Advanced System Programming

Agenda

- Static Library
- Dynamic Library
- Error Handling
- Asserts
- Process management commands
- File operation

Linux Libraries

Libraries in Linux

- Virtually all programs in Linux are linked against one or more libraries.

- Libraries contain code and data that provide services to independent programs.

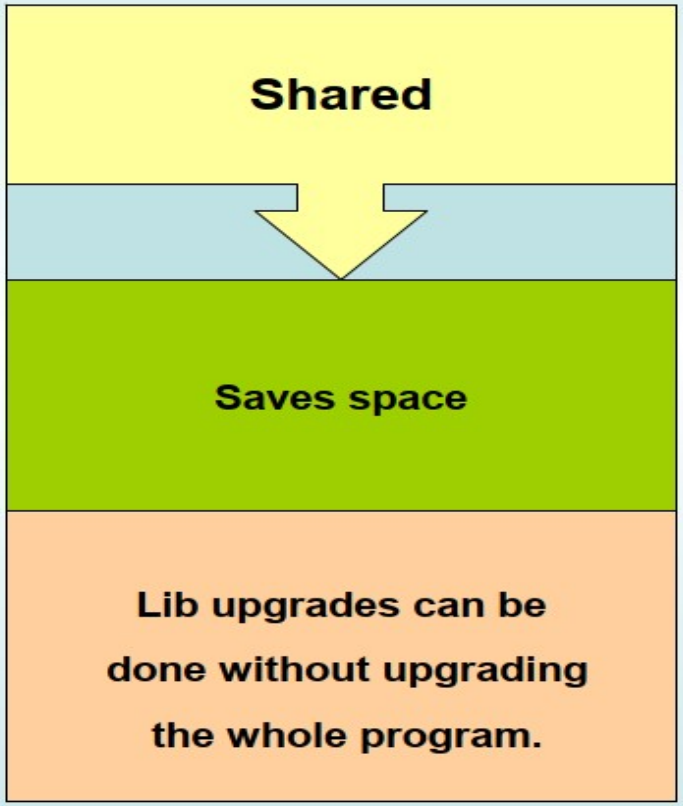
- There are two types of library in Linux:

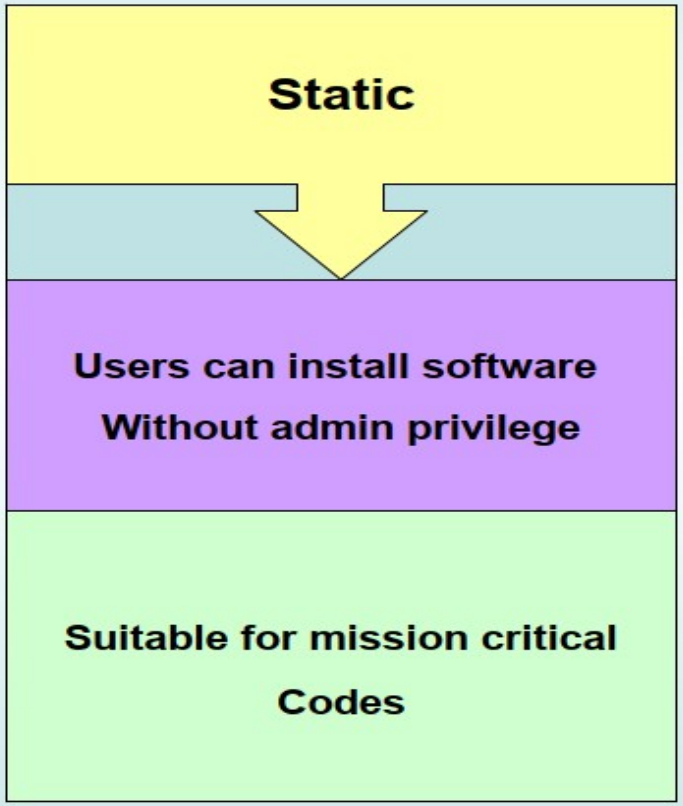
    - Static libraries (Archive file, same as windows .LIB file).

    - Shared libraries (Shared Object, same as windows DLL).

Linux Libraries

Shared vs. Static





Static Libraries

Static Library

- Is a collection of object files.

- Linker extracts needed object files from archive and attaches them to your program (as they were provided directly).

- When linker encounters an archive in command line, it searches the already passed objects to see if there is a reference to objects in this archive or not.

Static Library

ar cr <libraryname> <list of object files>

Example:

ar cr libfuncs.a obj1.o obj2.o

Static Libraries

Static Library

- If linker find the reference, it will extract the object from archive and put it in our exe.

- If linker could not find any references, it shows an error and stops.

IT IS IMPORTANT TO PASS THE COMMAND LINE OPTIONS IN CORRECT ORDER

Static Library

gcc <list of sources and object files> -L. –l<library name without lib prefix and .a suffix>

Example:

gcc -o statictest statictest.c -L. -lfuncs <correct way of calling>

gcc -L. -lfuncs -o statictest statictest.c <Incorrect way of calling will result in error>

Shared Library

- Is also a collection of objects.

- When it is linked into another program, the program does not contain the whole objects, but just references to the shared library.

- Is not a collection of object files, but a single big object file which is a combination of object files.

- Shared Libraries are Position Independent Codes, because the function in a SO, may be loaded at different addresses in different programs.

Shared Library

- The linker just includes the name of the “so” in executable file.

- The Operating System is responsible to find the specified “so” file.

- By default, system searches only “/lib” and “/usr/lib”.

- You can indicate another path by setting the LD_LIBRARY_PATH environment variable.

Shared Library

gcc -shared -fPIC -o <shared library name> <object files list>

Note: object files should also be compiled with –fPIC option example [gcc -c -fPIC -o obj1.o obj1.c]

Example:

gcc -shared -fPIC -o libfuncs.so obj1.o obj2.o

- LD_LIBRARY_PATH to set the shared library search path
- LD_DEBUG=<options>

Shared Library

Libraries in Linux

- The ldd command shows the shared libraries that are linked into an executable (and their dependencies).

