CMSC 313
Introduction to Computer Systems
Lecture 18
System-Level I/O and Data Representation

Alan Sussman
als@cs.umd.edu

Administrivia

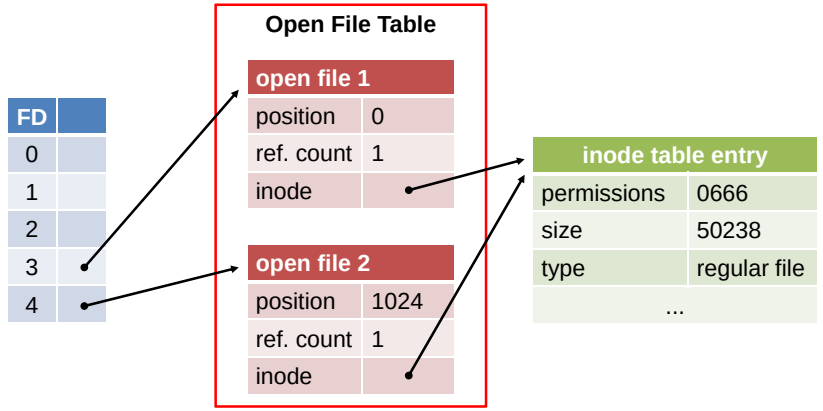
- Project 5 out Friday, due Nov. 19 at 6PM
- Exam #2 on Monday, Nov. 16
- Read Chapter 11, start on Sections 2.1-2.4

Chapter 11, Bryant and O'Hallaron

SYSTEM-LEVEL I/O

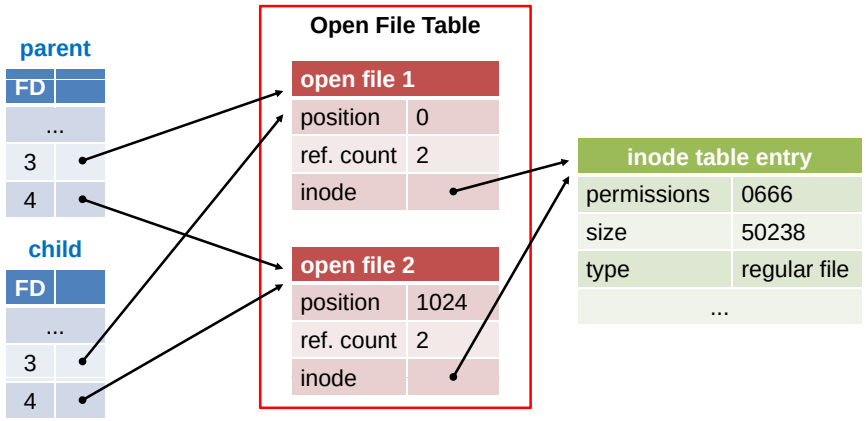
Opening the same file twice

```
fd1 = open("file.txt", O_RDONLY);
fd2 = open("file.txt", O_RDONLY);
read(fd2, buffer, 1024);
```



After a fork

```
fd1 = open("file.txt", O_RDONLY);
fd2 = open("file.txt", O_RDONLY);
read(fd2, buffer, 1024);
fork();
```



Interprocess Communication

- Processes can communicate with each other via various means, including signals and pipes
- Signals have a limited "vocabulary" - only 64 signals on some machines; other limitations as well
- Pipes allow arbitrary strings to be written from one process to another
- Two kinds of pipes: FIFOs (or "named pipes", created by **`mkfifo()`**), and anonymous pipes (often just called "pipes", created by **`pipe()`**)

Pipes

- Note: much of this information is from the pipe man page, in section 7 ("**`man 7 pipe`**")
  - the **`pipe()`** function discussed below is in section 2
- Provide a unidirectional IPC channel, with a read end and a write end
- Created with the **`pipe()`** function:

```
#include <unistd.h>
int pipe(int fd[2]);
```

  - **`fd`** needs to be an array of 2 elements
  - return value is 0 on success, -1 on error
  - On success, a new pipe is created, **`fd[0]`** will be the file descriptor for the read end, and **`fd[1]`** will be the file descriptor for the write end

Pipes, cont.

