CS 214: File System in Unix

Gang Qiao

Overview

- Linux file type
- Blocking I/O and non-blocking I/O
- File operation
- File metadata
- opendir and readdir

Linux file type

- A Linux file is a sequence of m bytes
- All I/O devices (network, disks,…) are modeled as files to generalize all input and output operations as uniform ones

- *Regular file*: .txt, .out, .o, .c, …
- *Directory*: a file composing a set of file links
- *Socket*: a pseudo-file that represents a network connection

  Once a socket has been created (using the proper primitives, and the proper parameters to identify the other host), writes to the socket are turned into network packets that get sent out, and data received from the network can be read from the socket

Blocking I/O & Non-blocking I/O

- When a program carries out I/O, an I/O function provided by the system library is invoked (system call)

- *Blocking I/O*: I/O function would return only after the I/O completes
- Programs would do nothing but to wait until the I/O function returns
- Also known as *Synchronous I/O*

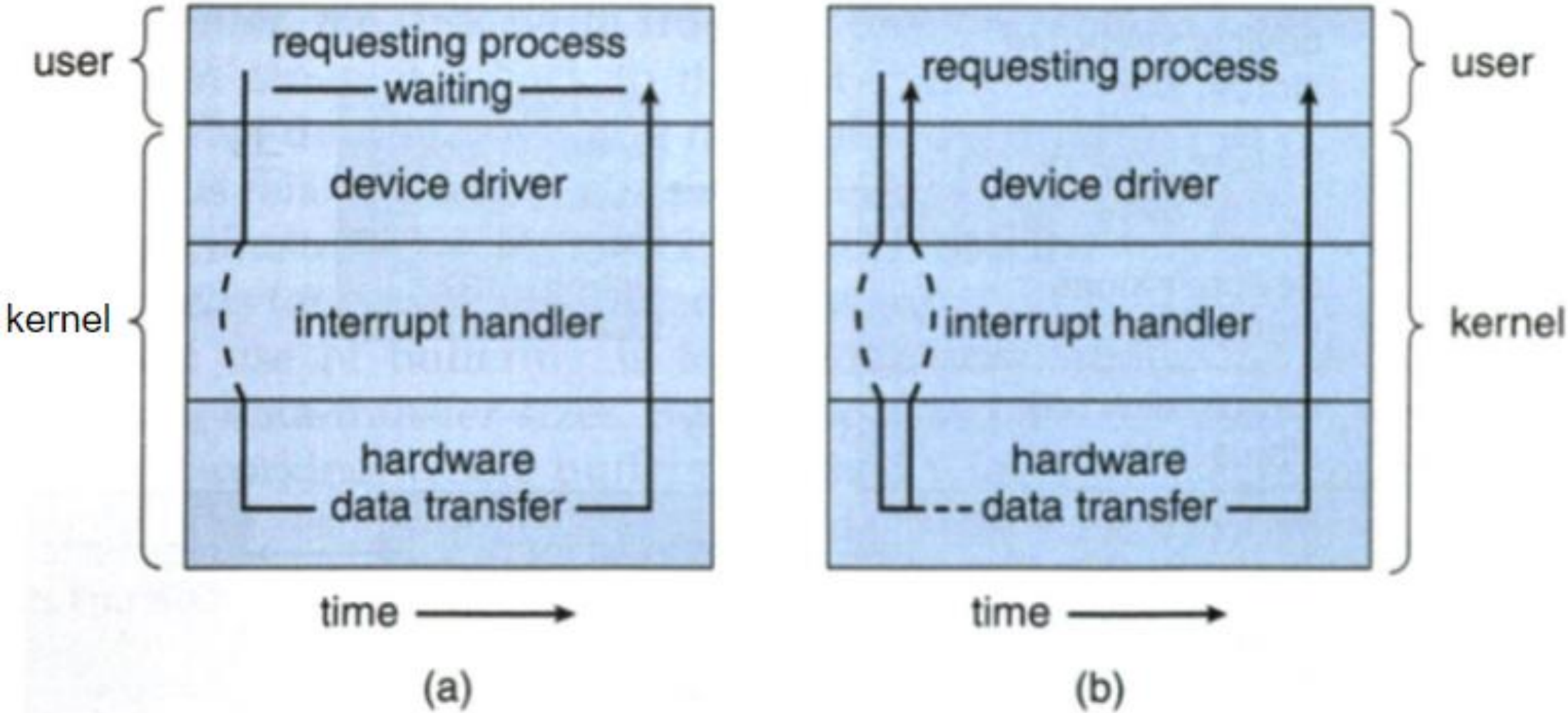
- *Non-blocking I/O*: I/O function returns as far as the I/O starts
- Programs could avoid doing nothing but waiting
- As I/O completes, such a message is delivered to the running program via signal, callback routine or other mechanisms
- Also known as *Asynchronous I/O*