- Static libs, can not point to another lib, so you should include all dependent libs in GCC command line.

- The included SO s, need to be available during execution.

- The linker will stop searching for libraries when it finds a directory containing the proper “.so” or “.a”.

- Priority of “.so” is higher than “.a” unless explicitly specified (-static option in gcc).

Shared Library

While compiling a source, you should indicate which libraries (which .SO) are needed.

While executing the code, the indicated SO should be available otherwise the code will not run.

Error Handling

- A program may encounter a situation which can not work correctly.

- In case of an error, your program may decide to:

  - Ignore the error and continue running.
  - Stop working immediately.
  - Decide what to do next (is error recoverable?)

- The ability of a program to deal with errors is called "*Error Handling*".

Error Handling

- The first step in handling an error is to determine it's happened.

- In your program, you are responsible of checking for errors.

- In Linux, when calling a system call, if some error happens, the system call will set the *errno* global variable.

- Most system calls, return -1 on error and set errno respectively.

- After performing any system call, it's up to you to check the return value of a call and deal with probable errors.

Error Handling

- The *errno* variable is global, so you should check it exactly after desired call.

- *errno* is thread-safe.

- There are some functions to work with errno and print meaningful error messages.

- Using *strerror()* and *strerror_r()* is an option to deal with errors.

- These functions will return a string describing the error code passed in the argument.

Error Handling

Table

number	hex	symbol	description
1	0x01	EPERM	Operation not permitted
2	0x02	ENOENT	No such file or directory
3	0x03	ESRCH	No such process
4	0x04	EINTR	Interrupted system call
5	0x05	EIO	Input/output error
6	0x06	ENXIO	No such device or address
7	0x07	E2BIG	Argument list too long
8	0x08	ENOEXEC	Exec format error
9	0x09	EBADF	Bad file descriptor
10	0x0a	ECHILD	No child processes
11	0x0b	EAGAIN	Resource temporarily unavailable
11	0x0b	EWOULDBLOCK	*(Same value as EAGAIN)* Resource temporarily unavailable
12	0x0c	ENOMEM	Cannot allocate memory
13	0x0d	EACCES	Permission denied
14	0x0e	EFAULT	Bad address
15	0x0f	ENOTBLK	Block device required
16	0x10	EBUSY	Device or resource busy
17	0x11	EEXIST	File exists
18	0x12	EXDEV	Invalid cross-device link
19	0x13	ENODEV	No such device

Error Handling

https://chromium.googlesource.com/chromiumos/docs/+/master/constants/errnos.md

Asserts

Error Handling

- You may also use the *assert* macro in your C program.

- One may use *assert* to properly generate some information in case of unpredicted situations.

- You can disable all *assert* s in your code providing – DNDEBUG option in your gcc command line.

- You should never perform any operation in your *assert* statement. Just check it.

Asserts

Error Handling

- You may also use the *assert* macro in your C program.

- One may use *assert* to properly generate some information in case of unpredicted situations.

- You can disable all *assert* s in your code providing – DNDEBUG option in your gcc command line.

- You should never perform any operation in your *assert* statement. Just check it.

Asserts

Syntax

The syntax for the assert macro in the C Language is:

void assert(int expression);

- **expression** : An expression that we expect to be true under normal circumstances.

Process management command - ps

Linux provides us a utility called ps for viewing information related with the processes on a system which stands as abbreviation for “Process Status”. ps command is used to list the currently running processes and their PIDs along with some other information depends on different options. It reads the process information from the virtual files in /proc file-system. /proc contains virtual files, this is the reason it’s referred as a virtual file system.

ps provides numerous options for manipulating the output according to our need.

Syntax

```
ps [options]
```

Process management command - ps

```
PROCESS STATE CODES
       Here are the different values that the s, stat and state output
       specifiers (header "STAT" or "S") will display to describe the state of
       a process.
       D    Uninterruptible sleep (usually IO)
       R    Running or runnable (on run queue)
       S    Interruptible sleep (waiting for an event to complete)
       T    Stopped, either by a job control signal or because it is being
            traced.
       W    paging (not valid since the 2.6.xx kernel)
       X    dead (should never be seen)
       Z    Defunct ("zombie") process, terminated but not reaped by its
            parent.

       For BSD formats and when the stat keyword is used, additional
       characters may be displayed:
       <    high-priority (not nice to other users)
       N    low-priority (nice to other users)
       L    has pages locked into memory (for real-time and custom IO)
       s    is a session leader
       l    is multi-threaded (using CLONE_THREAD, like NPTL pthreads do)
       +    is in the foreground process group
```

Process management command - top

top command is used to show the Linux processes. It provides a dynamic real-time view of the running system. Usually, this command shows the summary information of the system and the list of processes or threads which are currently managed by the Linux Kernel.

As soon as you will run this command it will open an interactive command mode where the top half portion will contain the statistics of processes and resource usage. And Lower half contains a list of the currently running processes. Pressing q will simply exit the command mode.

Process management command - kill

kill command in Linux (located in /bin/kill), is a built-in command which is used to terminate processes manually. kill command sends a signal to a process which terminates the process. If the user doesn't specify any signal which is to be sent along with kill command then default TERM signal is sent that terminates the process.

- **Kill –l :** Display all the available signals.

Process management command - killall

Killall command would kill the process by name.

Process management command - pidof

pidof command is used to find out the process IDs of a specific running program. It is basically an identification number that is automatically assigned to each process when it is created.

Syntax:

pidof [options] program1 program2 ... programN

Process management command - lsof

lsof command stands for **List Of Open File**. This command provides a list of files that are opened. Basically, it gives the information to find out the files which are opened by which process. With one go it lists out all open files in output console. It cannot only list common regular files but it can list a directory, a block special file, a shared library, a character special file, a regular pipe, a named pipe, an internet socket, a UNIX domain socket, and many others. it can be combined with grep command can be used to do advanced searching and listing.

