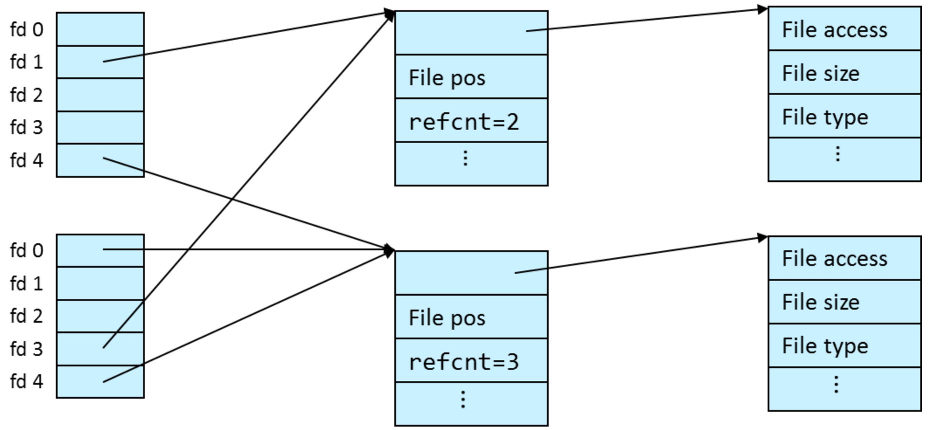
Input/Output

Files and Directories



Module Outline

- File Metadata
- Directories
- Kernel File Management
- Module Summary

```
struct stat {
  dev_t     st_dev;
  ino_t     st_ino;
  mode_t    st_mode;
  nlink_t   st_nlink;
  uid_t     st_uid;
  gid_t     st_gid;
  dev_t     st_rdev;
  off_t     st_size;
  blksize_t st_blksize;
  blkcnt_t  st_blocks;
  struct timespec st_atim;
  struct timespec st_mtim;
  struct timespec st_ctim;
};
```

File Metadata

File Metadata

- Metadata is data about data, in this case information about a file
  - filename
  - file type
  - file size
  - file creation, modification, and access time
  - file access permissions
  - …

- Per-file metadata maintained by kernel
  - not all metadata is stored in the same place
    - filename: stored in the directory that contains the files
    - everything else is stored in the inode of the file
      - inode: internal storage unit used by Unix file systems

File Metadata

- File metadata can be accessed with the `*stat*` family of Unix I/O API calls
  - return the file metadata in a `struct stat`

```
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 id (if special file)
  off_t             st_size;        // total size, in bytes
  unsigned long     st_blksize;     // preferred blocksize for filesystem I/O
  unsigned long     st_blocks;      // number of 512b blocks allocated
  // high-precision timestamps (precision: nanoseconds; since Linux kernel >=2.6)
  struct timespec   st_atim;        // time of last access
  struct timespec   st_mtim;        // time of last access
  struct timespec   st_ctim;        // time of last status change
  // low-precision timestamps  (precision: seconds; always available)
  time_t            st_atime;       // time of last access
  time_t            st_mtime;       // time of last modification
  time_t            st_ctime;       // time of last status change
};
```

File Metadata

- `struct stat`
  - manual: stat(2), inode(7)
  - `st_mode` enodes the type and access permissions of a file
    - use `S_IS***(stat.st_mode)` macros for convenience
      `if (S_ISREG(stat.st_mode)) // regular file`
    - use S_IR/W/X*** masks to check the access permissions
      `if (stat.st_mode & S_IRUSR) // owner has read permission`
  - `st_uid` / `st_gid`
    - use `getpwuid(stat.st_uid)` / `getgrgid(sb.st_gid)` to retrieve user/group information
  - `st_size` vs `st_blocks`
    - file is sparse if `st_size / 512 > st_blocks`

