File I/O

Hyo-bong Son (proshb@csl.skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu

SKK UNIVERSITY

Unix Files

- **A Unix file is a sequence of *m* bytes:**
  - $B_0, B_1, \ldots, B_k, \ldots, B_{m-1}$
- **All I/O devices are represented as files:**
  - **/dev/sda2** (hard disk partition)
  - **/dev/tty2** (terminal)
- **Even the kernel is represented as a file:**
  - **/dev/kmem** (kernel memory image)
  - **/proc** (kernel data structures)

Unix File Types

- **Regular file**
  - Binary or text file
  - Unix does not know the difference!
- **Directory file**
  - A file that contains the names and locations of other files.
- **Character special and block special files**
  - Terminals (character special) and disks (block special)
- **FIFO (named pipe)**
  - A file type used for interprocess communication
- **Socket**
  - A file type used for network communication between processes

Unix I/O

- **Characteristics**
  - The elegant mapping of files to devices allows kernel to export simple interface called Unix I/O.
  - All input and output is handled in a consistent and uniform way ("byte stream")
- **Basic Unix I/O operations (system calls):**
  - Opening and closing files
    - **open()** and **close()**
  - Changing the current file position (seek)
    - **lseek()**
  - Reading and writing a file
    - **read()** and **write()**

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

Opening Files

```
int open(char *filename, int flags, mode_t mode);
```

- **Return**
  - small identifying integer file descriptor if OK, -1 on error
- **Parameters**
  - <char *filename> indicates which file to open
  - <int flags> indicates how the process intends to access the file
    - O_RDONLY, O_WRONLY, O_RDWR
    - O_CREAT, O_TRUNC, O_APPEND
  - <mode_t mode> specifies the access permission bits of new files – optional field
    - S_IRUSR, S_IWUSR, S_IXUSR
    - S_IRGRP, S_IWGRP, S_IXGRP
    - S_IROTH, S_IWOTH, S_IXOTH

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);
}
```

- Closing an already closed file is a recipe for disaster in threaded programs
- Moral: Always check return codes, even for seemingly benign functions such as close()

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`**
  - **nbytes < 0** indicates that an error occurred.
  - short counts (**nbytes < sizeof(buf)**) are possible and are not errors!

Reading Files

```
ssize_t read(int fd, void *buf, size_t count);
```

- **Return**
  - Number of bytes read if OK, 0 on EOF, -1 on error
  - Short counts may occur!

- **Parameters**
  - <int fd> file descriptor fd to read
  - <void *buf> reads into the buffer starting at buf
  - <size_t count> reads up to count bytes

Writing Files

- **Writing a file copies bytes from memory to the current file position, and then updates current file position.**

```
char buf[512];
int fd;       /* file descriptor */
int nbytes;   /* number of bytes read */
/* Open the file fd ... */
/* Then write up to 512 bytes from buf to file fd */
if ((nbytes = write(fd, buf, sizeof(buf)) < 0) {
   perror("write");
   exit(1);
}
```

- **Returns number of bytes written from buf to file fd.**
  - **nbytes < 0** indicates that an error occurred.
  - As with reads, short counts are possible and are not errors!

Writing Files

```
ssize_t write(int fd, void *buf, size_t count);
```

- **Return**
  - Number of bytes read if OK, 0 on EOF, -1 on error
  - Short counts may occur!

- **Parameters**
  - <int fd> file descriptor fd to write
  - <void *buf> source to write start from buf
  - <size_t count> writes up to count bytes

File Offset

- **An open file's offset can be set explicitly by calling *lseek***

```
Off_t lseek(int fd, off_t offset, int whence)
```

- **Returns**
  - New file offset if OK, -1 on error
- **whence**
  - SEEK_SET: the file's offset is set to set to offset
  - SEEK_CUR: current value plus offset
  - SEEK_END: set to size of file plus offset

Hole File

root@proshb-pc: ~/Test