[File System -> Q4] blocking and nonblocking IO

http://faculty.salina.k-state.edu/tim/ossg/Device/blocking.html

Most I/O requests are considered **blocking** requests, meaning that control does not return to the application until the I/O is complete. The delayed from systems calls such read() and write() can be quite long. Using systems call that block is sometimes called **synchronous** programming. In most cases, the wait is not really a problem because the program can not do anything else until the I/O is finished. **However, in cases such as network programming with multiple clients or with graphical user interface programming, the program may wish to perform other activity as it continues to wait for more data or input from users.**

One solution for these situations is to use multiple threads so that one part of the program is not waiting for unrelated I/O to complete. Another alternative is to use **asynchronous** programming techniques with **nonblocking** system calls. An asynchronous call returns immediately, without waiting for the I/O to complete. **The completion of the I/O is later communicated to the application either through the setting of some variable in the application or through the triggering of a signal or call-back routine** that is executed outside the linear control flow of the application.

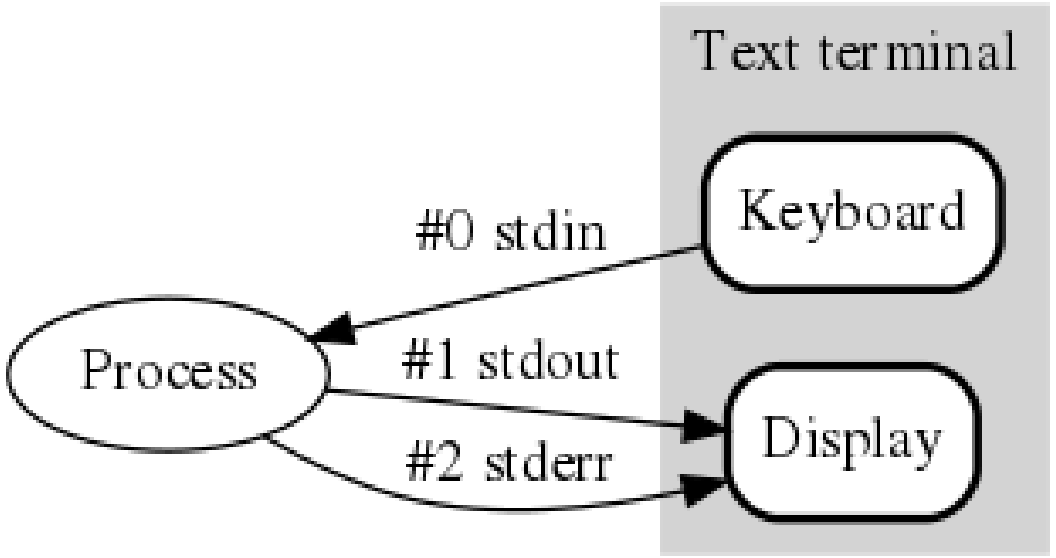


(a) blocking I/O system calls do not return until the I/O is complete.

(b) Nonblocking I/O system calls return immediately. The process is later notified when the I/O is complete.

Review: Standard I/O functions & predefined streams

- **Stream is needed when using standard I/O functions**
- Any read/write to a file via a standard I/O function is through stream
- **Operations on the file == operations to the stream**
- A stream is identified with *file descriptor*



- #0 standard input stream (associates with keyboard by default)
- #1 standard output stream (associates with display by default)
- #2 standard error stream (associates with display by default)

File operation

- ***Unix/Linux I/O functions***: access I/O devices via *system calls*
- **File descriptor is used for file operations**, no stream is needed
- open(), close(), read(), write(), …
- Low-level operations are performed directly using file descriptor

- ***Standard I/O functions***: access I/O devices in a higher-level
- A stream (represented with a FILE object) is needed to associate with a file
- Implemented by invoking Linux/Unix I/O functions
- Printf(), scanf(), fopen(), fread(), fscanf(), fgets(), sprintf(), sscanf(), …
- Streams interface could provide powerful formatted input and output functions

[File System -> Q5] File IO: read() vs fread()

read() is a low level, **unbuffered** file operation. It makes a **direct system call** on UNIX. It does a raw-file-read, and will call the OS API to read the number of bytes you requested.

fread() is a C library function, and provides **buffered** reads. It will ask the OS for a chunk of some size (compiler/library dependant, I think VC+ uses 512 bytes buffer by default, but you can change it), and at each call to fread() you will get the number of bytes from the chunk.