File Metadata

- `struct stat`
  - timestamps
    - st_?tim: nanosecond precision since Linux 2.6
    - st_?time: Unix / Linux <2.6 low-resolution timestamps (precision: seconds)
    - traditionally, Unix filesystems did not record a file's creation date (birth date)
      - newer file systems may support it, retrieve via `statx()` system call
  - `st_ctim[e]` vs `st_mtim[e]`
    - `ctim[e]`: timestamp of *last update to file inode* (file meta data)
    - `mtim[e]`: timestamp of *last update to file data*
    - in general, `mtim[e] ≤ ctim[e]`
  - `st_atim[e]`
    - time stamp of *last access to file*
    - often disabled for performance reasons (mount filesystem with -o noatime)

File Metadata

- C standard library functions to access file metadata (header sys/stat.h)

<table>
<tr><th>Operation</th><th>API</th><th>Remarks</th></tr>
<tr><td rowspan="5">Retrieve
file metadata</td><td><code>int stat(const char *pathname,
         struct stat *statbuf)</code></td><td></td></tr>
<tr><td><code>int lstat(const char *pathname,
          struct stat *statbuf)</code></td><td>does not follow symbolic links</td></tr>
<tr><td><code>int fstat(int fd,
          struct stat *statbuf)</code></td><td></td></tr>
<tr><td><code>int fstatat(int dirfd,
            const char *pathname,
            struct stat *statbuf,
            int flags)</code></td><td>useful when stat-ing files in a directory</td></tr>
<tr><td><code>int statx(int dirfd,
          const char *pathname,
          int flags,
          unsigned int mask,
          struct statx *statxbuf)</code></td><td>get extended file status</td></tr>
</table>

```
|-a
| `-b
|    |-c
|    | `-d
|    `-f
|-dir1
| `-seventyfive
|-dir2
| |-three
| `-seven
`-dir3
  |-five
  `-twentytwo
```

Directories

Directories

- C standard library functions to deal with directories (headers dirent.h, sys/types.h)

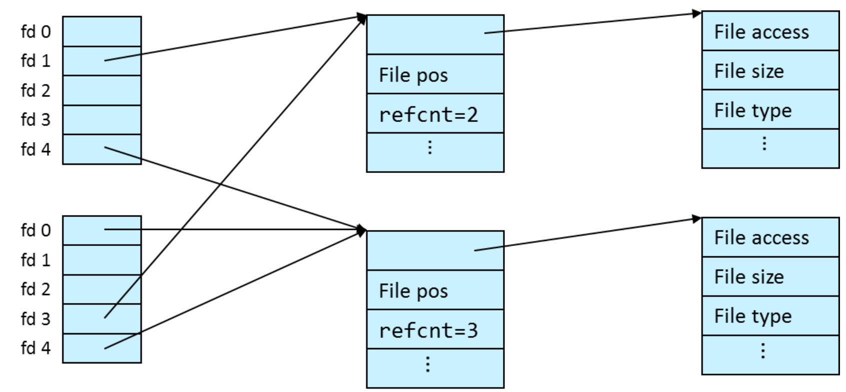
Operation	API	Variants
Open	`DIR* opendir(const char *name)`	`fdopendir`
Read entry	`struct dirent* readdir(DIR *dirp)`	
Close	`int closedir(DIR *dirp)`	
Retrieve descriptor	`int dirfd(DIR *dirp)`	
Make directory	`int mkdir(const char *pathname, mode_t mode)`	`mkdirat`

Example: Accessing Directories and File Metadata

```
#include <…>

int main(int argc, char *argv[])
{
  DIR *dir = opendir(argc > 1 ? argv[1] : ".");
  if (dir == NULL) { perror("Cannot open directory"); return EXIT_FAILURE; }

  int dd = dirfd(dir);

  struct dirent *e;
  errno = 0;
  while ((e = readdir(dir)) != NULL) {
    struct stat sb;
    if (fstatat(dd, e->d_name, &sb, 0) < 0) {
      perror("Cannot stat file");
    } else {
      printf("  %-32s %10ld\n", e->d_name, sb.st_size);
    }
    errno = 0;
  }
  if (errno != 0) perror("Cannot enumerate directory");

  closedir(dir);

  return EXIT_SUCCESS;
}
```