Syntax:

$lsof [option][user name]

Process management command - lsof

List all open files: This command lists out all the files that are opened by any process in the system.

 # lsof

The shows details of files which are opened. Process Id, the user associated with the process, FD(file descriptor), size of the file all together gives detailed information about the file opened by the command, process ID, user, its size etc.

FD represents as File descripter.

cwd : Current working directory.

txt : Text file.

mem : Memory file.

mmap : Memory mapped device.

Process management command - lsof

Type

DIR: Directory

REG: Regular file

CHR: Character special file

Process management command - lsof

List all files opened by a user: There are several users of a system and each user have different requirements and accordingly they use files and devices. To find a list of files that are opened by a specific user this command is useful.

Syntax:

 # lsof -u username

Example:

 # lsof -u test

Process management command - lsof

List all files which are opened by everyone except a specific user: With the help of this command you can list out all the files opened by all the process and all the user. But when we want to find the list of files that are opened by all users except a particular user then we can use:

Syntax:

 # lsof -u ^root

List all open files by a particular Process: This command can list out all the files opened by a particular process. -c followed by process names can find out all the files that are opened by that particular process that is named in the command.

Syntax:

 # lsof -c Mysql

Process management command - lsof

List all open files that are opened by a particular process: Each file is associated with some process ID. There can be many files that are opened by a particular process. By using lsof -p process ID, files opened by a particular process can be checked.

Syntax:

 # lsof -p process ID

Files opened by all other PID: As the above-given figure command lists out the files opened by a particular process ID. In the same way, you can use below command option to find out the list of files which are not opened by a particular process ID.

Syntax:

 # lsof -p ^process ID

Process management command - lsof

List parent process IDs: There is a large number of process running in a system and they have files opened for its usage. There may be many child processes of a process and this process can also be termed as the parent process. To find out the list of files opened by parent process Id lsof command is used with the option -R.

Syntax:

 # lsof -R

Files opened by a directory: It lists out the files which are opened by a particular directory. There are files as well as the directory in a system. So there can be several files opened by a directory as well as the regular file.

Syntax:

 # lsof +D directory path

Process management command - lsof

Files opened by network connections: Our Pc/system can be connected through various networks which helps in a variety of purpose. As we know that in Linux everything is a file, so we can even check the files that are opened by some network connections in the system.

Syntax:

 # lsof -i

File operation

Open function

Open and possibly create a file or device

```
#include <sys/types.h>
    #include <sys/stat.h>
    #include <fcntl.h>
    int open(const char *pathname, int flags);
    int open(const char *pathname, int flags, mode_t mode);
    int creat(const char *pathname, mode_t mode);
```

DESCRIPTION

Given a pathname for a file, open() returns a file descriptor, a small, nonnegative integer for use in subsequent system calls (read, write, lseek, fcntl, etc.).

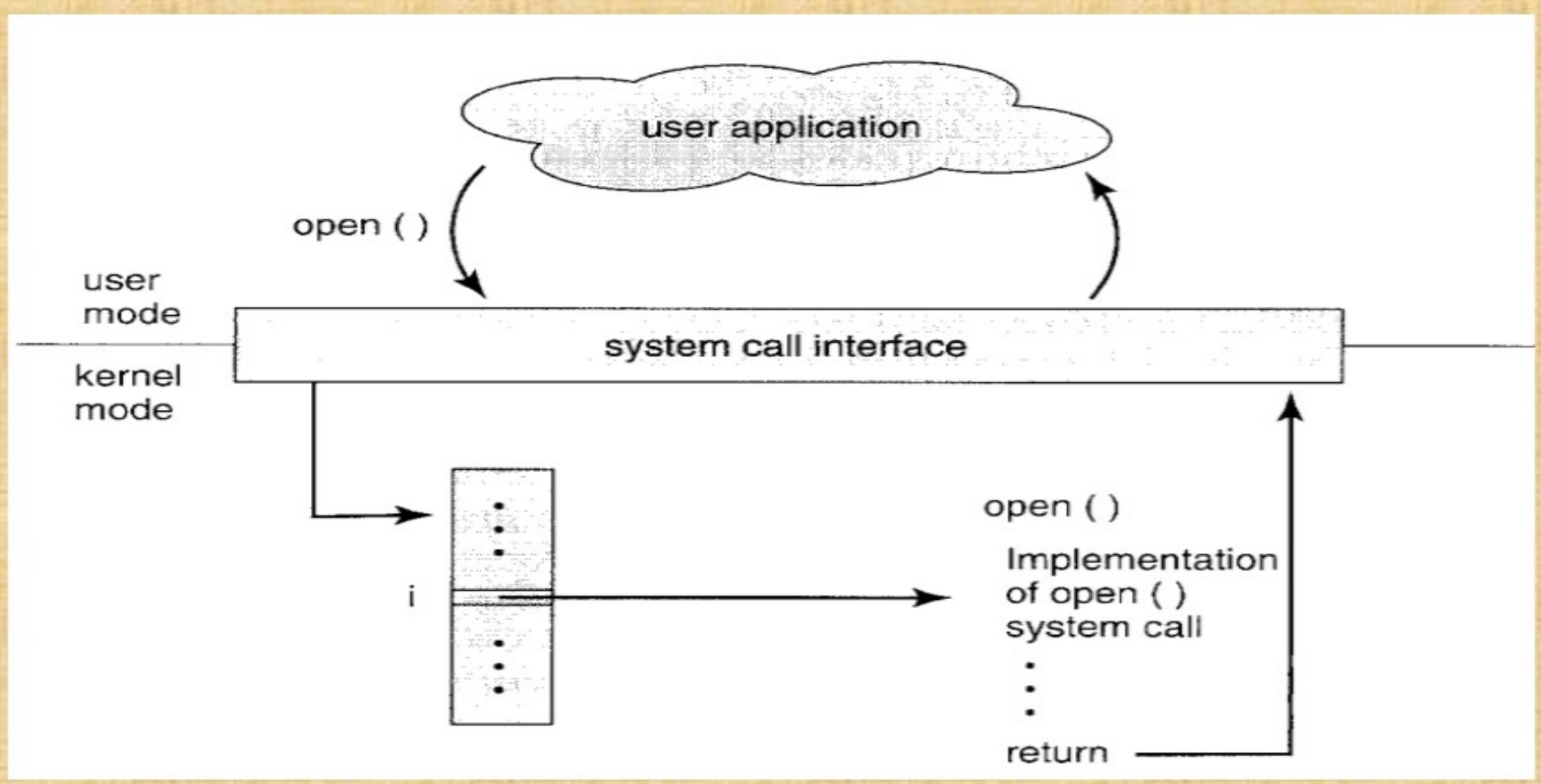
Flags : One of the access modes: O_RDONLY, O_WRONLY, or O_RDWR. These request opening the file read-only, write-only, or read/write, respectively.

