System-Level I/O: Basics

COMP 321

Dave Johnson

1

Doing I/O Directly with Just Kernel Calls

The basis on which all I/O by all programs is built

- Example: C language standard provides the "Standard I/O" library
  - But it is just implemented on top of system-level I/O kernel calls
- I/O ***can't*** be done without going through these kernel calls
  - Due to user mode / kernel mode separation, only the kernel can do I/O

Why not just always use the more familiar Standard I/O user library calls?

- Doing the I/O kernel calls yourself gives you more control
- Understanding system-level I/O kernel calls is necessary in order to really understand higher-level user I/O libraries such as Standard I/O

2

A Simple Example of System-Level I/O

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

int main()
{
        int fd;
        ssize_t nch;
        char buf[4096];

        fd = open("message", O_RDONLY);
        nch = read(fd, buf, sizeof(buf));
        write(1, buf, nch);
        return 0;
}
```

Use, e.g., "man 2 open" to see what header files to #include for each kernel call

A Simple Example of System-Level I/O

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

int main()
{
        int fd;
        ssize_t nch;
        char buf[4096];

        fd = open("message", O_RDONLY);
        nch = read(fd, buf, sizeof(buf));
        write(1, buf, nch);
        return 0;
}
```

Returns an int file descriptor number

The name of the file to open

Should check that open() did not return an error, but omitted here for simplicity

Open the file for read-only mode

A Simple Example of System-Level I/O

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

int main()
{
        int fd;
        ssize_t nch;
        char buf[4096];

        fd = open("message", O_RDONLY);
        nch = read(fd, buf, sizeof(buf));
        write(1, buf, nch);
        return 0;
}
```

Returns the number of bytes actually read

Read from this file

Should check that read() did not return an error, but omitted here for simplicity

Maximum number of bytes to read

Into this buffer

5

A Simple Example of System-Level I/O

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

int main()
{
        int fd;
        ssize_t nch;
        char buf[4096];

        fd = open("message", O_RDONLY);
        nch = read(fd, buf, sizeof(buf));
        write(1, buf, nch);
        return 0;
}
```

Write to file descriptor number 1

Write this number of bytes

Should check that write() did not return an error, but omitted here for simplicity

Write from this buffer

6

What is This File Descriptor Number 1 Stuff?

By convention, file descriptors 0, 1, and 2 have defined existing uses

- 0 = standard input file
- 1 = standard output file
- 2 = standard error file
- Typically, ***these are all already open***, and to the same place = the terminal
- But these can be redirected, including on the command line by the shell, e.g.,
  - ./program < file # redirect file descriptor 0 instead from "file"
  - ./program > file # redirect file descriptor 1 instead to "file"
- Even after these redirections, errors (file descriptor 2) show up on the screen
- Should really use constants defined in <unistd.h> (but ***0***, ***1***, ***2*** are ***well known***)
  - STDIN_FILENO, STDOUT_FILENO, STDERR_FILENO

Opening a File

```
int  open(const char *pathname,  int flags);
```

Opens a file for the indicated (i.e., flags) type of accesses

- Set flags to one of
  - O_RDONLY: open the file for reading from only
  - O_WRONLY: open the file for writing to only
  - O_RDWR: open the file for reading from and/or writing to
- Returns the ***lowest numbered*** file descriptor number that is not currently open in this process to something else
- Returns -1 on any error and sets global variable errno to indicate which error

Opening a File

```
int  open(const char *pathname,  int flags);
```

The flags argument may also include other bits, such as

- O_APPEND: On each write() to this open fd, always move to the end of the file first (every write() only appends to the file)
- O_TRUNC: If the file exists (and if possible), truncate the file to empty on the open()

- Specify flags as the "or" of what you need, e.g.,
  - open(name, O_RDWR | O_APPEND)
  - open(name, O_WRONLY | O_TRUNC)

Creating a File

```
int  creat(const char *pathname,  mode_t mode);
int  open(const char *pathname,  int flags,  mode_t mode);
```