statter.c

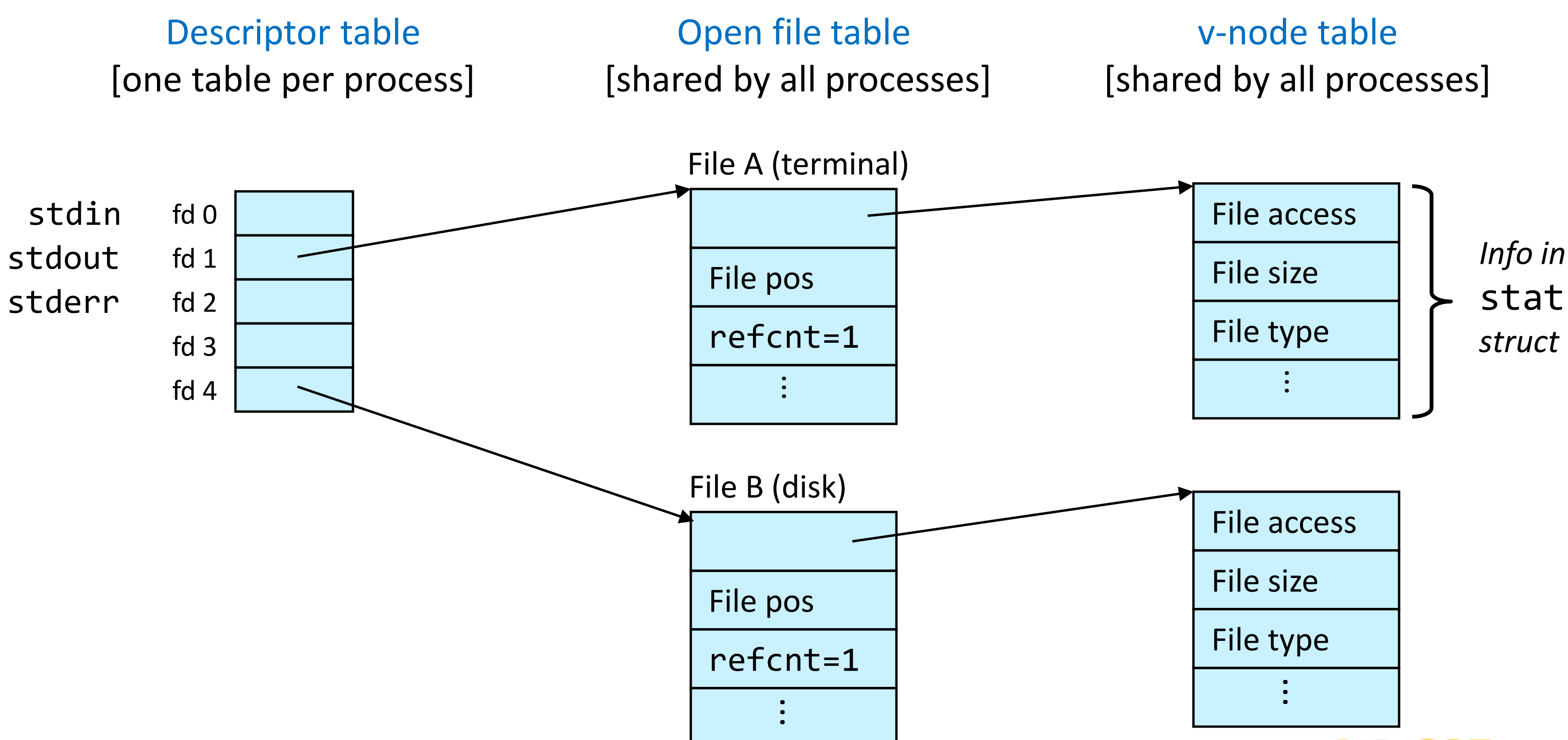
```
$ ./statter /
  opt                       4096
  sys                          0
  proc                         0
  devnull                     64
  usr                       4096
  lib64                    12288
  home                      4096
  .                         4096
  data                      4096
  var                       4096
  lost+found               16384
  ..                        4096
  dev                       4300
  ...
```



