Lecture # 2 – File I/O (Chapters 3)

- This chapter deals with unbuffered I/O (i.e. each read / write call invokes a system call as opposed to standard I/O (i.e. printf, scanf, gets, etc.).

- Unbuffered I/O boils down to 5 function calls – open, read, write, lseek, and close

- To the UNIX kernel, all open files are referred to by file descriptors. A File descriptor is non-negative integer.

- By convention, the following file descriptors are provided by the shell:

stdin	0
stdout	1
stderr	2

- open function

int open(const char *pathname, int oflag**, …** /*, mode_t mode */);

Note: returns file descriptor if OK, -1 if error

pathname is the name of the file to be opened

oflag is determined by ORing together several arguments

Mandatory:

O_RDONLY	Read only
O_WRONLY	Write only
O_RDWR	Read and write

Optional:

O_APPEND	Append to file for each write
O_CREAT	Create file if it does not exist (mode is required)
O_EXCL	Generate error if O_CREAT and file exists
O_TRUNC	If file exists, then truncate it to zero bytes

Example:

```
if ((fd = open(“/tmp/file1.txt”, O_RDONLY)) < 0)
{
        perror(“open /tmp/file1.txt”);
        exit(1);
}
```

Example:

```
if ((fd = open("/tmp/file2.txt", O_WRONLY | O_TRUNC | O_CREAT)) < 0)
{
        perror("open /tmp/file2.txt");
        exit(1);
}
```

- creat function

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

Note: returns file descriptor if OK, -1 if error
Note: This is same as open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);

- close function

int close(int filedes);

returns 0 if OK, -1 if error

Note: all open file descriptors are closed on process exit

- lseek function

off_t lseek(int filedes, off_t offset, int whence)

Returns new file offset if OK, -1 if error

whence:

SEEK_SET	file offset is set to offset
SEEK_CUR	file offset is set to current + offset (offset can be negative)
SEEK_END	file offset is set to end of file + offset

Example:

```
off_t        currpos;

currpos = lseek(fd, 0, SEEK_CUR);
```

Example 3.2 (file with a hole):

```
char    buf1[] = "abcdefghij";
char    buf2[] = "ABCDEFGHI";
```

```
int main(void)
{
        int     fd;

        if ((fd = creat("file.hole", FILE_MODE)) < 0)
        {
                perror("creat failed");
                exit(1);
        }

        if (write(fd, buf1, 10) != 10)
        {
                perror("buf1 write failed");
                exit(1);
        }

        if (lseek(fd, 40, SEEK_SET) == -1)
        {
                perror("lseek failed");
                exit(1);
        }

        if (write(fd, buf2, 10) != 10)
        {
                perror("buf2 write error");
                exit(1);
        }

        exit(0);
}
```

- read function

```
ssize_t         read(int filedes, void *buff, size_t nbytes);
```

Returns: number of bytes read
0 if end of file
-1 if error

- write function

```
ssize_t         write(int filedes, const void *buff, size_t nbytes);
```

Returns: number of bytes written
-1 if error

- I/O efficiency

Example:

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

int main(int argc, char **argv)
{
    int     n;
    int     bufsize;
    char    *buf;

    if (argc != 2)
    {
        fprintf(stderr, "usage: %s <bufsize>\n", argv[0]);
        exit(1);
    }

    bufsize = atoi(argv[1]);

    if (bufsize <= 0 || bufsize >= 131072)
    {
        fprintf(stderr, "%s: bufsize out of range 1 .. 131071\n", argv[0]);
        exit(1);
    }

    buf = (char *)malloc(bufsize);

    while ((n = read(0, buf, bufsize)) > 0)
        if (write(1, buf, n) != n)
            perror("write failed");

    if (n < 0)
        perror("read error");

    free(buf);
    exit(0);
}
```

How does buffer size effect program efficiency? (see p. 70)

- File sharing

UNIX supports the sharing of open files between different processes.

Data structures:

1. Each process has an entry in the **process table**. Each of these entries contains a list of open file descriptors. Associated with each file descriptor are file descriptor flags, and a pointer to a file table entry.

2. There is a **file table** for all open files. Each file table entry contains the file status flags, current file offset, and a pointer to the v-node table entry for the file.

3. Each open file (device) has a **v-node** structure. This entry contains information about the type of file, and pointers to functions that operate on the file. For most files, the v-node also contains the inode for the file. The inode contains info like owner, group, permissions, file size, and pointers to actual data blocks.

Note: See picture of data structures on page 72.

Each file opened by a process has an entry in the process table (via the file descriptors vector), and a file table entry.

All processes with a given file open share the same vnode structure (see page 73).

Question: Why do processes generally not share the file table entry?

- Atomic operations

File append:

```
if (lseek(fd, 0L, SEEK_END) < 0)
        perror("lseek failed");

if (write(fd, buff, 100) != 100)
        perror("write failed");
```

Question: What is the problem with this code in a multiuser environment?

Note: O_APPEND flag was created to resolve this issue.

File creation:

```
        if ((fd = open(pathname, O_WRONLY)) < 0)
                if (errno == ENOENT)
                {
                        if ((fd = creat(pathname, mode)) < 0)
                                perror("create error");
                }
                else
                        perror("open error");
```

Question: What is the problem with this code in a multiuser environment?

Note: O_CREAT and O_EXCL flags were created to resolve this issue.

- Dup and dup2 functions

```
int dup(int filedes);
int dup2(int filedes, int filedes2);
```

Returns new file descriptor if OK, -1 on error

Dup:

The dup function returns the lowest numbered available file descriptor as a copy of the file descriptor provided by the argument. **Note**: after dup call, we have two entries in process table vector which point to the same file table entry.

Dup2:

The dup2 function returns the filedes2 as a copy of filedes. If filedes2 is open, it is first closed. **Note**: dup2 is an atomic operation.
Example:

```
        dup2(filedes, filedes2);

                Is equivalent to:

        close(filedes2);
        dup(filedes);
```

Example: Perform I/O redirection to stdout

```
        if ((fd = open("/tmp/file.txt", O_WRONLY | O_CREAT | O_EXCL)) < 0)
        {
                perror("open failed");
                exit(1);
        }

        if (dup2(fd, 1) < 0)
        {
                perror("dup2 failed");
                exit(1);
        }

        close(fd);
```

- Fcntl function

int fcntl(int filedes, int cmd, … /* int arg */);

Returns depend on cmd, -1 on error

The fcntl function is used to change the properties of a file that is already open.

There are five different uses:

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

Note: See description of each function on p. 79, and code on p. 80

- Ioctl function

The ioctl function is the catchall for I/O operations. Originally, terminal I/O was one of the biggest users of ioctl.

int ioctl(int filedes, int request, …);