CSE320 System Fundamentals II System APIs

YOUNGMIN KWON / TONY MIONE

Unix I/O

Input/Output

- Process of copying data between main memory and external devices like disk drives, terminals, networks.

Unix I/O

- Unix I/O functions (read, write, …) are provided by the kernel
- Standard I/O library functions (printf, scanf, …) are implemented using Unix I/O functions

Why Unix I/O (when there is Standard I/O)

Understanding Unix I/O will help understand other system concepts

- e.g. Process creation and Opening a file

Sometimes, there are no other choices but to use Unix I/O

Unix I/O: Files

A Linux file is a sequence of bytes

- B_0, B_1, …, B_k, …, B_m

All I/O devices are *modeled as files*

- E.g. networks, disks, terminals
- Input and output are performed by reading from and writing to the appropriate files
- This mapping enables simple low level APIs known as Unix I/O

Unix I/O: Files

Opening files

- Announce an app's intention to access an I/O device.
- Kernel returns a descriptor
  - Small integer
  - Index into an 'open file' table

Changing the current file position

- A byte offset from the beginning of a file
- The kernel maintains a file position for each open file [where it makes sense]
- Seek operation can change the file position

Unix I/O: Files

Reading and writing files

- *read* copies some number of bytes from the current position of a file into memory
- *write* copies a number of bytes from memory to the current position of a file

Closing files

- Informs the kernel that the app has finished accessing the file
  - This allows the Kernel to free data used to administ

Some File Types

A regular file
- Contains arbitrary data
- Text file, Binary file

A directory
- A file containing an array of links that map file names to files
- `'.'` is a link to the directory itself, `'..'` is a link to the parent directory

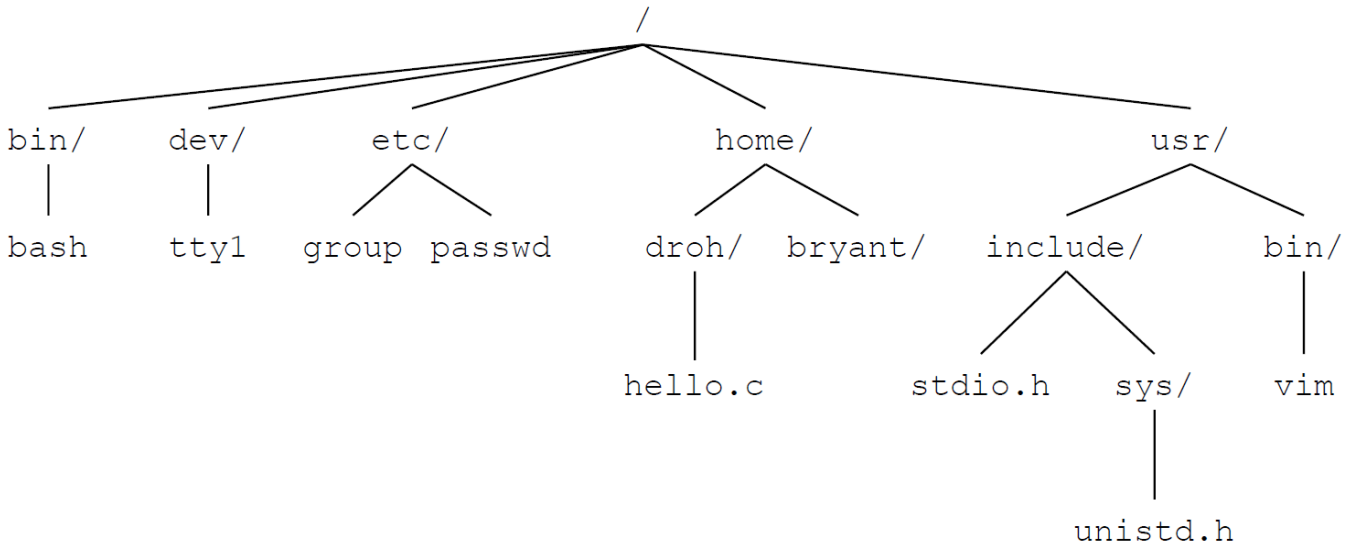
A socket
- A file used to communicate with another process

Other files
- Named pipes, symbolic links, character and block devices

Files

Linux kernel organizes all files in a single directory hierarchy anchored by the *root directory* "/"

