Operating Systems (Honor Track)

Abstractions 2: Files and I/O
A quick, programmer's viewpoint

Xin Jin

Spring 2022

Acknowledgments: Ion Stoica, Berkeley CS 162

Today: The File Abstraction

- High-Level File I/O: Streams
- Low-Level File I/O: File Descriptors
- *How* and *Why* of High-Level File I/O
- Process State for File Descriptors
- Common Pitfalls with OS Abstractions [if time]

Unix/POSIX Idea: Everything is a "File"

- Identical interface for:
  - Files on disk
  - Devices (terminals, printers, etc.)
  - Networking (sockets)
  - Local interprocess communication (pipes, sockets)
- Based on the system calls **`open()`**, **`read()`**, **`write()`**, and **`close()`**
- Additional: **`ioctl()`** for custom configuration that doesn't quite fit
- Note that the "Everything is a File" idea was a radical idea when proposed
  - Dennis Ritchie and Ken Thompson described this idea in their seminal paper on UNIX called "The UNIX Time-Sharing System" from 1974

Note: What does POSIX stand for?

- POSIX: Portable Operating System Interface (for uniX?)
  - Interface for application programmers (mostly)
  - Defines the term "Unix," derived from AT&T Unix
  - Created to bring order to many Unix-derived OSes, so applications are portable
    - » Partially available on non-Unix OSes, like Windows
  - Requires standard system call interface

The File System Abstraction

- File
  - Named collection of data in a file system
  - POSIX File data: sequence of bytes
    - » Could be text, binary, serialized objects, …
  - File Metadata: information about the file
    - » Size, Modification Time, Owner, etc.
- Directory
  - “Folder” containing files & directories
  - Hierachical (graphical) naming
    - » Path through the directory graph
    - » Uniquely identifies a file or directory
      - /home/ff/pkuos/public_html/sp22/index.html
  - Links and Volumes (later)

Connecting Processes, File Systems, and Users

- **Every process has a *current working directory* (CWD)**
  - Can be set with system call:
    `int chdir(const char *path); //change CWD`
- Absolute paths ignore CWD
  - /home/ff/pkuos
- Relative paths are relative to CWD
  - index.html, ./index.html
    - » Refers to index.html in current working directory
  - ../index.html
    - » Refers to index.html in parent of current working directory
  - ~/index.html
    - » Refers to index.html in the home directory

I/O and Storage Layers

Application / Service

High Level I/O — *Streams*

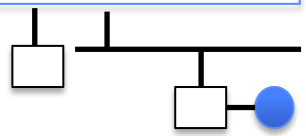
Low Level I/O — *File Descriptors*

Syscall — *open(), read(), write(), close(), …*

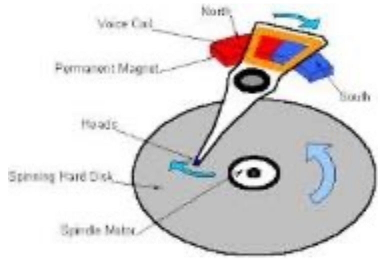
Open File Descriptions

File System — *Files/Directories/Indexes*

I/O Driver — *Commands and Data Transfers*



Disks, Flash, Controllers, DMA



C High-Level File API – Streams

- Operates on "streams" – unformatted sequences of bytes (whether text or binary data), with a position:

```
#include <stdio.h>
FILE *fopen( const char *filename, const char *mode );
int fclose( FILE *fp );
```

Mode Text	Binary	Descriptions
r	rb	Open existing file for reading
w	wb	Open for writing; created if does not exist
a	ab	Open for appending; created if does not exist
r+	rb+	Open existing file for reading & writing.
w+	wb+	Open for reading & writing; truncated to zero if exists, create otherwise
a+	ab+	Open for reading & writing. Created if does not exist. Read from beginning, write as append

- Open stream represented by pointer to a FILE data structure
  - Error reported by returning a NULL pointer

C API Standard Streams – `stdio.h`

- Three predefined streams are opened implicitly when the program is executed.
  - `FILE *stdin` – normal source of input, can be redirected
  - `FILE *stdout` – normal source of output, can too
  - `FILE *stderr` – diagnostics and errors

- STDIN / STDOUT enable composition in Unix

- All can be redirected
  - `cat hello.txt | grep “World!”`
  - **`cat`**’s **`stdout`** goes to **`grep`**’s **`stdin`**

C High-Level File API

```
// character oriented
int fputc( int c, FILE *fp );                  // rtn c or EOF on err
int fputs( const char *s, FILE *fp );          // rtn > 0 or EOF

int fgetc( FILE * fp );
char *fgets( char *buf, int n, FILE *fp );

// block oriented
size_t fread(void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);
size_t fwrite(const void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);

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

C Streams: Char-by-Char I/O

