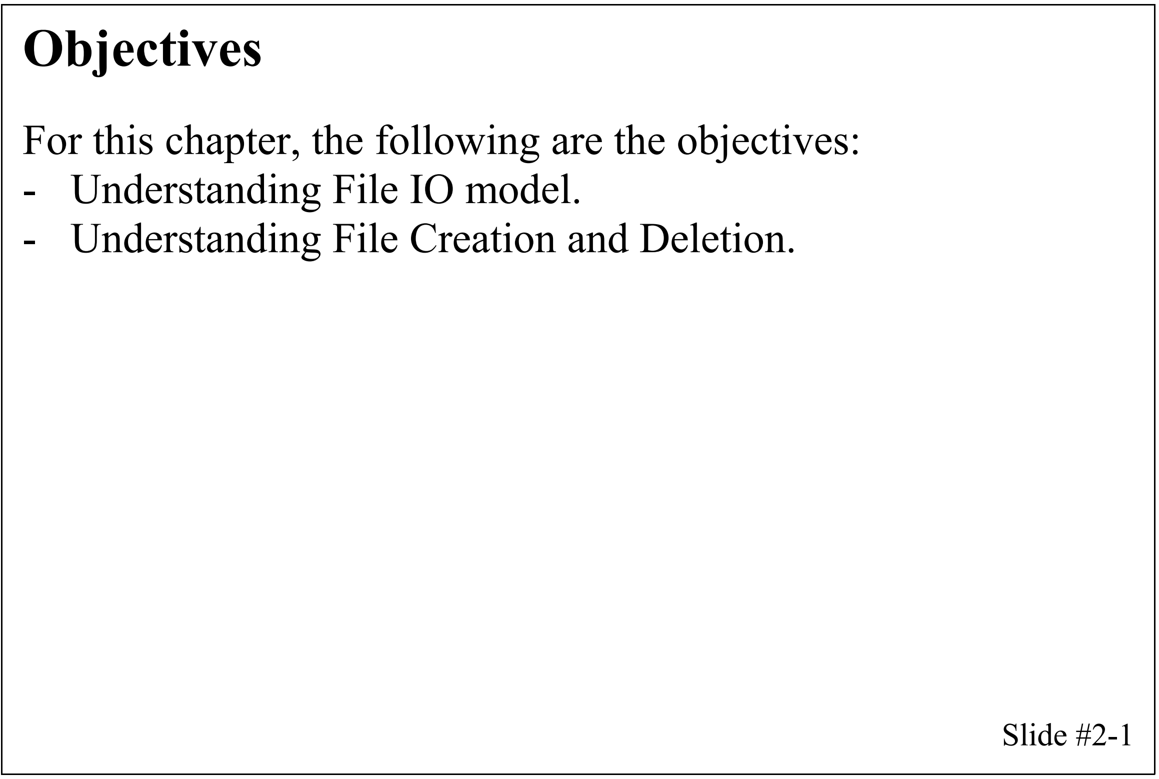
Chapter 2

File and File IO

Chapter Objectives

To understand File and File IO concepts used over the duration of the course



Objectives

For this chapter, the following are the objectives:

- Understanding File IO model.
- Understanding File Creation and Deletion.

Slide #2-1

Notes

In this chapter, we examine the terms File and File IO.

The objective of this chapter is to provide an understanding of the concepts on files and file IO that would be used over the duration of the course.

Chapter Organization

1. **Objective**: Introduction to
   - File
   - File IO.

2. **Description**: A file is the basic element of IO .
   Any IO (e.g. disk, tape or networking IO) is based on file IO -- a polymorphic approach to IO.
   This chapter provides an introduction to the concepts used in course.

3. **Concepts Covered in Chapter**:
   - Introduction to Files and Files IO.
   - The design and architecture of the UNIX Files and File IO model.
   - LINUX Notes

4. **Prior Knowledge**:
   same as Chapter #1

5. **Teaching & Learning Strategy**:
   Discussion questions are,
   - What are Files? .. fd? .. and, `FILE *`?

   - Files: Structure and Layout?,
   - Files created using `open()` ?,
   - File Systems Model: inodes and files .. and directories.
   - `rename()` and `unlink()`?

6. **Teaching Format**:
   Theory + Homework Assignments

7. **Study Time**: 120 Minutes (Lecture & Theory)
   + ~45 minutes (Homework Assignments)

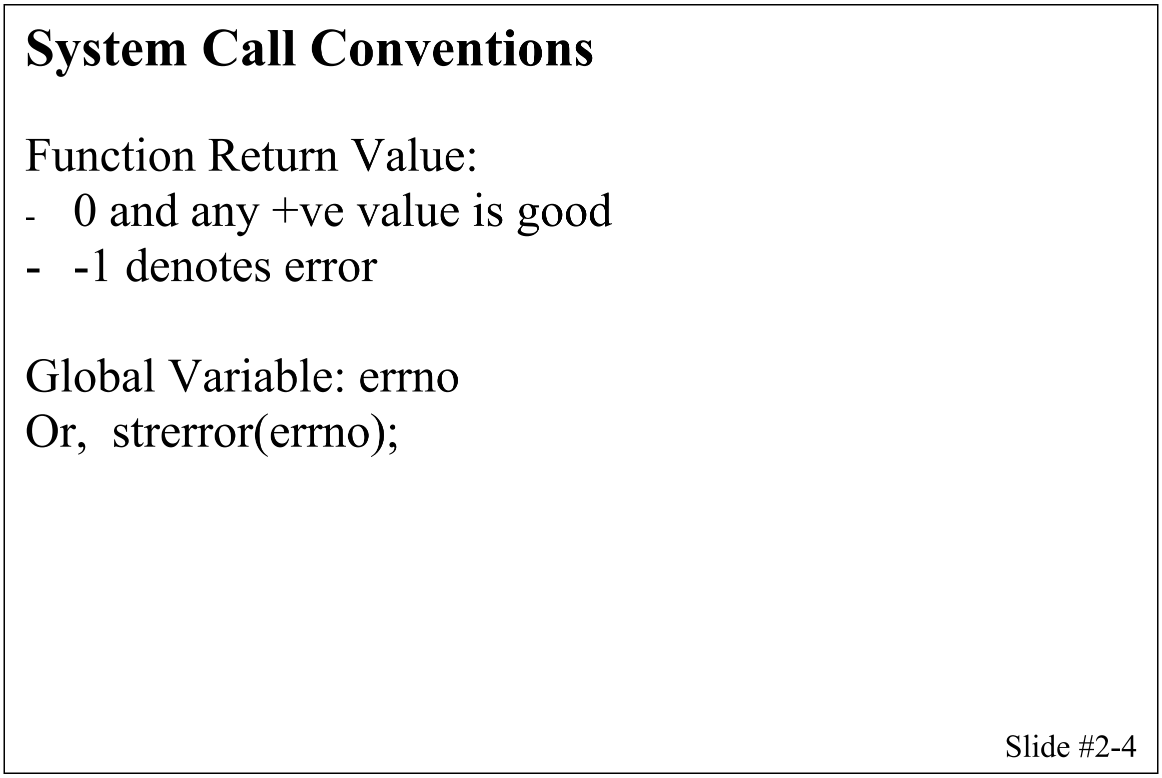
8. **Assessment**: Group Homework Assignments

9. **Homework Eval**: Group

10. References:

Stevens; APUE:	Ch #1, #2
Vahalia; UI:	Ch #1, #2.1-2.4
Robbins & Robbins; USP:	Ch #4, #5 and #6.

11. Stevens; APUE: Ch #1, #



System Call Conventions

Function Return Value:

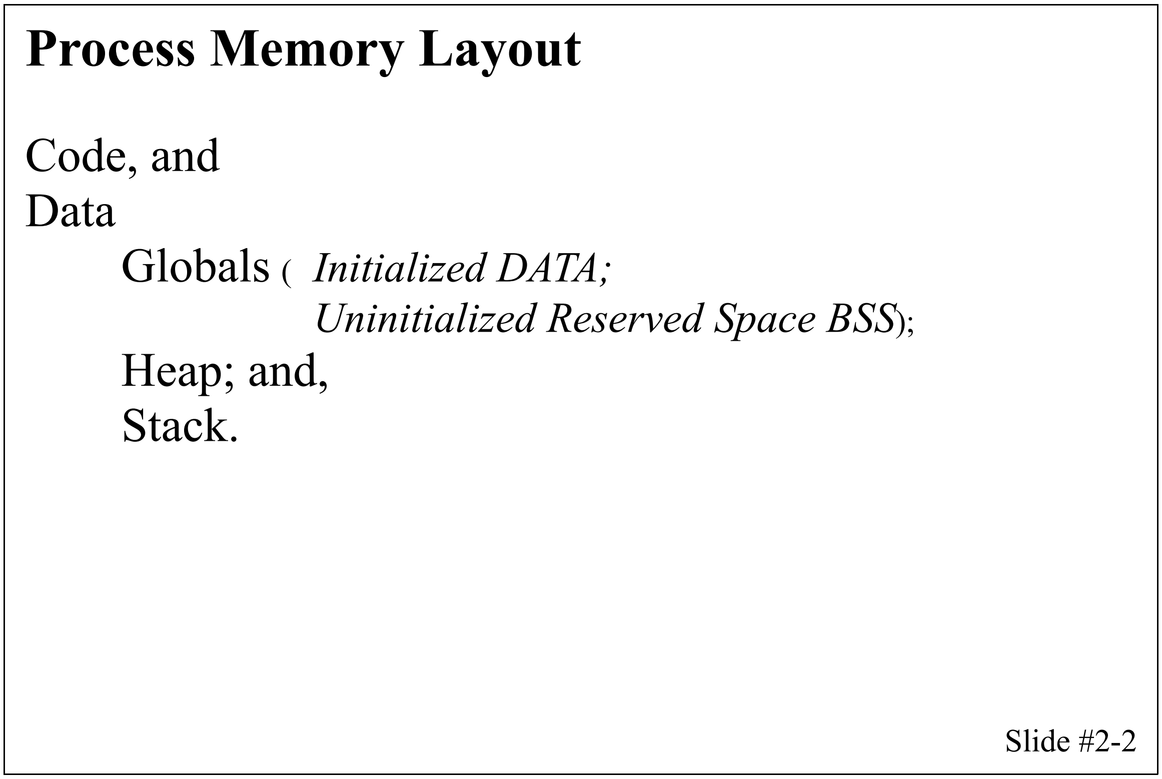
- 0 and any +ve value is good
- -1 denotes error

