

# System-Level I/O

Kai Shen

1

## I/O Overview

- Data input-from/output-to external devices (disk storage, network, terminal ...)
- I/O is privileged (must be managed by the operating system)
- Our study
  - Focus on the application interface
  - A bit of behind-the-scene look to help our understanding
  - Much more in OS and networking courses later
- Outline:
  - Operating system I/O (taking Unix as an example)
  - User-level runtime library (C language runtime)
  - Internal representation, sharing, and redirection

2

## Unix I/O and Files

- Data organization is at the center of I/O.
- A Unix *file* is a sequence of  $m$  bytes:
  - $B_0, B_1, \dots, B_k, \dots, B_{m-1}$
  - "unstructured"
- Basic Unix I/O operations (system calls):
  - Opening and closing files: `open()` and `close()`
  - Reading and writing a file: `read()` and `write()`
  - Changing the **current file position**: `lseek()`

Current file position = k

3

## Unix File Types

- Regular files
  - File containing user/app data (binary, text, whatever)
- Directory files
  - A file that contains the names and locations of other files
- Device files
  - Terminals (character device) and disks (block device)
- Unix pipes for inter-process communication
- Sockets for network communications
- `/proc`: An convenient way to access kernel information

4

## Opening Files

- Opening a file informs the kernel that you are getting ready to access that file

```
int fd; /* file descriptor */

if ((fd = open("/etc/hosts", O_RDONLY)) < 0) {
    perror("open");
    exit(1);
}
```

- Returns a small identifying integer *file descriptor*
  - `fd == -1` indicates that an error occurred
- Each process created by a Unix shell begins life with three open files associated with a terminal:
  - 0: standard input
  - 1: standard output
  - 2: standard error

5

## Closing Files

- Closing a file informs the kernel that you are finished accessing that file

```
int fd; /* file descriptor */
int retval; /* return value */

if ((retval = close(fd)) < 0) {
    perror("close");
    exit(1);
}
```

- Always check return codes, even for seemingly benign functions such as `close()`

6

## Reading Files

- Reading a file copies bytes from the current file position to memory, and then updates file position

```
char buf[512];
int fd; /* file descriptor */
int nbytes; /* number of bytes read */

/* Open file fd ... */
/* Then read up to 512 bytes from file fd */
if ((nbytes = read(fd, buf, sizeof(buf))) < 0) {
    perror("read");
    exit(1);
}
```

- Returns number of bytes read from file `fd` into `buf`
  - Return type `ssize_t` is signed integer
  - `nbytes < 0` indicates that an error occurred
  - Short counts** (`nbytes < sizeof(buf)`) are possible and are not errors!

7

## Short Counts

- Short counts can occur in these situations:
  - Encountering (end-of-file) EOF on reads
  - Reading text lines from a terminal
  - Reading and writing network sockets or Unix pipes
- Deal with short counts:
  - Check return count. Don't assume you've read/written all.
  - To reach desired count, you may need to read/write again & again.

8

### Simple Unix I/O example

- Copying standard in to standard out, one byte at a time

```

#include "csapp.h"

int main(void)
{
    char c;

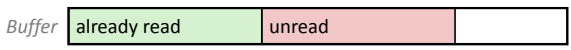
    while(read(STDIN_FILENO, &c, 1) != 0)
        write(STDOUT_FILENO, &c, 1);
    exit(0);
}
    
```

- Works but not efficient.

9

### Buffered I/O: Motivation

- Applications sometimes read/write one character (a line) at a time
- Implementing as Unix I/O calls expensive
  - read** and **write** require OS kernel calls
    - 10,000 clock cycles
- Solution: Buffered read
  - Use Unix **read** to grab block of bytes
  - User input functions take one byte at a time from buffer
    - Refill buffer when empty



10

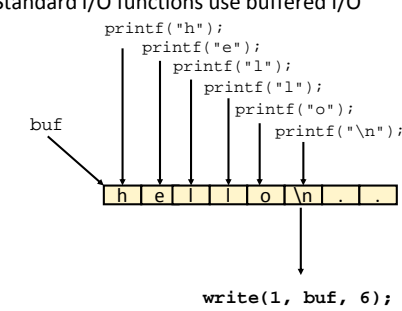
### C Utility I/O Functions

- The C library (`libc.so`) contains a collection of higher-level *I/O* functions
- Examples of standard I/O functions:
  - Opening and closing files (**fopen** and **fclose**)
  - Reading and writing bytes (**fread** and **fwrite**)
  - Reading and writing text lines (**fgets** and **fputs**)
  - Formatted reading and writing (**fscanf** and **fprintf**)
- I/O models open files as *streams*
  - Abstraction for a file descriptor and a buffer in memory.