```
int main(void) {
  FILE* input = fopen(“input.txt”, “r”);
  FILE* output = fopen(“output.txt”, “w”);
  int c;

  c = fgetc(input);
  while (c != EOF) {
    fputc(c, output);
    c = fgetc(input);
  }
  fclose(input);
  fclose(output);
}
```

C High-Level File API

```
// character oriented
int fputc( int c, FILE *fp );        // rtn c or EOF on err
int fputs( const char *s, FILE *fp );      // rtn > 0 or EOF

int fgetc( FILE * fp );
char *fgets( char *buf, int n, FILE *fp );

// block oriented
size_t fread(void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);
size_t fwrite(const void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);

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

C Streams: Block-by-Block I/O

```
#define BUFFER_SIZE 1024
int main(void) {
  FILE* input = fopen("input.txt", "r");
  FILE* output = fopen("output.txt", "w");
  char buffer[BUFFER_SIZE];
  size_t length;
  length = fread(buffer, BUFFER_SIZE, sizeof(char), input);
  while (length > 0) {
    fwrite(buffer, length, sizeof(char), output);
    length = fread(buffer, BUFFER_SIZE, sizeof(char), input);
  }
  fclose(input);
  fclose(output);
}
```

Aside: System Programming

- Systems programmers should always be paranoid!
  - Otherwise you get intermittently buggy code
- We should really be writing things like:

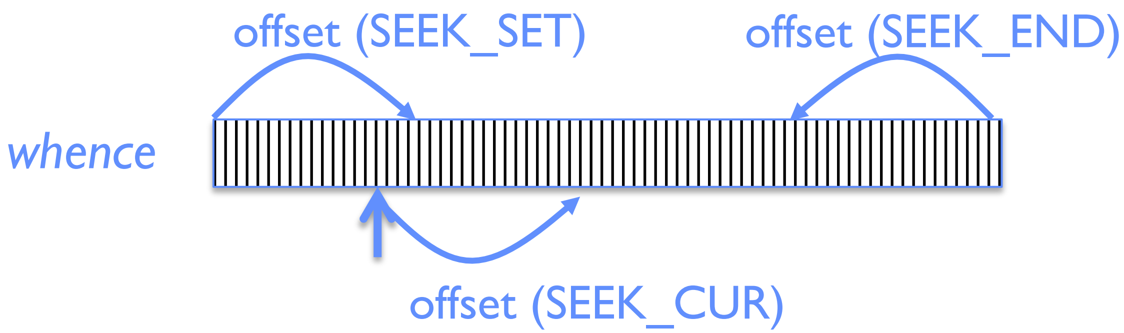
```
FILE* input = fopen(“input.txt”, “r”);
if (input == NULL) {
  // Prints our string and error msg.
  perror(“Failed to open input file”);
}
```

- Be **thorough about checking return values!**
  - Want failures to be systematically caught and dealt with
- I may be a bit loose with error checking for examples in class (to keep short)
  - Do as I say, not as I show in class!

C High-Level File API: Positioning The Pointer

```
int fseek(FILE *stream, long int offset, int whence); // Reposition stream position indicator
long int ftell (FILE *stream) // Get current position in stream
void rewind (FILE *stream) // Set position of stream to the beginning
```

- For `fseek()`, the `offset` is interpreted based on the `whence` argument (constants in `stdio.h`):
  - `SEEK_SET`: Then offset interpreted from beginning (position 0)
  - `SEEK_END`: Then offset interpreted backwards from end of file
  - `SEEK_CUR`: Then offset interpreted from current position



- Overall preserves high-level abstraction of a uniform stream of objects

Today: The File Abstraction

- High-Level File I/O: Streams
- Low-Level File I/O: File Descriptors
- *How* and *Why* of High-Level File I/O
- Process State for File Descriptors
- Common Pitfalls with OS Abstractions

Key Unix I/O Design Concepts

- Uniformity – everything is a file
  - file operations, device I/O, and interprocess communication through open, read/write, close
  - Allows simple composition of programs
    - » find | grep | wc …
- Open before use
  - Provides opportunity for access control and arbitration
  - Sets up the underlying machinery, i.e., data structures
- Byte-oriented
  - Even if blocks are transferred, addressing is in bytes
- Kernel buffered reads
  - Streaming and block devices looks the same, read blocks yielding processor to other task
- Kernel buffered writes
  - Completion of out-going transfer decoupled from the application, allowing it to continue
- Explicit close