RETURN VALUE

open() and creat() return the new file descriptor, or -1 if an error occurred (in which case, errno is set appropriately).

File operation

Handling open system call



File operation

Close function

Close a file descriptor

SYNOPSIS

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

DESCRIPTION

close() closes a file descriptor, so that it no longer refers to any file and may be reused.

RETURN VALUE

close() returns zero on success. On error, -1 is returned, and errno is set appropriately.

Read function

Read from a file descriptor

SYNOPSIS

#include <unistd.h>

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

DESCRIPTION

read() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf.

If count is zero, read() returns zero and has no other results. If count is greater than SSIZE_MAX, the result is

unspecified.

RETURN VALUE

On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. On error, -1 is returned, and errno is set appropriately.

File operation

Write function

Write to a file descriptor

SYNOPSIS

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

DESCRIPTION

write() writes up to count bytes from the buffer pointed buf to the file referred to by the file descriptor fd.

RETURN VALUE

On success, the number of bytes written is returned (zero indicates nothing was written). On error, -1 is returned, and errno is set appropriately.

File operation

lseek function

lseek - reposition read/write file offset

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

DESCRIPTION

The lseek() function repositions the offset of the open file associated with the file descriptor fd to the argument offset according to the directive whence as follows:

- SEEK_SET - The offset is set to offset bytes.
- SEEK_CUR - The offset is set to its current location plus offset bytes.
- SEEK_END – The offset is set to the size of the file plus offset bytes

RETURN VALUE

Upon successful completion, lseek() returns the resulting offset location as measured in bytes from the beginning of the file. Otherwise, a value of (off_t) -1 is returned and errno is set to indicate the error.

File operation

access

- Testing file permissions

SYNOPSIS

- Access (const char *pathname, permission_flags)
- Permission flags - R_OK, W_OK, X_OK for read, write and execute permissions.

DESCRIPTION

The access system call determines whether the calling process has access permission to a file.

- It can check any combination of read (r), write (w) and execute (x) permission.
- It can also check whether file exists or not.

RETURN VALUE

On success return value is 0, if process has all specified permissions,

If the file exists and process does not have the specified permissions, access returns -1 and errno is set to EACCESS.

File operation

dup and dup2 Functions:

- An existing file descriptor is duplicated by either of the following functions.

```
#include <unistd.h>
int dup(int filedes);
int dup2(int filedes, int filedes2);
```

Both return: new file descriptor if OK,-1 on error.

- The new file descriptor returned by dup is guaranteed to be the lowest-numbered available file descriptor.

File operation

- With dup2, we specify the value of the new descriptor with the filedes2 argument. If filedes2 is already open, it is first closed. If filedes equals filedes2, then dup2 returns filedes2 without closing it.
- dup(filedes); is equivalent to

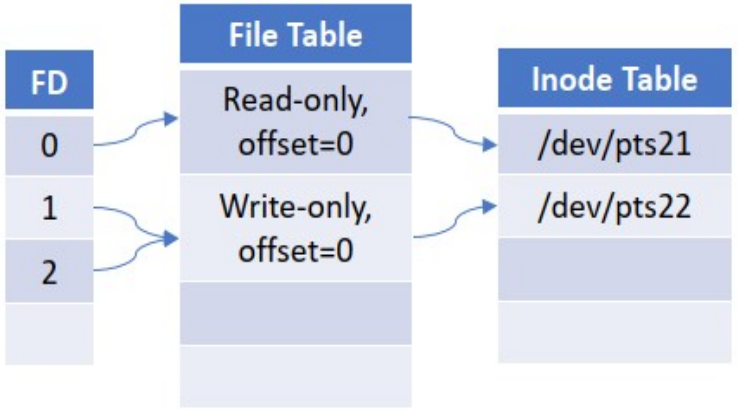
  fcntl(filedes, F_DUPFD, 0);
- dup2(filedes, filedes2); is equivalent to close(filedes2);

  fcntl(filedes, F_DUPFD, filedes2);

