

# COMP421

## Unix Environment for Programmers

### Lecture 11: File I/O

---

Jeff Wiegley, Ph.D.  
Computer Science

jeffw@csun.edu

09/12/2005

“We should have some ways of connecting programs by  
garden hose--screw in another segment when it becomes  
when it becomes necessary to message data in another  
way. This is the way of IO also.”

*—M. D. McIlroy, October 11, 1964*

# Basic File I/O

---

Operations on files are kept to a minimum:

- `open(...)` used to open or create files
- `read(...)` used to obtain data from a file
- `write(...)` used to place data into a file
- `close(...)` used to close a file
- `lseek(...)` used to rapidly move to a specific position within a file.
- `dup(...)`, `dup2(...)` used to duplicate file descriptors.
- `fcntl(...)` used to access/modify information about a file descriptor.
- `ioctl(...)` used to specify unusual file “operations” such as “eject tray” on CD-ROM “files”.

because nearly everything is treated as files it is important that these be well designed, well defined, efficient and non-changing.

# Opening files

---

`int open(pathname, flags, [mode])` is used to open or create a file for reading (or writing if `flags` includes proper bits).

flag bits include:

- `O_RDONLY`, `O_WRONLY`, `O_RDWR` specifies read/write capability.
- `O_APPEND` opens and performs an `lseek()` to the end of the file.
- `O_CREAT` creates the file if it doesn't already exist.
- `O_EXCL` causes `open` to fail if file already exists.
- `O_TRUNC` delete any data that may already be in the file.

example to open a file for appending data to the file:

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

int fd = open("testfile", O_RDWR|O_CREAT|O_APPEND);
```

## Closing files:

---

The Unix kernel buffers I/O operations in memory to increase performance. (Disks, tapes and punch cards were very, very slow).

For this reason as well as other accounting or procedural reasons it is necessary to “close” file descriptors when you are done using them.

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

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

closure of file descriptors is very important.

- Any pending/buffered data is flushed to the device.
- Any procedures necessary to deal with the hardware are performed.
- Every process has a limit on the number of open file descriptors it may have open at any one time.

web servers and such have to make sure they close what they aren't using or they won't be able to handle new requests.

## Reading data:

---

All reading is done with:

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

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

This is true of files, sockets and pipes and every other type of file. Other interprocess communication functions will be implemented on top of this.

Some things to be aware of:

- `buf` is simply an array of characters to be filled in by `read`.
- `count` is the length of the array. **It is not the number of characters to be read!**
- The integer returned by `read()` is the number of bytes that were actually read.
- If no data is ready then `read()` will block until data is ready.
- `read()` returns as many bytes as it can up to `count`.

## Managing read():

The amount of data `read()` will return is limited by either:

1. The argument `count`, or
2. The amount of data buffered by the kernel.<sup>a</sup>

If you want to limit the amount of data the `read()` returns pass it a smaller value of `count` (`count==1` will cause single bytes to be read).

If there are less bytes buffered than `count` then `read()` will only return what is available.

If you need to read an exact amount of data then you must nest the `read()` within a while loop and count how many bytes are left to go.

```
int need=572,gobbled;
while (need>0 && gobbled=read(fd,buf,9000)>0) need-=gobbled;
```

---

<sup>a</sup>the buffer is a finite size. If data isn't pulled out of the buffer then eventually the `write()` filling the buffer will block until data is extracted to make room.

## Writing to files:

---

`write()` is similar:

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

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

- `count` is the maximum number of bytes to write to the file descriptor.
- `write()` will attempt to write `count` number of bytes but is not guaranteed to write this many.
- Nesting in a loop is again needed to guarantee that a specific number of bytes are written.
- `write()` may block for a variety of reasons such as the target file descriptor's kernel buffer being full.

## Seeking to a particular position:

---

Some files such as database files have a very specific structure and grow to very large sizes.

To make operations on such files (and for hardware devices) the `lseek()` function allows for repositioning of the file pointer very rapidly.

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

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

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

The file pointer associated with the file descriptor is positioned to a specific location.

The position obtained is controlled by `whence`:

- `SEEK_SET` offset to `offset` bytes from the beginning of the file.
- `SEEK_CUR` offset to `offset` bytes from current position.
- `SEEK_END` offset to `offset` bytes past end of file.

## Duplicating descriptors:

---

Files can be shared simultaneously by multiple processes.

To facilitate sharing the functions `dup()` and `dup2()` exist to duplicate file descriptors.

```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

After duplication the two file descriptors are 100% interchangeable. performing an `lseek()` on one will be reflected in the other as well.<sup>a</sup>

Two very useful purposes of `dup()`

1. Setup interprocess communication prior to a `fork()`.
2. Setup file descriptors in preparation for “daemonizing” processes.

