



**Brain W. Kernighan, & Rob Pike**

## **The Unix Programming Environment**

**PHI**



**Kay A. Robbins, & Steve Robbins**

## **Unix<sup>TM</sup> Systems Programming**

**Communications, concurrency, and Threads**

**Pearson Education**

# system() Function

**system()** - execute a shell command

👉 SYNOPSIS:

```
#include <stdlib.h>

int system(const char *command);
```

- 👉 The **system()** library function uses **fork()** to create a child process that executes the shell command specified in **command** using **exec1()** as **:exec1("/bin/sh", "sh", "-c", command, (char \*) 0);**
- 👉 **system()** returns after the command has been completed.
- 👉 If **command** is **NULL**, then **system()** returns a status indicating whether a shell is available on the system.

# `system()` Function Return Value

The return value of **`system()`** is one of the following:

- ☞ If command is `NULL`, then a nonzero value if a shell is available, or 0 if no shell is available.
- ☞ If a child process could not be created, or its status could not be retrieved, the return value is -1.
- ☞ If a shell could not be executed in the child process, then the return value is as though the child shell terminated by calling `_exit()` with the status 127.
- ☞ If all system calls succeed, then the return value is the termination status of the child shell used to execute command. (The termination status of a shell is the termination status of the last command it executes.)

# read System Call

- 👉 UNIX provides sequential access to files and other devices through the **read** and **write** functions.
- 👉 The **read** function attempts to retrieve **nbyte** bytes from the file or device represented by **fildev** into the user variable **buf**.
- 👉 A large enough buffer must be provided to hold **nbyte** bytes of data.
- 👉 SYNOPSIS:

```
#include <unistd.h>
```

```
ssize_t read(int fildev, void *buf, size_t nbyte);
```

- (1) If successful, **read** returns the number of bytes actually read.
- (2) If unsuccessful, **read** returns **-1** and sets **errno**.

- 👉 The **ssize\_t** data type is a signed integer data type used for the number of bytes read, or **-1** if an error occurs.
- 👉 The **size\_t** is an unsigned integer data type for the number of bytes to read.

# Note: Read

- 👉 A **read** operation for a regular file may return fewer bytes than requested if, for example, it reached end-of-file before completely satisfying the request.
- 👉 A **read** operation for a regular file returns 0 to indicate end-of-file.
- 👉 When reading from a terminal, read returns 0 when the user enters an end-of-file character( CTRL+D ).

# write System Call

- 👉 The **write** function attempts to output **nbyte** bytes from the user buffer **buf** to the file represented by file descriptor **fildes**.
- 👉 SYNOPSIS:

```
#include <unistd.h>
```

```
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

- (1) If successful, write returns the number of bytes actually written.
- (2) If unsuccessful, write returns -1 and sets errno.

# open System Call

- ☞ The `open` function associates a file descriptor with a file or physical device.
- ☞ SYNOPSIS:

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

