System Programming

Week 3: File I/O

Seongjin Lee

Updated: 2016-09-21
02-fileio

insight@hanyang.ac.kr
http://esos.hanyang.ac.kr
Esos Lab. Hanyang University

Table of contents

1. File I/Os
1.1 Standard I/Os
1.2 I/O Efficiency

2. File Sharing
2.1 The Background
2.2 Atomic Operations
2.3 Descriptors
2.4 Manipulating Open Flags

3. Last Words

File I/Os

File Descriptors

A *file descriptor* (or *file handle*) is a small, non-negative integer which identifies a file to the kernel.

File Descriptors

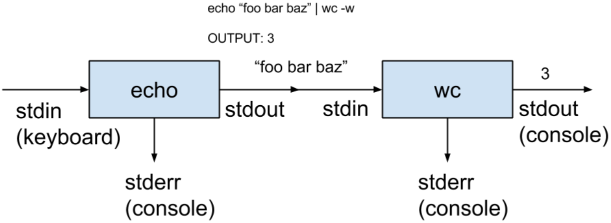
A *file descriptor* (or *file handle*) is a small, non-negative integer which identifies a file to the kernel.

Traditionally, `stdin`, `stdout` and `stderr` are 0, 1 and 2 respectively.

File Descriptors

A *file descriptor* (or *file handle*) is a small, non-negative integer which identifies a file to the kernel.

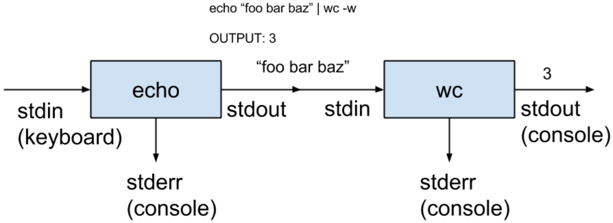
Traditionally, `stdin`, `stdout` and `stderr` are 0, 1 and 2 respectively.



File Descriptors

A *file descriptor* (or *file handle*) is a small, non-negative integer which identifies a file to the kernel.

Traditionally, `stdin`, `stdout` and `stderr` are 0, 1 and 2 respectively.



echo “foo bar baz” | wc -w

OUTPUT: 3

Relying on “magic numbers” is BAD. Use `STDIN_FILENO`, `STDOUT_FILENO` and `STDERR_FILENO` defined in <unistd.h> or `stdin`, `stdout`, and `stderr` defined in <stdio.h>.

How many files can you open? (./codes/fdcount.c)

```c
long open_max(void)
{
    if (openmax == 0) {
        errno = 0;
    /* first time through */
        if ((openmax = sysconf(_SC_OPEN_MAX)) < 0) {
            if (errno == 0)
                openmax = OPEN_MAX_GUESS; /* it is indeterminate */
            else
                fprintf(stderr, "sysconf error for _SC_OPEN_MAX");
        }
    }
    return(openmax);
}

int main(){
   printf("The number of file decriptors a process can open: %d\n", (int)RLIMIT_
        NOFILE);
   printf("The number of file decriptors an user can open: %ld\n", openmax );
   return 0;
}
```

How to compile

```
$ cd codes; cc -Wall -g -o fdcount fdcount.c
```

File I/Os : Standard I/Os

Basic File I/Os

There are five fundamental UNIX file I/O related functions:

- `open(2)`
- `close(2)`
- `lseek(2)`
- `read(2)`
- `write(2)`

open(2)

```
#include <fcntl.h>
int open(const char *path, int oflag, ... /* mode_t mode */ );
int openat(int fd, const char *path, int oflag, ... /* mode_t mode */ );
```

Both return: file descriptor if OK, 1 on error

The *path* parameter is the name of the file to open or create.

Options are specified by the *oflag*

Options are

oflag must be one (and only one) of:

○ `O_RDONLY` – Open for reading only

○ `O_WRONLY` – Open for writing only

○ `O_RDWR` – Open for reading and writing

and may be OR'd with any of these:

○ `O_APPEND` – Append to end of file for each write

○ `O_CREAT` – Create the file if it doesn't exist. Requires *mode* argument

○ `O_EXCL` – Generate error if `O_CREAT` and file already exists. (atomic)

○ `O_TRUNC` – If file exists and successfully open in `O_WRONLY` or `O_RDWR`, make length = 0

○ `O_NOCTTY` – If pathname refers to a terminal device, do not allocate the device as a controlling terminal

○ `O_NONBLOCK` – If pathname refers to a FIFO, block special, or char special, set nonblocking mode (open and I/O)

○ `O_SYNC` – Each write waits for physical I/O to complete

openat(2)