---

<sup>a</sup>`fork()` also “copies” file descriptors. But this done by copying a process table entry and the resulting file descriptors are independent resources of different processes.

## fcntl():

---

The kernel maintains a list of *v-nodes* for every process. Every v-node is associated with an open file descriptor.

The `fcntl()` function is basically a method for accessing or modifying information in the v-node such as current offset, ownership, file descriptor number.

## ioctl():

---

Unix likes to treat everything as files but certain “devices” have needs beyond reading, writing and positioning.

`ioctl()` is the catch all function for making non-file-ish requests to devices represented by a file descriptor.

This includes operations such tray ejection, tape mounting and turning on device specific features such as DMA.

## Blocking an asynchronous computation:

`read()` normally blocks when no data is ready but everything else is fine.

`write()` normally blocks when buffer systems are already full.

This can make modern programs very difficult to write.

Modern programs tend to be asynchronous and perform many tasks simultaneously. GUIs are highly asynchronous. Early or non-threaded web servers handle multiple HTML requests simultaneously.

If a read or write operation blocks then the entire program application blocks. This can lead to deadlock scenarios.

There are a couple of ways to handle the problem...

## Non-Blocking I/O:

---

The file descriptor can be opened with `O_NONBLOCK` or `O_NDELAY`.

This forces `read()` and `write()` to fail if they would have otherwise blocked.

In such a case `read()` (`write()`) will immediately return -1 and the global variable `errno` (included by `#include <errno.h>`) will be set to `EAGAIN` (`errno==EAGAIN` is true.)

This is an easy and straightforward method to solve asynchronous processing.

It complicates the application a bit.

It doesn't scale well to handle lots of file descriptors.

## select():

---

`select()` is a function that can “poll” a collection of file descriptors simultaneously and returns only when one of the descriptors is “ready”.

File descriptors are represented as sets (`fd_set`).

The application adds desired file descriptors to read, write or exception sets and the sets are passed to `select()` for polling.

```
int select(int n, fd_set *readfds, fd_set *writefds,  
          fd_set *exceptfds, struct timeval *timeout);
```

`n` is the value of the largest file descriptor number **plus 1!**

`timeout` is a structure that can be used to limit the amount of time that the call to `select()` will block.

When `select()` returns the file descriptor sets can be tested to see which file descriptors are ready. File descriptors in the read set are guaranteed not to block a read if the set test yields true for that file descriptor.

## select() continued:

---

`select()` still complicates applications.

Works well on large sets of file descriptors.

- Non-threaded web servers.
- Modem communication programs.
- Parent processes coordinating multiple children.
- Coordination of distributed processes.

Only provides limited asynchronous processing. (It will still block until at least file descriptor is ready or until a timeout occurs.)

## (Non-)Blocking I/O and CPU cycles:

When system calls, such as `read()`, block the kernel is aware and can put the process to sleep where it doesn't use any CPU cycles. The kernel wakes the process when an event occurs that satisfies the blocking condition.

Non-blocking I/O thwarts this performance advantage because CPU cycles are consumed "polling".

`select()` has the advantage that it blocks until a file descriptor is ready.

A proper implementation of `select()` can still allow the kernel to suspend the execution of the application and use the CPU cycles for other running processes.

## Buffered I/O (`#include <stdio.h>`):

The system-level functions of `read()`, `write()`, etc. are implemented to be simple, efficient and general purpose.

One disadvantage to these routines is that they are not buffered.<sup>a</sup>

This makes it difficult to read specific items such as integers and lines. (`read()` and `write()` have no knowledge that integers are 4 bytes long or that lines end with `\n`). `<stdio.h>`

To make life easier for the programmer the library of functions known as `stdio.h` was created to address this.

---

<sup>a</sup>The kernel buffers IO to the physical device behind these routines to improve device performance but there is no buffering between the routines and the user's application. Hence the need for loops as described early.

## Common `stdio.h` functions:

`stdio.h` provides many functions, similar to those presented earlier, that are buffered. They don't function on raw file descriptors; instead they operate on an abstract type called `FILE *` (a pointer to a `FILE` structure).

- `fopen()`, `fdopen()` to open files.
- `fclose()` to close files opened by the `f...()` functions.
- `printf()`, `fprintf()` to write to files with specified formatting.
- `scanf()`, `fscanf()` to read and parse data from files.
- `fgets()` to read lines of input at a time.

All of these functions know how to perform without additional loops.

These understand numerical formats and can parse such data from strings into the mathematical value represented similar to `Integer.parseInt()` in Java.

These functions are built on top of the `fcntl()`, `unistd()` functions.