CS 105

"Tour of the Black Holes of Computing"

Input and Output

Topics

- Unix I/O philosophy
- Accessing files
- Reading and writing
- Pipes and filters

The Unix I/O Philosophy

Before Unix, doing I/O was a pain

- Different approaches for different devices, different for files on different devices
- OS made it impossible to do some simple things (e.g. `objdump` a program)

Unix introduced a unified approach

- All files are treated the same
- All devices appear to be files
- Access methods are the same for all files and devices
  - Exception: Berkeley royally screwed up networking
- OS doesn't care about file contents ⇒ any program can read/write any file

Unix Pathnames

Every file (or device) is identified by an *absolute pathname*

- Series of characters starting with and separated by slashes
  - Example: `/home/geoff/bin/mindiffs`
  - Slashes separate *components*
  - All but last component must be *directory* (sometimes called a "folder")
  - Net effect is the folders-within-folders model you're familiar with
- All pathnames start at "root" directory, which is named just "/"

For convenience, *relative pathname* starts at *current working directory*

- Starts without slash
- If CWD is `/home/geoff`, `bin/mindiffs` is same as `/home/geoff/bin/mindiffs`
- CWD is per-process (but inherited from parent)

Pathname Conventions

Some directories have standardized uses:

- `/` is the *root* of the file system tree—everything starts there
- `/bin` and `/usr/bin` contain executable programs ("binaries")
- `/home/blah` is home directory for user `blah`
  - Convenient shorthand (only works in shell): `~/foo` means `/home/geoff/foo` for me
  - blah's executables go in `/home/blah/bin` (aka `~blah/bin`)
- `/etc` has system-wide configuration files
- `/lib` and `/usr/lib` have libraries (also `lib64` on some machines)
- `/dev` contains all devices
  - `/dev/hda` might be hard disk, `/dev/audio` is sound
  - `/dev/null` throws stuff away or gives EOF; `/dev/zero` gives binary zeros; `/dev/random` gives random binary data
- `/proc` and `/sys` contain pseudo-files for system management
  - E.g. `/proc/cpuinfo` tells you all about your CPU chip
- Many others, less important to know about

Unix File Conventions

Earlier systems tried to "help" with file access

- Example: divide file into "records" so you could read one at a time
- Often got in way of what you wanted to do

Unix approach: file (or device) is uninterpreted stream of bytes

- Up to application to decide what those bytes mean
- Implication: if you want to bring up `emacs` on `ctarget`, that's just fine
  - Can produce surprises but also gives unparalleled power

Text files have special convention

- Series of lines, each ended by newline (`'\n'`)
- Implication: last character of any proper text file is newline (editors can enforce)
- Many programs also interpret each line as *fields* separated by whitespace
  - Following that convention unlocks the awesome power of *pipes*

Accessing Files

Programs access files with *open-process-close* model

- Opening a file sets up to use it (like opening a book)
- Processing is normally done in pieces or chunks
- Close tells operating system you're done with that file
  - OS will close it for you if you exit without closing (sloppy but common)

The `open` System Call

To access a new or existing file, use `open`:

- `fd = open(pathname, how [, permissions])`
- *`pathname`* is string giving absolute or relative pathname
- *`how`* is logical OR saying how you want to access file
  - `O_RDONLY` if you are just planning to read it
  - `O_WRONLY` if you intend to write it
    - » Include `O_CREAT | O_TRUNC` and *`permissions`* if you want to (re)create it
    - » *`permissions`* are usually 0666 or symbolic equivalent (PITA, IMHO)
  - `O_RDWR` to both read and write
- *`fd`* is returned small-integer *file descriptor*, used in all other calls
  - -1 on error, as usual
  - fd 0 is already connected to *standard input* of the process
  - fd 1 is *standard output*, used for the "normal" results of the program
  - fd 2 is *standard error*, used for messages intended for humans

Closing a File

`result = close(fd)`

- Closing says "I'm all done, release resources"
- CLOSING CAN FAIL!!!
  - Returns -1 on error
  - Some I/O errors are delayed for efficiency reasons
  - Good programs *must* check result of close
- After closing, *`fd`* is invalid (but same number might be reused by OS later)

OK, That's the Easy Stuff

Actually there's more easy stuff…but it's not as important

- `link`: create alternate name (efficient but now mostly obsolete)
- `symlink`: create alternate name (more flexible than `link`, now most popular)
- `unlink`: oddly, it's how you delete files
- `stat/fstat`: find out information about files (size, owner, permissions, etc.)
- `chdir`: like `cd` command but for processes instead of command line
- Too many more to list all; learn 'em when you need 'em

Reading and Writing

Fundamental truth: files don't necessarily fit in memory

- Implies programs have to deal with files one piece at a time
- `stdio` library (covered later) makes that easier for text files
- Understanding underlying mechanisms is important

Every open file has an associated *file position* maintained by the OS

- Position starts at 0
- Updated automatically by every `read` or `write`
- Next operation takes place at new position
- If necessary, can discover or reset position with `lseek`