Each process has a *current working directory*



Opening and Closing Files

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

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

// flags
O_RDONLY // Reading only
O_WRONLY // Writing only
O_RDWR   // Reading and Writing

O_CREAT  // If file doesn't exist, create a truncated one
O_TRUNC  // If file already exists, truncate it
O_APPEND // Set the file position to the end of the file

// modes
S_IRUSR, S_IRGRP, S_IROTH  // User, group, other can read this file
S_IWUSR, S_IWGRP, S_IWOTH  // User, group, other can write this file
S_IXUSR, S_IXGRP, S_IXOTH  // User, group, other can execute this file
```

Opening and Closing Files

```c
#include <unistd.h>
int close(int fd);
```

```c
#define DEF_MODE  S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH
#define DEF_UMASK S_IWGRP|S_IWOTH

umask(DEF_UMASK);        //umask bits will be unset
int  fd = open("foo.txt", O_CREAT|O_TRUNC|O_WRONLY, DEF_MODE);
close(fd);
```

Reading and Writing Files

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t n);
ssize_t write(int fd, const void *buf, size_t n);
```

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
int main() {
    int fd1 = open("foo.txt", O_CREAT|O_WRONLY, S_IRUSR|S_IWUSR);
    int fd2 = open("foo.txt", O_RDONLY, 0);
    close(fd1);
    close(fd2);
    printf("fd1: %d, fd2: %d\n", fd1, fd2);
}
```

Reading and Writing Files

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
int main() {
    char buf[100];
    int fd1, fd2, n, i;
    // read from the standard input
    n = read(STDIN_FILENO, buf, sizeof(buf));

    // write to a file
    fd1 = open("foo.txt", O_CREAT|O_WRONLY, S_IRUSR|S_IWUSR);
    for(i = 0; i < n; )
        i += write(fd1, buf + i, n - i);
    close(fd1);

    // read from a file
    fd2 = open("foo.txt", O_RDONLY, 0);
    for(i = 0; i < n; )
        i += read(fd2, buf + i, n - i);
    close(fd2);

    // write to the standard output
    for(i = 0; i < n; )
        i += write(STDOUT_FILENO, buf + i, n - i);
}
```

Reading File Metadata

```
#include <unistd.h>
#include <sys/stat.h>
int stat(const char *filename, struct stat *buf);
int fstat(int fd, struct stat *buf);
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* protection */
    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 */
    blksize_t st_blksize; /* blocksize for file system I/O */
    blkcnt_t  st_blocks;  /* number of 512B blocks allocated */
    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 */
};
```