(read character by character, which is usually of low speed)
Unbuffered I/O involves reading or writing data one element at a time: If the data stream is character-based, like a file on disk in a UNIX system, that means reading or writing the data character-by-character. **Because the overhead involved in each I/O request is so great (due to the physical nature of the underlying device) and the number of requests is so high, this tends to be very slow. (why2?)**

(read chunk by chunk)
Buffered I/O involves reading or writing data **in chunk unit**: If a request to read a single character is made the system will actually read, say, 512 characters from the device into the chunk, then return just the first character from the buffer to the calling routine.

open()

- Import libraries: sys/types.h, sys/stat.h, fcntl.h
- Prototype:
- int open(const char **pathname*, int *flags*);
- int open(const char **pathname*, int *flags*, mode_t *mode*);
- Function: open and possibly create a file or device. OS kernel does the work for the program.
- *pathname*: the complete (or relative) path of the file on system
- *flags*: define the way in which the file needs to be opened i.e. in read-only, write-only or read-write mode
- *mode*: represent the permissions in case a new file is created using the open() function with the O_CREAT flag
- *return value*: new file descriptor if successful, -1 otherwise

open(): flags

- int open(const char *_pathname_, int ***flags***, mode_t _mode_);
- _flags_: define the way in which the file needs to be opened i.e. in read-only, write-only or read-write mode
- O_RDONLY, O_WRONLY, O_RDWR

- Optional instructions
- O_CREAT: if the file doesn't exist, then created an empty file
- O_TRUNC: if the file already exists and is a regular file and the open mode allows writing (i.e., is O_RDWR or O_WRONLY), it will be truncated to length 0
- O_APPEND: open the file in append mode, i.e. set the starting position to the end of the file before each write operation

- Combination using "|"
- fd = open("foo.txt", O_WRONLY|O_APPEND, 0);

open(): mode

- int open(const char **pathname*, int *flags*, mode_t ***mode***);
- *mode*: represent the permissions in case a new file is created using the open() function with the O_CREAT flag

- S_IRUSR / S_IWUSR / S_IXUSR: User(owner) can read / write / execute this file
- S_IRWXU: User(owner) can read, write and execute this file

- S_IRGRP / S_IWGRP / S_IXGRP: Members of the owner's group can read / write / execute this file
- S_IRWXG: Members of the owner's group can read, write and execute this file

- S_IROTH / S_IWOTH / S_IXOTH: Others (anyone) can read / write / execute this file
- S_IRWXO: Others (anyone) can read, write and execute this file

open(): operations

- Three operations/permissions on a file: read, write and execute
- Change the permissions: *chmod* command
- Q: write the chmod call to give all users in the test.c's creator's group write and read access
- A: chmof g+rw test.c

open(): examples

- fd = open(“bar.txt”, O_RDONLY|O_CREAT, S_IRWXU);

- #define DEF_MODE S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH
- fd = open(“bar.txt”, O_CREAT|O_TRUNC|O_WRONLY, DEF_MODE);

close()

- Import library: unistd.h
- Prototype: int close(int *fd*);
- Function: close an opened file represented by a file descriptor *fd*
- *return value*: 0 on success, -1 on error, ***errno*** is set propriately

- Import library: errno.h
- Errno: indicate what goes wrong in the event of an error in the system
- Errno is set by system calls and some library functions

read()

- Import library: unistd.h
- Prototype: ssize_t read(int *fd*, void **buf*, size_t *count*);
- Function: read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*
- *return value*: number of bytes read if successful, 0 on EOF, -1 on error

- ssize_t: signed long in essence, which could be a negative number (-1 on error)
- size_t: unsigned long in essence, which should be larger than or equal to 0

write()

- Import library: unistd.h
- Prototype: ssize_t write(int *fd*, const void **buf*, size_t *count*);
- Function: write up to *count* bytes from the buffer starting at *buf* to the file referred to by the file descriptor *fd*
- *return value*: number of bytes written if successful, -1 on error

File metadata

- Metadata of a file in Linux is represented as a predefined structure ***stat***
- Use ***stat*** and ***fstat*** functions to get the metadata of a file

- struct stat {
  - dev_t st_dev; /* device */
  - ino_t st_ino; /* inode */
  - mode_t st_mode; /* protection & 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 access */
  - time_t st_mtime; /* time of last modification */
  - time_t st_ctime; /* time of last change */
- };

stat() & fstat()

- Import libraries: unistd.h, sys/stat.h

- Stat function
- Prototype: int stat(**const char *_filename_**, struct stat *_buf_);
- *return value*: 0 if OK, -1 on error

