Goal:

* 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

Recall: Synchronization btwn Threads

* Mutual Exclusion: Ensuring only thread does a particular thing at a time
* Critical Section: Code that exactly one thread can execute at once
  + Result of mutual exclusion
* Lock: An object only one thread can hold at a time
  + Provides mutual exclusion
* Offers two **atomic** operations
  + Lock.Acquire()
  + Lock.Release()
* Need other tools for “cooperation” 🡪 Semaphores

Semaphore: A kind a generalized lock

* A semaphore has a non-negative integer value and supports two operations
  + P() or down(): an atomic operation that waits for semaphore to become positive then decrements it by 1
  + V() or up(): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any
* Mutex
  + Called a “binary semaphore”
  + Initial value of semaphore = 1
  + Semaphore.down()

\*Critical Section\*

Semaphore.up()

* Signaling other threads, eg ThreadJoin
  + Init value semaphore = 0
  + ThreadJoin{

Semaphore.down()

}

ThreadFinish{

Semaphore.up()

}

A picture containing text, diagram, screenshot, line

Description automatically generated

Starting New Program (w/Fork & Exec):

A screenshot of a computer code

Description automatically generated with low confidence How to use wait:

A screenshot of a computer code

Description automatically generated with medium confidence

How to signal between two processes:

(sigaction w/ sa\_handler)

1. Declare struct sigaction
2. Set flags, mask, sa\_handler
3. Bind sig\_handler to SIGINT
4. Everytime SIGINT signal used, sighandler is called
   1. SIGINT – ctrl-c
   2. SIGTERM – default for kill command
   3. SIGSTP – ctrl-z
   4. SIGKILL, SOGSTOP – terminate/stop process (can’t be changed with sigaction)

System Call Interface – Set of standardized functions that cross user/kernel divide

Libc 🡪 Standardized functions (including sys calls) linked with your C source code

pthread library: POSIX thread library

POSIX : Portable Operating System Interface (for Unix?)

* Interface for application programmers
* Defines the term “Unix”, dervived from A&T Unix
* Crated to bring order to many Unix-derived OSes, so applications are portable
* Requires standard system call interface

(Everything is a “File” 🡪 Unix/POSIX Idea)

Identical interface for:

* Files on disk
* Devices (terminals, printers, etc)
* Regular files on disk
* Networking (sockets)
* Local IPC (pipes, sockets)

Based on the system calls open(), read(), write(), and close()

Additional: ioctl() for custom configuration that doesn’t quite fit

The File System abstraction:

* File
  + Named collection of data in a file system
  + POXIS File data: sequence of bytes
    - Could be text, binary, serialized objects
  + File Metadata: information about the file
* Directory
  + “Folder” containing files & directories
  + Hierarchical (graphical) naming
    - Path graph directory graph
    - Uniquely identifies a file or directory
  + Links and Volumes (later)
* Every process has CWD (current working directory)
  + Can be set with a system call 🡪 int chdir(const char\* path);
* Absolute paths ignore CWD (/home/oski/cs162)
* Relative paths are relative to CWD (../index/html, ~/index.html, ./index.html)

**Streams:**

Operating on “streams: - unformatted sequence of bytes with a position)

#include <stdio.h>

FILE\* fopen (const char\* filename, const char \*mode)

int fclose(FILE \*fp)

Open stream represented by a **pointer** to a **FILE** data structure

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

FILE \*stderr – diagnostics and errors

Can communicate with different processes with

STDIN / STDOUT enable compositon 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)
* int fputs(const char\* s, FILE\* fp)
* int fgetc(FILE\* fp)
* char \*fgets(char\* buf, int n, FILE\* fp)

Ex:

int main(void){

FILE\* input = fopen(“input.txt”, “r”);

FILE\* output = fopen(“output.txt”, “w”);

int c;

c = fgetc(input);

while(c != EOF){

fputc(output, c);

c = fgetc(input);

}

fclose(input);

fclose(output);

}

//blocked 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);

#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);

}

Positioning the Pointer:

int fseek(FILE\* stream, long int offset, int whence)

* SEEK\_SET 🡺 offset interpreted from the beginning
* SEEK\_END 🡺 offset interpreted backwards from end of file
* SEEK\_CUR 🡺 offset interpreted from current position

long int ftell(FILE\* stream)

void rewind(FILE\* stream)

**Low Level File I/O:**

* Unix 🡪 Uniformity, everything is a file
  + Allows for simple composition of programs
* Open before use
  + Provides opportunity for access control arbitration
  + Sets up the underlying machinery, ie, data structures
* Byte oriented
  + Even if blocks are transferred, addressing is in bytes
* Kernel Buffered reads
  + Streaming and block devices look the same, read blocks yielding processor to other tasks
* Kernel Buffered writes
  + Completion of out-going transfer decoupled from the application, allowing it to continue