```
 1 #include <stdlib.h>
 2 #include <fcntl.h>
 3
 4 #define FILE_MOD (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
 5
 6 char buf1[] = "abcdefghij";
 7 char buf2[] = "ABCDEFGHIJ";
 8
 9 int main(void)
10 {
11     int fd;
12
13     if((fd = creat("file.hole", FILE_MOD)) < 0 )
14     {
15         perror("creat error");
16     }
17
18     if(write(fd, buf1, 10) != 10)
19     {
20         perror("buf1 write error");
21     }
22
23     if(lseek(fd, 16384, SEEK_SET) == -1)
24     {
25         perror("lseek error");
26     }
27
28     if(write(fd, buf2, 10) != 10)
29     {
30         perror("buf2 write error");
31     }
32     close(fd);
33 }
-- INSERT --                                   31,3-6        Top
```

Hole File

- **Contents of the hole file**

```
root@proshb-pc: ~/Test
root@proshb-pc:~/Test# od -c file.hole
0000000   a   b   c   d   e   f   g   h   i   j  \0  \0  \0  \0  \0  \0
0000020  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
0040000   A   B   C   D   E   F   G   H   I   J
0040012
root@proshb-pc:~/Test#
```

- **Difference between two files**

```
root@proshb-pc: ~/Test
root@proshb-pc:~/Test# ls -ls file.hole file.nohole
 8 -rw-r--r-- 1 root root 16394 Mar 12 17:43 file.hole
20 -rw-r--r-- 1 root root 16394 Mar 12 17:43 file.nohole
root@proshb-pc:~/Test#
```

  - Both files are the same size, but consumption of disk blocks is different.

Dealing with Short Counts

- **When short counts can occur?**
  - Encountering (end-of-file) EOF on reads.
  - Reading from disk files (except for EOF)
  - Reading text lines from a terminal.
  - Reading and writing network sockets or Unix pipes.
  - Writing to disk files.

Dealing with 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.
- **Short counts does not occur in these situations:**
  - Reading from disk files (except for EOF)
  - Writing to disk files.
- **How should you deal with short counts in your code?**
  - One way is to use the RIO (Robust I/O) package from your textbook's **csapp.c** file (Appendix B).

File Metadata

- **Data about data, in this case file data.**
  - Maintained by kernel, accessed by users with the **stat** and **fstat** functions.

