commands for fcntl()

commands	Description
F_GETLK	Gets discretionary file locks
F_SETLK	Sets discretionary lock, no wait
F_SETLKW	Sets discretionary lock, wait if necessary
F_GETOWN	Retrieves the process id or process group no that will receive the SIGIO and SIGURG signal
F_SETOWN	Sets the process id or process group no.

The fsync() call :- The fsync() system call flushes all of the data written to file descriptor fd to disk or other underlying device.

```
# include <unistd.h>
int fsync(int fd);
# if def _POSIX_SYNCHRONIZED_IO
int fdatasync(int fd);
# endif
```

* The Linux file system may keep the data in memory for several seconds before writing it to disk in order to more efficiently handle IO.

* A zero is returned if successful; otherwise -1 will be returned and errno will be set.

The ftruncate() call :- The ftruncate() system call truncates the file referenced by file descriptor fd to the length specified by length.

```
int ftruncate(int fd, size_t length);
```

FD_SET(int fd, fd_set *set);

FD_Zero(fd_set *set);

* 1st parameter is the number of file descriptors in the file descriptor sets.
* 2nd, 3rd and 4th parameters are pointers to file descriptor sets that indicate which file descriptor you would like to be able to read write, or receive exception notification on, respectively.
* The last parameter is the time out value.
* All but the 1st parameter may be null.
* on return the file descriptor sets will be modified to indicate which descriptors are ready for immediate I/O operations.
* The time out will also be modified [to indicate which descriptors are ready for immediate I/O operations.] on return, although that is not the case on most system other than Linux.

The fstat() call ⇒ * The fstat() system call returns information about the file referred to by the file descriptor files, placing the result in the struct stat pointed to by buf().
* A return value of zero is success and -1 is failure

```
# include <sys/stat.h>
# include <unistd.h>
int fstat(int filedes, struct stat *buf);
```

The fchown() call ⇒ * The fchown() system call lets you change the owner and group associated with an open file.

```
# include <sys/types.h>
# include <unistd.h>
int fchown(int fd, uid_t owner, gid_t group);
```

* The 1st parameter is the file descriptor
* The 2nd the numerical user id
* The 3rd the numerical group id.

The fchmod() call ⇒ The fchmod() call changes the mode (permission bits) of the file referenced by fildes to mode

```
# include <sys/types.h>
# include <sys/stat.h>
int fchmod(int fildes, mode_t mode);
```

The fchdir() call :→ * changes the directory referred to by the open file descriptor fd.

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

The flock() call: * It request or removes an advisory lock on the file referred to by file descriptor fd

```
#include <sys/file.h>
int flock(int fd, int operation)
```

* The second parameter, operation, will be LOCK_SH for a shared lock, LOCK_EX for an exclusive lock, or LOCK_UN to unlock; the value LOCK_NB can be OR-ed with any of the other options to prevent blocking.

* Linux also implements mandatory locks, if they are enabled on your kernel.

The pipe() call :→ It creates a pipe and returns two file descriptor in the two integer arrays pointed to by filedes

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

* pipe() can be used in conjuction with fork(), dup2(), and execve() to create pipes to other programs with redirected input and/or output

Types of Files: * A variety of files are manipulated using file descriptor I/O or by file pointer I/O (stdio)

4 Regular files — All of the system calls, except for ioctl() and fchdir() (which applies to directories) apply to ordinary files.

Tape I/O :→ * Tape drives normally support sequential access to one or more unnamed files separated by end of file makers.

* some tape drives support variable block sizes but others use a fixed block size or block sizes that are a multiple of some size.
* Most, perhaps all, tape drives have some maximum block size.
* Nine track (open reel) tape drives support variable block sizes with a maximum block size of something like 16 bytes.
* Quarter inch cartridge drives require block sizes to be multiples of 512 bytes.
* DAT drives support variable length block sizes.
* If you want to preserve filename information, use an archive program such as tar, which will write a single file containing an archive of many files with filename, ownership, and permission information.
* Two files are written to tape for every data file; The 1st file contains a header and the second contain the data

** Linux tape drive interface, you simply open the tape drive (/dev/nst0) and perform read() and write calls.

* Each write() writes a single block to tape; you control the block size by simply controlling how many bytes you write at a time.
* to read variable length blocks, and learn what block size was read, simply issue a read() with a size large enough to hold the largest block expected.
* End of file markers are placed on tape by writing a block size of zero

The ioctl() call - * It is a catch all for setting or retrieving various parameters associated with a file or to perform other operation on the file.

* The arguments to ioctl(), vary depending on the underlying device

```
#include <sys/ioctl.h>
int ioctl(int d, int request, ...)
```

* The argument d must be an open file descriptor.

The fcntl() call - * similar to ioctl() but it sets or retrives a different set of parameters.

```
#include<unistd.h>
#include <fcntl.h>
int fcntl (int fd, int cmd);
int fcntl (int fd, int cmd, long arg);
```

* 1st argument is the file descriptor
* 2nd argument is the command
* 3rd is specific to the particular command

command for fcntl()

command	Description
F_DUPFD	Duplicates file descriptor. Use dup2()
F_GETFD	Gets close-on-exec flag. The file will remain open across exec() family calls if the low order bit is 0
F_SETFD	sets close-on-exec flag
F_GETFL	Gets the flags set by open
F_SETFL	changs the the flags set by open

I/O Routine

The close() Call :-
* To close a file descriptor
* The single argument is the file descriptor number returned by open().

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

* Any locks held by the process on the file are released, &
* If closing the file causes the link count to reach zero, the file will be deleted.

The read() call :
* The read() system call is used to read data from the file corresponding to a file descriptor

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

* The 1st argument is the file descriptor that was returned from a previous open()
* 2nd argument is a pointer to a buffer to copy the data from
* 3rd argument gives the number of bytes to read.
* Read() returns the no. of bytes read or a value of -1 if an error occurs.

The write() call :
* The write() is used to write data to the file corresponding to a file descriptor

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

Serial port I/O :- * using the serial ports is fairly easy.
* open the port O_RDWR and use ioctls or termios to set the communications parameters
* Serial ports have two instances:
** a call in device
** a call out device

printer port → * The parallel printer driver creates three character special device:
/dev/lp0, /dev/lp1, and /dev/lp2.

* To Talk to the printer directly, you simply open write data to the appropriate device.

sound card → * The sound driver creates a number of character special devices for the various subdevices on a typical soundcard.

* To play back digitized sound samples, you simply open /dev/audio (or /dev/dsp or /dsp1)

* use ioctl's() to set the sample rate, number of channels, number of bits per sample, and other parameters and then start writing your audio samples to the open file descriptor