CSC 357 Lecture Notes Week 4

Unbuffered File I/O
UNIX Files and Directories

I. Relevant reading:

A. Stevens chapters 3 and 4.

B. Skim chapter 2.

II. C and UNIX standards (Stevens Ch 2)

A. Two levels of standards.

B. ISO C standard defines language proper, and C standard library.

C and UNIX standards, cont’d

1. Appendix A of K&R is the reference manual for the language proper.

2. Appendix B of K&R is a summary of the major library components.

3. The ISO (International Standards Organization) maintains the official standard.

C and UNIX standards, cont'd

C. IEEE POSIX defines the full library standard.

1. The standard is based on UNIX, but any operating system may meet the standard.

2. Systems that do are all *POSIX compliant*.

3. POSIX includes the ISO standard C library, but not the specification of the language proper.

C and UNIX standards, cont'd

D. POSIX is a specification of library functions, not an implementation.

   1. Many implementations of UNIX.

   2. IEEE has official POSIX certification program.

C and UNIX standards, cont’d

3. Four implementations of UNIX in Stevens:
   a. Solaris
   b. Linux
   c. Mac OS X
   d. FreeBSD

III. UNIX unbuffered file I/O (Stevens Ch 3).

A. Five functions -- `open`, `read`, `write`, `lseek`, and `close`.

B. Operate on file descriptors, at UNIX kernel level.

C. Lower-level than the "f" series, like `fopen`.

Unbuffered I/O, cont'd

1. These lower-level functions are referred to as *unbuffered*.

2. The OS does perform buffering on `FILE*` streams, but not with files accessed through lower level file descriptors.

3. Sec 5.4 of Stevens talks about buffering details.

IV. File descriptors (Stevens Sec 3.2).

A. At the kernel level, all files are referred to by a *file descriptor*, which is a non-negative integer.

B. The `open` function returns a file descriptor.

C. Functions like `read` and `write` take file descriptors as inputs.

V. `open` (Stevens Sec 3.3).

A. Open a file, returning file descriptor, or -1 if error.

B. Signature:

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

open, cont'd

1. *pathname* is name of file to open or create
2. *oflag* is used to specify options
3. the optional *mode* is only applicable when a new file is being created

open, cont'd

C. Options values are constructed by a bitwise-inclusive-OR of flags.

1. Exactly one of the following:

`O_RDONLY`	Open for reading only.
`O_WRONLY`	Open for writing only.
`O_RDWR`	Open for reading and writing.

open, cont'd

2. Any combination of the following may be used:

O_APPEND	Append to end
O_CREAT	Create the file
O_EXCL	Fail O_CREAT if file exists
O_TRUNC	Truncate length to 0

O_NOCTTY	Do not have a terminal
O_NONBLOCK	Do not block on open

open, cont'd

3. POSIX synchronization options are:

O_DSYNC	Wait for write to complete, no attrs
O_RSYNC	Have reads wait for pending writes
O_SYNC	Wait for write to complete, yes attrs

open, cont’d

4. There are other platform-specific options for such things as symbolic links, locks, and 64-bit file offsets.

open, cont'd

D. Example:

```
open("data", O_RDWR | O_APPEND)
```

VI. `creat` (Stevens Sec 3.4).

A. Create a file.

B. Equivalent to following `open`:

```
open(pathname,
     O_WRONLY | O_CREAT | O_TRUNC,
     mode)
```

VII. `close` (Stevens Sec 3.5).

A. Close an open file, returning 0 if OK, -1 if error.

B. Signature:

`int close(int` *filedes*`);`

C. When a process terminates, all open files are closed by the kernel.

VIII. **lseek** (Stevens Sec 3.6).

A. The `lseek` function sets the read/write offset of an open file, returning new offset if OK, -1 if error.

1. All open files have an offset position that defines from what byte a read starts or to what byte a write starts.

2. The offset is initialized to 0 by `open`, unless `O_APPEND` is specified.

B. Signature:

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

C. Interpretation of *offset* based value of *whence*:

- SEEK_SET, set *offset* from beginning of file
- SEEK_CUR, set to current value plus *offset*; *offset* value can be positive or negative
- SEEK_END, set to size of file plus *offset*

lseek, cont'd

D. Programmer can determine the value of the current offset without changing, e.g.,

```
off_t curpos;
curpos = lseek(fd, 0, SEEK_CUR);
```