Kernel File Management

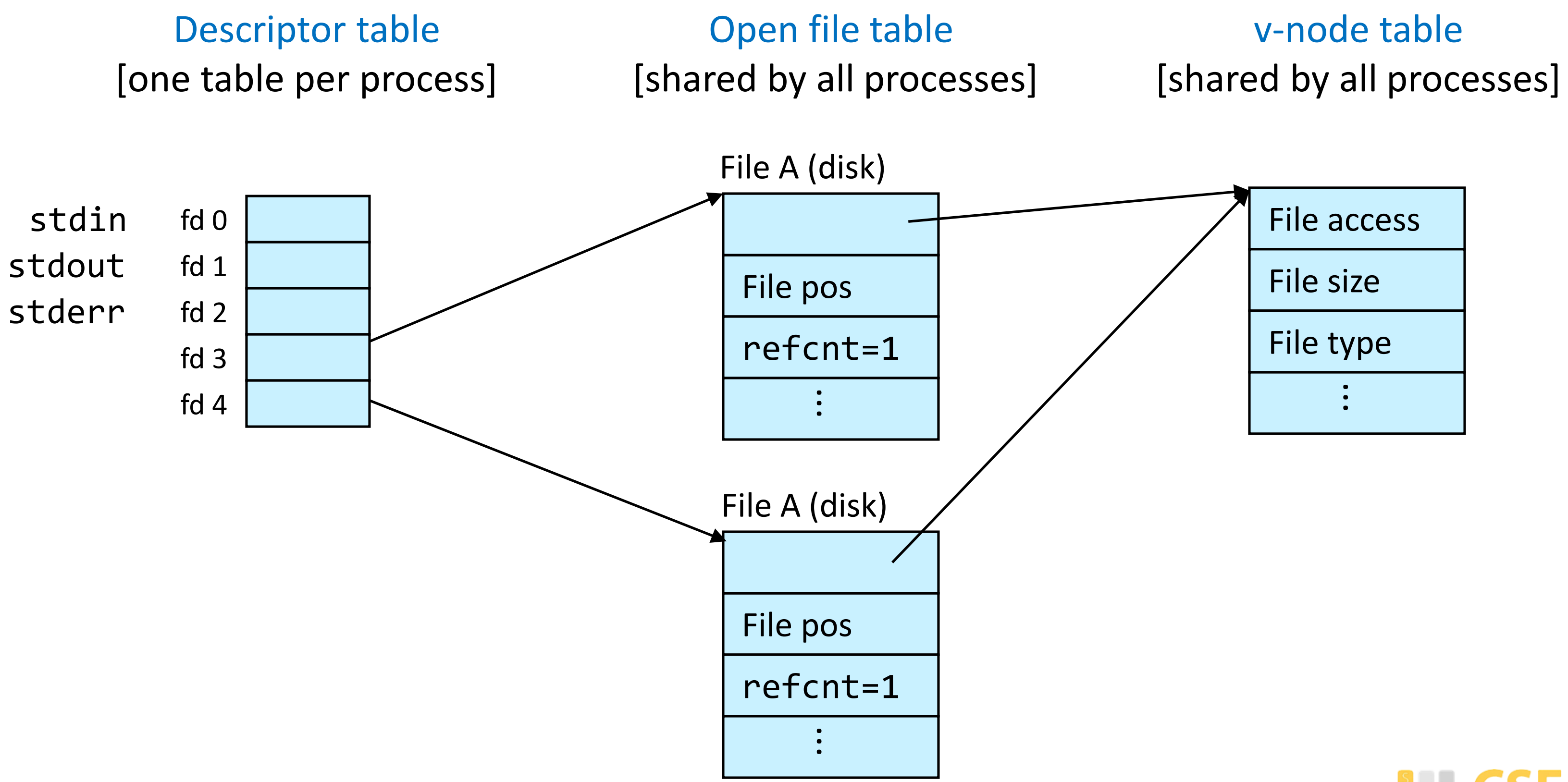
How the Unix Kernel Represents Open Files

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



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



I/O Redirection

- I/O redirection is one of the core concepts of Unix

```
$ ls > output.txt

$ ls | sort -R

$ cat < input.txt
```

- How does a shell implement I/O redirection?