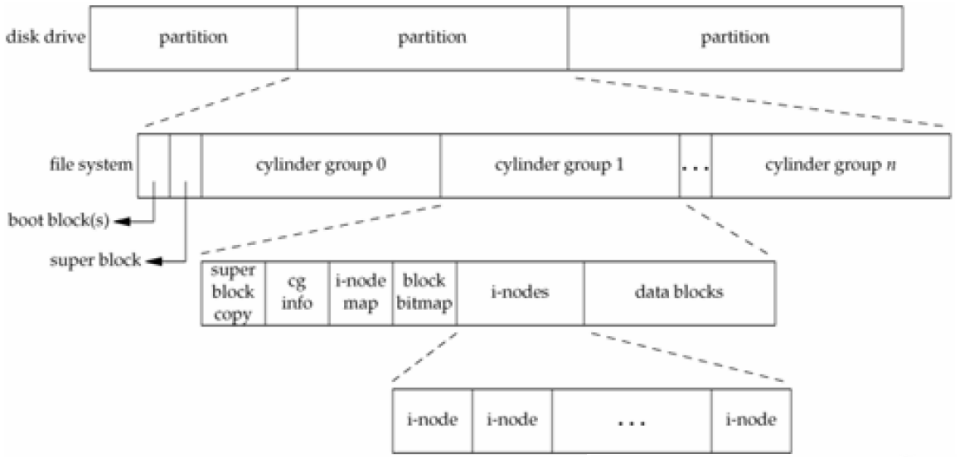
Reading File Metadata

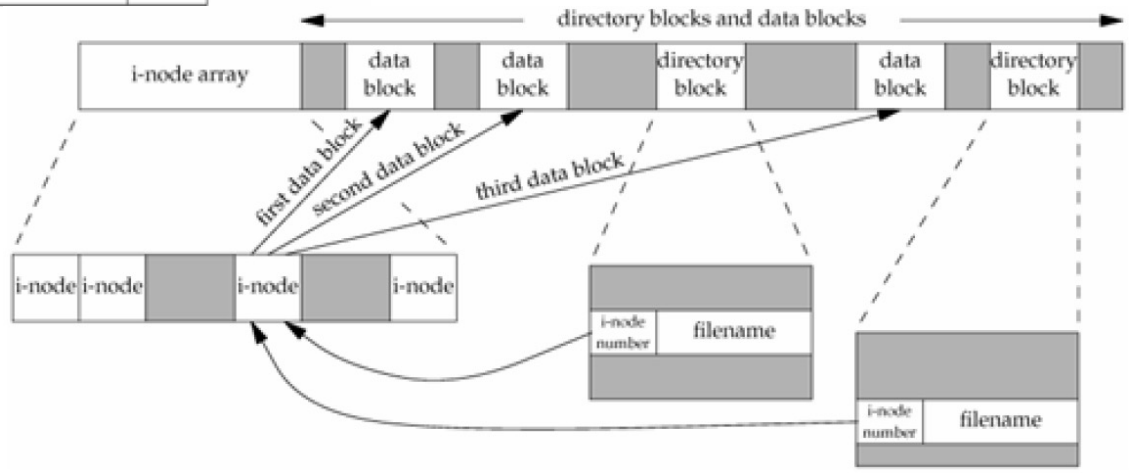
```c
#include <unistd.h>
#include <sys/stat.h>
#include <time.h>
#include <stdio.h>

void printstat(char *fname);

int main()
{
    printstat(".");
    printstat("./foo.c");
    printstat("/dev/tty0");
}
```

Reading File Metadata

```
void printstat(char *fname)
{
    struct stat sb;
    stat(fname, &sb);

    printf("\n---------------------------------------\n");
    printf("File name:          %s\n", fname);
    printf("File type:          ");
    switch (sb.st_mode & S_IFMT) {
    case S_IFBLK:  printf("block device\n");     break;
    case S_IFCHR:  printf("character device\n"); break;
    case S_IFDIR:  printf("directory\n");        break;
    case S_IFIFO:  printf("FIFO/pipe\n");        break;
    case S_IFLNK:  printf("symlink\n");          break;
    case S_IFREG:  printf("regular file\n");     break;
    case S_IFSOCK: printf("socket\n");           break;
    default:       printf("unknown?\n");         break;
    }
    printf("I-node number:      %ld\n", (long)sb.st_ino);
    printf("File size:          %lld bytes\n", (long long)sb.st_size);
    printf("Last status change: %s", ctime(&sb.st_ctime));
    printf("Last access:        %s", ctime(&sb.st_atime));
    printf("Last modification:  %s", ctime(&sb.st_mtime));
}
```