`openat(2)` function is equivalent to the open() function except in the case where the path specifies a relative path in an atomic fashion

- `O_EXEC` – Open for execute only
- `O_SEARCH` – Open for search only (applies to directories)
- `O_DIRECTORY` – If path resolves to a non-directory file, fail and set errno to `ENOTDIR`.
- `O_DSYNC` – Wait for physical I/O for data, except file attributes
- `O_RSYNC` – Block read operations on any pending writes.
- `O_PATH` – Obtain a file descriptor purely for fd-level operations. (Linux >2.6.36 only)

close(2)

```
#include <unistd.h>
int close(int fd); // Returns: 0 if OK, 1 on error
```

Closing a file releases any record locks that the process may have on the file.

When a process terminates, all of its open files are closed automatically by the kernel.

Many programs take advantage of this fact and don't explicitly close open files.

lseek example

```
cd ./codes ; make hole
```

```
James@maker:codes> make hole
cc -g -Wall -O0 -c hole.c
cc -g -Wall -O0 -o hole hole.o
James@maker:codes> ./hole
James@maker:codes> ls -l file.hole
-rw------- 1 James staff 16394 Sep 15 23:46 file.hole
James@maker:codes> od -c file.hole
0000000 a b c d e f g h i j \0 \0 \0 \0 \0 \0
0000020 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
*
0040000 A B C D E F G H I J
0040012
James@maker:codes>
```

Try making a file without a hole and compare the size of the two files.

creat(2)

```
#include <fcntl.h>
int creat(const char *path, mode_t mode); \\ Returns: file descriptor opened for
     write-only if OK, 1 on error
```

it is equivalent to `open(path, O_WRONLY | O_CREAT | O_TRUNC, mode);`

One deficiency with creat is that the file is opened only for writing.

If we were creating a temporary file that we wanted to write and then read back, we had to call creat, close, and then open. A better way is to use the open function, as in `open(path, O_RDWR | O_CREAT | O_TRUNC, mode);`

lseek(2)

```
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence); \\ Returns: new file offset if OK,
      1 on error
```

By default, this offset is initialized to 0 when a file is opened, unless the O_APPEND option is specified.

The interpretation of the offset depends on the value of the whence argument.

- SEEK_SET the file's offset is set to offset bytes from the beginning of the file.
- SEEK_CUR the file's offset is set to its current value plus the offset. The offset can be positive or negative.
- SEEK_END the file's offset is set to the size of the file plus the offset. The offset can be positive or negative.

read(2)

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t nbytes); \\ Returns: number of bytes read,
      0 if end of file, 1 on error
```

There are several cases where `read` returns less than the number of bytes requested:

- **regular file:** EOF is reached before the requested number of bytes has been read.
- **terminal device:** Normally, up to one line is read at a time.
- **network:** Buffering within the network may cause less than the requested amount to be returned.
- **a pipe or FIFO:** If the pipe contains fewer bytes than requested, read will return only what is available.
- **interrupted by a signal** and a partial amount of data has already been read.

write(2)

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t nbytes); \\ Returns: number of
     bytes written if OK, 1 on error
```

The return value is usually equal to the nbytes argument; otherwise, an error has occurred.

- A common cause for a write error is either filling up a disk or exceeding the file size limit for a given process.

For a regular file, `write` starts at the file's current offset.

If the `O_APPEND` option was specified when the file was opened, the file's offset is set to the current end of file before each write operation.

After a successful write, the file's offset is incremented by the number of bytes actually written.

File I/Os : I/O Efficiency

I/O Efficiency

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

#ifndef BUFFSIZE
#define BUFFSIZE 4096
#endif

int
main(void)
{
   int  n;
   char buf[BUFFSIZE];

   while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0){
      if (write(STDOUT_FILENO, buf, n) != n){
         fprintf(stderr, "write error\n");
         exit(1);
      }
   }