Global Variable: errno
Or, strerror(errno);

Slide #2-4

Notes

1. 0 and any +ve value is good. -1 denotes error.
2. errno
   The global variable errno will be set if an error occurred. errno will not be cleared if a subsequent system call has succeeded.
3. strerr(errno)
   The function strerror() takes an error number and returns the constant describing the nature of the error.
4. Proper Error Handling and Debugging



Process Memory Layout

Code, and
Data
    Globals (*Initialized DATA;*
              Uninitialized Reserved Space BSS);
    Heap; and,
    Stack.

Slide #2-2

Notes:

1. **Process Memory Layout**

    1.1. Code
    1.2. Data consists of
    - Globals, string constants, and initialized variables
    - Heap, and,
    - Stack.

2. **What is BSS?**

    "BSS" refers to Block Started by Symbol - the way uninitialized global variables are handled by the compiler. The BSS does not appear in the executable image of application. e.g.

```
char cvar;          // uninitialized variable allocated in .bss
char cvar2=25;      // initialized variable allocated into .data
```

3. **Further Reading**

    http://www.cosmicsoftware.com/faq/faq23.php
    http://en.wikipedia.org/wiki/Block_Started_by_Symbol

```
/* #1.   Global Un-initialized Array */
#include <stdio.h>

int myarray[50000];

int main(void) {
   int c;

   myarray[0]=1;
   c=getchar();        // to insert a pause for pmap analysis
   return 0;
}
```

```
/* #2.   Global Initialized Array */
#include <stdio.h>

int myarray[50000]={1};

int main(void) {
   int c;

   myarray[0]=1;
   c=getchar();        // to insert a pause for pmap analysis

   return 0;
}
```

```
/* #3.   Stack Un-Initialized Array */
#include <stdio.h>

int main(void) {
   int c;
   int myarray[50000];

   myarray[0]=1;
   c=getchar();        // to insert a pause for pmap analysis

   return 0;
}
```

```
/* #4.   Stack Initialized Array */
#include <stdio.h>

int main(void) {
   int c;
   int myarray[50000]={1};

   myarray[0]=1;
   c=getchar();        // to insert a pause for pmap analysis

   return 0;
}
```

1. considering ls -l output, notice that the size of the executable varies, depending on whether the variable was initialized of not. clearly, it is discernable that ch2_2 and ch2_4 are both, where the array definition is initialized.

```
$ ls -l
-rwxr-xr-x  1 rvinjamu me   9245 Jul 18 07:36 ch2_1
-rwxr-xr-x  1 rvinjamu me 209277 Jul 18 07:37 ch2_2
-rwxr-xr-x  1 rvinjamu me   6773 Jul 18 07:40 ch2_3
-rwxr-xr-x  1 rvinjamu me 206937 Jul 18 07:43 ch2_4
```

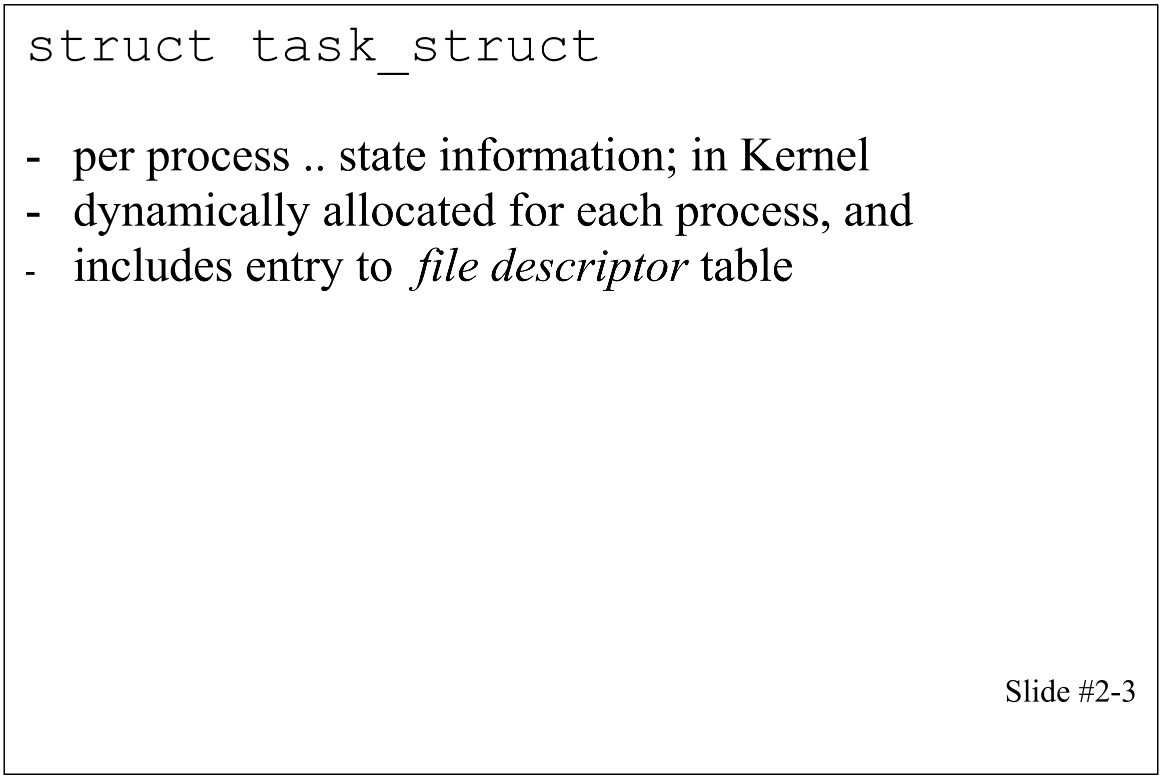
2. next, consider the output of size

```
$ size
   text    data     bss     dec     hex filename
   1132     512  200032  201676   313cc ch2_1
   1132  200520       8  201660   313bc ch2_2
   1132     512       8    1652     674 ch2_3
 201241     520       8  201769   31429 ch2_4
```

3. Notes

3.1. Allocation of 200k (=50k int array * 4 bytes) varies depending whether the variable is initialized or not, regardless of being a global variable or a local variable.

3.2. notice that ch2_1 has a size reported by ‘ls’ of 9k bytes. The bss is not allocated till run time.



`struct task_struct`

- per process .. state information; in Kernel
- dynamically allocated for each process, and
- includes entry to *file descriptor* table

Slide #2-3

Notes

Task Structure

1.1. Every process under Linux is dynamically allocated a `struct task_struct` structure. Currently, sized about 1.7 KB .. in `include/linux/sched.h.`

1.2. Loosely corresponds to elements of UNIX kernel '`struct proc`' and `'struct user'` combined together.

1.2.1. As memory **was** a very scarce resource, UNIX has traditionally separated process state information into two parts:

1.2.1.1. One part, kept memory-resident at all times (called 'proc structure'; that, includes process state, scheduling information etc.) and,

1.2.1.2. Another part, needed only when the process is running (called 'u area'; that, includes file descriptor table, disk quota information etc.).

1.2.2. Linux does not implement such separation; the process state information is maintained in a kernel memory-resident data structure at all times.

1.3. Some fields in `struct task_struct` include.

1.3.1. File Descriptor Table field in `p->files`.

1.3.2. Scheduler Related Fields `p->has_cpu`, `p->processor`, `p->counter`, `p->priority`, `p->policy` and `p->rt_priority` .. more later.

1.3.3. Address space fields `p->mm` and `p->active_mm` that point to the process' address space, described by `mm_struct` structure, and active address space, if the process doesn't have a real one (e.g. kernel threads). More Later.

1.3.4. Task Personality Fields `p->exec_domain` and `p->personality` .. the way certain system calls behave in order to emulate the "personality" of foreign flavours of UNIX.

