are POSIX compliant and will run on non-Unix systems
- better to use library functions when possible

Typically return **-1** on error and set the error code **`errno`**

Libc wrapper to print message to stdout

```c
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

stdio.h - C Standard Library I/O Functions

system calls provide operations to manipulate files.

libc provides a non-portable low-level API to manipulate files (wrapper functions)

stdio.h provides a portable higher-level API to manipulate files.

- part of standard C library
- available in every C implementation that can do I/O
- functions are portable, convenient & efficient
- on Unix-like systems they will call open()/read()/write() ... with buffering

Use stdio.h functions for file operations unless you have a good reason not to

- e.g .program with special I/O requirements like a database implementation

stdio library to print message to stdout

```
char bytes[] = "Hello, Zac!\n";
printf("%s",bytes);
```

printf will do the write system call for us!

See more ways to print using stdio.h with hello_stdio.h

Source code for hello_stdio.c

Live Coding

syscall vs libc vs stdio.h

hello.c printing to stdout
read_char.c reading byte from stdin

Libc wrapper to open a file

`int open(char *pathname, int flags);`

- open file at **`pathname`**, according to **`flags`**
- **`flags`** is a bit-mask defined in **`<fcntl.h>`**

`int open(char *pathname, int flags, mode_t mode);`

- Use this version when potentially creating a new file
- **`mode`** is an octal number to give the file sensible user access permissions

if successful they return file descriptor (small non-negative int)
if unsuccessful they return **`-1`** and set **`errno`** to value indicating reason

Libc wrapper to open a file

Flag	Use
`O_RDONLY`	open for reading
`O_WRONLY`	open for writing
`O_APPEND`	append on each write
`O_RDWR`	open object for reading and writing
`O_CREAT`	create file if doesn't exist
`O_TRUNC`	truncate to size 0

flags can be combined e.g. `(O_WRONLY|O_CREAT)`

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

Libc wrapper to close a file

```
int close(int fd);
```

- release open file descriptor **`fd`**
- if successful, return **`0`**
- if unsuccessful, return **`-1`** and set errno
- could be unsuccessful if **`fd`** is not an open file descriptor
  - e.g. if **`fd`** has already been closed
- number of file descriptors may be limited (maybe to 1024)
  - limited number of file open at any time, so use **`close()`**

Libc library wrapper for read system call

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

- read (up to) **`count`** bytes from **`fd`** into **`buf`**
  - **`buf`** should point to array of at least **`count`** bytes
  - read cannot check **`buf`** points to enough space
- if successful, number of bytes actually read is returned
- if no more bytes to read, **`0`** returned
- if error, **`-1`** is returned and **errno** set
- file descriptor **current position** in file is updated

Libc library wrapper for read system call

```
ssize_t write(int fd, const void *buf, size_t count);
```

- attempt to write **`count`** bytes from **`buf`** into stream identified by **`fd`**
- if successful, number of bytes actually written is returned
- if unsuccessful**, -1** returned and **errno is set**
- file descriptor **current position** in file is updated

Code Demo

open_read.c

open_write.c

open_issue.c

stdio.h - fopen()

```
FILE *fopen(const char *pathname, const char *mode);
```

- **mode** is string of 1 or more characters including:
  - r open file for reading.
  - w open file for writing
    truncated to 0 zero length if it exists
    created if does not exist
  - a open file for writing
    writes append to it if it exists
    created if does not exist

FILE *

fopen returns a **FILE** pointer

- FILE is an opaque struct - we can not access fields
- FILE stores file descriptor
- FILE may also for efficiency store buffered data

Demo: Modify open_read.c and open_write.c to use stdio.h

stdio.h fclose()

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

- calls close
- number of streams open at any time is limited (to maybe 1024)
- writes unwritten buffered data to the stream

stdio.h reading and writing

```
int fgetc(FILE *stream) ;                    // read a byte
int fputc(int c, FILE *stream);              // write a byte

// read/write array of bytes (fgetc/fputc + loop often better)
size_t fread(void *ptr, size_t size, size_t nmemb,
             FILE *stream);

size_t fwrite(const void *ptr, size_t size, size_t nmemb,
              FILE *stream);
```

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 zero bytes

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

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);
scanf("%s", array);                 // Ok in general.
                                    // 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, ...);
```

Exercise

Implement cp command

1. byte at a time libc.c
2. byte at a time stdio.h
3. using fgets - what is the problem with this approach?

Exercise

Implement cp command

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

We also have implementations using syscall, libc.

Which is the best approach?

IO Performance & Buffering libc vs stdio

Lets 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
```