```
int open(const char *path, int oflag, ...);
```

- (1) If successful, `open` returns a nonnegative integer representing the open file descriptor.
- (2) If unsuccessful, `open` returns `-1` and sets `errno`.

- ☞ The `path` parameter of `open` points to the pathname of the file or device.
- ☞ The `oflag` parameter specifies status flags and access modes for the opened file.
- ☞ A third parameter must be included to specify access permissions if a file is created.

# File Descriptor

- ✍ Files are designated with in C program either by **file pointers** or by **file descriptor**.
- ✍ A file **descriptor** is an integer value that represents a file or device that is open.
- ✍ It is an index into the process file descriptor table.
- ✍ The file descriptor table is in the process user area and provides access to the system information for the associated file or device.
- ✍ File pointers and file descriptors provide logical designations called *handles* for performing device-independent input and output.



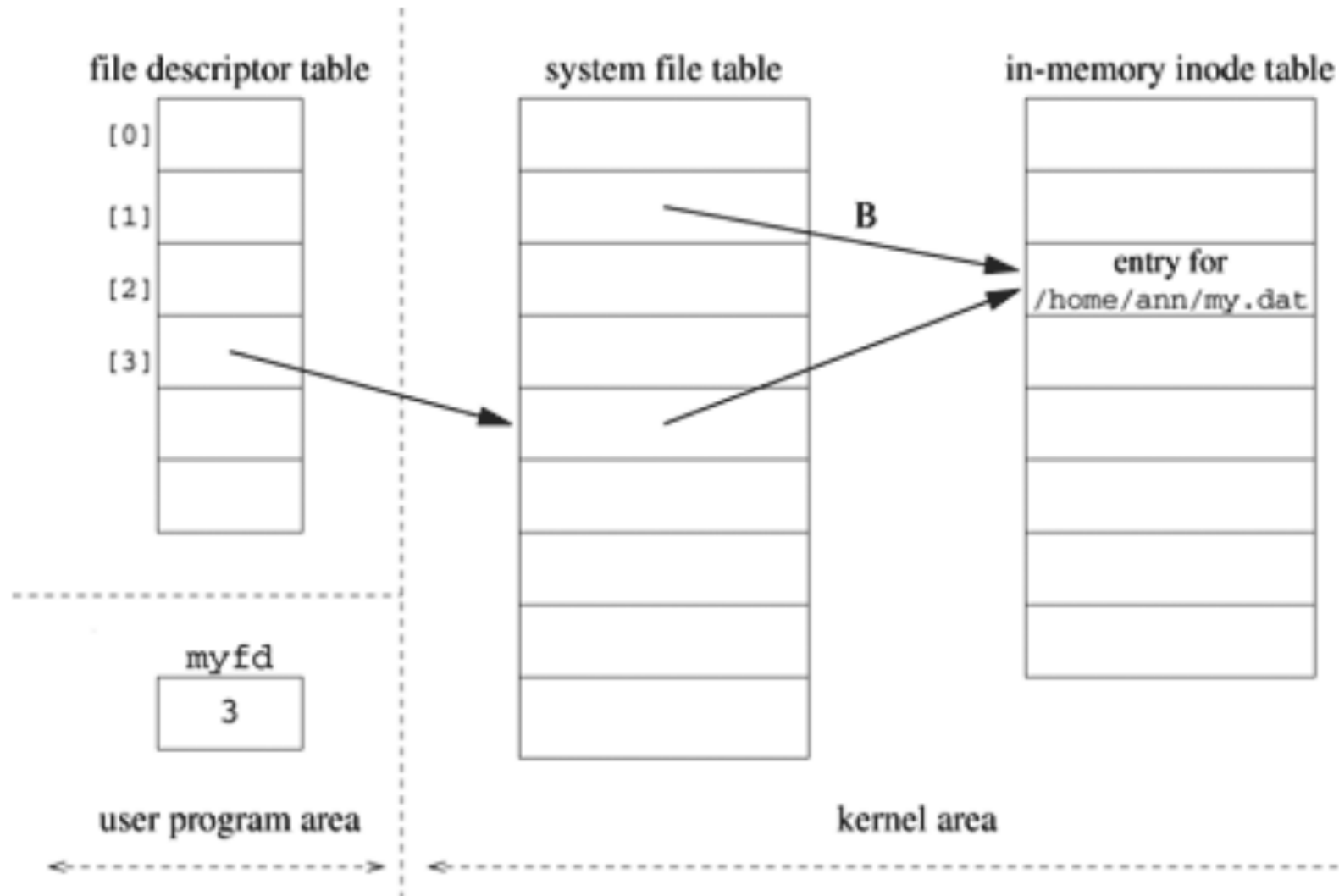
# File Descriptor

- ✍ The symbolic names for the **file pointers** that represent standard input, standard output and standard error are **`stdin`**, **`stdout`** and **`stderr`**, respectively. These symbolic names are defined in **`stdio.h`**.
- ✍ The symbolic names for the **file descriptors** that represent standard input, standard output and standard error are **`STDIN_FILENO`**, **`STDOUT_FILENO`** and **`STDERR_FILENO`**, respectively. These symbolic names are defined in **`unistd.h`**.
- ✍ The **numeric values** that represent standard input, standard output and standard error are 0, 1, and 2 respectively.




# File Descriptor

A schematic of the file descriptor table after a program executes the following.