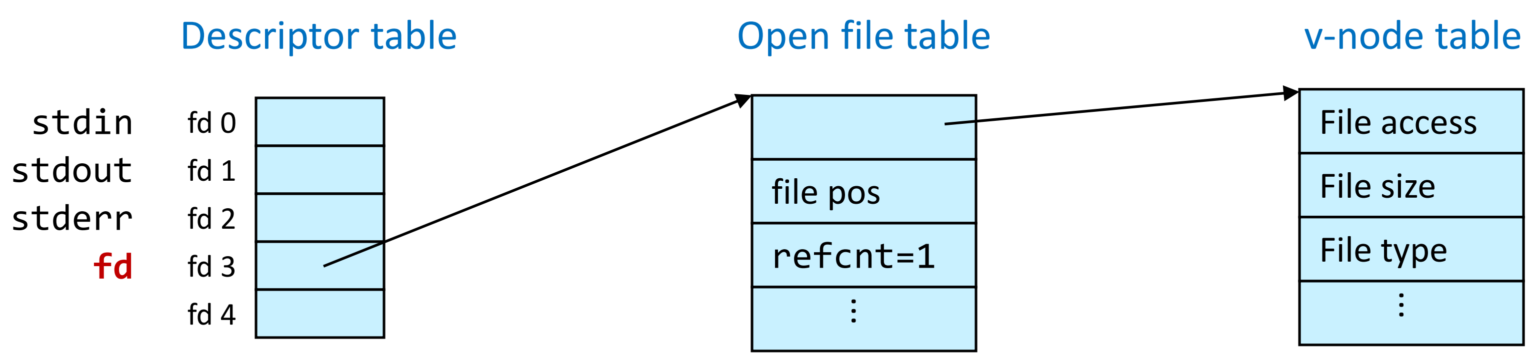
File Descriptor Manipulation

- C standard library functions allow manipulation of file descriptors (header unistd.h)

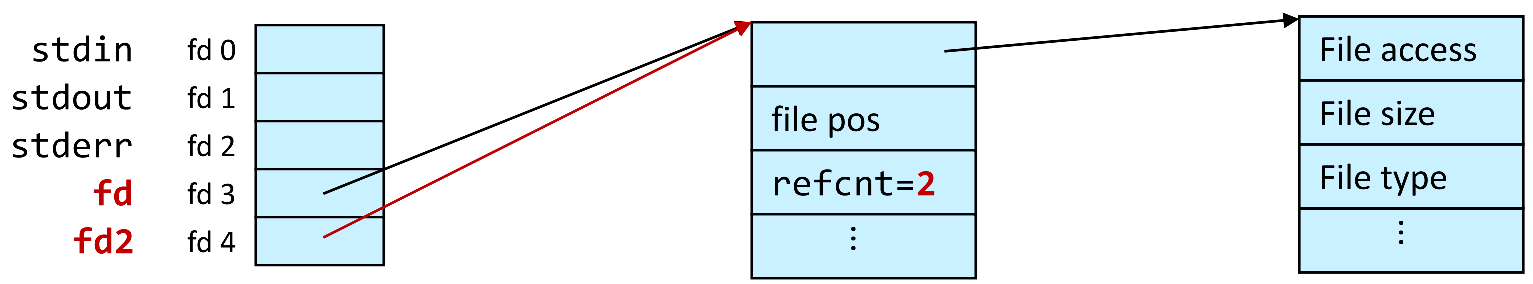
Operation	API	Remarks
`Duplicate file descriptor`	`int dup(int oldfd)`	Returned fd points to same entry in open file table as oldfd
`Duplicate file descriptor to specific fd`	`int dup2(int oldfd, int newfd)`	Entry in newfd is overwritten with value in oldfd
`Retrieve file descriptor of file stream`	`int fileno(FILE *stream)`	

File Descriptor Manipulation: dup( )