File operation

Example: Input Redirection

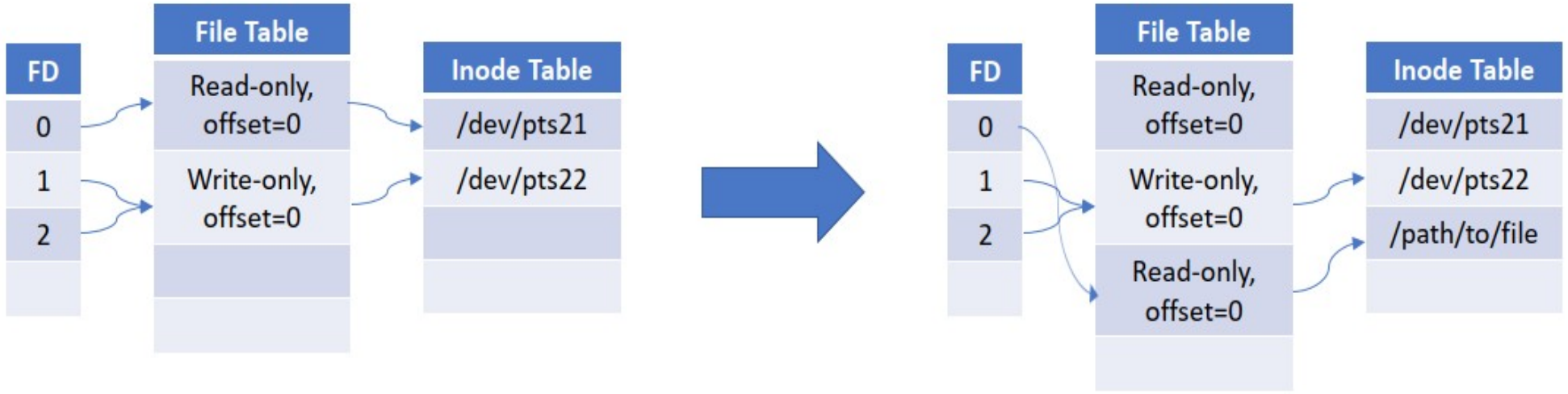
- Goal: instead of reading from stdin, read from another source



File operation

Example: Input Redirection

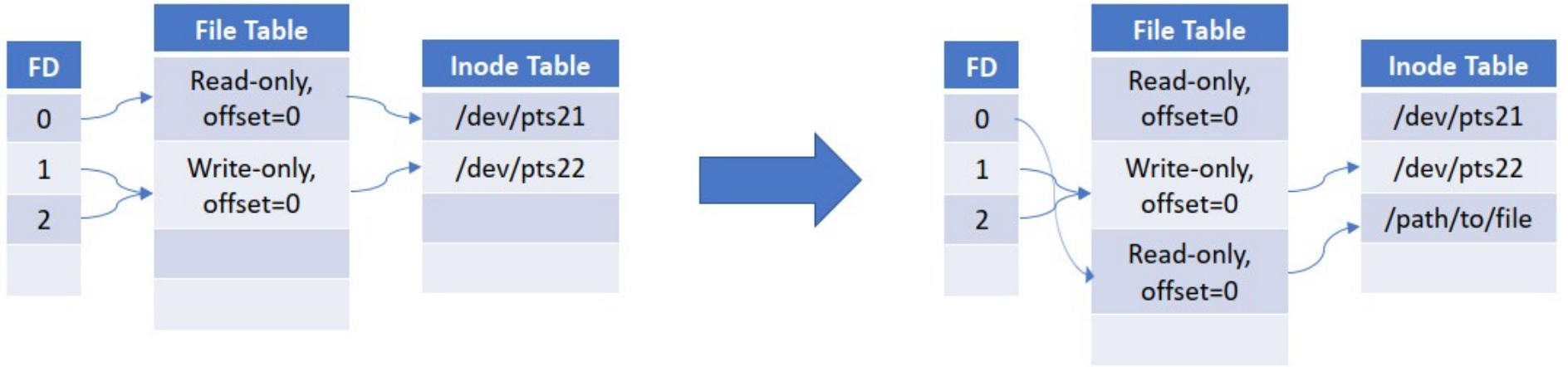
- Goal: instead of reading from stdin, read from another source



File operation

Example: Input Redirection

- Goal: instead of reading from stdin, read from another source

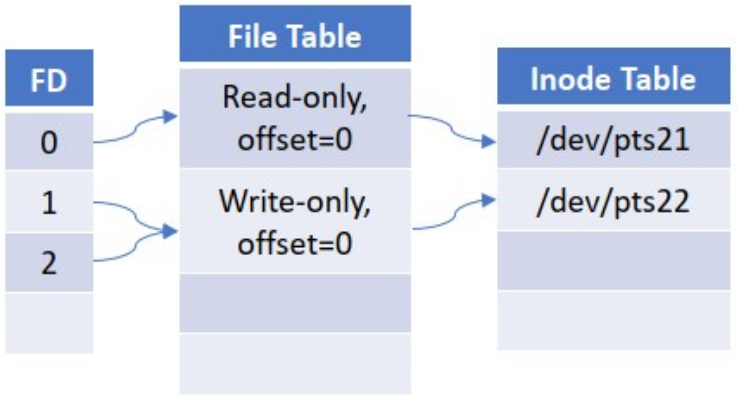



```
int ifd = open(newfile, O_RDONLY);
if (ifd >= 0) {
        close(0);
        dup(ifd);
        close(ifd);
}
```

File operation

Example: Input Redirection

- Goal: instead of reading from stdin, read from another source

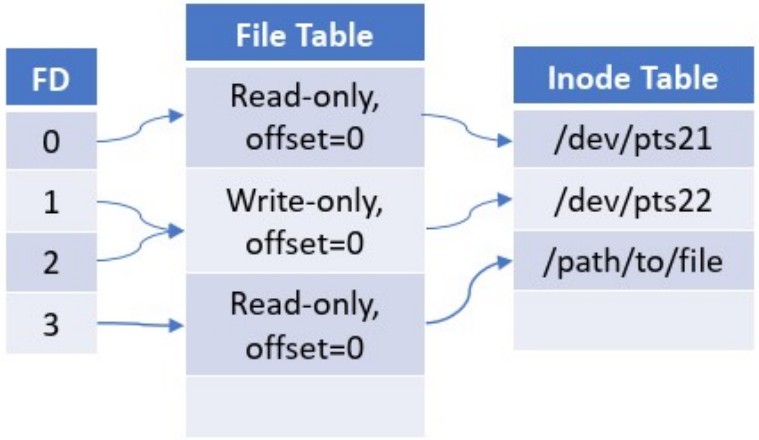



```
int ifd = open(newfile, O_RDONLY);
if (ifd >= 0) {
        close(0);
        dup(ifd);
        close(ifd);
}
```