1.3.5. File System Information field `p->fs`:

1.3.5.1. root directory's dentry, and, mountpoint,

1.3.5.2. alternate root directory's dentry, and, mountpoint,

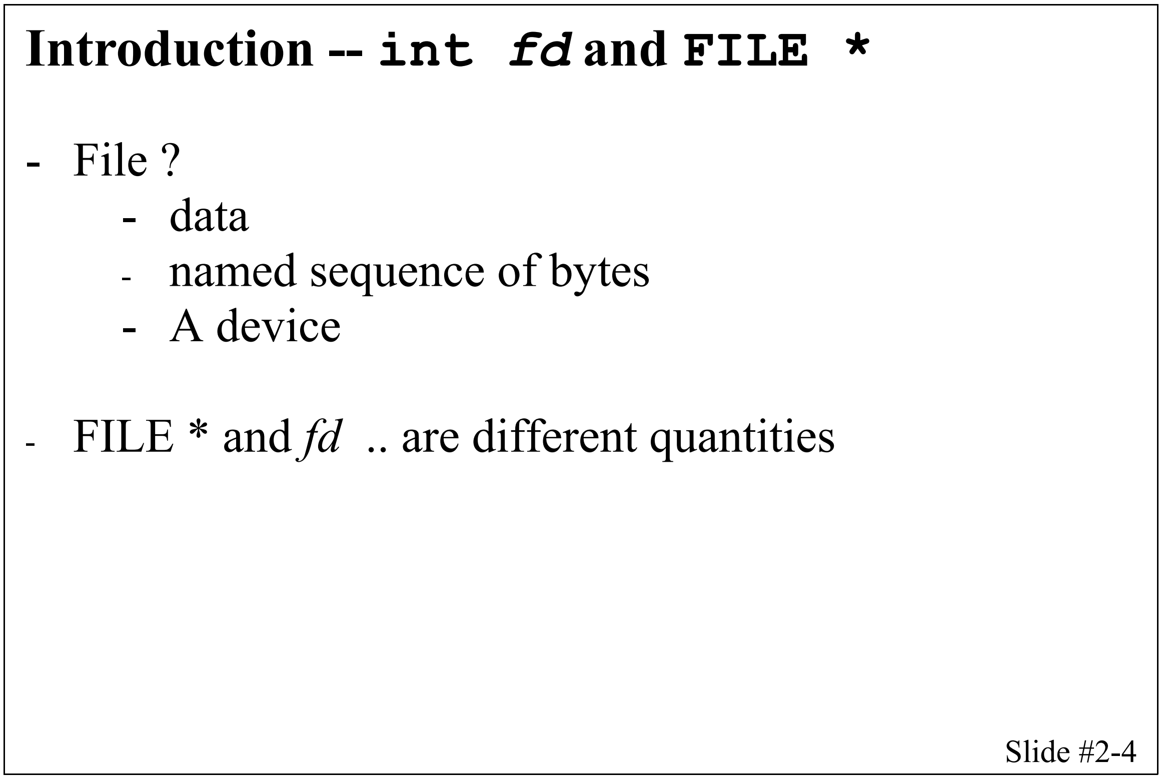
1.3.5.3. current working directory's dentry, and, mountpoint.

1.3.6. Signal handler field .. `p->sig`

Further Reading

1.4. Task Structure in 2.4 http://www.faqs.org/docs/kernel_2_4/lki-2.html

1.5. Concepts http://www.informit.com/articles/article.asp?p=370047&rl=1



Introduction -- `int fd` and `FILE *`

- File ?
  - data
  - named sequence of bytes
  - A device

- FILE * and *fd* .. are different quantities

Slide #2-4

Notes

1. IO

1.1. IO in UNIX is based on the concept of ‘file’ (.. not FILE).

1.2. All IO uses the same approach ..
“polymorphic” approach to File IO .. that is, without regard to “geometry” and hardware specifications of the underlying device; and, as a logical extension, devices can be “virtual”, too.

1.2.1. All files are `open()`ed , and accessed using `read()` and `write()`, and when done .. are `close()`’d (.. explicit or implicit).

1.2.2. Each user is prevented from accessing and interfering with another user’s memory.

1.3. A process running user-space code, is also known as “user” process and is said to be operating in “user mode”.

2. fd table

2.1. The Kernel maintains File Descriptors open for each process .. an ‘fd’ table.

2.2. The fd table is in the Task Structure in Linux (or, u area in UNIX).

http://linux.about.com/od/commands/l/blcmdl_2a.htm

`FILE` **and** `FILE *`

Structure
.. defined in <`stdio.h`>
.. `fopen()` allocates memory using `malloc()`

.. in user-space

fd

- int .. *datatype*
- index# - into an array .. *the File Descriptor table*.
- 0, 1 and 2 are always taken by .. *stdin, stdout, stderr*

Slide #2-6

Notes

1. FILE .. user-space structure.
   Allocated on heap by fopen() and, *may* be declared as ..

```
typedef struct
{  int              cnt;   /* number of available chars in buffer */
   unsigned char   *ptr;   /* next character from/to here in buf */
   unsigned char   *base;  /* the buffer */
   unsigned char    flag;  /* the state of the stream */
   unsigned char    fd;    /* file descriptor */
} FILE;
```

2. Used by fopen();fgets(),getchar();putc(),puts(),putchar();
   Also, `printf(), scanf();`

3. At a high level, `fopen()` does the following:

   - `malloc()` for struct FILE
   - prepares flags and calls `open()`
   - if open() successful, populate `fd` member in FILE struct
   - else, deallocate .. FILE struct
   - return .. FILE *

Notes

4. File Descriptor "fd" **Table** .. is

   4.1. for each process,

4.2. an index# or vector# into file descriptor table,

4.3. points the kernel file table entry

4.4. `open()` creates an entry in the kernel file table, that is mapped into the kernel file descriptor table.

5. **Kernel** File **Table** .. maintains

5.1. resource metadata and utilization in kernel tables .. as a resource manager

5.2. an entry for each open file .. in the kernel file table.

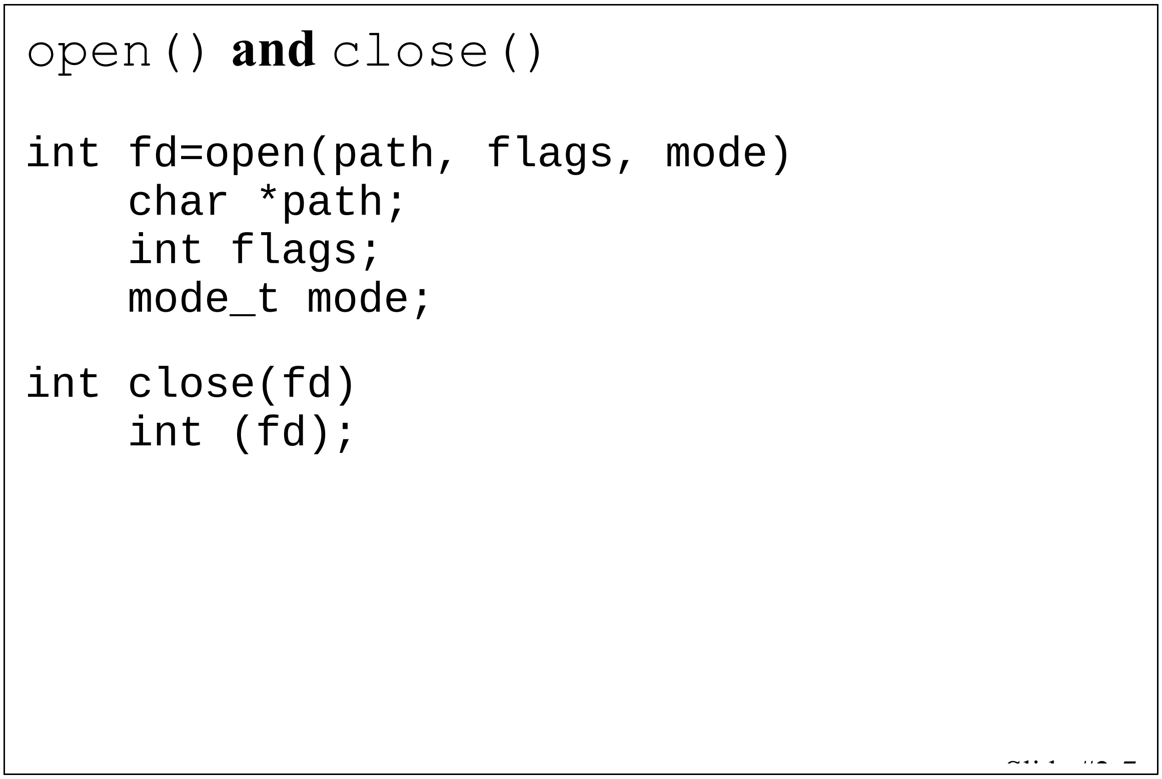
5.3. multiple entries for <u>one</u> file, if the same file is opened multiple times within the same process.