```
/* Metadata returned by the stat and fstat functions */
struct stat {
    dev_t         st_dev;      /* device */
    ino_t         st_ino;      /* inode */
    mode_t        st_mode;     /* protection and file type */
    nlink_t       st_nlink;    /* number of hard links */
    uid_t         st_uid;      /* user ID of owner */
    gid_t         st_gid;      /* group ID of owner */
    dev_t         st_rdev;     /* device type (if inode device) */
    off_t         st_size;     /* total size, in bytes */
    unsigned long st_blksize;  /* blocksize for filesystem I/O */
    unsigned long st_blocks;   /* number of blocks allocated */
    time_t        st_atime;    /* time of last file access */
    time_t        st_mtime;    /* time of last file modification */
    time_t        st_ctime;    /* time of last inode change */
};                            /* statbuf.h included by sys/stat.h */
```

Accessing File Metadata

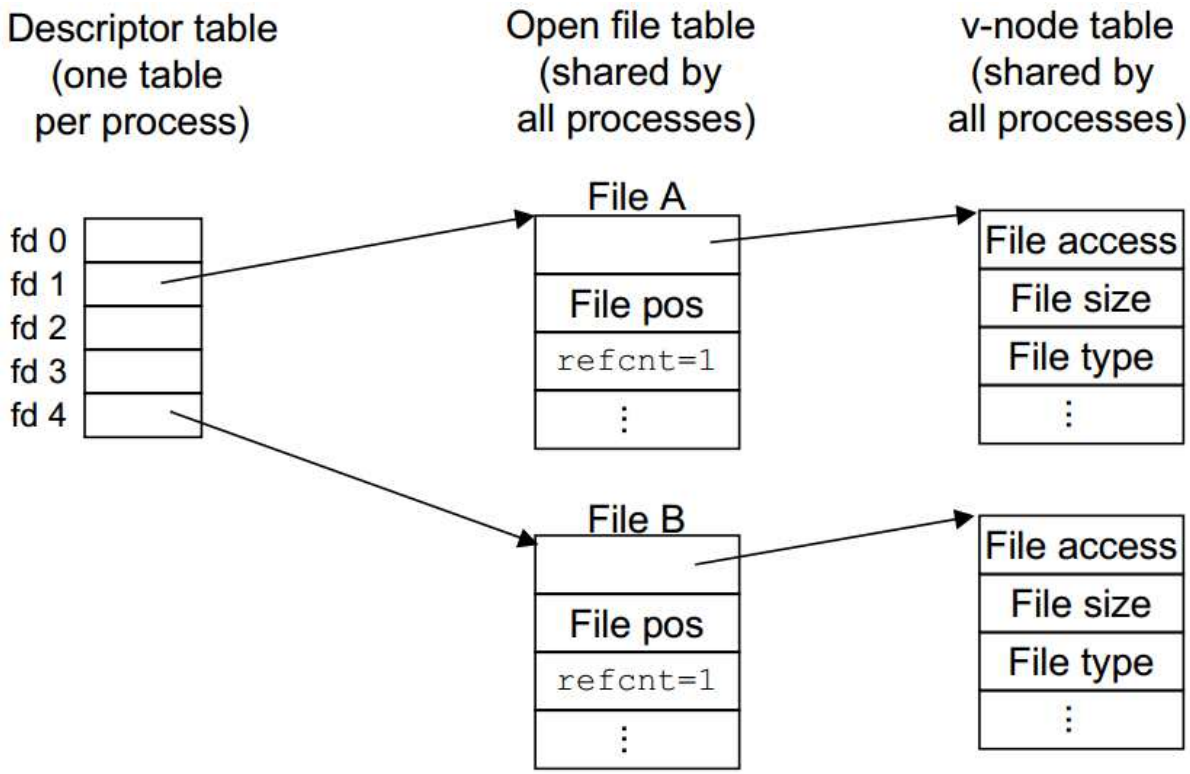
```
/* statcheck.c - Querying and manipulating a file's meta data */

int main (int argc, char **argv)
{
    struct stat st;
    char *type, *readok;

    stat(argv[1], &st);
    if (S_ISREG(st.st_mode)) /* file type*/
        type = "regular";
    else if (S_ISDIR(st.st_mode))
        type = "directory";
    else
        type = "other";
    if ((st.st_mode & S_IRUSR)) /* OK to read?*/
        readok = "yes";
    else
        readok = "no";

    printf("type: %s, read: %s\n", type, readok);
    exit(0);
}
```

```
bass> ./statcheck statcheck.c
type: regular, read: yes
bass> chmod 000 statcheck.c
bass> ./statcheck statcheck.c
type: regular, read: no
```