Creates a new, empty file, returning a file descriptor number

- On open(), if creating a new file (e.g., mode includes O_CREAT), then the mode argument gives the new file's protection (discussed later)
  - Otherwise, mode on open() is ignored or can be omitted
  - Calling creat() is equivalent to calling open() with flags = (O_WRONLY | O_CREAT | O_TRUNC)
- Returns the ***lowest numbered*** file descriptor number that is not currently open in this process to something else
- Returns -1 on any error and sets global variable errno to indicate which error

The Current File Offset in an Open File Descriptor

Each open file descriptor has an associated position (i.e., offset) within the file

- The model is ***something like*** accessing a file stored on magnetic tape, such as

Image from Wikimedia Commons

Image from obsoletemedia.org

The Current File Offset in an Open File Descriptor

Each open file descriptor has an associated position (i.e., offset) within the file

- The current file offset is initialized to 0 (i.e., the beginning of the file) when the file descriptor is opened (e.g., from open or creat)
- A ***single*** current fd offset is used jointly for both reading and writing this fd
  - Any ***read*** from this fd advances the fd's offset by the number of bytes actually ***read***
  - Any ***write*** to this fd advances ***this same*** offset by the number of bytes actually ***written***
- Example: repeated reading from the file sequentially transfers each next part of the file, until reaching the end of the file

Reading a File

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

Reads from the contents of a file into memory beginning at address buf

- The file must already be open; fd specifies which file to read from
- File data is transferred starting at fd current offset, sequentially into buf
- The maximum number of bytes to read is given by count
  - Tries to read requested count, returns the number of bytes actually read
  - Number read may be less than requested (a "***short count***")
  - This is ***not*** an error! We'll talk more about this later
- Returns 0 if fd was already positioned at the end of the file (no more to read)
- Returns -1 on any error and sets global variable errno to indicate which error

Writing a File

```
ssize_t  write(int fd,  const void *buf,  size_t count);
```

Writes into the contents of a file from memory beginning at address buf

- The file must already be open – fd specifies which file to write to
- File data is transferred sequentially from buf, starting at fd current position
- The maximum number of bytes to write is given by count
  - Tries to write requested count, returns the number of bytes actually written
  - Number written may be less than requested (a "***short count***")
  - This is ***not*** an error! We'll talk more about this later
- Returns -1 on any error and sets global variable errno to indicate which error

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

int main()
{
    char c;
    ssize_t nch;

    while ((nch = read(STDIN_FILENO, &c, 1)) == 1)
        write(STDOUT_FILENO, &c, 1);

    if (nch == 0)
        printf("Stopped at end of file.\n");
    else if (nch < 0)
        printf("Stopped on read error.\n");

    return 0;
}
```

Very inefficient since it does two kernel calls for each character being copied

Reads from standard input, 1 byte at a time

Writes to standard output, 1 byte at a time


```
#include <stdio.h>
#include <unistd.h>
#define SIZE    4096

int main()
{
    char buff[SIZE];
    ssize_t nch;

    while (1) {
        if ((nch = read(STDIN_FILENO, buff, SIZE)) <= 0) break;
        write(STDOUT_FILENO, buff, nch);
    }
    if (nch == 0)
        printf("Stopped at end of file.\n");
    else if (nch < 0)
        printf("Stopped on read error.\n");
    return 0;
}
```

Reads from standard input, up to SIZE bytes at a time

Writes to standard output, in the sizes that were read

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#define SIZE    4096

int main()
{
    char buff[SIZE];
    ssize_t nch;
    int in_fd, out_fd;

    in_fd = open("input", O_RDONLY);
    out_fd = open("output", O_WRONLY | O_CREAT | O_TRUNC, 0666);
    while (1) {
        if ((nch = read(in_fd, buff, SIZE)) <= 0) break;
        write(out_fd, buff, nch);
    }

    return 0;
}
```

Opens "input" to read from (opens read-only)

Opens "output" to write to (opens for writing) – if "output" doesn't exist, it is created, or otherwise it is truncated