File operation

Example: Input Redirection

- Goal: instead of reading from stdin, read from another source

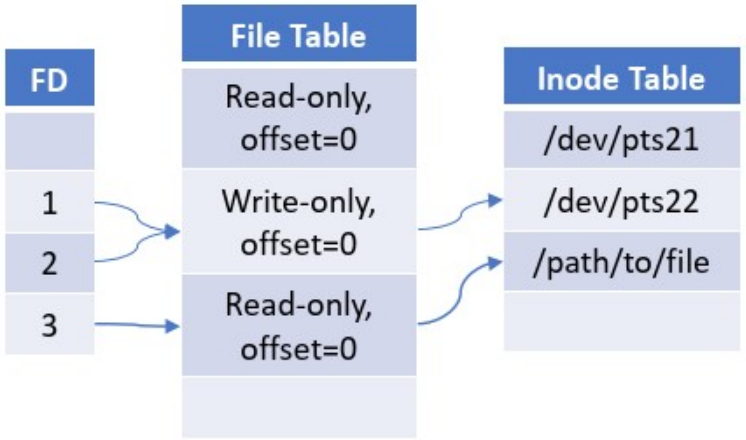



```
int ifd = open(newfile, O_RDONLY);
if (ifd >= 0) {
        close(0);
        dup(ifd);
        close(ifd);
}
```

File operation

Example: Input Redirection

- Goal: instead of reading from stdin, read from another source

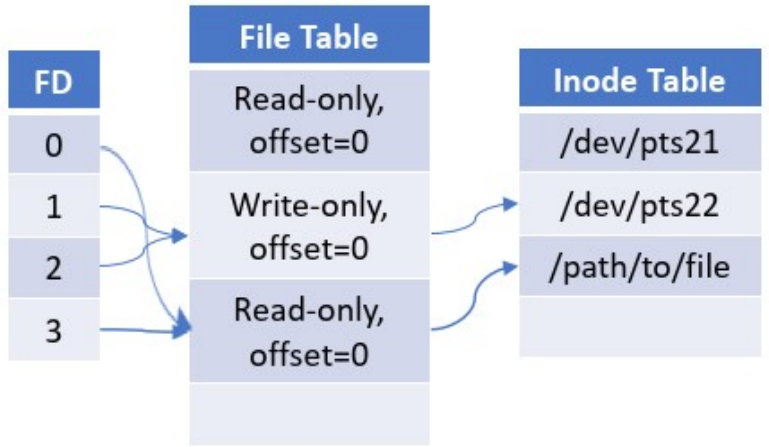



```
int ifd = open(newfile, O_RDONLY);
if (ifd >= 0) {
        close(0);
        dup(ifd);
        close(ifd);
}
```

File operation

Example: Input Redirection

- Goal: instead of reading from stdin, read from another source

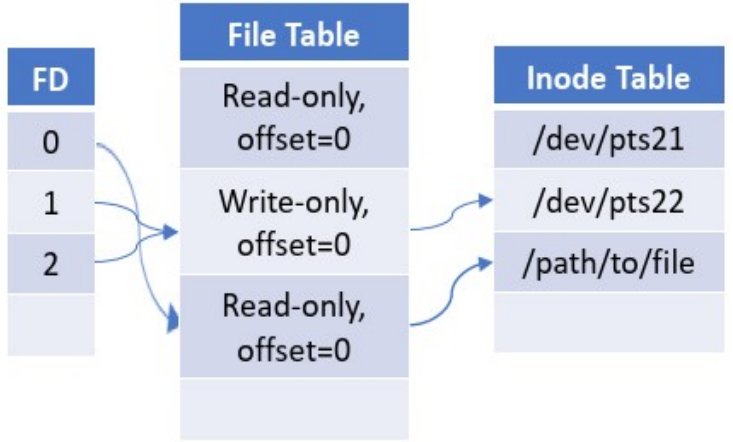



```
int ifd = open(newfile, O_RDONLY);
if (ifd >= 0) {
        close(0);
        dup(ifd);
        close(ifd);
}
```

File operation

Example: Input Redirection

- Goal: instead of reading from stdin, read from another source



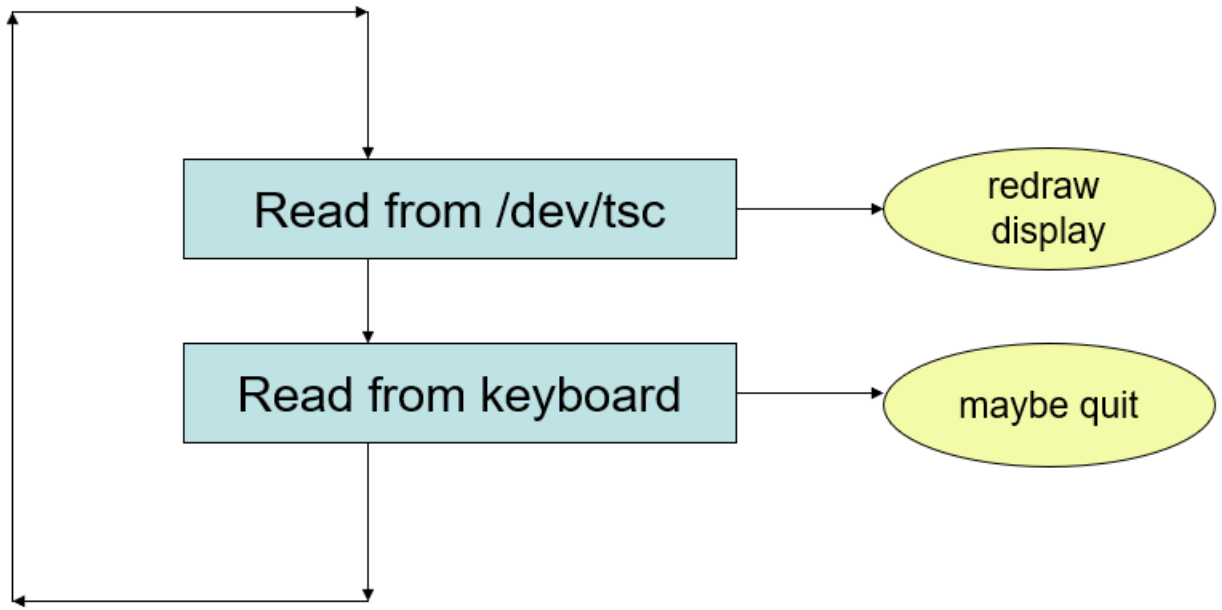

```
int ifd = open(newfile, O_RDONLY);
if (ifd >= 0) {
        close(0);
        dup(ifd);
        close(ifd);
}
```