File System Internal Format



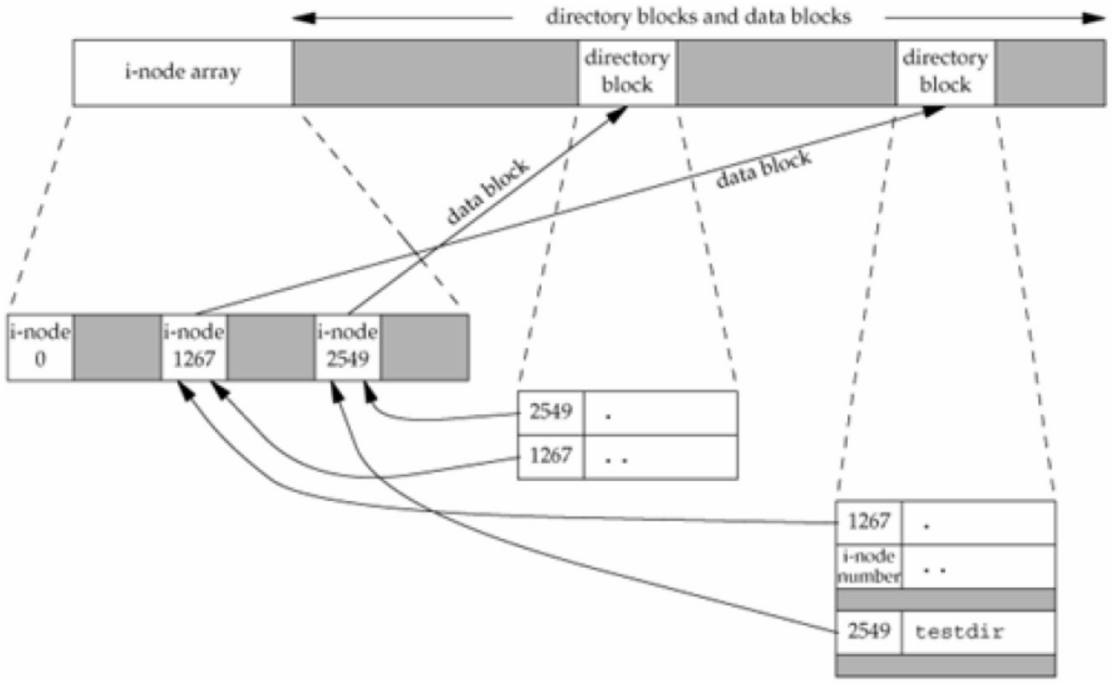


Some Fields of super_block

```
// struct super_block: information about the file system
//
struct super_block {
    struct list_head s_list;         /* list of all superblocks */
    dev_t s_dev;                     /* identifier */
    unsigned long s_blocksize;       /* block size in bytes */
    unsigned char s_dirt;            /* dirty flag */
    struct file_system_type *s_type; /* filesystem type */
    struct super_operations *s_op;   /* superblock methods */
    unsigned long s_flags;           /* mount flags */
    unsigned long s_magic;           /* filesystem's magic number */
    struct dentry *s_root;           /* directory mount point */
    int s_count;                     /* superblock ref count */
    int s_need_sync;                 /* not-yet-synced flag */
    struct list_head s_inodes;       /* list of inodes */
    struct list_head s_dirty;        /* list of dirty inodes */
    fmode_t s_mode;                  /* mount permissions */
...
};
```

Some Fields of inode

```
// struct inode: metadata about a file
//
struct inode {
    struct list_head i_sb_list;     /* inodes in the superblock */
    struct list_head i_dentry;      /* dentries referencing this inode*/
    unsigned long i_ino;            /* inode number */
    unsigned int i_nlink;           /* number of hard links */
    uid_t i_uid;                    /* user id of owner */
    gid_t i_gid;                    /* group id of owner */
    loff_t i_size;                  /* file size in bytes */
    struct timespec i_atime;        /* last access time */
    struct timespec i_mtime;        /* last modify time */
    struct timespec i_ctime;        /* last change time */
    umode_t i_mode;                 /* access permissions */
    struct inode_operations *i_op; /* inode ops table */
    struct file_operations *i_fop; /* default inode ops */
    struct super_block *i_sb;       /* associated superblock */
    void *i_private;                /* fs private pointer */
...
};
```



Sample filesystem after creating the directory `testdir`

Reading Directory Contents

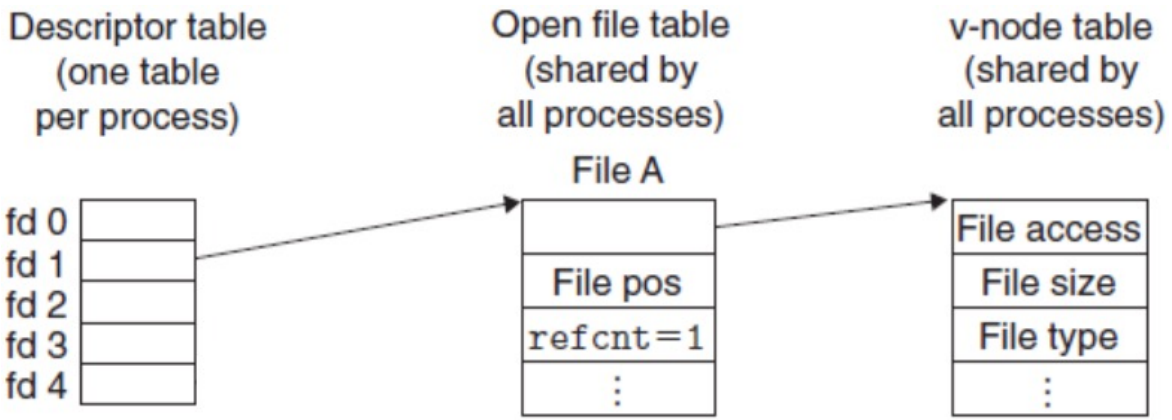
```
#include <dirent.h>
DIR *opendir(const char *name);
int closedir(DIR *dirp);
struct dirent *readdir(DIR *dirp);

#include <dirent.h>
#include <stdio.h>
int main(int argc, char **argv) {
    DIR *dp;
    struct dirent *dep;
    char *name = (argc != 2 ? "/": argv[1]);

    dp = opendir(name);
    while((dep = readdir(dp)) != NULL)
        printf("%s\n", dep->d_name);

    closedir(dp);
    return 0;
}
```