   if (n < 0){
      fprintf(stderr, "read error\n");
      exit(1);
   }

   return(0);
}
```

I/O Efficiency cnt'd

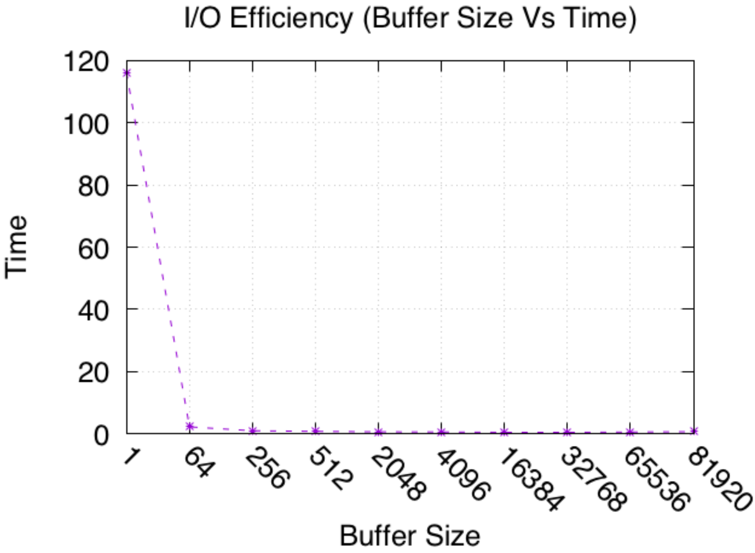
- It reads from standard input and writes to standard output, assuming that these have been set up by the shell before this program is executed. Indeed, all normal UNIX system shells provide a way to open a file for reading on standard input and to create (or rewrite) a file on standard output. This prevents the program from having to open the input and output files, and allows the user to take advantage of the shell's I/O redirection facilities.
- The program doesn't close the input file or output file. Instead, the program uses the feature of the UNIX kernel that closes all open file descriptors in a process when that process terminates.
- This example works for both text files and binary files, since there is no difference between the two to the UNIX kernel.

I/O Efficiency cnt'd

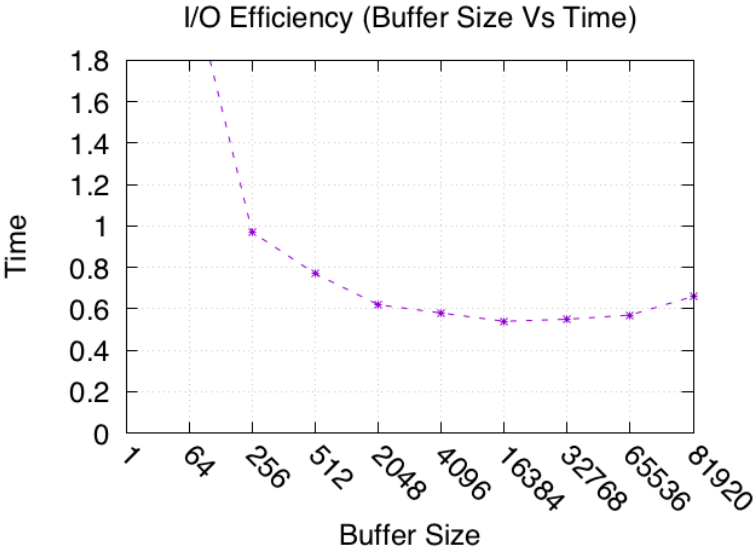
```
cd codes; make iotest
```

```
iotest: mycat.c testfiles
   for NUM in 1 64 256 512 2048 4096 16384 32768 65536 81920; do \
      echo "BUFFSIZE = $$NUM"; \
      cc -Wall -DBUFFSIZE=$$NUM -o $@ $< ; \
      i=$$(( $$i + 1 )) ; \
      time -p ./$@ < testfiles/temp_file > testfiles/file$$i.copy; \
      echo; \
   done;

testfiles:
   mkdir -p testfiles
   dd if=/dev/urandom of=testfiles/temp_file count=104800; \