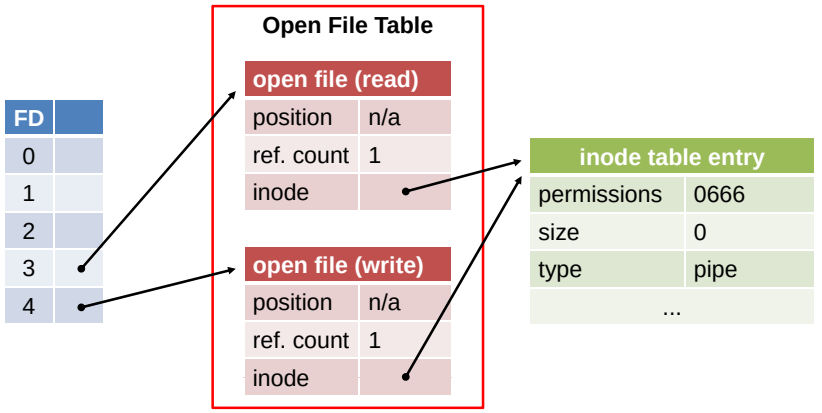
- An anonymous pipe's file descriptors are only visible in the process that created the pipe (and its children)
- I/O is blocking – read() from an empty pipe does not return until data is in the pipe, and write() to a full pipe does not return until sufficient space exists to complete the write
- Reads on the pipe will return EOF if no process has the write end open; writes to a pipe that has no readers cause an error
- Can't move the file position around (no seeking)

Pipes, cont.

- Generally, we create a pipe, then fork()
  - parent and child can then communicate via the pipe
- Because of the blocking I/O, we need to close ends of the pipe that we're not using, both immediately after the fork and once we're finished reading/writing
  - otherwise, read won't signal EOF appropriately or cause errors when we want to

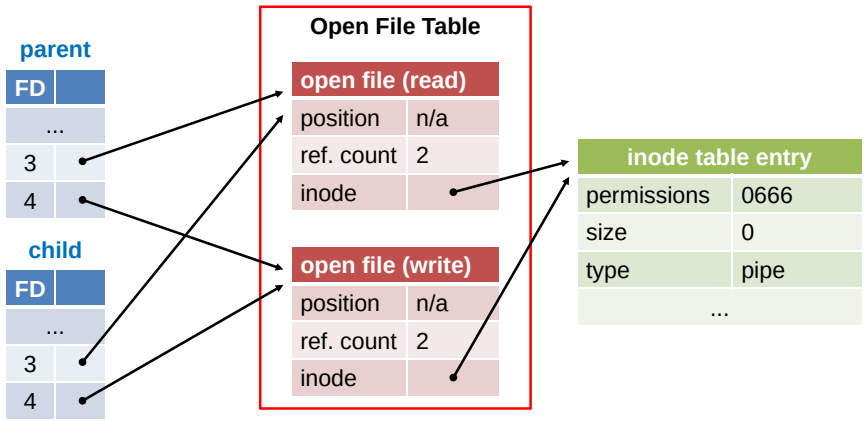
Opening a pipe

```
int fd[2];
pipe(fd);
```



After a fork

```
int fd[2];
pipe(fd);
fork();
```



An example with `pipe()`

```
/* #include statements and most error checking omitted */

int main() {
  int fd[2];
  char buf[BUFSIZ];  /* BUFSIZ is defined in stdio.h */

  if (pipe(fd) < 0)
    err(EX_OSERR, "pipe error");
  if (fork()) {  /* parent code */
    close(fd[1]);
    read(fd[0], buf, BUFSIZ);
    printf("Message from child: %s\n", buf);
  }
  else {  /* child code */
    char msg[] = "Hi there!";
    close(fd[0]);
    write(fd[1], msg, sizeof(msg));
  }
  return 0;
}
```