1. Used to determine if file is capable of seeking.

2. See example on Page 64 of Stevens.

lseek, cont'd

E. When `lseek` is used to set a file's offset larger than its current size, file has"a hole" in it.

1. OS may take advantage of this by allocating fewer file blocks.

2. Unwritten bytes read back as 0s.

3. See example on pp. 65-66 of Stevens.

lseek, cont'd

F. Type `off_t` allows OS to provide different size integers for file offsets, and hence max size file.

lseek, cont'd

1. Most platforms support both 32-bit and 64-bit file offsets, the latter being > 2 GB (2^{31}-1).

2. Here are defs of `off_t` on hornet:

lseek, cont'd

```
#if defined(_LP64) || _FILE_OFFSET_BITS == 32
typedef long off_t;
#else
typedef __longlong_t off_t;
#endif
```

IX. **`read`** (Stevens Sec 3.7).

A. Read from an open file, returning number of bytes read, 0 if eof, -1 if error

B. Signature:

```
ssize_t read(int fildes,
             void *buf,
             size_t nbytes);
```

read, cont'd

1. `ssize_t` return value is number of bytes read, 0 on eof

2. *fildes* is file to read from

3. *buf* is buffer of at least *nbytes*

read, cont'd

C. There are several cases in which the number of bytes read is less than requested, including:

1. If eof is reached during the read, the number of bytes read may be less than requested.

2. When reading from a terminal device, normally only one line at a time is read.

read, cont’d

3. When reading from a network, buffering may cause fewer bytes than requested to be read.

4. When reading from a pipe, only the number of available bytes is read.

read, cont'd

5. When reading from a record-oriented device, sometimes only a record at a time is read.

6. When the read is interrupted by a signal, the read may only be partially completed.

read, cont’d

D. The read operation starts at the current file offset.

E. After successful read, file offset is incremented by number of bytes actually read.

F. Typedefs `ssize_t` and `size_t` allow flexibility in number of bytes readable and requestable.

X. **write** (Stevens Sec 3.8).

A. Write data to an open file, returning number of bytes written if OK, -1 if error.

B. Signature:

```
ssize_t write(int fildes,
              const void *buf,
              size_t nbytes);
```

write, cont'd

C. Write starts at current file offset of the given *filedes*, unless `O_APPEND` set on open.

D. After successful write, offset incremented by number of bytes actually written.

E. Typical causes for write failure are full disk or exceeding the file size limit for a process.

XI. I/O Efficiency (Stevens Sec 3.9).

A. This section has some interesting data on the effect of programmer-selected buffer size on execution time of `read` and `write`.

B. We'll discuss further in an upcoming lecture.

XII. File sharing (Stevens Section 3.10).

A. Two or more processes[1] can share the same file.

B. They have common pointer to same file data.

---

[1] As defined in Chapter 1 of Stevens, a *process* is an independently executing program.

File sharing, cont'd

C. The processes have independent copies of:

1. the file descriptor and its flags
2. file status flags
3. current file offset

File sharing, cont'd

D. Pictures on pp. 72 and 73 illustrate well.

E. If processes only read file, no problems.

F. If they each try to write, they can interfere with each other.

G. A classic "readers/writers" situation.

XIII. Atomic operations (Stevens Section 3.11).

A. Problem with operation sequence `lseek` followed immediately by `write`.

1. Process can seek, but be suspended before write.

2. If during suspension another process does seek and write, unexpected results can occur.

Atomic operations, cont'd

B. Suppose processes A and B have a shared file.

1. Process A seeks to end, then is suspended.

2. Process B then seeks to end, writes 100 bytes.

3. Process A gets reactivated to do its write, but it's now 100 bytes in front of the end.

Atomic operations, cont'd

C. To address this problem, there are functions `pwrite` and `pread`.

D. Signatures:

```
ssize_t pwrite(
        int fildes,
        const void *buf,
        size_t nbytes,
        off_t offset);
```

Atomic operations, cont'd

```
ssize_t pread(
     int fildes,
     void *buf,
     size_t nbytes,
     off_t offset);
```

Atomic operations, cont’d

E. Also potential problem with creating file.

1. Process A checks if a file exists, with intent not to create if it does.

2. Process A is suspended, B gets control.

3. Process B creates file that A just checked.

Atomic operations, cont'd

4. Process A gets control back, thinks file does not exist, and proceeds to re-create it.

5. Problem if B wrote to file before A got control back, then A re-creates with truncation.

Atomic operations, cont'd

F. Term *atomic operation* refers to operation composed of multiple uninterruptible steps.

1. Subset of steps cannot be performed.
2. All steps run to completion, or none runs.

XIV. `dup` and `dup2` (Stevens Section 3.12).

A. File descriptors can be duplicated.

B. The only difference between `dup`'d descriptors is *file descriptor flags*.

C. Share same *status* flags, current offset, file data.

D. We'll discuss the relevance later.

XV. `fsync`

A. UNIX kernels typically use buffer caches to make read/write operations more efficient.

B. Contents of cache memory and file may differ.

C. For applications that care, `fsync` function forces synchronization of cache and associated file.

XVI. `fcntl` (Stevens Section 3.14).

A. Provides for control of open files.

B. Signature:

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

fcntl, cont'd

1. *cmd* is `#define`d in `<fcntl.h>`.