Reading Data

`nbytes = read(`*`fd, buffer, buffer-size`*`)`

- *`fd`* is a file descriptor returned by a previous open (or 0, for `stdin`)
- *`buffer`* is the *address* of an area in memory where the data should go
  - Often a `char[]` array
  - But can be (e.g.) the address of a `struct`
- *`buffer-size`* is the maximum number of bytes to read (usually array or struct size)
- `nbytes` is how many bytes were actually read

`read` will collect data from the given file and stick it in *`buffer`*

- Subsequent `read` will return the data *after* what the last `read` gave you
- So `read, read, read` will give you all the data in the file—one chunk at a time

`read` will *NEVER* return more than what you asked for

- But it has the right to return less! You may have to re-ask for more data

`read` returns 0 when there is no more data ("end of file" or EOF)

The Canonical File Loop

```
while (1) {
    nbytes = read some data into a "buffer" (often from stdin)
    if (nbytes == -1)
        handle error
    else if (nbytes == 0)
        break;        // End of file (EOF); we're all done
    process nbytes of data in some way
    write results (often to stdout) from same or another buffer
}
```

Writing Data

***nbytes* = `write`(*fd, buffer, buffer-size*)**

- *fd* is a file descriptor returned by a previous open (or 1 or 2, for `stdout` or `stderr`)
- *buffer* is the *address* of an area in memory where the data comes from
- *buffer-size* is the number of bytes to write (usually size of array or struct)
- `nbytes` is how many bytes were actually written

`write` will collect data from the given *buffer* and write it to the chosen file

- Next `write` will add data after where the last `write` changed things
- Thus `write`, `write`, `write` will gradually grow the file

`write` will *NEVER* write more than what you asked for

- But it has the right to write less!
- You may have to re-ask to finish the work

Fun fact: if `write` fails you might not find out until `close` (for efficiency)

Sample (Bad) Program: `cat`

Copy `stdin` to `stdout` (works on files of any size):

```
int main()
{
    int n;
    char buf[1];
    while ((n = read(0, buf, sizeof buf)) > 0) {
        if (write(1, buf, n) == -1)
            return 1;
    }
    if (close(1) == -1)
        return 1;
    return 0;
}
```


Improving `cat`

Inconvenient to use

- Must connect desired file to stdin (using < sign)
- Nicer to be able to put file name on command line (as real `cat` does)
- See https://www.cs.hmc.edu/~geoff/interfaces.html for thorough discussion

As written, horribly inefficient

- One system call per byte (roughly 6000 cycles each)
- OS can transfer 8K bytes in as little as 2K cycles
  - Transfer done in 8-byte longs, >1 cycle per long
- Straightforward modification

Error checking and reporting are…primitive

- Again, straightforward

Handles “short reads” but must also handle “short writes”

Binary I/O

There is no law saying that `buf` has to be an array of chars:

```
struct info {
    int count;
    double total;
};
...
    struct info stuff;
    off_t cur_pos = lseek(data_fd, 0, SEEK_CUR);
    nbytes = read(data_fd, &stuff, sizeof stuff);
    ++stuff.count;
    stuff.total += value;
    lseek(data_fd, cur_pos, SEEK_SET);
    nbytes = write(data_fd, &stuff, sizeof stuff);
```

The Guts of fgrep

```
n = strlen(search_string);
while (1) {
    nbytes = read(fd, buf, sizeof buf);
    for (int i = 0;  i < sizeof buf - n;  i++) {
        if (strncmp(&buf[i], search_string, n) == 0)
            /* Print line containing search_string */
    }
}
```

Big problem: What if line or search string runs across two buffers?

Fixing fgrep

Solution to problem: Process one entire line at a time

- Read 8K (or whatever) into a *data buffer*
- Copy one line at a time into a separate *line buffer*
  - If line continues past buffer end (i.e., no newline found), refill data buffer
- Repeat for next line

Same should be done for output

- Collect whatever you're writing into *output buffer*
- When buffer gets full, *flush* it to output file
- This way there's one system call per 8K of output

Happens often enough that there's a library to do it: `stdio`

Using stdio

The "standard I/O" package takes care of intermediate buffers for you

- `fopen`, `fclose`
- `getc`, `putc`: read and write characters (*extremely* efficient; don't be scared of them)
- `fgets`, `fputs`: handles one line at a time
- `fread`, `fwrite`: deals with *n* bytes; useful for binary I/O
- `fseek`, `ftell`, `rewind`: equivalents of lseek
- `scanf`, `fscanf`: bad input parsing; only useful in primitive situations
- `printf`, `fprintf`: formatted output; old friends by now
- `setbuf`, `setlinebuf`, `fflush`: force output to appear

fopen and fclose

```
#include <stdio.h>
FILE* some_stream = fopen(pathname, mode);
```