17

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

int main()
{
    char c;
    ssize_t nch;
    int in_fd1, in_fd2, out_fd;

    in_fd1 = open("input", O_RDONLY);
    in_fd2 = open("input", O_WRONLY);
    out_fd = open("output", O_WRONLY | O_CREAT | O_TRUNC, 0666);

    while (1) {
        if ((nch = read(in_fd1, &c, 1)) != 1) break;
        write(out_fd, &c, 1);
        write(in_fd2, "X", 1);
    }
    return 0;
}
```

A silly thing to do with multiple file descriptors

For example purposes only!

in_fd1 and in_fd2 current positions are ***independent***

Copies from "input" to "output", 1 character at a time, but ***also*** overwrites each character in "input" with an "X"

18

Explicitly Moving a File Descriptor's File Offset

```
off_t  lseek(int fd,  off_t offset,  int whence);
```

Changes current offset for file descriptor fd to new position in the file's data

- The new fd offset is set to "offset" number of bytes relative to "whence"
  - SEEK_SET: fd offset is set to offset bytes (absolute)
  - SEEK_CUR: fd offset is set to current position plus offset bytes
  - SEEK_END: fd offset is set to the size (bytes) of the file plus offset bytes
- In general, "offset" may be negative (e.g., offset = -100, whence = SEEK_CUR)
- Returns
  - The fd's new file offset, on success
  - -1 on any error and sets global variable errno to indicate which error

Explicitly Moving a File Descriptor's File Offset

```
off_t  lseek(int fd,  off_t offset,  int whence);
```

Image from Wikimedia Commons

Examples

lseek(fd, 0, SEEK_SET)

- "Rewinds" the file

lseek(fd, 55, SEEK_CUR)

- Skips forward 55 bytes

lseek(fd, -10, SEEK_END)

- Moves to 10 bytes before the end of the file

```
int main()
{
   char c;
   ssize_t nch;
   int in_fd, out_fd;
   off_t offset;

   in_fd = open("input", O_RDONLY);
   out_fd = open("output", O_WRONLY | O_CREAT | O_TRUNC, 0666);

   offset = lseek(in_fd, -1, SEEK_END);
   while (1) {
      if ((nch = read(in_fd, &c, 1)) != 1) break;
      write(out_fd, &c, 1);
      if (offset == 0) break;
      offset = lseek(in_fd, -2, SEEK_CUR);
   }
   return 0;
}
```

A silly thing to do with lseek

For example purposes only!

Copies characters from "input" to "output" but reads them in reverse order!

If "input" is a copy of the program's source code, turns it into this in "output"

```
}
;0 nruter
}
;)RUC_KEES ,2- ,df_ni(keesl = tesffo
;kaerb )0 == tesffo( fi
;)1 ,c& ,df_tuo(etirw
;kaerb )1 =! ))1 ,c& ,df_ni(daer = hcn(( fi
{ )1( elihw
;)DNE_KEES ,1- ,df_ni(keesl = tesffo
;)6660 ,CNURT_O | TAERC_O | YLNORW_O ,"tuptuo"(nepo = df_tuo
;)YLNODR_O ,"tupni"(nepo = df_ni
;tesffo t_ffo
;df_tuo ,df_ni tni
;hcn t_eziss
;c rahc
{
)(niam tni
```

Closing a File Descriptor