- Fstat function
- Prototype: int fstat(**int _fd_**, struct stat *_buf_);
- *return value*: 0 if OK, -1 on error

C Program to list all files and sub-directories in a directory

```c
#include <stdio.h>
#include <dirent.h>

int main(void)
{
    struct dirent *de;  // Pointer for directory entry

    // opendir() returns a pointer of DIR type.
    DIR *dr = opendir(".");

    if (dr == NULL)  // opendir returns NULL if couldn't open directory
    {
        printf("Could not open current directory" );
        return 0;
    }

    // Refer http://pubs.opengroup.org/onlinepubs/7990989775/xsh/readdir.html
    // for readdir()
    while ((de = readdir(dr)) != NULL)
            printf("%s\n", de->d_name);

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

NAME top

opendir, fdopendir - open a directory

SYNOPSIS top

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

DIR *opendir(const char *name);
DIR *fdopendir(int fd);
```

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

fdopendir():
    Since glibc 2.10:
        _POSIX_C_SOURCE >= 200809L
    Before glibc 2.10:
        _GNU_SOURCE

DESCRIPTION top

The **opendir**() function opens a directory stream corresponding to the directory *name*, and returns a pointer to the directory stream. The stream is positioned at the first entry in the directory.

The **fdopendir**() function is like **opendir**(), but returns a directory stream for the directory referred to by the open file descriptor *fd*. After a successful call to **fdopendir**(), *fd* is used internally by the implementation, and should not otherwise be used by the application.

SYNOPSIS top

```
#include <dirent.h>

struct dirent *readdir(DIR *dirp);
```

DESCRIPTION top

The **readdir**() function returns a pointer to a *dirent* structure representing the next directory entry in the directory stream pointed to by *dirp*. It returns NULL on reaching the end of the directory stream or if an error occurred.

In the glibc implementation, the *dirent* structure is defined as follows:

```
struct dirent {
    ino_t          d_ino;       /* Inode number */
    off_t          d_off;       /* Not an offset; see below */
    unsigned short d_reclen;    /* Length of this record */
    unsigned char  d_type;      /* Type of file; not supported
                                   by all filesystem types */
    char           d_name[256]; /* Null-terminated filename */
};
```

The only fields in the *dirent* structure that are mandated by POSIX.1 are *d_name* and *d_ino*. The other fields are unstandardized, and not present on all systems; see NOTES below for some further details.

The fields of the *dirent* structure are as follows:

d_ino This is the inode number of the file.

d_off The value returned in *d_off* is the same as would be returned by calling telldir(3) at the current position in the directory stream. Be aware that despite its type and name, the *d_off* field is seldom any kind of directory offset on modern filesystems. Applications should treat this field as an opaque value, making no assumptions about its contents; see also telldir(3).

d_reclen
This is the size (in bytes) of the returned record. This may not match the size of the structure definition shown above; see NOTES.

d_type This field contains a value indicating the file type, making it possible to avoid the expense of calling lstat(2) if further actions depend on the type of the file.

When a suitable feature test macro is defined (**_DEFAULT_SOURCE** on glibc versions since 2.19, or **_BSD_SOURCE** on glibc versions 2.19 and earlier), glibc defines the following macro constants for the value returned in *d_type*:

DT_BLK This is a block device.

DT_CHR This is a character device.

DT_DIR This is a directory.

DT_FIFO This is a named pipe (FIFO).

DT_LNK This is a symbolic link.

https://stackoverflow.com/questions/8436841/how-to-recursively-list-directories-in-c-on-linux

C program that recursively find all files in a directory and its sub directories

```c
#include <unistd.h>
#include <sys/types.h>
#include <dirent.h>
#include <stdio.h>
#include <string.h>

void listdir(const char *name, int indent)
{
    DIR *dir;
    struct dirent *entry;

    if (!(dir = opendir(name)))
        return;

    while ((entry = readdir(dir)) != NULL) {
        if (entry->d_type == DT_DIR) {
            char path[1024];
            if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0)
                continue;
            snprintf(path, sizeof(path), "%s/%s", name, entry->d_name);
            printf("%*s[%s]\n", indent, "", entry->d_name);
            listdir(path, indent + 2);
        } else {
            printf("%*s- %s\n", indent, "", entry->d_name);
        }
    }
    closedir(dir);
}
```

```c
int main(void) {
    listdir(".", 0);
    return 0;
}
```

Read the directory stream and judge its type in a loop

//this is a directory

//the directory stream automatically contains dir '.' and '..'

load the subdirectory into path and recursively call listdir (depth-first search)