6. Code Example

```
#include <stdio.h>

char *rfn= “myfile.txt”;
int main() {
   FILE *rfp;
   rfp=fopen(rfn, “r”);
   .
   .
   .
   fclose(rfp);
}
```

Code Example -- open(), and close()

```
#include <stdio.h>

char *rfn= “myfile.txt”;
int main() {
   int rfd;
   rfd=open(rfn, O_RDONLY);
   .
   .
   .
   close(rfd);
}
```



open() **and** close()

```
int fd=open(path, flags, mode)
    char *path;
    int flags;
    mode_t mode;

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

Notes

1. int fd=open(path, flags, mode) .. takes 3 parameters.
   `fd` .. returns index# of lowest available element in u_area fd table, and, -1 on error.
   `path` .. full name for file, includes absolute/relative path
   `flag` .. IO characteristics .. e.g. O_RDONLY, O_WRONLY, O_RDWR, O_APPEND,O_TRUNC, O_CREAT
   `mode` .. **permission modes** for file at creation. Not needed at other times.

open flags

O_RDONLY	open for read only
O_WRONLY	open for write only
O_RDWR	open for read and for writes
O_APPEND	open for **append** on each write.
O_CREAT	create file if it does not exist
O_EXCL	error if create and file exists
O_TRUNC	truncate if file exists
O_NONBLOCK	no blocking on `read()`
O_NOCTTY	disallow tty to become control terminal

.

2. **open flags and fopen() ..** Not all flags are used by fopen().

   `2.1.    fopen(“myfl”,“r”) .. open(“myfl”,O_RDONLY);`

2.2. fopen(“myfl”,“w”) .. open(“myfl”,O_WRONLY|O_CREAT|O_TRUNC)

7. mode_t

2.3. perm. bits .. single int .. each bit=>special perms. drwxrwxrwx ..
2.4. POSIX data type .. similar to other “_t” data types, belong to “int” class
2.5. can change permissions .. chmod is a wrapper on **chmod()** ..
2.6. **chmod()** args e.g. S_Imabc .. m=>R,W or X; abc=>USR,GRP or OTH

S_IRUSR	read permission for owner, user
S_IRGRP	read permission for group
S_IROTH	read permission for others
S_IWUSR	write permission for owner, user
S_IWGRP	write permission for group
S_IWOTH	write permission for others
S_IXUSR	execute permission for owner, user
S_IXGRP	execute permission for group
S_IXOTH	execute permission for others
S_IRWXU	read, write,execute permission for owner, user
S_IRWXG	read, write,execute permission for group
S_IRWXO	read, write,execute permission for others

8. Code Example -- open(), and close()

```
#include <stdio.h>

char *rfn= “myfile.txt”;
int main() {
   int rfd;
   rfd=open(rfn, O_RDONLY);
   .
   .
   .
   close(rfd);
}
```

9. Code Example -- open(), and multiple calls for close() on same descriptor.

```
#include <stdio.h>

char *rfn= “myfile.txt”;
int main() {
   int rfd;
   rfd=open(rfn, O_RDONLY);
   .
   .
   .
   close(rfd);
   close(rfd); /* multiple close() same fd  is *mostly* ok. */
}
```

10. Code Example -- open(), and multiple calls for open() and close()

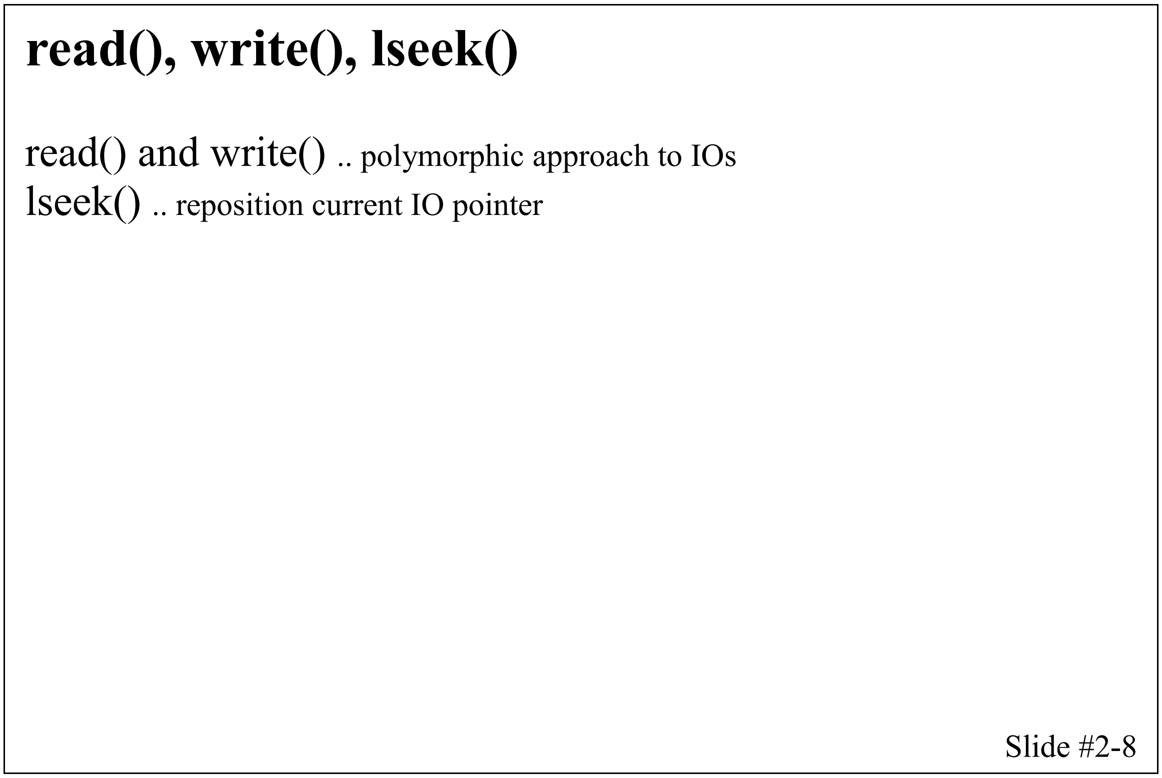
```
#include <stdio.h>

char *rfn= “myfile.txt”;
int main() {
   int rfd1, rfd2;
   rfd1=open(rfn, O_RDONLY);
   rfd2=open(rfn, O_RDONLY);
   .
   .
   .
   close(rfd1);
   close(rfd2); /* .. different descriptor .. ok. */
}
```

11. Code Example – open() and using flags

```
#include <stdio.h>
int main(int argc, char **argv) {
   int rfd, openflags;
   char *rfn;
   for (argv++; *argv; argv++){
      rfn= *argv;
      openflags = O_RDONLY;
      rfd=open(rfn, openflags);
      printf(“file \”%s\” open fd=%d; flags=%#010x\n”,
            rfd, flags);
      .
   .
   .
      /* close(rfd); */
            /* What happens  if close() is commented out ?*/
   }

exit(0);

}
```

Further Reading

http://linux.about.com/od/commands/l/blcmdl2_open.htm
http://linux.about.com/od/commands/l/blcmdl2_close.htm



read(), write(), lseek()

read() and write() .. polymorphic approach to IOs
lseek() .. reposition current IO pointer

Slide #2-8

Notes

1. **read()**

1.1. `ssize_t read(`*`fd`*`,void *buf,size_t nbytes)` .. 3 parameters.
1.2. facilitates binary IO.
1.3. Lowest-level system call, and, is used by stdlib functions that need read functionality .. getchar()
1.4. Reads bytes from a file (.. or any device) from the position indicated by the current offset in kernel file table entry.
1.5. Returns #bytes read. ‘0’ on End-Of-File.

2. **write()**

2.1. `ssize_t write(`*`fd`*`,void *buf,size_t nbytes)` .. 3 parameters.
2.2. facilitates binary IO, like read().
2.3. Lowest-level system call, and, is used by stdlib functions that need write functionality .. putchar()
2.4. Writes bytes from file (..or device) .. from the position indicated by the current offset in kernel file table entry.
2.5. Returns #bytes written. Usually same# as requested. ‘-1’ on error ..

3. **lseek()**

   3.1. `off_t lseek(`*`fd`*`,off_t offset,int whence)` .. 3 parameters.

   3.2. offset can be -ve.

   3.3. `whence` .. positions IO ptr ..

   `SEEK_SET` .. offset from beginning of file,

   `SEEK_CUR` .. offset relative to current position in file,

   `SEEK_END` .. offset from end position of file,

4. **Code Example** -- open(), close() and read(), write()

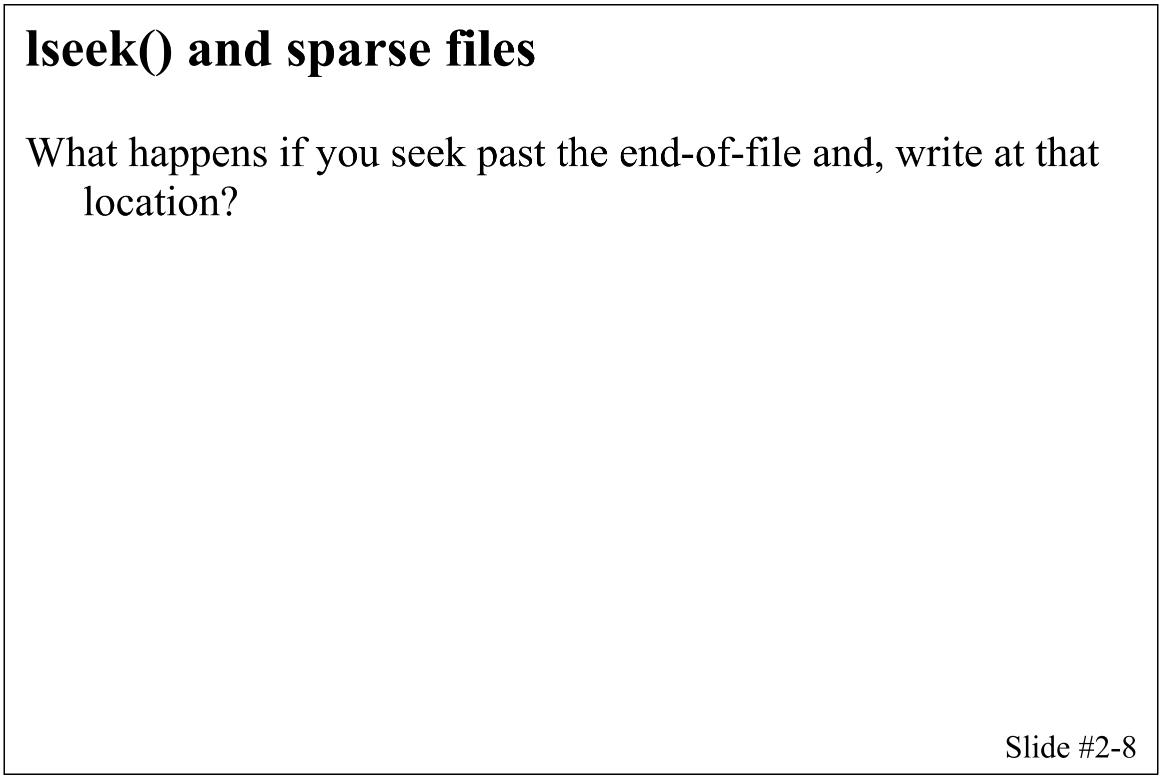
```
#include <stdio.h>

char *rfn= “myfile.txt”;
char *wfn= “myfile.out”;

int main() {
   int rfd, wfd;           // read and write fd’s
   int nr,nw;          // number read and written
   char buf[1024];

   rfd=open(rfn, O_RDONLY);
   wfd=open(wfn, O_WRONLY||O_APPEND||O_CREAT||O_TRUNC);

   while (nr=read(rfd,buf,sizeof(buf))) {
      .
      .
      .
      nw=write(wfd,buf,strlen(buf));
   }
.
.
.
close(rfd);
close(wfd);
}
```

5. **Code Example** -- open(), close(), read(), write() and lssek()