```
int  close(int fd);
```

Closes an existing open file descriptor

- Doesn't matter whether opened, e.g., from open() or creat() or otherwise
- Returns 0 on success
- Returns -1 on any error, such as if this file descriptor is ***not*** currently open
- Any later accesses to closed file descriptor will return -1 with errno = EBADF
- But this file descriptor number is now available for the next open(), etc.
  - That will always return the ***lowest numbered*** file descriptor number that is not currently open in this process to something else
  - Warning: this reuse of that file descriptor number can create/hide bugs if you didn't mean to close that file descriptor

Why Open a File Before Reading/Writing It?

Much more efficient than finding the file by pathname on every access

- Example: pathname "/usr/share/man/man2/open.2.gz" – (from "man 2 open")
  - Must search in "/" directory for "usr", then search there for "share", then search there for "man", then search there for "man2", and finally search there for "open.2.gz"
  - More efficient to do just once on the open, not on every read, write, etc.

Allows the kernel to easily remember this open file's current file offset

- Start remembering on open, keep remembering only while fd is open

More efficient to check file protections only at file open

- Remember verified O_RDONLY, O_WRONLY, or O_RDWR just like fd offset

Unix File Types

- ***Regular file***: The most common type of file, may contain anything, "regular"
  - All the examples we have looked at here have used only regular files
- ***Directory file***: A file that gives names to files and gives the files' locations
  - Generally the second most common type of file
- ***Block special file***: A file that represents a "block"-oriented device
- ***Character special file***: A file that represents a "character"-oriented device
- ***FIFO***: Used for communication between processes (a "pipe" or named "pipe")
- ***Socket***: Used for network communication (e.g., over the Internet)
- ***Symbolic link***: A file that gives an "alternate" name for some other file

Getting Information About Files

```
int  stat(const char *restrict pathname,  struct stat *restrict statbuf);
int  fstat(int fd,  struct stat *statbuf);
```

Return metadata about a file into the "struct stat" pointed to by statbuf

- ***Metadata*** is data about data, here the information about the file
- Two equivalent interfaces
  - ***stat***: the file is indicated by char *pathname (without opening the file)
  - ***fstat***: the file is indicated by an already open file descriptor fd
- On Linux, the struct stat format is defined in "man 3type stat"
  (on other systems, generally in "man 2 stat")

Contents of a "struct stat"

```
struct stat {
        dev_t     st_dev;     /* ID of device on which the file resides */
        ino_t     st_ino;     /* inode number representing the file */
        mode_t    st_mode;    /* file protection and file type */
        nlink_t   st_nlink;   /* number of hard links to the file (to the file's inode) */
        uid_t     st_uid;     /* user ID of file owner */
        gid_t     st_gid;     /* group ID of file owner */
        dev_t     st_rdev;    /* device type (if the inode represents a special file) */
        off_t     st_size;    /* total size, in bytes */
        blksize_t          st_blksize;   /* "preferred" block size for filesystem I/O */
        blkcnt_t           st_blocks;    /* number of 512-byte blocks allocated */
        struct timespec    st_atim;      /* time of last access to file */
        struct timespec    st_mtim;      /* time of last modification to file data */
        struct timespec    st_ctim;      /* time of last status change (last change to inode) */
};
```



```
int main()
{
    struct stat statbuf;

    fstat(STDIN_FILENO, &statbuf);
    printf("Type of file on standard input: ");
    switch (statbuf.st_mode & S_IFMT) {
        case S_IFREG:   printf("regular file\n");          break;
        case S_IFDIR:   printf("directory file\n");        break;
        case S_IFBLK:   printf("block device\n");          break;
        case S_IFCHR:   printf("character device\n");      break;
        case S_IFIFO:   printf("FIFO\n");                  break;
        case S_IFSOCK:  printf("socket\n");                break;
        case S_IFLNK:   printf("symbolic link\n");         break;
        default:        printf("unknown?\n");              break;
    }

    return 0;
}
```

open vs. Open, read vs. Read, write vs. Write, etc.

You should <u>always</u> check for error returns from <u>all</u> kernel calls, <u>always</u>

- (My examples didn't fully do this, just due to space and time considerations)
- All kernel calls return -1 on any error
  - and set global variable errno to tell you the problem (see "man 3 errno")
- The textbook provides csapp "wrapper" functions to do error checking for you
- For example, Read vs. read:
  - But these always exit(0) the program (unix_error always does exit(0))
  - And they print an "unfriendly" (and unhelpful?) message

```
ssize_t Read(int fd, void *buf, size_t count)
{
        ssize_t rc;
        if ((rc = read(fd, buf, count)) < 0)
                unix_error("Read error");
        return rc;
}
```