```
myfd = open("/home/ann/my.dat", O_RDONLY);
```

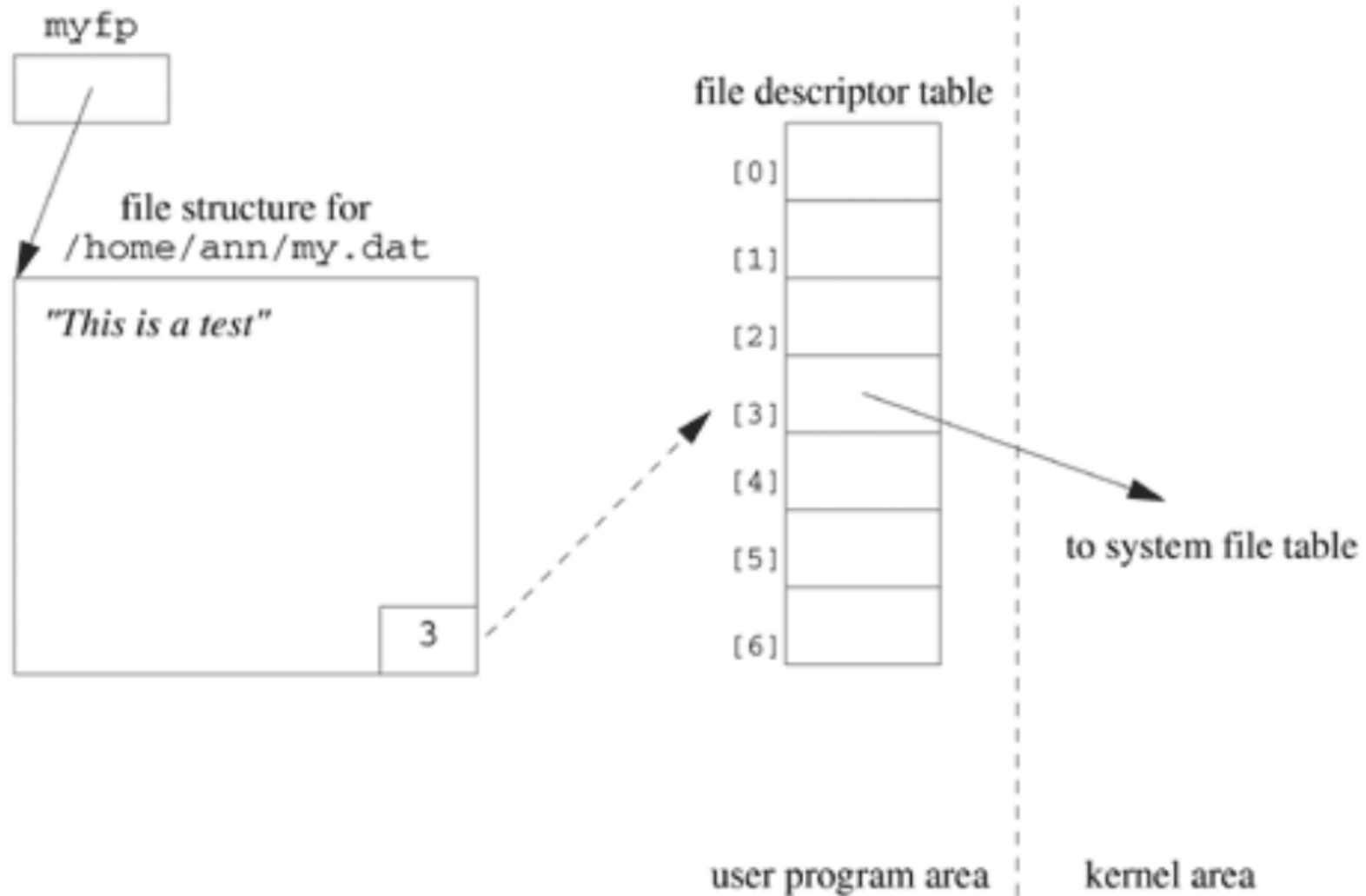


# File Descriptor

-  The **open** function creates an entry in the file descriptor table that points to an entry in the system file table.
-  The **open** function returns the value 3, specifying that the file descriptor entry is in position three of the process file descriptor table.
-  The system file table, which is shared by all the processes in the system, has an entry for each active **open**.

# File Pointer

```
FILE *myfp;  
if ((myfp = fopen("/home/ann/my.dat", "w")) == NULL)  
    perror("Failed to open /home/ann/my.dat");  
else  
    fprintf(myfp, "This is a test");
```



# oflag Argument Setting

 The POSIX values for the **access mode flags** in **oflag** argument:

1. **O\_RDONLY** : read-only access
2. **O\_WRONLY** : write-only access
3. **O\_RDWR** : read-write-only access

**Note:** specify exactly one of above designating read-only, write-only or read-write access.

 The **oflag** argument is also constructed by taking the bitwise OR (|) of the desired combination of the access mode and the **additional flags**.

 The additional flags:

1. **O\_APPEND**
2. **O\_CREAT**
3. **O\_EXCL**
4. **O\_NOCTTY**
5. **O\_NONBLOCK**
6. **O\_TRUNC**

# Additional Flags

**O\_APPEND:** The O\_APPEND flag causes the file offset to be moved to the end of the file before a write, allowing you to add to an existing file.

**O\_CREAT:** The O\_CREAT flag causes a file to be created if it doesn't already exist. If O\_CREAT flag is included, a third argument to `open()` must be passed to designate the permissions.

**O\_EXCL:** If you want to avoid writing over an existing file, use the combination O\_CREAT | O\_EXCL. This combination returns an error if the file already exists.

**O\_NOCTTY:** The O\_NOCTTY flag prevents an opened device from becoming a controlling terminal.

**O\_NONBLOCK:** The O\_NONBLOCK flag controls whether the `open()` returns immediately or blocks until the device is ready.

**O\_TRUNC:** O\_TRUNC truncates the length of a regular file opened for writing to 0.

# open () Examples

The following code segment opens the file /home/students/my.dat for reading

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

int myfd;
myfd = open("/home/students/my.dat", O_RDONLY);
```