```
#include <stdio.h>

char *rfn= “myfile.txt”;
char *wfn= “myfile.out”;

int main() {
   int rfd, wfd;            // read and write fd’s
   int nr,nw, nl;           // number read and written
   char buf[1024];

   rfd=open(rfn, O_RDONLY);
   wfd=open(wfn, O_WRONLY||O_APPEND||O_CREAT||O_TRUNC);

   while (nr=read(rfd,buf,sizeof(buf))) {
      .
      .
      .
      nw=write(wfd,buf,strlen(buf));
      nl=lseek(rfd,10,SEEK_CUR);
   }
   .
   .
   .
   close(rfd);
   close(wfd);
}
```

6. **Further Reading :**

   http://linux.about.com/od/commands/l/blcmdl2_read.htm
   http://linux.about.com/od/commands/l/blcmdl2_write.htm
   http://linux.about.com/od/commands/l/blcmdl2_lseek.htm



lseek() and sparse files

What happens if you seek past the end-of-file and, write at that location?

Slide #2-8

Note

1. A sparse file can be created by using

```
dd if=/dev/zero of=myfile bs=1k seek=128 count=1
```

2. The above line writes only 1k.

3. Yet, “1s –l | grep myfile” shows a 128k file

```
-rw-r--r--  1 raghav root 132096 Aug 24 10:46 myfile
```

4. Try, “ls –lah | grep myfile”

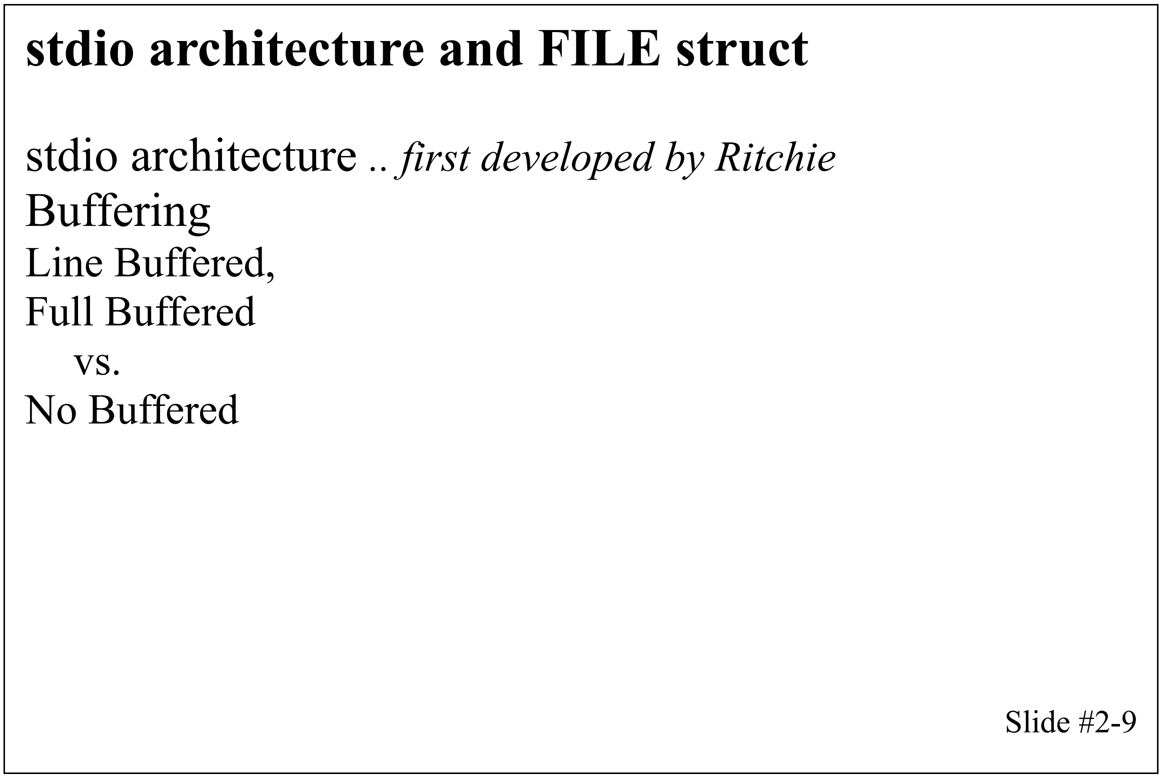
```
ls -lah | grep myfile
-rw-r--r--   1 raghav root  129K Aug 24 10:46 myfile
```

5. And, “du –lah | grep myfile”

```
du -lah | grep myfile
8.0K    ./myfile
```

6. Also,

```
dd if=/dev/zero of=myfile bs=1k seek=128 count=4;
du -lah | grep myfile
8.0K    ./myfile
```



stdio architecture and FILE struct

stdio architecture .. *first developed by Ritchie*

Buffering

Line Buffered,

Full Buffered

vs.

No Buffered

Slide #2-9

Notes:

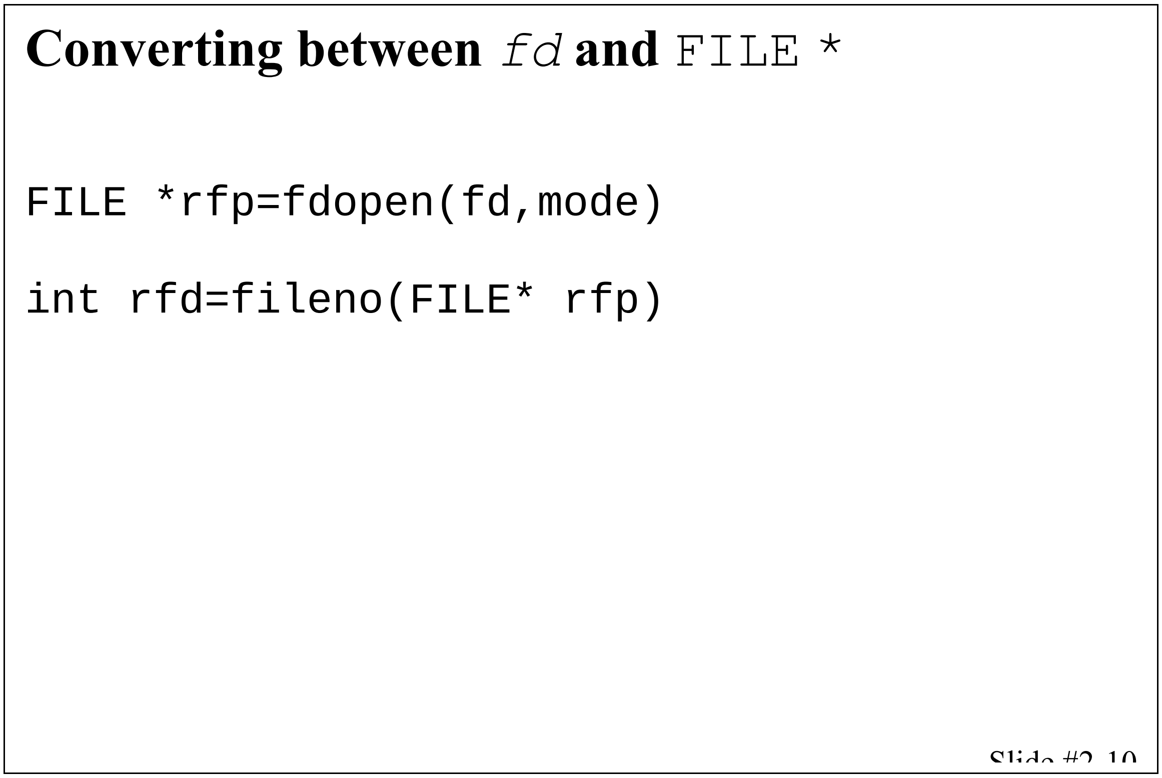
1. **stdio concepts**
   1.1. `stdin,stdout` and `stderr` are FILE * structs .. correspond to fd's `0,1,2`
   1.2. fd's 0,1,2 are opened automatically and assigned to global vars
   `stdin, stdout, stderr`
   1.3. default buffering ..

1.3.1.	FILE	1 blk buffer (default);
1.3.2.	stdin, stdout	linebuf;
1.3.3.	stderr	No Buffering

2. **fflush()**
   2.1. `int fflush(FILE *fp);` /* flush FILE * buffer ..*/

3. **setvbuf()**
   /*called before any IO. To change buf size or mode `_IOFBF,_IOLBF,_IONBF` */