File operation

MMAP Handson

File operation

Illustrates I/O Multiplexing



More than just one source for input

File operation

Remember ‘read()’ semantics

If device-driver’s ‘read()’ function is called before the device has any data available, then the calling task is expected to ‘sleep’ on a wait-queue until the device has at least one byte of data ready to return.

The keyboard’s device-driver behaves in exactly that expected way: it ‘blocks’ if we try to read when there are no keystrokes.

File operation

Blocking versus Multiplexing

If we want to read from multiple devices, we are not able to do it properly with the read() system-call

If one device has no data, our task will be blocked, and so can’t read data that may become available from some other device

File operation

Idea: use ‘nonblocking’ i/o

When we ‘open()’ a device-file, we could specify the ‘O_NONBLOCK’ i/o mode

File operation

```
char buf[BUFSIZ];
ssize_t nr;

start:
nr = read (fd, buf, BUFSIZ);
if (nr == -1) {
        if (errno == EINTR)
                goto start; /* oh shush */
        if (errno == EAGAIN)
                /* resubmit later */
        else
                /* error */
}
```

File operation

Linux provides two system calls for truncating the length of a file, both of which are defined and required (to varying degrees) by various POSIX standards. They are:

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

int ftruncate (int fd, off_t len);
```

and:

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

int truncate (const char *path, off_t len);
```

Both system calls truncate the given file to the length given by `len`. The `ftruncate( )` system call operates on the file descriptor given by `fd`, which must be open for writing. The `truncate( )` system call operates on the filename given by `path`, which must be writable. Both return `0` on success. On error, they return `-1`, and set `errno` as appropriate.

File operation

The `select( )` system call provides a mechanism for implementing synchronous multiplexing I/O:

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select (int n,
            fd_set *readfds,
            fd_set *writefds,
            fd_set *exceptfds,
            struct timeval *timeout);

FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

A call to `select( )` will block until the given file descriptors are ready to perform I/O, or until an optionally specified timeout has elapsed.

The `timeout` parameter is a pointer to a `timeval` structure, which is defined as follows:

```
#include <sys/time.h>

struct timeval {
        long tv_sec;          /* seconds */
        long tv_usec;         /* microseconds */
};
```

File operation

The `poll( )` system call is System V's multiplexed I/O solution. It solves several deficiencies in `select( )`, although `select( )` is still often used (again, most likely out of habit, or in the name of portability):

```
#include <sys/poll.h>

int poll (struct pollfd *fds, unsigned int nfds, int timeout);
```

Unlike `select( )`, with its inefficient three bitmask-based sets of file descriptors, `poll( )` employs a single array of `nfds pollfd` structures, pointed to by `fds`. The structure is defined as follows:

```
#include <sys/poll.h>

struct pollfd {
        int fd;           /* file descriptor */
        short events;     /* requested events to watch */
        short revents;    /* returned events witnessed */
};
```

Each `pollfd` structure specifies a single file descriptor to watch. Multiple structures may be passed, instructing `poll( )` to watch multiple file descriptors. The `events` field of each structure is a bitmask of events to watch for on that file descriptor. The user sets this field. The `revents` field is a bitmask of events that were witnessed on the file descriptor. The kernel sets this field on return. All of the events requested in the `events` field may be returned in the `revents` field. Valid events are as follows:

File operation

POLLIN

There is data to read.

POLLRDNORM

There is normal data to read.

POLLRDBAND

There is priority data to read.

POLLPRI

There is urgent data to read.

POLLOUT

Writing will not block.

File operation

POLLWRNORM

Writing normal data will not block.

POLLWRBAND

Writing priority data will not block.

POLLMSG

A SIGPOLL message is available.

In addition, the following events may be returned in the revents field:

POLLER

Error on the given file descriptor.

POLLHUP

Hung up event on the given file descriptor.

POLLNVAL

The given file descriptor is invalid.

File operation

File Operations

- File handle : FILE *
- Open a file : fopen
- Input/Output
  - Character IO : getc, putc
  - String IO : fgets, fputs
  - Formatted IO : fscanf, fprintf
  - Raw IO : fread, fwrite
- Close a file : fclose
- Other operations :
  - fflush, fseek, freopen

File operation

Files are opened for reading or writing via `fopen( )`:

```
#include <stdio.h>

FILE * fopen (const char *path, const char *mode);
```

This function opens the file `path` according to the given modes, and associates a new stream with it.

The `fclose( )` function closes a given stream:

```
#include <stdio.h>

int fclose (FILE *stream);
```

Any buffered and not-yet-written data is first flushed. On success, `fclose( )` returns `0`. On failure, it returns `EOF` and sets `errno` appropriately.

File operation

The function `fdopen( )` converts an already open file descriptor (`fd`) to a stream:

```
#include <stdio.h>

FILE * fdopen (int fd, const char *mode);
```

The possible modes are the same as for `fopen( )`, and must be compatible with the modes originally used to open the file descriptor. The modes `w` and `w+` may be specified, but they will not cause truncation. The stream is positioned at the file position associated with the file descriptor.