ftest: setfl.c testfiles
   for FLAGS in a b c d; do \
      echo "FLAGS = $$FLAGS"; \
      cc -Wall -o $@ $< ; \
      time -p ./$@ -$$FLAGS < testfiles/temp_file > testfiles/file_$$FLAGS; \
      echo ; \
   done;
```

I/O Efficiency cnt'd



I/O Efficiency cnt'd



File Sharing

File Sharing : The Background

File Sharing

The UNIX System supports the sharing of open files among different processes.

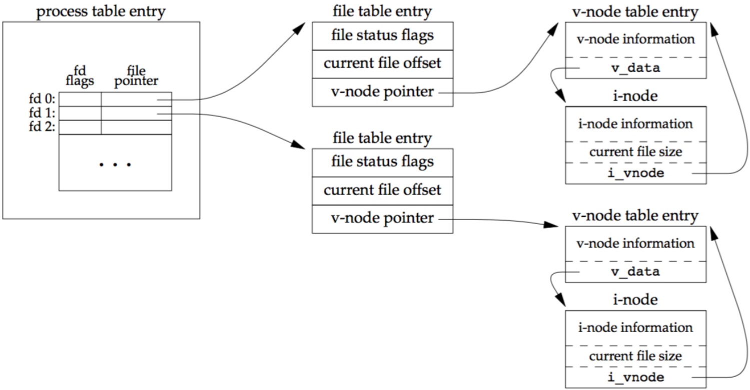
The kernel uses three data structures to represent an open file, and the relationships among them determine the effect one process has on another with regard to file sharing.

1. Process table
2. File table
3. v-node structure

Bit about the Data Structures

1. Each process has an entry (which is a table of file descriptor) in the process table.
   - The file descriptor flags (Fig. 3.7 in the text)
   - A pointer to a file table entry
2. The kernel maintains a file table for all open files, which consists of
   - file status flags (`O_APPEND, O_SYNC, O_RDONLY`, etc)
   - current file offset
   - pointer to a v-node table entry for the file
3. Each open file has a v-node structure that contains the type of file and pointers to functions that operate on the file
   - vnode information
   - inode information (such as current file size)

Arrangement of the Tables



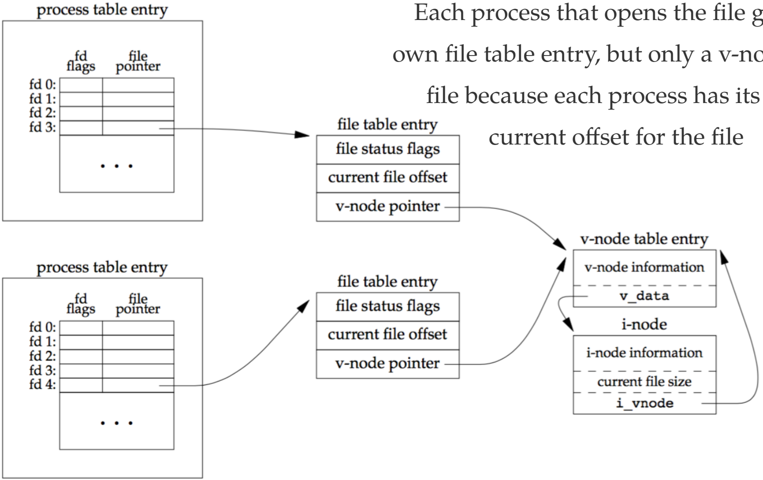
Kernel data structures for open files (Fig. 3.7)

on v-node

- The v-node was invented to provide support for multiple file system types on a computer.
- Linux uses a file system-independent i-node and file system-dependent i-node.

Example of File Open

Each process that opens the file gets its own file table entry, but only a v-node for a file because each process has its own current offset for the file



Two independent processes with the same file open (Stevens Fig. 3.8)

File Sharing

What happens with the operations that we've described earlier.

- After each `write` completes, the current file offset in the file table entry is incremented by the number of bytes written (if `current_file_offset` > `current_file_size` then `current_file_size` ← `current_file_offset` in i-node table entry)
- If a file is opened with `O_APPEND`, each `write` sets the `current_file_offset` = `current_file_size`
- `lseek` adjusts `current_file_offset` in file table entry
- `lseek` modifies `current_file_offset` int the file table entry. NO I/O.

File Sharing : Atomic Operations

Appending to a File: Older version

Unix that didn't support the `O_APPEND` option to open used following code to append

```
if (lseek(fd, 0L, 2) < 0){ /* position to EOF, 2: SEEK_END*/
    fprintf(stderr, "lseek error");
    exit(1);
}

if (write(fd, buf, 100) != 100){ /* and write */
    fprintf(stderr, "write error");
    exit(1);
}
```

It worked for single process

Read p.78 for more info

Atomic Append to a File: pread(2) and pwrite(2)

Two functions that allow applications to seek and perform I/O atomically: `pread` and `pwrite`

- `pread`: `lseek` + `read`
- `prwite`: `lseek` + `write`
- Exceptions are
  - There is no way to interrupt the two operations
  - The crruent file offset is not updated

```
#include <unistd.h>
ssize_t pread(int fd, void *buf, size_t nbytes, off_t offset);
// Returns: number of bytes read, 0 if end of file, 1 on error

ssize_t pwrite(int fd, const void *buf, size_t nbytes, off_t offset);
// Returns: number of bytes written if OK, 1 on error
```

File Sharing : Descriptors

dup and dup2 Functions

An existing file descriptor is duplicated by following functions

- dup : duplicate existing file descriptor
- dup2 : duplicate to a particular file descriptor value
- dup returns the lowest number unused file descriptor

```
#include <unistd.h>
int dup(int fd); // fd : old-descriptor
int dup2(int fd, int fd2); // fd2 : new descriptor

