

CS810D - Advanced Programming in the UNIX Environment

Department of Computer Science
Stevens Institute of Technology
Jan Schaumann

`jschauma@cs.stevens.edu`

`http://www.cs.stevens.edu/~jschauma/810-APUE/`

Standard I/O

Basic File I/O: almost all UNIX file I/O can be performed using these five functions:

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

Processes may want to share resources. This requires us to look at:

- atomicity of these operations
- file sharing
- manipulation of file descriptors

creat(2)

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

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

Returns: file descriptor if OK, -1 on error

creat(2)

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

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

Returns: file descriptor if OK, -1 on error

This interface is made obsolete by open(2).

open(2)

```
#include <fcntl.h>

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

Returns: file descriptor if OK, -1 on error

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

close(2)

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

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

Returns: 0 if OK, -1 on error

- closing a filedescriptor releases any record locks on that file (more on that in future lectures)
- file descriptors not explicitly closed are closed by the kernel when the process terminates.

lseek(2)

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

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

Returns: new file offset if OK, -1 on error

The value of *whence* determines how offset is used:

- SEEK_SET bytes from the beginning of the file
- SEEK_CUR bytes from the current file position
- SEEK_END bytes from the end of the file

“Weird” things you can do using `lseek(2)`:

- seek to a negative offset
- seek 0 bytes from the current position
- seek past the end of the file

read(2)

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

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

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

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

- EOF reached before requested number of bytes have been read
- Reading from a terminal device, one "line" read at a time
- Reading from a network, buffering can cause delays in arrival of data
- Record-oriented devices (magtape) may return data one record at a time

`read` begins reading at the current offset, and increments the offset by the number of bytes actually read.

write(2)

```
#include <unistd.h>

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

Returns: number of bytes written if OK, -1 on error

- `write` returns `nbytes` or an error has occurred (disk full, file size limit exceeded).
- For regular files, `write` begins writing at the current offset (unless `O_APPEND` has been specified, in which case the offset is first set to the end of the file.)
- After the write, the offset is adjusted by the number of bytes actually written.

I/O Efficiency

Caveats with the program “simple-cat.c” from the last class:

- assumes that *stdin* and *stdout* have been set up appropriately

I/O Efficiency

Caveats with the program “simple-cat.c” from the last class:

- assumes that *stdin* and *stdout* have been set up appropriately
- relies on the fact that when a process terminates, the kernel will close all open files

I/O Efficiency

Caveats with the program “simple-cat.c” from the last class:

- assumes that *stdin* and *stdout* have been set up appropriately
- relies on the fact that when a process terminates, the kernel will close all open files
- works for “text” and “binary” files since there is no such distinction in the UNIX kernel

I/O Efficiency

Caveats with the program “simple-cat.c” from the last class:

- assumes that *stdin* and *stdout* have been set up appropriately
- relies on the fact that when a process terminates, the kernel will close all open files
- works for “text” and “binary” files since there is no such distinction in the UNIX kernel
- how do we know the optimal `BUFSIZE`?

File Sharing

Since UNIX is a multi-user/multi-tasking system, it is conceivable (and useful) if more than one process can act on a single file simultaneously. In order to understand how this is accomplished, we need to examine some kernel data structures which relate to files. (Compare Stevens, pp 70 ff)

File Sharing

Since UNIX is a multi-user/multi-tasking system, it is conceivable (and useful) if more than one process can act on a single file simultaneously. In order to understand how this is accomplished, we need to examine some kernel data structures which relate to files. (Compare Stevens, pp 70 ff)

- each process table entry has a table of file descriptors, which in turn contain
 - the file descriptor flags
 - a pointer to a file table entry

File Sharing

Since UNIX is a multi-user/multi-tasking system, it is conceivable (and useful) if more than one process can act on a single file simultaneously. In order to understand how this is accomplished, we need to examine some kernel data structures which relate to files. (Compare Stevens, pp 70 ff)

- each process table entry has a table of file descriptors, which in turn contain
 - the file descriptor flags
 - a pointer to a file table entry
- the kernel maintains a file table; each entry contains
 - file status flags
 - current offset
 - pointer to a vnode table entry

File Sharing

Since UNIX is a multi-user/multi-tasking system, it is conceivable (and useful) if more than one process can act on a single file simultaneously. In order to understand how this is accomplished, we need to examine some kernel data structures which relate to files. (Compare Stevens, pp 70 ff)

- each process table entry has a table of file descriptors, which in turn contain
 - the file descriptor flags
 - a pointer to a file table entry
- the kernel maintains a file table; each entry contains
 - file status flags
 - current offset
 - pointer to a vnode table entry
- a vnode structure contains
 - vnode information
 - inode information (such as current file size)

File Sharing

Knowing this, here's what happens with each of the calls we discussed earlier:

- after each `write` completes, the current file offset in the file table entry is incremented. (If `current_file_offset > current_file_size`, change current file size in i-node table entry.)
- If file was opened `O_APPEND` set corresponding flag in file status flags in file table. For each `write`, current file offset is first set to current file size from the i-node entry.
- `lseek` simply adjusts current file offset in file table entry
- to `lseek` to the end of a file, just copy current file size into current file offset.

So far so good - reading a file simultaneously is no problem, then. How about writing?

Atomic Operations

An operation is composed of a sequence of steps. An operation is atomic if either all of the steps are performed or none of the steps are performed. Suppose UNIX didn't have `O_APPEND` (early versions didn't). To append, you'd have to do this:

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

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

At any point between the `lseek` and `write` calls, the kernel could suspend this process and switch to another. What if this other process were doing the same thing to the same file? Therefore we need an atomic operation to accomplish this. The `O_APPEND` flag to open provides us with it for this case.

dup(2) and dup2(2)

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

```
int dup(int oldd);
```

```
int dup2(int oldd, int newd);
```

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

An existing file descriptor can be duplicated with `dup(2)` or duplicated to a particular file descriptor value with `dup2(2)`. As with `open(2)`, `dup(2)` returns the lowest numbered unused file descriptor.

Note the difference in scope of the file *descriptor* flags and the file *status* flags.

fcntl(2)

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

int fcntl(int filedes, int cmd, ... /* int arg */);
```

Returns: depend on *cmd* if OK, -1 on error

`fcntl(2)` is one of those "catch-all" functions with a myriad of purposes. Here, they all relate to changing properties of an already open file. It can:

cmd	effect	return value
F_DUPFD	duplicate <i>filedes</i> (FD_CLOEXEC file descriptor flag is cleared)	new <i>filedes</i>
F_GETFD	get the file descriptor flags for <i>filedes</i>	descriptor flags
F_SETFD	set the file descriptor flags to the value of the third argument	not -1
F_GETFL	get the file status flags	status flags
F_SETFL	set the file status flags	not -1

...as well as several other functions.

ioctl(2)

```
#include <unistd.h> /* SVR4 */
#include <sys/ioctl.h> /* 4.3+BSD */

int ioctl(int filedes, int request, ...);
```

Returns: -1 on error, something else if OK

Another catch-all function, this one is designed to handle device specifics that can't be specified via any of the previous function calls. For example, terminal I/O, magtape access, socket I/O, etc.

Homework

- Reading:
 - manual pages for the functions covered
 - Stevens Chap. 3
- Thinking:
 - Stevens # 3.5 (bourne shell syntax “> &”)
- Coding:
 - required: `tcp(1)` (see website)
 - extra credit: `tcpm(1)` (see website)