- `fd = open("file.txt", O_RDONLY, 0);`

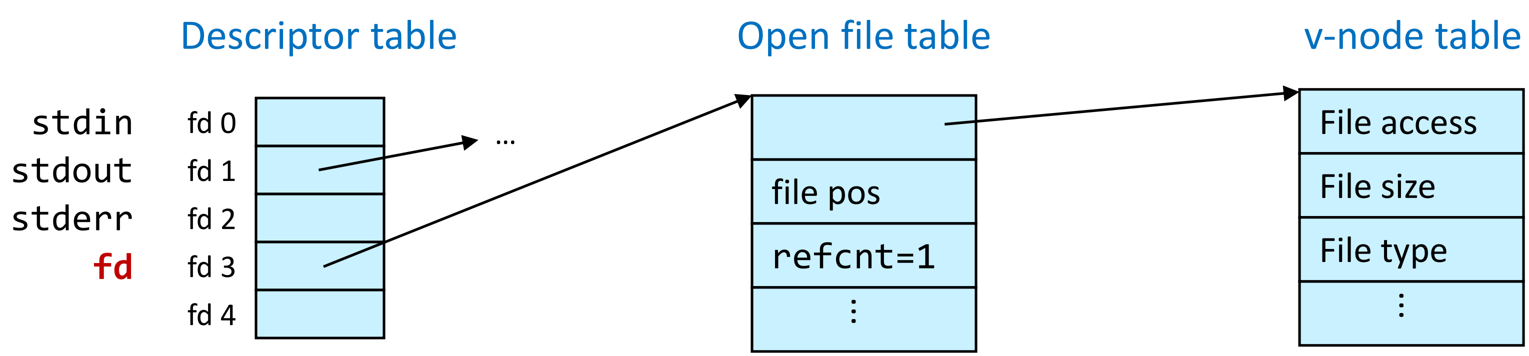


- `fd2 = dup(fd);`

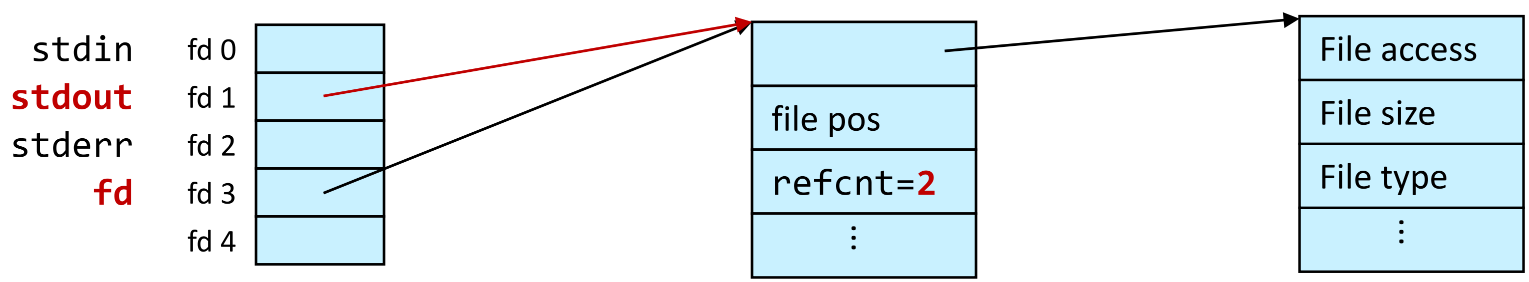


File Descriptor Manipulation: dup2( )

- `fd = open("output.txt", O_WRONLY, 0);`



- **dup2**(fd, STDIO_FILENO);



Fun with File Descriptors

- What is the output of this program if the input file contains “System Programming”?

```
#include <…>

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);
  if ((fd1 == -1) || (fd2 == -1) || (fd3 == -1)) {
    fprintf(stderr, "Cannot open input file.\n"); return EXIT_FAILURE;
  }

  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;
}
```