Sharing Files

Descriptor Table
- Each process has its own descriptor table
- Each open descriptor entry points to a file table entry

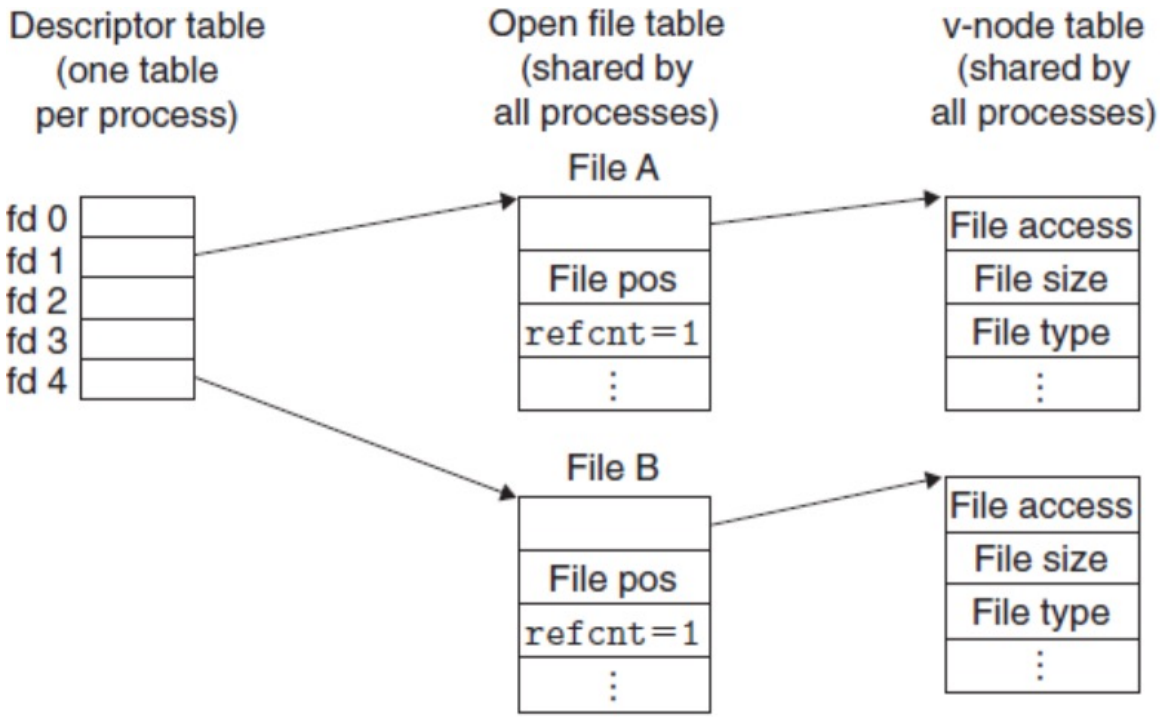
File Table
- The set of open files is represented by a file table shared by all processes
- File position, Reference count, Pointer to an entry in the v-node table

V-node table
- V-node table is shared by all processes
- Each entry contains most of the stat structure including s_mode, st_size