Interface:

int open(const char\* filename, int flags, [mode\_t mode])

int create(const char\* filename, mode\_t mode)

int close (int filedes)

* Integer returns form open() is a file descriptor
  + Error indicated by return < 0: the global errno variablr set with error
* Operations on file descriptors
  + Open system call created an open file description in system-wide table of open tables
  + Open file description object in the kernel represents an instance of an open file
  + Q: Why give user an integer instead of a pointer to the file descriptor in kernel?
  + A: Security

Read/Write/Open using file descriptors:

int fileno(FILE\* stream)

FILE\* fdopen(int filedes, const char\* opentype)

ssize\_t read(int filedes, void\*buffer, size\_t maxsize)

ssize\_t write(int filedes, const void\* buffer, size\_t size)

off\_t lseek(int filedes, off\_t offset, int whence)

POSIX I/O: Design Patterns:

* Open before use
* Byte oriented
* Close when done

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
  + Return to user when data is “handed off” to kernel

Low-Level I/O: Other Operations:

* Operations specific to terminals, devices, networking
  + ioctl
* Duplicating descriptors
  + int dup2(int old, int new)
  + int dup(int old)
* Pipes – Channel
* File Locking
* Memory Mapping Files

High-Level vs Low-Level FILE API

* Both perform system call in their respective wrapper function
* A screenshot of a computer program

  Description automatically generated with low confidenceLow Level avoids extra regular work
* fread will buffer 4K bytes at a time in in local memory data structure, so subsequent reads are faster

A screenshot of a computer code

Description automatically generated with low confidence

What’s in the FILE\* returned by fopen?

* File descriptor
* Buffer (array)
* Lock (in case multiple threads use the FILE concurrently)
* … and more (but we don’t need right now)

When you call fwrite, what happens to the data provided?

* It gets written the FILE’s buffer
* If the FILE’s buffer is full, then it is flushed
  + Which means it’s written to the underlying file descriptor
* The C standard lib may flush the FILE more frequently
* A picture containing text, font, screenshot, white

  Description automatically generatedWhen you write code, make the weakest possible assumptions about the data is flushed from FILE buffers

A picture containing text, font, screenshot, algebra

Description automatically generated

You code should behave correctly regardless of when C standard library flushes its buffer

* Add your own calls to fflush so that the data is written when you need to
* Calls to fclose flush the buffer before deallocating the memory and closing the file descriptor

With the low-level file API, we don’t have this problem

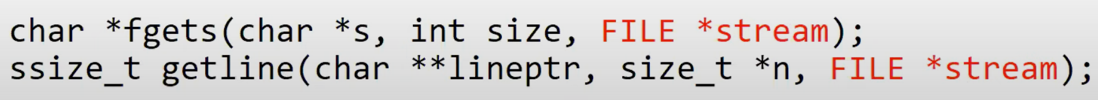
* After write completes, data is visible to any subsequent reads
* Kernel hides all buffering from user (but you don’t get performance advantage of user mode buffering)

Syscalls are 25x more expensive than function calls

* read/write a file byte by byte? Max through out ~10MB/second
* fgetc? Keeps with your SSD (fgetc is a buffered command, big chunk is buffered)

Why buffer in userspace?

* System call operations less capable
  + Simplifies operating system
* Now “read until new line” operation in kernel
  + Kernel agnostic about formatting
  + Solution: Make a big read syscall, find first new line in userspace
    - Do this instead (w/ high level API):



Process State of File Descriptors:

* Recall on successful call to open():
  + File descriptor is returned to user
  + An open file description is created in kernel
* For each process, Kernel maintains mapping from fd to open file description
  + One future system calls, kernel looks up open file description using file descriptor and uses it to service the system call
* What’s in an Open File Description:
  + Inode 🡪 where blocks on disk for your file
  + Offset 🡪 Offset in stream
* A screenshot of a computer program

  Description automatically generated with low confidenceAbstract Representation of Process:
  + User Space: Threads, Reg, Address Space/Memory
  + Kernel: File Descriptor Table

Instead of Closing, let’s fork()!

A diagram of a file

Description automatically generated with low confidence

Why is aliasing the open file descriptor a good idea?

* It allows for shared resources between processes
* Recall, in POSIX, everything is a “file” (files on disk, devices, sockets, IPC ie pipes) 🡪 So parent and child have access to same resources
* Standard File Descriptors 🡪 STDIN, STDOUT, STDERR (allocated for any process)

dup, dup2:

A screenshot of a computer

Description automatically generated with medium confidence