```
int setvbuf(FILE *fp, char *buf, int mode, size_t size);
```



Converting between *`fd`* and `FILE *`

```
FILE *rfp=fdopen(fd,mode)

int rfd=fileno(FILE* rfp)
```

Slide #2-10

Notes:

1. Converting from/to fd
   1.1. `FILE *rfp=fdopen(int rfd, char *mode);` //Convert fd to `FILE *`
      1.1.1. stdlib function .. allocates new `FILE` struct over an existing `fd`.
      1.1.2. Note .. `mode`, should match the value of mode as originally `open()`'ed.
   1.2. `int fileno(FILE *rfp)` // Macro .. Get fd given a FILE *

2. Further Reading

   http://linux.about.com/library/cmd/blcmdl3_fdopen.htm

stdio architecture and FILE struct

file redirection
IO redirection
stdin, stdout, stderr redirection
.. e.g. '|' and tee

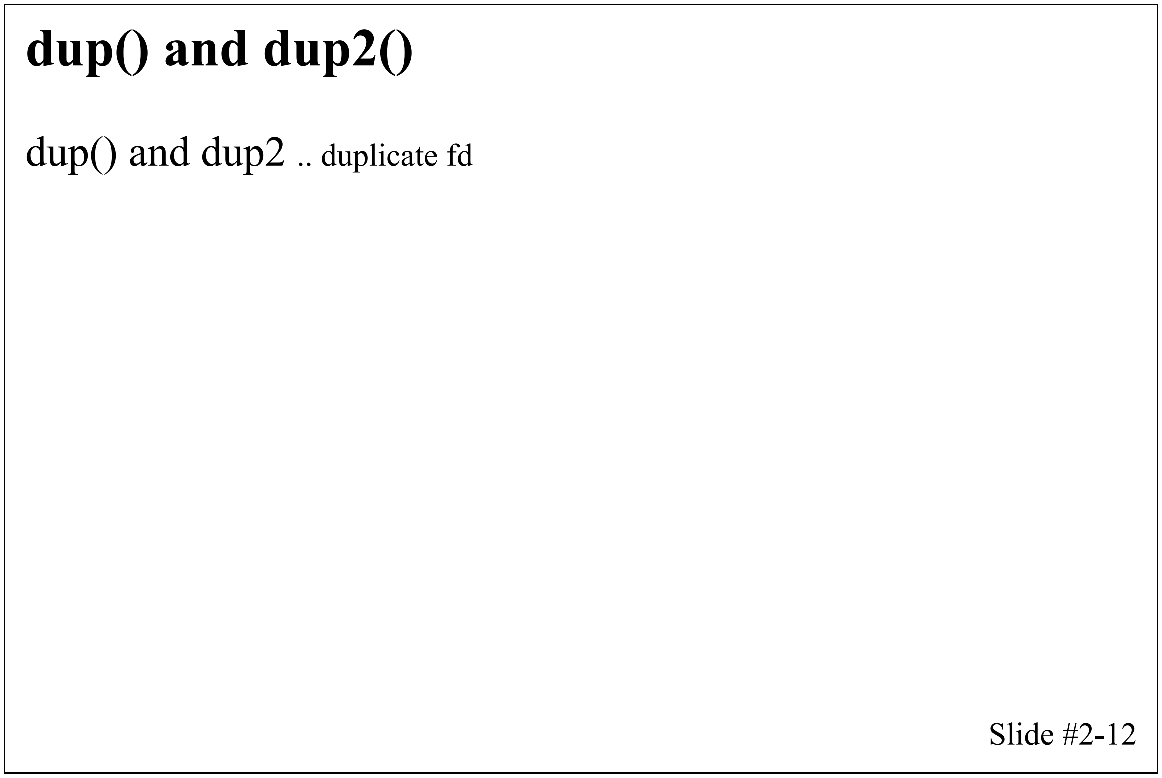
Notes
Code Example

1. **Code Example** – stdin redirection .. classic approach.

```
#define _POSIX_SOURCE 1
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>

char *rfn= “myfile.txt”;

int main(int argc, char **argv) {
   int c,newfd;
   if (argc == 2) rfn=argv[1];

   fprintf(stderr, “Reading a char from stdin\n”);
   c=getchar();
   fprintf(stderr, “char read is %c\n”, c);

   close(0);
   newfd=open(rfn,O_RDONLY);

   if (newfd < 0) {
      fprintf(stderr, “Error: open ‘%s’ failed: %s”,
         rfn,strerror(errno));
      exit(1);
   }
   fprintf(stderr, “Reading from fd=%d .. new stdin\n”, newfd);

   while ((c=getchar())!=EOF) {
      putchar(c);
   }

   close(0);
   exit(0);
}
```

2. **Further Reading**

   http://linux.about.com/library/cmd/blcmdl3_fdopen.htm



dup() and dup2()

dup() and dup2 .. duplicate fd

Slide #2-12

Notes

1. dup() and dup2()
   1.1. `dup()`
      1.1.1. creates new fd. `int dup(int *fd*)`
      1.1.2. Returns lowest available fd. two fd's referencing same Kernel File Entry.
   1.2. `dup2()`
      1.2.1. Closes new fd, if already open. `int dup2(*oldfd,newfd*)`
      1.2.2. Returns new fd. two fd's pointing to the same Kernel File Entry.

2. What does the following example do?

```
#define  _POSIX_SOURCE 1

int main() {
   c=getchar();
   rfd=open(rfn,O_RDONLY);
   close(0);
   dup(rfd);
   close(rfd);

   while ((c=getchar()) != EOF) {
      putchar(c);
   }
   exit(0);
}
```

2. Code Example – stdin redirection .. using dup()

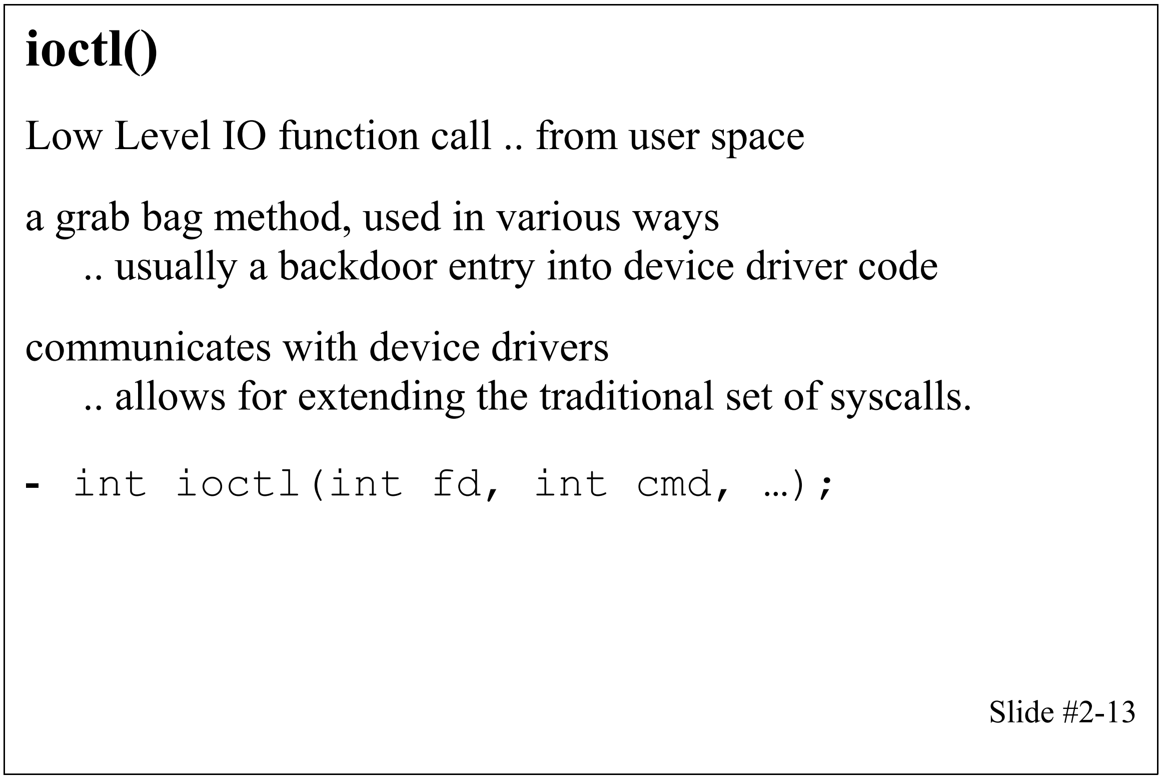
```
#define _POSIX_SOURCE 1

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>

char *rfn= “myfile.txt”;

int main(int argc, char **argv) {
   int c,rfd,newfd;

   if (argc == 2) {
      rfn=argv[1];
   }

   fprintf(stderr, “Reading a char from stdin\n”); c=getchar();
   fprintf(stderr, “char read is %c\n”, c);

   newfd=open(rfn,O_RDONLY);

   if (newfd < 0) {
      fprintf(stderr,“Error: open %s failed:%s”,rfn,
         strerror(errno));
      exit(1);
   }

   fprintf(stderr, “File ‘%s’ uses fd=%d.\n”, rfn, newfd);

   close(0);

   if (dup(newfd) != 0) {  // will go into slot 0
      fprintf(stderr, “Error: Could not dup file %s”, rfn);
      exit(1);
   }

   close(newfd);  // close fd==3

   while ((c=getchar())!=EOF) {
      putchar(c);
   }

   close(0);
   exit(0);
}
```

3. Code Example – stdin redirection .. using dup2()