2. Optional *arg* varies based on value of *cmd*.

C. Myriad different *cmd*s and *arg*s.

fcntl, cont'd

D. Many settable when file is opened, but

1. `fcntl` allows file props to be changed without close and reopen;

2. for stdio and pipes, `fcntl` is only way to set file props, when an appl'n did not itself open.

XVII. `ioctl` (Stevens Section 3.15).

A. Provides control of file descriptors associated with devices.

B. Signature:

```
int iocntl(
        int fildes,
        int request,
        ... );
```

ioctl, cont'd

1. *request* and optional third arg interpreted by device driver

2. interpretation performed in device-specific way

XVIII. `/dev/fd`

A. UNIX has uniform treatment of files and devices.

1. There's a standard dir named "`/dev`".

2. We'll see more about `/dev` in coming lectures.

/dev/fd, cont'd

B. At level of file descriptors, many UNIX systems provide a `/dev/fd` subdirectory

1. By convention, file descriptors `0`, `1`, `2` correspond to `stdin`, `stdout`, `stderr`.

2. Enforces uniformity of files and devices.

/dev/fd, cont'd

C. Association of `stdio` with numeric file descriptors is not POSIX.

1. POSIX requires the def of `STDIO_FILENO`, `STDOUT_FILENO`, `STDERR_FILENO`.

2. Despite this, many UNIX apps rely on hard numeric mapping.

XIX. Files and directories (Stevens Chapter 4).

A. Fundamental part of any OS.

B. UNIX treats files and directories pretty uniformly.

C. Also treats files and devices uniformly.

D. Also provides the *symbolic link* file type.

Files and directories, cont'd

E. At system call level, there are `stat` functions.

F. Also other useful system functions that operate on files and directories.

XX. `stat`, `lstat`, `fstat` (Stevens Section 4.2).

A. Functions return file info in a `struct stat`, defined in `<sys/stat.h>`.

B. Signatures:

stat, lstat, fstat, cont'd

```
int stat(
        const char* restrict[2] pathname,
        struct stat* restrict buf);
```

[2] `restrict` is keyword added to 1999 ISO C

stat, lstat, fstat, cont'd

```
int lstat(
        const char* restrict pathname,
        struct stat* restrict buf);

int fstat(
        int fildes,
        struct stat* buf);
```

stat, lstat, fstat, cont'd

1. Returned data in *buf* parameter, which must point to caller-declared structure.

2. For `fstat`, *filedes* is fd of open file.

3. Return val is 0 if OK, -1 if error.

stat, lstat, fstat, cont'd

C. Diff between `stat` and `lstat` is `lstat` returns info about sym link file, not file ref'd by link;

i.e., `stat` follows the symbolic link pointer, `lstat` does not.

stat, lstat, fstat, cont'd

D. Here's def of `struct stat` on falcon/hornet:

```
struct stat {
    dev_t           st_dev;
    ino_t           st_ino;
    mode_t          st_mode;
    nlink_t         st_nlink;
    uid_t           st_uid;
    gid_t           st_gid;
```

struct stat, cont'd

```
    dev_t          st_rdev;
    off_t          st_size;
    timestruc_t st_atim;
    timestruc_t st_mtim;
    timestruc_t st_ctim;
    blksize_t   st_blksize;
    blkcnt_t    st_blocks;
    char           st_fstype
                        [_ST_FSTYPSZ];
};
```

struct stat, cont'd

1. Struct fields declared as sys-defined datatypes, from `<sys/types.h>` and elsewhere.

2. Use of `struct stat` will figure prominently in programming assignment 3.

XXI. File types (Stevens Section 4.3).

A. Most common are regular data files and dirs.

B. UNIX defines seven different files types:

1. *Regular file*, which holds data; kernel does not distinguish between text and binary.

2. *Directory file*, which contains names of other files and pointers to file info.

Files and dirs, cont'd

3. *Block special file*, which provides buffered I/O access to devices such as disk drives.

4. *Character special file*, which provides unbuffered I/O access to devices.

Files and dirs, cont'd

5. *FIFO*, for communication between processes, also called *named pipe*.

6. *Socket*, for inter-process communication accross network

7. *Symbolic link*, points to another file; akin to *short cut* in Windows.

Files and dirs, cont'd

C. Page 90 of Stevens has a useful code example.

1. Program that prints file-type of each command-line arg.

2. Uses `lstat` to obtain file info.

Files and dirs, cont'd

D. Later on pages 121-125, another code example that uses `lstat` to traverse dir hierarchy.