```
int stat(const char *filename, struct stat *buf)
int fstat(int fd, struct stat *buf)
```

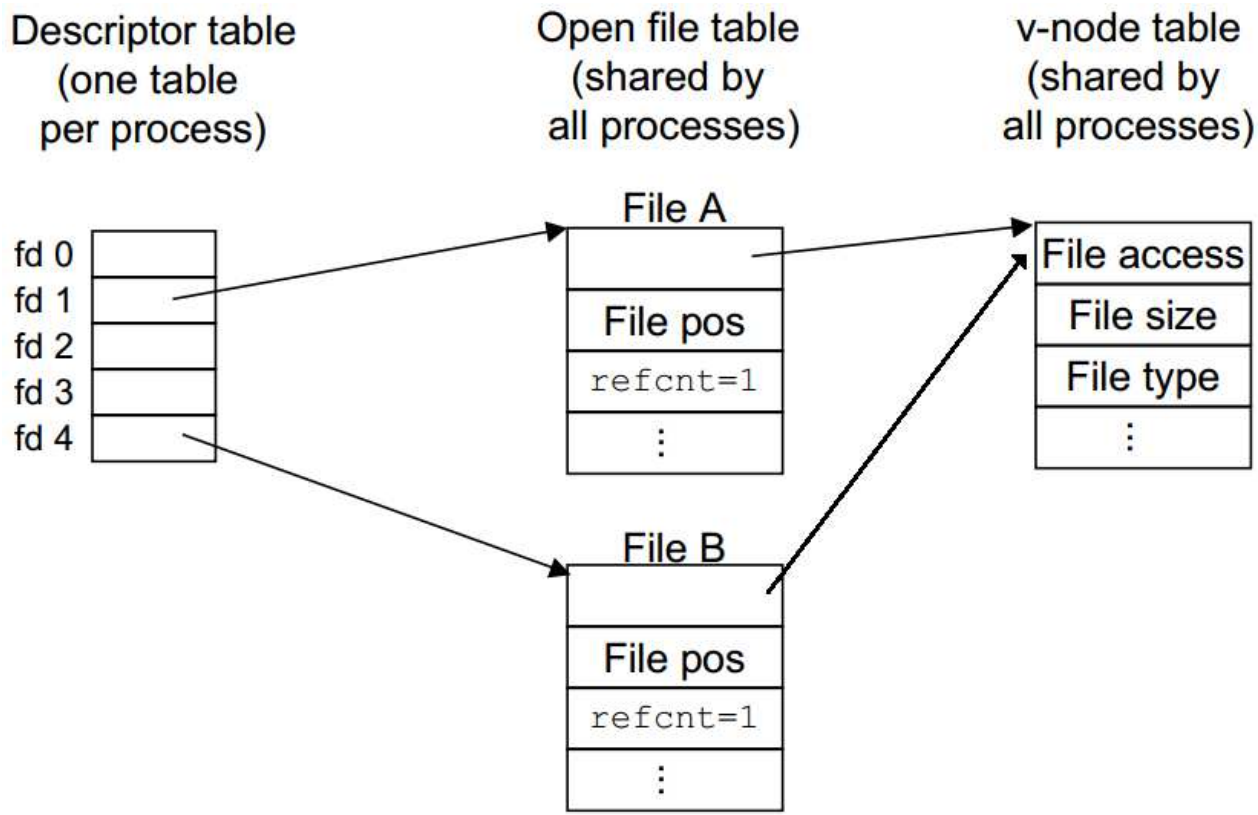
Sharing Files

- **The kernel represents open files using three related data structures:**
  - Descriptor table, file table, v-node table



Sharing Files

- **Two descriptors sharing the same disk file through two open file table entries.**



I/O Redirection

- **Hoinky**

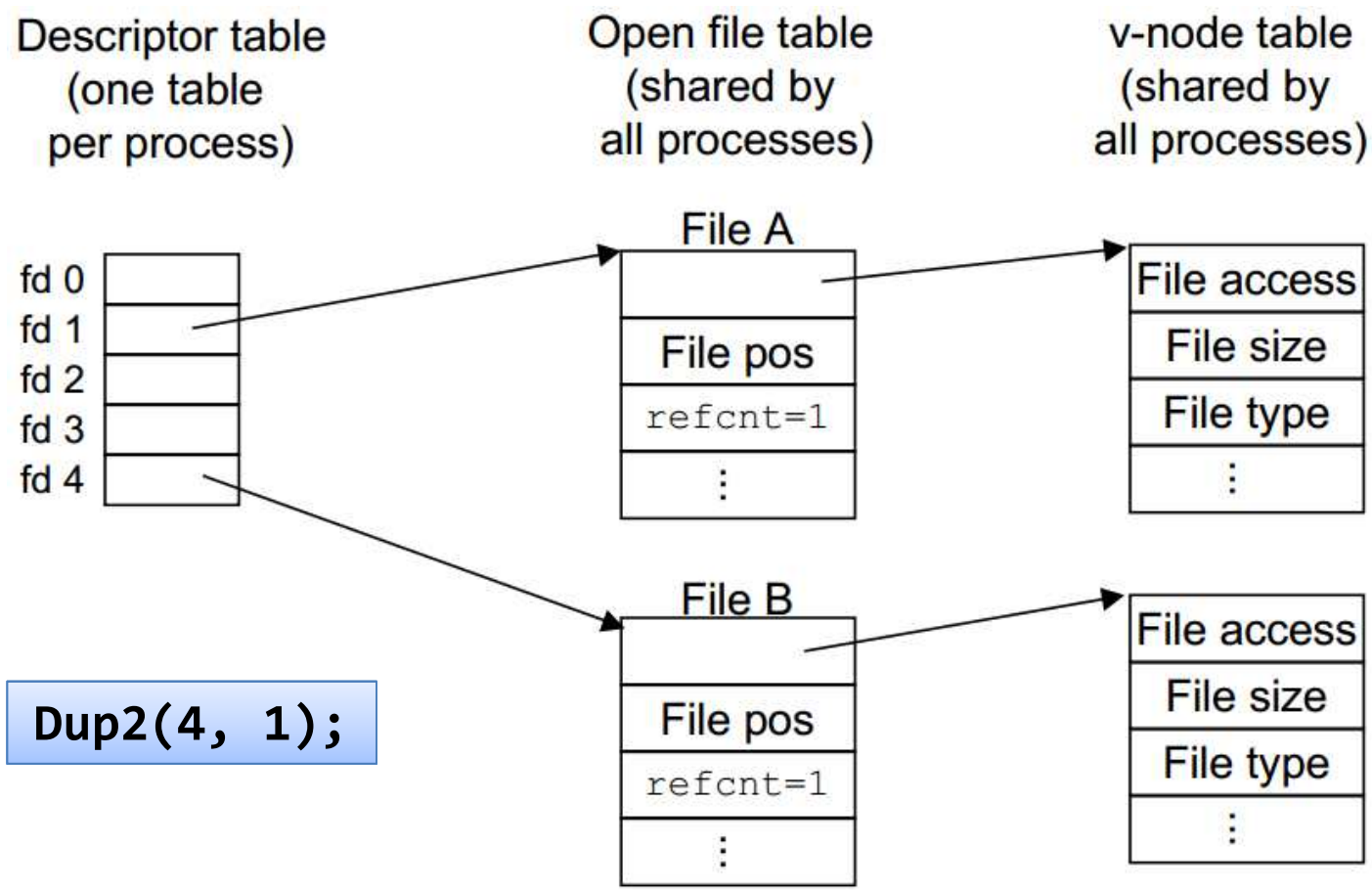
```
$ ls > foo.txt
```

- **dup & dup2 duplicates existing file descriptors**

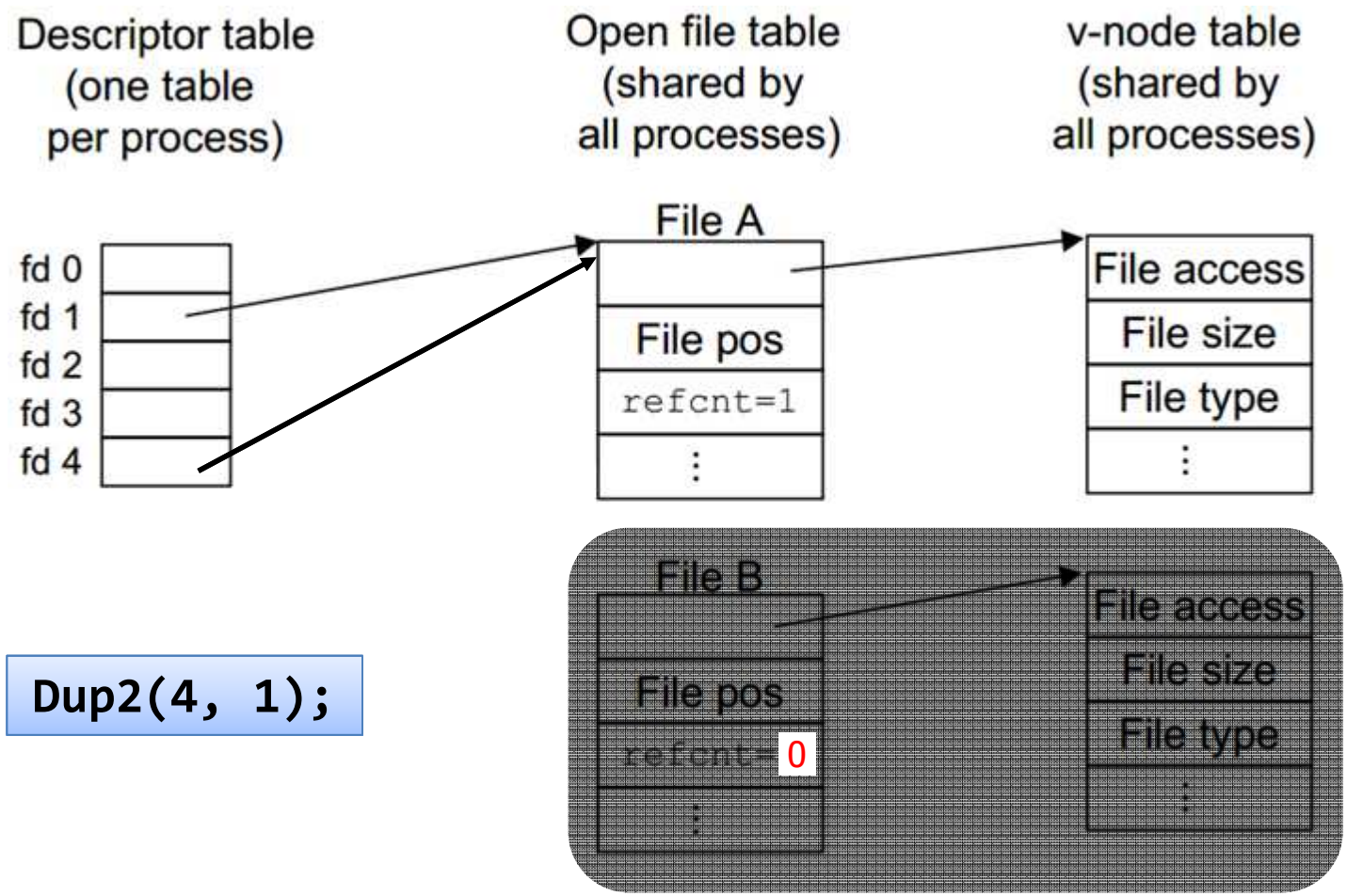
```
int dup(int filedes);
int dup2(int filedes, int filedes2);
```

  - 'dup' is guaranteed to return the lowest-numbered available file descriptor.