Sharing Files

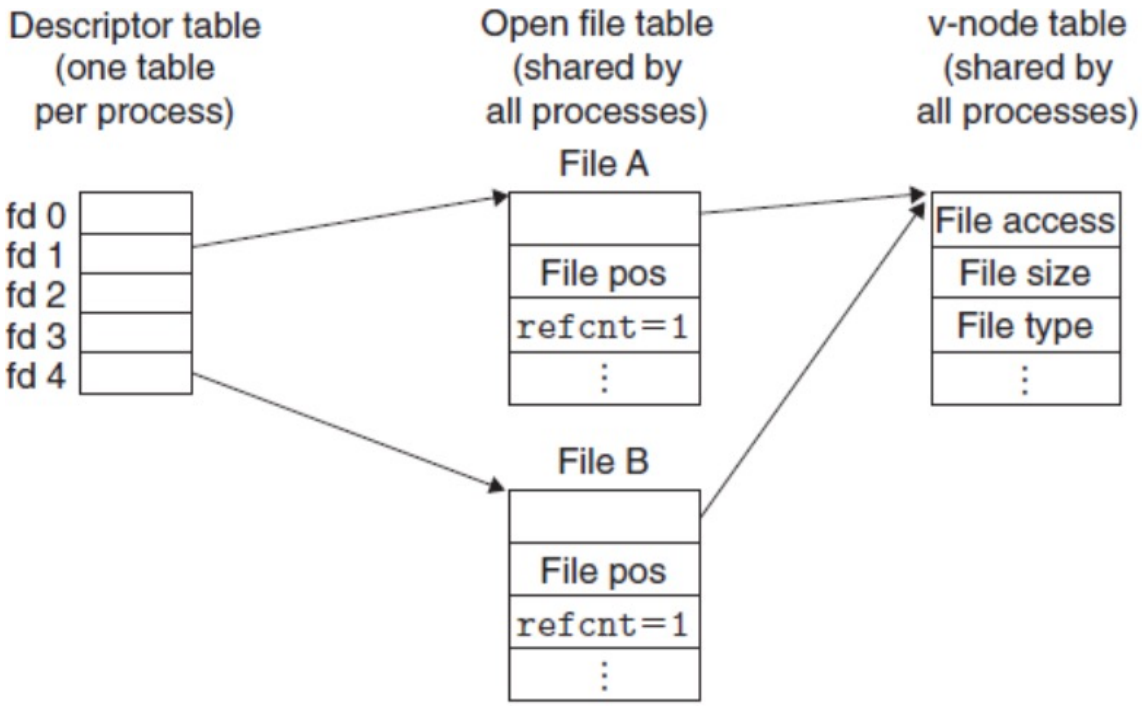
Typically, two descriptors reference distinct files



Sharing Files

A process can open the same file twice

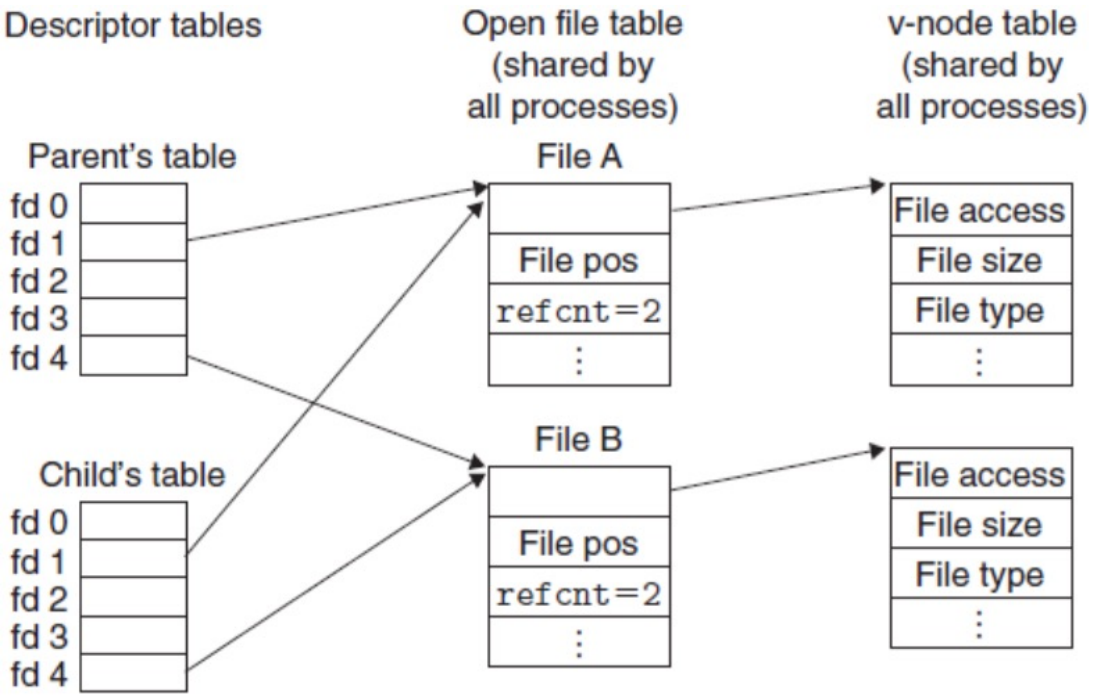
- Each descriptor has its own file position
- Both refer to the same physical file so they point to the same v-node table entry