I/O redirection

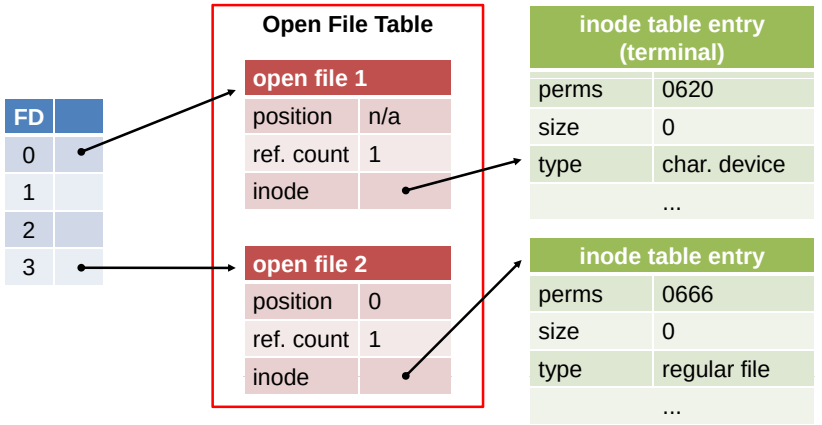
- The **`dup2()`** function copies an open file table entry from one file descriptor to another

```
#include <unistd.h>
int dup2(int old_fd, int new_fd);
```

  - returns -1 on error, **`new_fd`** on success
  - **`new_fd`** is closed first if necessary (if it's already open)
- This can be used for things like I/O redirection, or to allow reading from a file named on the command line (instead of stdin)

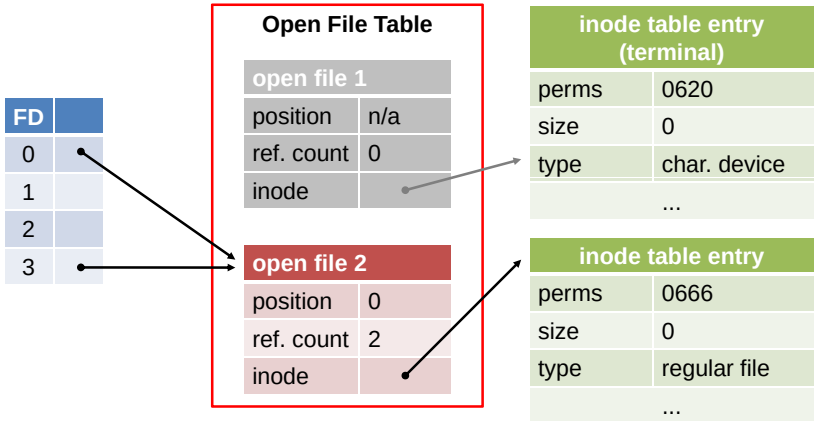
Before **`dup2()`**

```
int fd = open(filename, O_RDONLY);
```



After **`dup2()`**

```
int fd = open(filename, O_RDONLY);
dup2(fd, STDIN_FILENO);
```



Example using **`dup2()`**

- Allow reading from stdin or named file:

```
int main(int argc, char **argv) {
  ...
  if (argc > 1) {
    int fd = open(argv[1], O_RDONLY);
    dup2(fd, STDIN_FILENO);
    close(fd);
  }
  /* read from stdin from here on */
  ...
}
```

Standard I/O vs. Unix I/O

- Standard I/O (**`FILE *`**, **`fgets()`**, **`printf()`**, etc.) generally features buffers; Unix I/O does not
- Mixing the two can cause weird things to happen; what does this output?

```
printf("u");
write(STDOUT_FILENO, "m", 1);
printf("d\n");
```

- **`mud`**
- Even stranger things happen with reading...

Mixing buffered/unbuffered reads

`buf.c`:

```
#include <stdio.h>
#include <unistd.h>

#define BUFFER_SZ 100
static char buffer[BUFFER_SZ];

int main() {
  char line_s[11];
  char line_u[11] = {0};
  setbuffer(stdin, buffer, BUFFER_SZ);
  fgets(line_s, 11, stdin);
  read(STDIN_FILENO, line_u, 10);
  printf("L1: %s\n", line_s);
  printf("L2: %s\n", line_u);
  fgets(line_s, 11, stdin);
  read(STDIN FILENO, line_u, 10);
  printf("L3: %s\n", line_s);
  printf("L4: %s\n", line_u);
  return 0;
}
```

`data.txt`:

```
aaaaaaaaaabbbbbbbbbbccccccccc
ddddddddddeeeeeeeeeeffffffffff
gggggggggghhhhhhhhhhiiiiiiiiii
jjjjjjjjjjkkkkkkkkkkllllllllll
mmmmmmmmmmnnnnnnnnnnoooooooooo
```