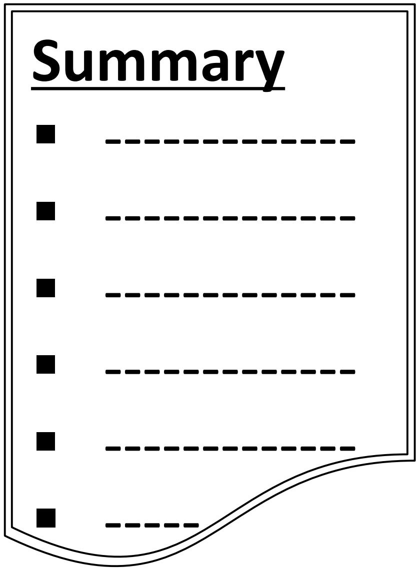
fwfd1.c

Fun with File Descriptors

- What are the contents of the generated output file?

```
#include <…>

int main(int argc, char *argv[])
{
  int fd1, fd2, fd3;
  char *fname = argv[1];

  if ((fd1 = open(fname, O_CREAT|O_TRUNC|O_RDWR, S_IRUSR|S_IWUSR)) == -1) {
    perror("Cannot open output file"); return EXIT_FAILURE;
  }

  write(fd1, "CSAP", 4);

  fd3 = open(fname, O_APPEND|O_WRONLY, 0);
  write(fd3, "M1522", 5);

  fd2 = dup(fd1);                    // Allocates descriptor
  write(fd2, "SNU", 3);
  write(fd3, “800", 3);

  return 0;
}
```

fwfd2.c



Module Summary

Most Frequently Used Unix I/O System Calls

$ man –S 2 <func>

```
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

int      open(const char *pathname, int flags[, mode_t mode]);
int      creat(const char *pathname, mode_t mode);

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

off_t   lseek(int fd, off_t offset, int whence);
int      stat(const char *path, struct stat *buf);

int      close(int fd);
```

Most Frequently Used Standard I/O System Calls

$ man –S 3 <func>

```
#include <stdio.h>

FILE*   fopen(const char *pathname, const char *mode);

size_t  fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t  fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
int     fflush(FILE *stream);

int     feof(FILE *stream);
int     ferror(FILE *stream);

off_t   fseek(FILE *stream, long offset, int whence);
int     f[get/set]pos(FILE *stream, fpos_t *pos);

int     fclose(FILE *fp);
```

Most Frequently Used API to Manage Directories

$ man –S 3 <func>

```
#include <sys/types.h>
#include <dirent.h>

DIR*    opendir(const char *name);

struct dirent *readdir(DIR *dirp);

int     dirfd(DIR *dirp);

int     closedir(DIR *dirp);
```

Pros and Cons of Unix I/O

- **Pros**
  - 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.

- **Cons**
  - Dealing with short counts is tricky and error prone.
  - Efficient reading of text lines requires some form of buffering, also tricky and error prone.
  - Both of these issues are addressed by the standard I/O packages.

Pros and Cons of Standard I/O

- **Pros**
  - Buffering increases efficiency by decreasing the number of read and write system calls
  - Short counts are handled automatically

- **Cons**
  - Provides no function for accessing file metadata
  - Standard I/O functions are not async-signal-safe, and not appropriate for signal handlers.
  - Standard I/O is not appropriate for input and output on network sockets
    - There are poorly documented restrictions on streams that interact badly with restrictions on sockets (CS:APP3e, Sec 10.9)

Choosing I/O Functions

- General rule: use the highest-level I/O functions you can
  - Many C programmers are able to do all of their work using the standard I/O functions

- When to use standard I/O
  - When working with disk or terminal files

- When to use raw Unix I/O
  - Inside signal handlers, because Unix I/O is async-signal-safe.
  - In rare cases when you need absolute highest performance.

Aside: Working with Binary Files

- Binary File Examples
  - Object code, Images (JPEG, GIF)

- Functions you shouldn't use on binary files
  - Line-oriented I/O such as fgets, scanf, printf
    - Different systems interpret 0x0A ('\n') (newline) differently:
      - Linux and Mac OS X: LF(0x0a) ['\n']
      - HTTP servers & Windoes: CR+LF(0x0d 0x0a) ['\r\n']

  - String functions
    - strlen, strcpy
    - Interprets byte value 0 (end of string) as special

For Further Information

- The Unix bible:
  - W. Richard Stevens & Stephen A. Rago, Advanced Programming in the Unix Environment, 3rd Edition, Addison-Wesley, 2013, ISBN 978-0321637734
    - Updated from Stevens's 1993 classic text.