Low-Level File I/O: The RAW system-call interface

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>

int open (const char *filename, int flags [, mode_t mode])
int creat (const char *filename, mode_t mode)
int close (int filedes)
```

Bit vector of:

- Access modes (Rd, Wr, …)
- Open Flags (Create, …)
- Operating modes (Appends, …)

Bit vector of Permission Bits:

- User|Group|Other X R|W|X

- Integer return from `open()` is a *file descriptor*
  - *Error indicated by return < 0:* the global `errno` variable set with error (see man pages)
- Operations on *file descriptors*:
  - Open system call created an *open file description* entry in system-wide table of open files
  - *Open file description* object in the kernel represents an instance of an open file
  - Why give user an integer instead of a pointer to the file description in kernel?

C Low-Level (pre-opened) Standard Descriptors

```
#include <unistd.h>
STDIN_FILENO -  macro has value 0
STDOUT_FILENO - macro has value 1
STDERR_FILENO - macro has value 2

// Get file descriptor inside FILE *
int fileno (FILE *stream)

// Make FILE * from descriptor
FILE * fdopen (int filedes, const char *opentype)
```

Low-Level File API

- Read data from open file using file descriptor:

```
ssize_t read (int filedes, void *buffer, size_t maxsize)
```

  - Reads up to `maxsize` bytes – **might actually read less!**
  - returns bytes read, 0 => EOF, -1 => error

- Write data to open file using file descriptor

```
ssize_t write (int filedes, const void *buffer, size_t size)
```

  - returns number of bytes written

- Reposition file offset within kernel (this is independent of any position held by high-level FILE descriptor for this file!

```
off_t lseek (int filedes, off_t offset, int whence)
```

Example: `lowio.c`

```
int main() {
  char buf[1000];
  int     fd = open("lowio.c", O_RDONLY, S_IRUSR | S_IWUSR);
  ssize_t rd = read(fd, buf, sizeof(buf));
  int    err = close(fd);
  ssize_t wr = write(STDOUT_FILENO, buf, rd);
}
```

- How many bytes does this program read?

POSIX I/O: Design Patterns

- Open before use
  - Access control check, setup happens here
- Byte-oriented
  - Least common denominator
  - OS responsible for hiding the fact that real devices may not work this way (e.g. hard drive stores data in blocks)
- Explicit close

POSIX I/O: Kernel Buffering

- Reads are buffered inside kernel
  - Part of making everything byte-oriented
  - Process is **blocked** while waiting for device
  - Let other processes run while gathering result
- Writes are buffered inside kernel
  - Complete in background (more later on)
  - Return to user when data is “handed off” to kernel

- This buffering is part of global buffer management and caching for block devices (such as disks)
  - Items typically cached in quanta of disk block sizes
  - We will have many interesting things to say about this buffering when we dive into the kernel

Low-Level I/O: Other Operations

- Operations specific to terminals, devices, networking, …
  - e.g., `ioctl`
- Duplicating descriptors
  - `int dup2(int old, int new);`
  - `int dup(int old);`
- Pipes – channel
  - `int pipe(int pipefd[2]);`
  - `Writes to pipefd[1] can be read from pipefd[0]`
- File Locking
- Memory-Mapping Files
- Asynchronous I/O

Today: The File Abstraction

- High-Level File I/O: Streams
- Low-Level File I/O: File Descriptors
- *How* and *Why* of High-Level File I/O
- Process State for File Descriptors
- Some Pitfalls with OS Abstractions [if time]

High-Level vs. Low-Level File API

High-Level Operation:

size_t fread(…) {

Do some work like a normal fn…

asm code … syscall # into %eax

put args into registers %ebx, …

special trap instruction

> Kernel:
>
> get args from regs
>
> dispatch to system func
>
> Do the work to read from the file
>
> Store return value in %eax

get return values from regs

Do some more work like a normal fn…

};

Low-Level Operation:

ssize_t read(…) {

asm code … syscall # into %eax

put args into registers %ebx, …

special trap instruction

> Kernel:
>
> get args from regs
>
> dispatch to system func
>
> Do the work to read from the file
>
> Store return value in %eax

get return values from regs

};

High-Level vs. Low-Level File API

Program 1

```
printf("Beginning of line ");
sleep(10); // sleep for 10 seconds
printf("and end of line\n");
```

Program 2

```
write(STDOUT_FILENO, "Beginning of line ", 18);
sleep(10);
write("and end of line \n", 16);
```

- Group discussion
  - What are the behaviors of the two programs? Why?

- Program 1
  - Streams are buffered in user memory
  - Prints out everything at once
- Program 2
  - Operations on file descriptors are visible immediately
  - Outputs "Beginning of line" 10 seconds earlier than “and end of line”

Conclusion

- POSIX idea: “everything is a file”
- All sorts of I/O managed by open/read/write/close