// Both return: new file descriptor if OK, 1 on error
```

The returned new file descriptor as the value of the functions shares the same file table entry as the fd argument

dup and dup2 Functions

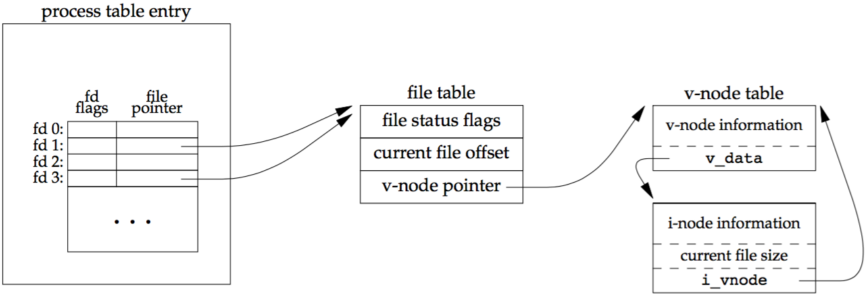


Fig 3.9 Kernel data structures after dup

sync, fsync, and fdatasync Functions

There is a page cache in the kernel through which most disk I/O passes.

- When writing a data to a file, it is copied to the buffer and queued for writing to disk at some later time
- data in buffer are written to the disk explicitly using following functions

sync, fsync, and fdatasync Functions

```
#include <unistd.h>
int fsync(int fd);
int fdatasync(int fd);
// Returns: 0 if OK, 1 on error
void sync(void);
```

- sync queues all the modified block buffers for writing and returns.
  - it does not wait for the disk writes to take place
  - it is called periodically (30 seconds) from a system daemon
- `fsync` refers only to a file, specified by the file descriptor `fd`
- `fdatasync` is similar to `fsync`, but affects only the data portions of a file

File Sharing : Manipulating Open Flags

fcntl Function

`fcntl` function can change the properties of a file that is already open

```
#include <fcntl.h>
int fcntl(int fd, int cmd, ... /* int arg */ );
// Returns: depends on cmd if OK (see following), 1 on error
```

The purposes

1. Duplicate an existing descriptor (*cmd* = `F_DUPFD` or `F_DUPFD_CLOEXEC`)
2. Get/set file descriptor flags (*cmd* = `F_GETFD` or `F_SETFD`)
3. Get/set file status flags (*cmd* = `F_GETFL` or `F_SETFL`)
4. Get/set asynchronous I/O ownership (*cmd* = `F_GETOWN` or `F_SETOWN`)
5. Get/set record locks (*cmd* = `F_GETLK`, `F_SETLK`, or `F_SETLKW`)

fcntl example

`cd codes; make setfl` or `make ftest`

```
    if ((flags = fcntl(STDOUT_FILENO, F_GETFL, 0)) < 0) {
      perror("Can’t get file descriptor flags");
      exit(1);
    }
      flags |= O_SYNC;
    if (fcntl(STDOUT_FILENO, F_SETFL, flags) < 0) {
      perror("Can’t set file descriptor flags");
      exit(1);
    }

    while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0 ){
      if ( write(STDOUT_FILENO, buf, n) != n ) {
        fprintf(stderr, "write error\n");
        exit(1);
      }
```

Use the Makefile and check the performance shown in Fig 3.14

```
ftest: setfl.c testfiles
  for FLAGS in a b c d; do \
    echo "FLAGS = $$FLAGS"; \
    cc -Wall -o $@ $< ; \
    time -p ./$@ -$$FLAGS < testfiles/temp_file > testfiles/file_$$FLAGS; \
    echo ; \
  done;
```

ioctl Function

The catchall for I/O operations.

```
#include <unistd.h> /* System V */
#include <sys/ioctl.h> /* BSD, OSX, and Linux */
int ioctl(int fd, int request, ...);
// Returns: 1 on error, something else if OK
```

It is designed to handle devices, such as terminal I/O, mag tape, etc, that can't be specified by previous calls.

/dev/fd

Newer systems provide a directory named `/dev/fd` whose entries are files named 0, 1, 2, and so on.
Opening the file `/dev/fd/`*n* is equivalent to duplicating descriptor *n*, assuming that descriptor *n* is open

```
> ls -l /dev/sdtin /dev/stdout /dev/stderr

> ls -l /dev/fd/

> echo first > file1
> echo third > file2
> echo second | paste file1 /dev/fd/0 file2
```

Last Words

Homework

- Reading

```
man { open | creat | lseek | read | pread | write |
    pwrite | close | dup | dup2 | fcntl | sync | fsync
    | fdatasync | ioctl }
```

  Chapter 4

- Solve
  - Question 3.4 and 3.5