  - 'dup2 specifies the value of the new descriptor with the filedes2

Sharing Files



Sharing Files




```
Dup2(4, 1);
```

Pros/Cons of Unix I/O

- **Pros**
  - The most general and lowest overhead form of I/O.
    - All other I/O packages are implemented on top of Unix I/O functions.
  - Unix I/O provides functions for accessing file metadata.
- **Cons**
  - System call overheads for small-sized I/O.
  - Dealing with short counts is tricky and error prone.
  - Efficient reading of text lines requires some form of buffering, also tricky and error prone.
  - These issues are addressed by the standard I/O.

Summary

- **Unix file I/O**
  - `open()`, `read()`, `write()`, `close()`, …
  - A uniform way to access files, I/O devices, network sockets, kernel data structures, etc.

- **When to use raw Unix I/O**
  - When you need to fetch file metadata.
  - When you read or write network sockets or pipes.
  - In rare cases when you need absolute highest performance.

Exercises

- **Copying standard input to standard output one byte at a time.**

- **Find 10 words appeared most in the given novel**
  - Novel: http://www.gutenberg.org/files/829/829-0.txt
  - Words are separated by '₩n' and '₩t', otherwise ignore special characters and attach two words.
  - Hello2me → hellome
  - http://www.skku.edu → httpwwwskkuedu
  - Output example: [RANK 0][the][4096] …