Sharing Files

Open files and then call fork – What happens?

- Child has its own duplicate *copy of parent's descriptor table*
- *Open file tables are shared* and thus the *file positions*



Sharing Files

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

void fork_then_open();
void open_then_fork();

int main(int argc, char **argv)
{
    fork_then_open();
    open_then_fork();
}
```

```c
void fork_then_open()
{
    int pid = fork(); // fork then open
    int fd = open("foo.txt", O_CREAT|O_RDWR, S_IRUSR|S_IWUSR);

    if(pid == 0) {
        sleep(1);
        char buf[100] = {0};
        read(fd, buf, sizeof(buf));
        close(fd);
        printf("[%s]\n", buf);
        exit(0);
    }
    else {
        int status;
        write(fd, "1234567890", 11);
        close(fd);
        waitpid(pid, &status, 0);
    }
}
```

```c
void open_then_fork()
{
    int fd = open("bar.txt", O_CREAT|O_RDWR, S_IRUSR|S_IWUSR);
    int pid = fork(); // open then fork

    if(pid == 0) {
        sleep(1);
        char buf[100] = {0};
        // lseek(fd, 0, SEEK_SET);
        read(fd, buf, sizeof(buf));
        close(fd);
        printf("[%s]\n", buf);
        exit(0);
    }
    else {
        int status;
        write(fd, "1234567890", 11);
        close(fd);
        waitpid(pid, &status, 0);
    }
}
```

I/O Redirection

Redirect output to a file, read input from a file

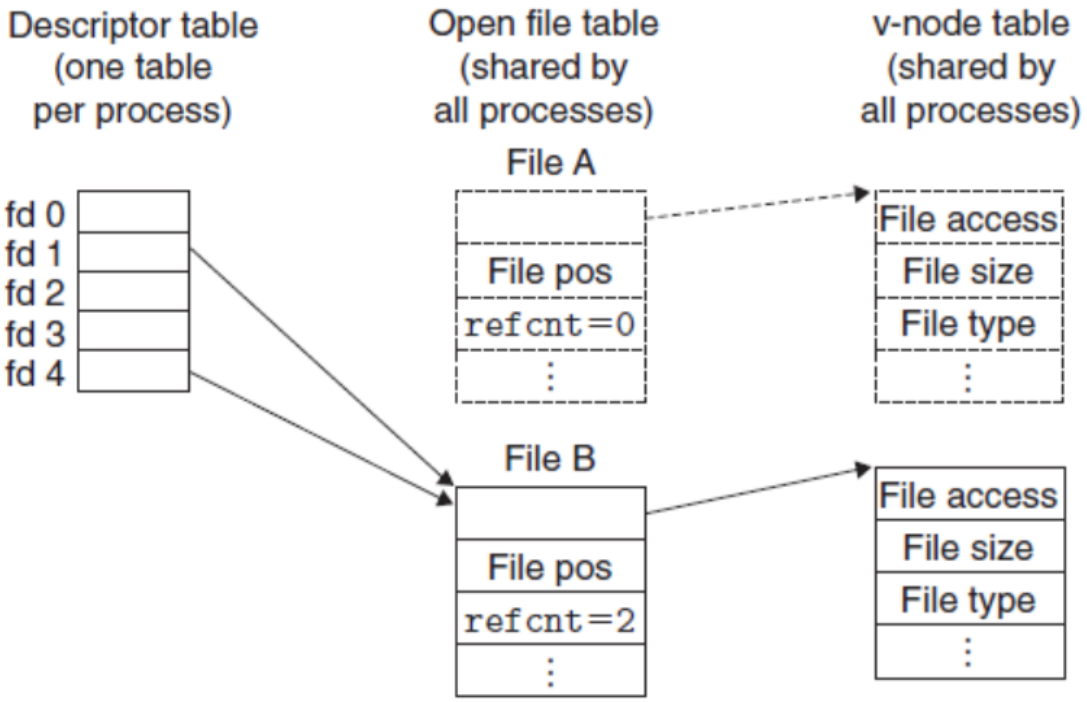
- `$ ls > foo.txt`
- `$ wc < foo.txt`
- `$ ls | wc`

dup2:

- `int dup2(int oldfd, int newfd);`