File operation

The function `fgets( )` reads a string from a given stream:

```
#include <stdio.h>

char * fgets (char *str, int size, FILE *stream);
```

This function reads up to *one less* than `size` bytes from `stream`, and stores the results in `str`. A null character (`\0`) is stored in the buffer after the bytes read in. Reading stops after an `EOF` or a newline character is reached. If a newline is read, the `\n` is stored in `str`.

For some applications, reading individual characters or lines is insufficient. Sometimes, developers want to read and write complex binary data, such as C structures. For this, the standard I/O library provides `fread( )`:

```
#include <stdio.h>

size_t fread (void *buf, size_t size, size_t nr, FILE *stream);
```

A call to `fread( )` will read up to `nr` elements of data, each of `size` bytes, from `stream` into the buffer pointed at by `buf`. The file pointer is advanced by the number of bytes read.

File operation

Individual characters and lines will not cut it when programs need to write complex data. To directly store binary data such as C variables, standard I/O provides `fwrite( )`:

```
#include <stdio.h>

size_t fwrite (void *buf,
               size_t size,
               size_t nr,
               FILE *stream);
```

A call to `fwrite( )` will write to `stream` up to `nr` elements, each `size` bytes in length, from the data pointed at by `buf`. The file pointer will be advanced by the total number of bytes written.

```
#include <stdio.h>

int fseek (FILE *stream, long offset, int whence);
```

If `whence` is set to `SEEK_SET`, the file position is set to `offset`. If `whence` is set to `SEEK_CUR`, the file position is set to the current position plus `offset`. If `whence` is set to `SEEK_END`, the file position is set to the end of the file plus `offset`.

Upon successful completion, `fseek( )` returns `0`, clears the `EOF` indicator, and undoes the effects (if any) of `ungetc( )`. On error, it returns `-1`, and `errno` is set appropriately. The most common errors are invalid stream (`EBADF`) and invalid `whence` argument (`EINVAL`).

File operation

Writing to a file using fprintf( )

fprintf( ) works just like printf and sprintf except that its first argument is a file pointer.

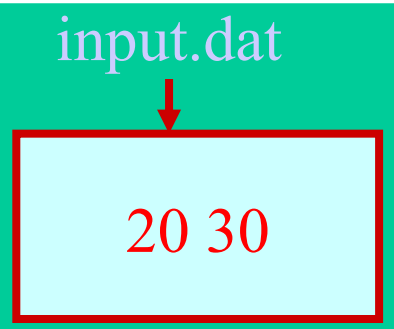
```
FILE *fptr;
fptr= fopen ("file.dat","w");
/* Check it's open */
fprintf (fptr,"Hello World!\n");
```

File operation

Reading Data Using fscanf( )

- We also read data from a file using fscanf( ).

```
FILE *fptr;
fptr= fopen ("input.dat","r");
/* Check it's open */
if (fptr==NULL)
  {
   printf("Error in opening file \n");
  }
fscanf(fptr,"%d%d",&x,&y);
```



x=20
y=30

File operation

Unlike `lseek( )`, `fseek( )` does not return the updated position. A separate interface is provided for this purpose. The `ftell( )` function returns the current stream position of `stream`:

```
#include <stdio.h>

long ftell (FILE *stream);
```

```
#include <stdio.h>

void rewind (FILE *stream);
```

This invocation:

```
rewind (stream);
```

resets the position back to the start of the stream. It is equivalent to:

```
fseek (stream, 0, SEEK_SET);
```

except that it also clears the error indicator.

File operation Handson

File operation

```
#include <unistd.h>

ssize_t pread (int fd, void *buf, size_t count, off_t pos);
```

This call reads up to `count` bytes into `buf` from the file descriptor `fd` at file position `pos`.

```
#include <unistd.h>

ssize_t pwrite (int fd, const void *buf, size_t count, off_t pos);
```

This call writes up to `count` bytes from `buf` to the file descriptor `fd` at file position `pos`.

These calls are almost identical in behavior to their non-`p` brethren, except that they completely ignore the current file position; instead of using the current position, they use the value provided by `pos`. Also, when done, they do not update the file position. In other words, any intermixed `read( )` and `write( )` calls could potentially corrupt the work done by the positional calls.

Both positional calls can be used only on seekable file descriptors. They provide semantics similar to preceding a `read( )` or `write( )` call with a call to `lseek( )`, with three differences. First, these calls are easier to use, especially when doing a tricky operation such as moving through a file backward or randomly. Second, they do not update the file pointer upon completion. Finally, and most importantly, they avoid any potential races that might occur when using `lseek( )`. As threads share file descriptors, it would be possible for a different thread in the same program to update the file position after the first thread's call to `lseek( )`, but before its read or write operation executed. Such race conditions can be avoided by using the `pread( )` and `pwrite( )` system calls.

File operation

```
ssize_t readv (int fd,
               const struct iovec *iov,
               int count);
```

The `writev( )` function writes at most `count` segments from the buffers described by `iov` into the file descriptor `fd`:

```
#include <sys/uio.h>

ssize_t writev (int fd,
                const struct iovec *iov,
                int count);
```

The `readv( )` and `writev( )` functions behave the same as `read( )` and `write( )`, respectively, except that multiple buffers are read from or written to.

Each `iovec` structure describes an independent disjoint buffer, which is called a *segment*:

File operation

- The main advantages offered by readv and writev are:
  - It allows working with non contiguous blocks of data i.e. buffers need ot be part of an array, but separately allocated.
  - The I/O is ‘atomic’ i.e if we do writev, all the elements in the vector will be written in one contiguous operation, and writes done by other processes will not occur in between them.

Timing results comparing `writev` and other techniques

	Linux (Intel x86)			Mac OS X (PowerPC)		
Operation	User	System	Clock	User	System	Clock
two `write`s	1.29	3.15	7.39	1.60	17.40	19.84
buffer copy, then one `write`	1.03	1.98	6.47	1.10	11.09	12.54
one `writev`	0.70	2.72	6.41	0.86	13.58	14.72

Thank you