```
#define _POSIX_SOURCE 1

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>

char *rfn= “myfile.txt”;

int main(int argc, char **argv) {
   int c,rfd,newfd;

   if (argc == 2) {
      rfn=argv[1];
   }

   fprintf(stderr, “Reading a char from stdin\n”);
   c=getchar();
   fprintf(stderr, “char read is %c\n”, c);

   newfd=open(rfn,O_RDONLY);

   if (newfd < 0) {
      fprintf(stderr, “Error: open %s failed:%s”,
         rfn,strerror(errno));
      exit(1);
   }

   fprintf(stderr, “File ‘%s’ uses fd=%d.\n”, rfn, rfd);

   if (dup2(rfd,0) != 0) {  // will go into slot 0
   fprintf(stderr, “Error: Could not dup file %s”, rfn);
   exit(1);
   }

   close(rfd);  // close fd==3

   while ((c=getchar())!=EOF) {
      putchar(c);
   }

   close(0);
   exit(0);
}
```

3. **Further Reading :**

   http://linux.about.com/library/cmd/blcmdl2_dup.htm



ioctl()

Low Level IO function call .. from user space

a grab bag method, used in various ways
.. usually a backdoor entry into device driver code

communicates with device drivers
.. allows for extending the traditional set of syscalls.

- `int ioctl(int fd, int cmd, …);`

Slide #2-13

Notes

1. ioctl

The *ioctl* function call in user space has the following prototype:

```
int ioctl(int fd, int cmd, …);
```

1.1. The first argument is the fd .. got from open().

1.2. The second argument is a number, that issues a command to the device.

- These numbers are unique across the system to prevent errors caused by issuing the right command to the wrong device.
- Refer to *include/asm/ioctl.h* and *Documentation/ioctl-number.txt*.
More Later.

1.3. For the third argument,

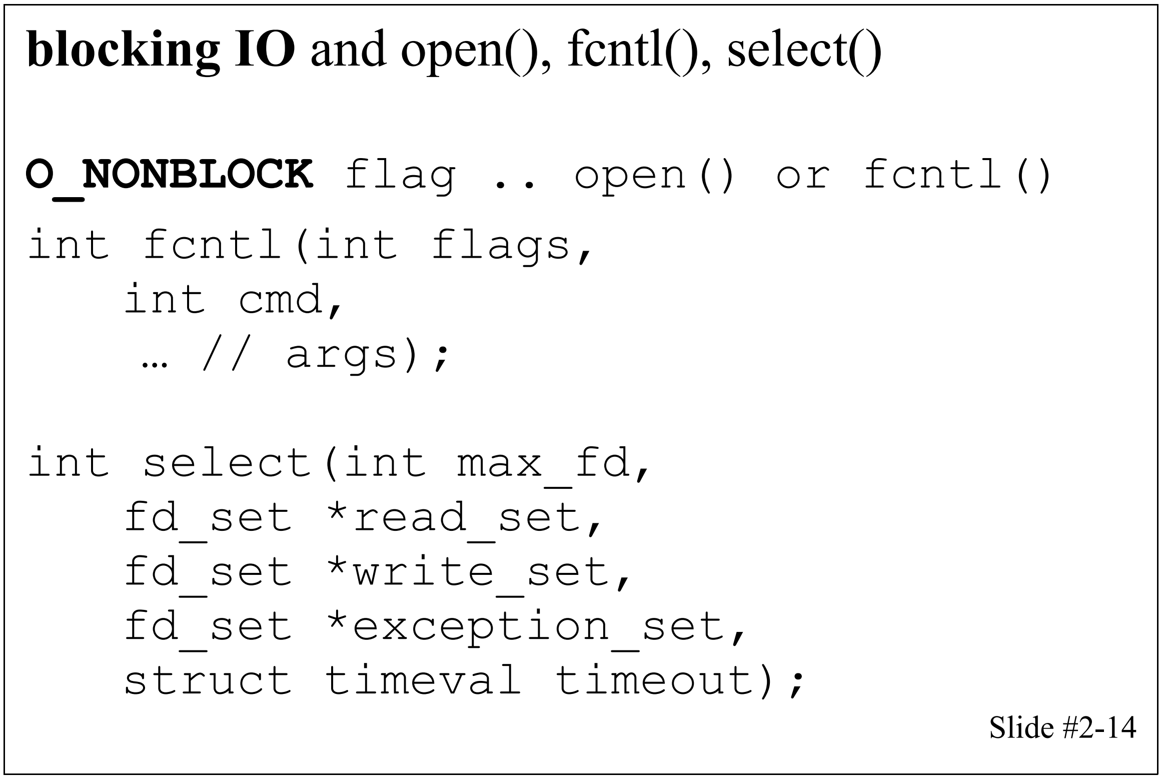
- the dots in the prototype **usually** represent a variable number of arguments; however, in this case, it refers to a single optional argument, usually called `char *argp`.
- The dots are there simply to avoid type checking during compilation.

Code Example

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

```
#include <errno.h>
#define WRITE 1
#define READ 2
#define MYNUM 5558
#define MYSTR "Eureka!"
main() {
   int fd, len, wlen;
   char str[128];
   long inum = MYNUM;
   long onum;

   strcpy(str, MYSTR);
   // open
   if((fd = open("/dev/CDD", O_RDWR | O_APPEND)) == -1) {
      fprintf(stderr,"ERR:open():%s\n",strerror(errno));
      exit(0);
   }
   // write
   wlen = strlen(str);
   if ((len = write(fd, str, wlen)) == -1) {
      fprintf(stderr,"ERR:write():%s\n",strerror(errno));
      exit(1);
   }

   // read
   if ((len = read(fd, str, sizeof(str))) == -1) {
      fprintf(stderr,"ERR:read():%s\n",strerror(errno));
      exit(1);
   }
   fprintf(stdout, "%s\n", str);

   // write using ioctl()
   if ((len = ioctl(fd, WRITE, &inum)) == -1) {
      fprintf(stderr,"ERR:ioctl-write():%s\n",
         strerror(errno));
      exit(1);
   }

   // read using ioctl()
   if ((len = ioctl(fd, READ, &onum)) == -1) {
      fprintf(stderr,"ERR:on ioctl-read():%s\n",
         strerror(errno));
      exit(1);
   }
   fprintf(stdout, "read .. %#0x(%d)\n", onum,onum);

   close(fd);
}
```

1. **Further Reading :**
   http://www.linuxjournal.com/node/6908
   http://openit.disco.unimib.it/lxrfarina/source/include/sys/ioctl.h



blocking IO and open(), fcntl(), select()

`O_NONBLOCK` `flag .. open() or fcntl()`

```
int fcntl(int flags,
    int cmd,
     … // args);

int select(int max_fd,
    fd_set *read_set,
    fd_set *write_set,
    fd_set *exception_set,
    struct timeval timeout);
```

Slide #2-14

Notes

1. **stdin ..** for interactive input to the program

2. **fcntl()** .. change attributes on fd's already **open**ed. e.g. to change blocking/non-blocking attributes.

3. **Blocking IO ..** blocking IO vs non-blocking IO

   3.1. blocking IO .. by default
   .. read() <u>waits</u> for data to be available, write() <u>waits</u> for data to be flushed

   3.2. non-blocking IO
   .. is enabled by setting `O_NONBLOCK` flag

   3.2.1. when `open()`'ing file, or,

   3.2.2. Code example

```
#define  _POSIX_SOURCE 1
…
int main() {
…
     rfd=open(rfn,O_NONBLOCK);
…
}
```

   3.2.3. calling `fcntl()` on an open file.

```
#define  _POSIX_SOURCE 1

int main() {

   int flags, status;

   flags=fcntl(fd,F_GETFL, 0);
   if flags == -1 {
      …
      exit(1);
   }

      flags |= O_NONBLOCK;
   status=fcntl(fd, F_SETFL, flags);
   …
   while(1) {
      c=getchar();
      if (c==EOF) {
         …  // test for errno and SIGNALs here ..
         break;
      }
      else
         putchar(c);
      }
   }
      flags &= ~O_NONBLOCK;
   status=fcntl(fd, F_SETFL, flags);
}
```

4. locking files .. fcntl() also flock() in BSD vs lockf() in SYSV.

5. **select()**
   5.1. For polling multiple fd's. -- select() blocks until IO is possible on fdset, then, each fd is checked to see if IO is possible using that fd.
   5.2. *may* be implemented as layer over poll() .. e.g. in BSD. Or, vice versa
   5.3. was not part of POSIX until recently.

6. **poll()**
   6.1. poll() is *mostly* identical to select() .. in terms of functionality ..|

7. **Code Example** – using select() ..