11

### Buffering in C Utility I/O

- Standard I/O functions use buffered I/O



- Buffer flushed to output fd on `"\n"` or **fflush()** call

12

### Unix I/O vs. C Utility I/O

- C Utility I/O is implemented using low-level Unix I/O

```

fopen fdopen
fread fwrite
fscanf fprintf
sscanf sprintf
fgets fputs
fflush fseek
fclose
            
```

```

open read
write lseek
stat close
            
```

C application program

C library

Operating system I/O (accessed via system calls)

- Which ones should you use in your programs?

13

### Unix I/O vs. C Utility I/O

- Pros for Unix I/O
  - Unix I/O is the most general and lowest overhead form of I/O.
    - All other I/O packages are implemented using Unix I/O functions.
  - Unix I/O provides functions for accessing file metadata.
  - Unix I/O functions are async-signal-safe and can be used safely in signal handlers.
- Pros for C Utility I/O
  - Buffering increases efficiency by decreasing the number of **read** and **write** system calls
  - Short counts are handled automatically

14

### How the Unix Kernel Represents Open Files?

- Two descriptors referencing two distinct open disk files. Descriptor 1 (stdout) points to terminal, and descriptor 4 points to open disk file

**Descriptor table**  
[one table per process]

**Open file table**  
[shared by all processes]  
[open file instance]

**v-node table**  
[shared by all processes]  
[physical file on device]

```

stdin fd 0
stdout fd 1
stderr fd 2
fd 3
fd 4
            
```

File A (terminal)

File pos  
refcnt=1  
:

File B (disk)

File pos  
refcnt=1  
:

File access

File size  
File type  
:

File access

File size  
File type  
:

15

### File Sharing

- Two distinct descriptors sharing the same disk file through two distinct open file table entries
  - E.g., Calling **open** twice with the same **filename** argument

**Descriptor table**  
[one table per process]

**Open file table**  
[shared by all processes]  
[open file instance]

**v-node table**  
[shared by all processes]  
[physical file on device]

```

stdin fd 0
stdout fd 1
stderr fd 2
fd 3
fd 4
            
```

File A (disk)

File pos  
refcnt=1  
:

File B (disk)

File pos  
refcnt=1  
:

File access

File size  
File type  
:

16

### How Processes Share Files: Fork()

- A child process inherits its parent's open files
  - Note: situation unchanged by `exec` functions (use `fcntl` to change)
- Before `fork()` call:
  - Descriptor table** [one table per process]
  - Open file table** [shared by all processes] [open file instance]
  - v-node table** [shared by all processes] [physical file on device]

17

### How Processes Share Files: Fork()

- A child process inherits its parent's open files
- After `fork()`:
  - Child's table same as parent's, and +1 to each `refcnt`

- Descriptor table** [one table per process]
- Open file table** [shared by all processes] [open file instance]
- v-node table** [shared by all processes] [physical file on device]

18

### I/O Redirection

- Question: How does a shell implement I/O redirection?
  - `unix> ls > foo.txt`
- Answer: By calling the `dup2(oldfd, newfd)` function
  - Copies (per-process) descriptor table entry `oldfd` to entry `newfd`

19

### I/O Redirection Example

- Step #1: open file to which `stdout` should be redirected
  - Happens in child executing shell code, before `exec`

- Descriptor table** [one table per process]
- Open file table** [shared by all processes] [open file instance]
- v-node table** [shared by all processes] [physical file on device]

20

### I/O Redirection Example (cont.)

- Step #2: call `dup2(4, 1)`
  - cause `fd=1` (stdout) to refer to disk file pointed at by `fd=4`

**Descriptor table**  
[one table per process]

stdin	fd 0
stdout	fd 1
stderr	fd 2
	fd 3
	fd 4

**Open file table**  
[shared by all processes]  
[open file instance]

File A
File pos
refcnt=0
⋮
File B
File pos
refcnt=2
⋮

**v-node table**  
[shared by all processes]  
[physical file on device]

File access
File size
File type
⋮

21

### Fun with File Descriptors

```

int main(int argc, char *argv[])
{
    int fd1, fd2, fd3;
    char c1, c2, c3;
    char *fname = argv[1];
    fd1 = open(fname, O_RDONLY, 0);
    fd2 = open(fname, O_RDONLY, 0);
    fd3 = open(fname, O_RDONLY, 0);
    dup2(fd2, fd3);
    read(fd1, &c1, 1);
    read(fd2, &c2, 1);
    read(fd3, &c3, 1);
    printf("c1 = %c, c2 = %c, c3 = %c\n", c1, c2, c3);
    return 0;
}
    
```

- What would this program print for file containing "abcde"?
  - c1 = a, c2 = a, c3 = b

22

### Disclaimer

These slides were adapted from the CMU course slides provided along with the textbook of "Computer Systems: A programmer's Perspective" by Bryant and O'Hallaron. The slides are intended for the sole purpose of teaching the computer organization course at the University of Rochester.

23