- Copy the descriptor entry in oldfd to the entry in newfd.
- If newfd was already open, close newfd before copying oldfd
- I/O Redirection: before run execve, open the file and copy its descriptor entry to entry 1 (output) or to entry 0 (input)

I/O Redirection

After dup2(4,1) when entry 1 was pointing to File A

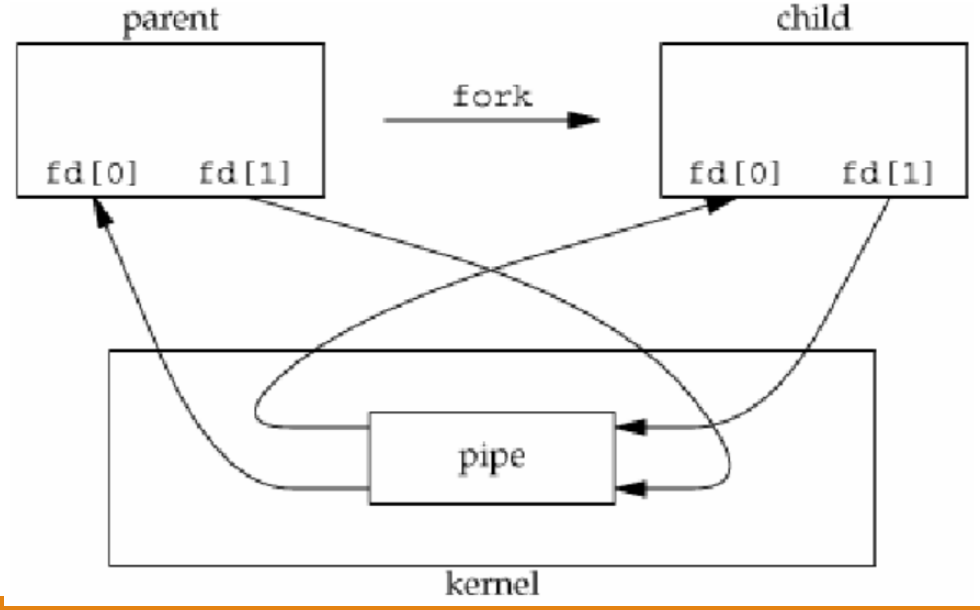


```c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main() {
    int fd1 = open("foo.txt", O_CREAT|O_TRUNC|O_WRONLY, S_IRUSR|S_IWUSR);
    write(fd1, "1234567890", 11);
    close(fd1);

    char buf[4] = {0};
    int fd2 = open("foo.txt", O_RDONLY, 0);
    int fd3 = open("foo.txt", O_RDONLY, 0);
    read(fd2, buf, 3);
    printf("fd2: %s\n", buf);
    read(fd3, buf, 3);
    printf("fd3: %s\n", buf);

    read(fd2, buf, 3);
    printf("fd2: %s\n", buf);

    dup2(fd2, fd3);   // copy the entry for fd2 to the entry for fd3
    read(fd3, buf, 3);
    printf("fd3: %s\n", buf);
}
```

Pipe

Unix IPC mechanism

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



```c
#include <unistd.h>

int main() {
    int fd[2];
    pipe(fd);

    pid_t pid = fork();
    if(pid == 0) {
        close(fd[0]);
        dup2(fd[1], 1);
        char *param[] = {"/bin/ls", NULL};
        execve(param[0], param, NULL);

    }
    else {
        close(fd[1]);
        dup2(fd[0], 0);
        waitpid(pid, NULL, 0);
        char *param[] = {"/usr/bin/wc", NULL};
        execve(param[0], param, NULL);
    }
    return 0;
}
```

Questions?