Execution:

```
$ ./buf < data.txt
L1: aaaaaaaaaa
L2: kkkkkkkkkk
L3: bbbbbbbbbb
L4: llllllllll
```


Parts of Sections 2.1-2.4, Bryant and O'Hallaron

DATA REPRESENTATION

Representing characters

- We need:
  - to be able to represent common characters
  - to have standards so computers can interoperate
- Common formats
  - ASCII
    - is the most commonly used character code
    - uses 7 bits for characters (stored in 8 bits normally)
  - EBCDIC
    - an 8-bit code, used now only by some IBM mainframes
  - UNICODE
    - a family of encodings - 8, 16, and 32 bits per character
    - allows a greater variety of characters
    - is able to represent virtually any character in use today in any language, and some no longer in use

ASCII

- Represents normal characters on US keyboards
  - **`A`**-**`Z`** (the characters numbered 65-90)
  - **`a`**-**`z`** (97-122)
  - **`0`**-**`9`** (48-57)
  - space (32)
  - control characters (0-31, 127)
    - the first 26 (after 0) of the 33 ASCII control characters have names Ctrl-A - Ctrl-Z
    - for example, ASCII character 13, Ctrl-M, is CR (carriage return) (**`\r`** in C); ASCII char. 9, Ctrl-I, is HT (horizontal tab) (**`\t`** in C)
  - punctuation: **`!@#$%^&*()_+-=[]{}|\;:"'<>?,./`** (the remaining characters)
- The UNIX command "**`man ascii`**" shows the ASCII character set (for those who don't read the project footnotes)

UNICODE

- Different representations
- UTF-32: a 32-bit representation of all characters
  - all characters are the same size
  - uses lots of space (twice as much as UTF-16 for most things, four times as much as ASCII for many things)
- UTF-16: a 16-bit representation of characters
  - some characters are stored in two-character forms
  - is popular since most things can be represented in 16 bits
- UTF-8: an 8-bit representation of characters
  - provides backwards compatibility with ASCII
    - the low 7 bits are exactly ASCII
    - if the high bit is on it indicates part of UNICODE extensions
  - popular for web and other applications

Representing unsigned integers

- **All data** is stored in binary
  - all digits are 0 or 1
- In an unsigned number every bit position *i* represents the value 2*i*, where *i* is 0 for the rightmost bit. The value of a number is the sum of the values of the bit positions containing a 1.
- Example bit position values for an 8-bit number:

128	64	32	16	8	4	2	1

- The range of values that can be stored is 0 to $2^n - 1$, where *n* is the number of bits used
  - for example, using 16 bits, the numbers 0 to 65,535 can be represented

Representing signed integers

- Signed integers are usually stored using two's complement
  - the leftmost bit indicates if a number is positive (0) or negative (1)
- To get the two's complement representation of a negative value, flip all the bits of the positive value and add 1
- Two's complement allows easy addition of positive and negative numbers (just ignore overflow)
- The range of values that can be stored is -2^{n-1} to $2^{n-1}-1$ for *n* bits
  - for example, using 16 bits, the numbers -32,768 to 32,767 can be represented
- Two's complement isn't the same thing as an unsigned number with a sign bit
  - -5 is ~(5) + 1 = 11111010 + 1 = 11111011, not 10000101

How are floats/doubles represented?

- Each number has three parts:
  - its sign (s), which is 0 for positive numbers, and 1 for negative numbers
  - a mantissa (m), which represents a number between 0 and 1
    - it's represented as a binary number, i.e., ½ = 0.1
    - it's normalized into [1,2) (the exponent is adjusted as needed)
  - an exponent (e), which designates the position of the decimal point
- A number is $(-1)^s \times m \times r^e$, where r is the radix
  - the number $6132.789_{10} = 1 \times 6.132789 \times 10^3$ (the radix is 10 for this example)
  - the number $0.05_{10} = 1 \times 5.0 \times 10^{-2}$ (radix is also 10)
  - the number $-1001.1110_2 = -1 \times 1.001110 \times 2^3$ (here the radix is 2)
- This is much like scientific notation, with the addition of the sign as a factor, and the ability to use a base other than 10