- Returns a *stream handle*, or `NULL` on error
- *`pathname`* same as for open
- *`mode`* is a string:
  - `"r"` to read, `"w"` to write new file; other options available
  - Sadly, `"rb"` and `"wb"` needed to handle binary files on some stupid OSes

```
int error = fclose(some_stream);
```

- Returns `0` on success, `EOF` (a `#define`d constant) on error

Character and Line I/O

`int ch = fgetc(some_stream);`

`int result = fputc(ch, some_stream);`

- Both return `EOF` on *either* end-of-file (`fgetc` only) or error
  - Must use `ferror` or `feof` to distinguish

`char line[some_size];`

`char* result = fgets(line, max_size, some_stream);`

`int result = fputs(line, some_stream);`

- `fgets` includes trailing newline (if any) and guaranteed `'\0'` (compare `gets`)
- `fgets` returns `NULL` on error or EOF, otherwise useless pointer to *line*
- `fputs` expects trailing null byte; you must supply newline at end
- `fputs` returns `0` on success, EOF on error

`printf` and `fprintf`

stdout
stderr

`int nbytes = printf(format, ...);`

`int nbytes = fprintf(some_stream, format, ...);`

- Both return number of bytes printed, or `-1` on error
- `printf` automatically goes to standard output (`stdout`)
- *format* determines how to interpret remaining options
  - Most characters shown as-is
  - Percent sign means "substitute next argument here"
  - Complex and powerful notation

Example:

```
printf("The %s Department has %d professor%s.\n",
  dept_name, dept_size, dept_size == 1 ? "" : "s");
```


The Output-Buffering Problem

Sending data to a file or device is expensive

- Refer back to byte-at-a-time implementation of `cat`

The `stdio` package automatically *buffers* output and sends it in bunches

Sometimes you want to see output right away

- Prompts to user
- Output on terminal
- Information in log file

`stdio` offers three options and a function to help

- Normal buffering: saves up 4K or 8K, writes all at once (highly efficient)
- Line buffering: write immediately after every newline
- No buffering: write every character immediately (inefficient; rarely a good idea)

Controlling Output Buffering

`stdio` tries to make sensible automatic choices

- Chooses line buffering if an output file (including `stdout`) is connected to a terminal
- Otherwise uses normal buffering

Multiple ways to override the default choice:

- `fflush(some_stream)` says "send out everything you've saved, *now*"
- `setlinebuf(some_stream)` says "I want line buffering even if it's not going to a terminal"
  - Useful, e.g., for log files
- `setbuf(some_stream, NULL)` says "Don't buffer anything; write every character *now*"
  - Rarely a good idea; better to write a few characters and then use `fflush`
- `fflush` returns `EOF` on error; others can't fail

File Sharing

Every open (and thus `fopen`) creates a new entry in OS's *open file table*

- Contains identity of file plus (important) *current file position*

Forked children inherit parent's open file descriptors

- By implication, they share current file position
- Nice for output: means both parent and child can append to same file without clobbering each other's data
  - But need to be careful about when flushing happens!
- Confusing for input: if child reads line 1, parent will next see line 2
  - Even more confusing: if both use `stdio` buffering, child will see first 8K, parent next 8K
  - Could result in intermixed lines

`exec` does *not* close descriptors (mostly), so those also shared

- End result: when you run a program, it's still connected to your terminal

Some other ways to share, but not critical here

Unix Filters

Most Unix commands are "filters" that can read from `stdin` and write to `stdout`

- Interpret data as lines of fields, separated by whitespace
- Do one simple task ("Do one thing and do it well")

Result: you can do powerful tasks by feeding output of one command to input of another

- Simple example:
  - `ls /etc > temp1`
  - `grep "time" < temp1 > temp2`
  - `wc -l < temp2`
  - `rm temp1 temp2`
- FWIW, returns 3 on my machine, 2 on Wilkes

Unix Pipes

Drawback of filters: all those temporary files

Solution: a *pipe* connects standard output of one command to standard input of another

Returning to previous example:

- `ls /etc | grep "time" | wc -l`
- This example also illustrates why `stderr` was invented

20 Filters Worth Learning About

`grep, egrep, fgrep`
`sort`
`tr`
`echo`
`diff`
`cmp`
`wc`
`sed` [only basics]
`awk` [complex]
`uniq`
`head, tail`
`comm`
`find` [weird syntax]
`xargs` [only useful with `find`]
`cut`
`join`
`tee`

BTW, Here's How I Made That List

```
cat ~/bin/* 2>/dev/null | fgrep ' | ` \
  | tr '|' \\012 | awk '{print $1}' \
  | sort | uniq | grep '^[a-z]' \
  | egrep -iv '^[a-z0-9_]+=` \
  | less
```

Piece by piece:

1. Collect all my shell scripts, ignoring errors, and look for lines with pipes
2. Convert pipe symbols to newlines and print first nonblank field on each line
3. Sort result, choose unique lines, choose only those starting with a letter
4. Discard lines that start with a variable name followed by `"="`
5. Feed it into `less` so I can eyeball the 147 lines of output

9 commands strung together: this is exactly the power of pipes!