How would you modify above Example to open /home/another/my.dat for nonblocking read?

```
Perform OR the O_RDONLY and the O_NONBLOCK flags.

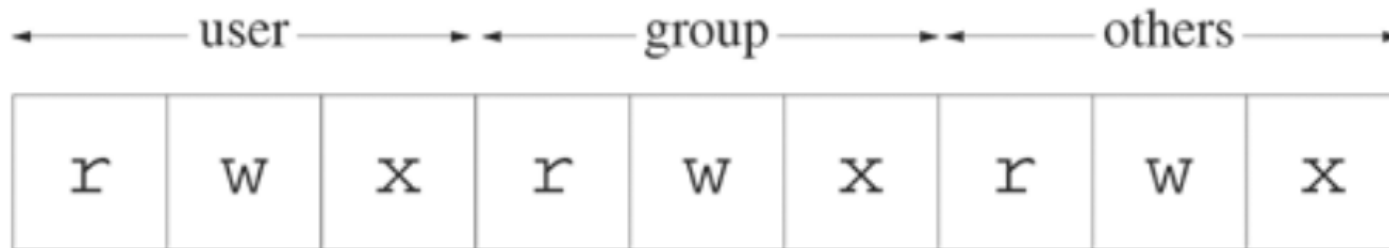
myfd = open("/home/students/my.dat", O_RDONLY |
            O_NONBLOCK);
```

# Third argument to `open()`

- ☞ Each file has three classes associated with it: a user (or owner), a group and everybody else (others).
- ☞ The possible permissions or privileges are read(r), write(w) and execute(x). These privileges are specified separately for the user, the group and others.
- ☞ When you open a file with the `O_CREAT` flag, you must specify the permissions as the third argument to `open` in a mask of type `mode_t`.
- ☞ POSIX defines symbolic names for masks corresponding to the permission bits. These names are defined in **`sys/stat.h`**



# POSIX symbolic names for file permissions



Symbol	meaning
S_IRUSR	read by owner
S_IWUSR	write by owner
S_IXUSR	execute by owner
S_IRWXU	read, write, execute by owner
S_IRGRP	read by group
S_IWGRP	write by group
S_IXGRP	execute by group
S_IRWXG	read, write, execute by group
S_IROTH	read by others
S_IWOTH	write by others
S_IXOTH	execute by others
S_IRWXO	read, write, execute by others
S_ISUID	set user ID on execution
S_ISGID	set group ID on execution

# Example

The following code segment creates a file, info.dat, in the current directory. If the info.dat file already exists, it is overwritten. The new file can be read or written by the user and only read by everyone else.

```
int fd;

mode_t fdmode = (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);

if ((fd = open("info.dat", O_RDWR | O_CREAT, fdmode)) ==
    -1)
    perror("Failed to open info.dat");
```

# close () System Call

The close function has a single parameter, **fildes**, representing the open file whose resources are to be released.

```
#include <unistd.h>
```

```
int close(int fildes);
```

- (1) If successful, close returns 0.
- (2) If unsuccessful, close returns -1 and sets errno.

# Filters

- ✍ A filter reads from standard input, performs a transformation, and outputs the result to standard output.
- ✍ Filters write their error messages to standard error.
- ✍ **Examples:** `head`, `tail`, `more`, `sort`, `grep`, `sed`, **and** `awk` etc.
- ✍ **cat as a filter:** The `cat` command takes a list of filenames as command-line arguments, reads each of the files in succession, and echoes the contents of each file to standard output. However, if no input file is specified, `cat` takes its input from standard input and writes its results to standard output. In this case, `cat` behaves like a filter.

# Redirection

- ✍ A file descriptor is an index into the file descriptor table of that process.
- ✍ Each entry in the file descriptor table points to an entry in the system file table, which is created when the file is opened.
- ✍ A program can modify the file descriptor table entry so that it points to a different entry in the system file table. This action is known as **redirection**.
- ✍ Most shells interpret the greater than character (`>`) on the command line as redirection of standard output and the less than character (`<`) as redirection of standard input.
- ✍ **Example:** `$ cat > myfile.txt`, redirects standard output to `myfile.txt` with `>`.