```
#define #_POSIX_SOURCE 1
#include <stdio.h>
#include <sys/types.h>
#include <sys/time.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>

char *rfn= "myfile.txt";

int main(int argc, char **argv) {
   fd_set rfd,wfd,efd;
   struct timeval tmout;

   int myfd1 = STDIN_FILENO;
   int myfd2 = 0;
   int max_fd = 0;
   int nrdy = 0, c = 0;
   while (1) {
      tmout.tv_sec=10;
      tmout.tv_usec=10000;

      FD_ZERO(&rfd);
      FD_ZERO(&wfd);
      FD_ZERO(&efd);
      FD_SET(myfd1,&rfds);
      FD_SET(myfd2,&rfds);
      maxfd=1;

      nrdy = select (maxfd, &rfd, &wfd, &efd, &tmout);

      if nrdy(<0){
         if(errno==EINTR) {
            continue; // select() not restarted, after
             // an interrupt.
         }

         fprintf(stderr, “select() failed: %s\n”, strerror(errno));
      }

if (nrdy == 0) {
   fprintf(stderr, “Timed Out.\n”);
   break;
}

if (FD_ISSET(myfd1,&rfd)) {
fprintf(stderr, “Can Read from fd=%d”, myfd1);

      while ((c=getchar()) != EOF) {
            if (c==’\n’) break;
}

if (c==EOF) break;

}
   }

exit(0);
}
```

2. Code Example .. using poll() ...

```
/*  CDD2app.c */

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/poll.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>

#define CMD1 1
#define CMD2 2
#define MYNUM 0x88888888
#define MYSTR "Eureka!"
#define MYSTR2 " Hello World!"
#define LONGSTR "This is a long string! ABCDEFGHIJKLMNOPQRSTUVWXYZ 0123456789"

main() {
    int fd, len, wlen;
    char str[128];
    int num, rnum;

    struct pollfd pollfd;

    strcpy(str, MYSTR);

    // open
    if((fd=open("/dev/CDD2/CDD2_a",O_RDWR|O_APPEND))== -1){
        fprintf(stderr,"ERR:open():%s\n",strerror(errno));
        exit(0);
    }

    // write
    wlen = strlen(str);
    if ((len = write(fd, str, wlen)) == -1) {
        fprintf(stderr,"ERR:write():%s\n",strerror(errno));
        exit(1);
    }

    // read
    if ((len = read(fd, str, sizeof(str))) == -1) {
        fprintf(stderr,"ERR:read():%s\n",strerror(errno));
        exit(1);
    }
    fprintf(stdout, "%s\n", str);

    // write
    wlen = strlen(str);
    if ((len = write(fd, str, wlen)) == -1) {
        fprintf(stderr,"ERR:write():%s\n",strerror(errno));
        exit(1);
    }

    // write2
    strcpy(str, MYSTR2);
    wlen = strlen(str);
    if ((len = write(fd, str, wlen)) == -1) {
        fprintf(stderr,"ERR:write():%s\n",strerror(errno));
        exit(1);
    }
    // read
    if ((len = read(fd, str, sizeof(str))) == -1) {
        fprintf(stderr,"ERR:read():%s\n",strerror(errno));
        exit(1);
    }
    fprintf(stdout, "%s\n", str);

    // write2
    strcpy(str, LONGSTR);
    wlen = strlen(str);
    if ((len = write(fd, str, wlen)) == -1) {
        fprintf(stderr,"ERR:write():%s\n",strerror(errno));
        exit(1);
    }

    // lseek()  .. -ve offset from start of file
    if(lseek(fd, -999, SEEK_SET) <0) {      // -ve test
    // if(lseek(fd, -999, SEEK_CUR) <0) {   // -ve test
    // if(lseek(fd, -999, SEEK_END) <0) {   // -ve test
        fprintf(stderr,"ERR:lseek():%s\n",strerror(errno));
    }

    // lseek()  .. +ve offset beyond end of file
    if(lseek(fd, 999, SEEK_SET) <0) {       // -ve test
    // if(lseek(fd, 999, SEEK_CUR) <0) {    // -ve test
    // if(lseek(fd, 999, SEEK_END) <0) {    // -ve test
        fprintf(stderr,"ERR:lseek():%s\n",strerror(errno));
    }

    // lseek()  .. +test for -ve offset
    // if(lseek(fd,-1, SEEK_SET) <0) {      // -ve test
    // if(lseek(fd,-1, SEEK_CUR) <0) {
    // if(lseek(fd,0, SEEK_END) <0) {           // +ve test
    if(lseek(fd,-10, SEEK_END) <0) {            // +ve test
        fprintf(stderr,"ERR:lseek():%s\n",strerror(errno));
    }

    // read .. to show data consumption.
    if ((len = read(fd, str, sizeof(str))) == -1) {
        fprintf(stderr,"ERR:read():%s\n",strerror(errno));
        exit(1);
    }
    fprintf(stdout, "%s\n", str);

    // write2 .. data is consumed .. do a fresh write here.
    strcpy(str, LONGSTR);
    wlen = strlen(str);
    if ((len = write(fd, str, wlen)) == -1) {
        fprintf(stderr,"ERR:write():%s\n",strerror(errno));
        exit(1);
    }

    // lseek()  .. +test for +ve offset
    if(lseek(fd,15, SEEK_SET) <0) {         // +ve test
    // if(lseek(fd,15, SEEK_CUR) <0) {
    // if(lseek(fd,15, SEEK_END) <0) {      // -ve test
        fprintf(stderr,"ERR:lseek():%s\n",strerror(errno));
    }

    // read .. data is available .. do a read.
    if ((len = read(fd, str, sizeof(str))) == -1) {
        fprintf(stderr,"ERR:read():%s\n",strerror(errno));
        exit(1);
    }
    fprintf(stdout, "%s\n", str);

    close(fd);

    // open
    if((fd = open("/dev/CDD2/CDD2_b", O_RDWR | O_APPEND)) == -1) {
        fprintf(stderr,"ERR:open():%s\n",strerror(errno));
        exit(0);
    }


                           // zero'd pollfd
    memset(&pollfd, 0, sizeof(struct pollfd));
    pollfd.fd=fd;                                   // init
    pollfd.events |= POLLIN | POLLOUT;              // init

    if ((num = poll(&pollfd,1,-1)) < 0) {
        fprintf(stderr,"ERR:poll():%s\n",strerror(errno));
        exit(1);
    }
    else if (num) {     // poll() returned an event

        memset(str,0,sizeof(str));
        if (pollfd.revents & POLLOUT) {

            // write
            strcpy(str, MYSTR);
            wlen = strlen(str);
            if ((len = write(fd, str, wlen)) == -1) {
                fprintf(stderr,"ERR:write():%s\n",
strerror(errno));
                exit(1);
            }

            // write
            strcpy(str, MYSTR2);
            wlen = strlen(str);
            if ((len = write(fd, str, wlen)) == -1) {
                fprintf(stderr,"ERR:write():%s\n",
            strerror(errno));
                exit(1);
            }
        }

        if(pollfd.revents & POLLIN) {

            // read..can use "while" loop to consume chars
            if ((len = read(fd, str, sizeof(str))) == -1) {
                fprintf(stderr,"ERR:read():%s\n",
strerror(errno));
                exit(1);
            }
            fprintf(stdout, "%s\n", str);

        }
    }
    close(fd);
}
```

8. Further Reading :

http://linux.about.com/library/cmd/blcmdl2_poll.htm
http://linux.about.com/od/commands/l/blcmdl2_select.htm
http://linux.about.com/library/cmd/blcmdl2_select_tut.htm

Filesystems and Inodes

File Systems

.. has boot blk, super blk, inodes, data blks.
.. each file and directory has a corresponding inode.

Inodes are information nodes.

.. `stat()` system call reads information in inode

Slide #2-15

Notes

1. **File System** .. different fs have different approaches, however, fs's have similar inode implementation.
   1.1. Consists of boot block, super block, inodes, and data blocks.
   1.2. Each file and directory has a corresponding inode. An inode contains metadata information, and specifies where the data for the file can be found in the file system.

1.3. A directory is a binary file which maps file names and inode numbers.

2. **stat()**

2.1. inodes store file creation, modification time and time of last access, in st_ctime, st_mtime and st_atime fields that are of type `struct time_t`.

2.2. To convert to human readable form use .. `static char * ctime(time_t timeval)`;

stat(), opendir() and **readdir()**

```
int stat(char *path, struct stat* sbuf);
```

Slide #2-16

Notes

1. **stat()**
   1.1. gets metadata and status information on file... creation/modification/last-access dates, size.
   1.2. stat() .. structure

```
struct  stat {
    dev_t   st_dev;
                       /* rsrvd for dev expnd, */
    long    st_pad1[3];
                       /* sysid definition */
    ino_t   st_ino;
    Mode_t  st_mode;
    nlink_t st_nlink;
    uid_t   st_uid;
    gid_t   st_gid;
    dev_t   st_rdev;
    long    st_pad2[2];
    off_t   st_size;
         /* reserve for future off_t expansion */
    long    st_pad3;
    time_t  st_atime;
    time_t  st_mtime;
    time_t  st_ctime;
    long    st_blksize;
    blkcnt_t st_blocks;
    char    st_fstype[_ST_FSTYPSZ];
    long    st_pad4[8]; /* expansion area */
};
```

2. directory files..

2.1. are files.

2.2. They are binary files, where the permission bit S_ISDIR is set.
It consists of filename and inode number pairs.

2.3. When a file is open()ed, the kernel looks through the current directory “file” (“.” .. available in the u_area) for an entry with the “filename” and uses the inode# to access the contents of the file.

2.4. opendir()

2.4.1. `open`’s directory files;

2.4.2. returns DIR *;

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

DIR *opendir(const char *dirname);
```

2.5. readdir()

2.5.1. reads DIR *;

2.5.2. returns ptr to struct direct (BSD) or struct dirent (POSIX)

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

struct dirent *readdir(DIR *dirp);
```

2.5.3. The following sample code will search the current directory for the entry name:

```
dirp = opendir(".");
while ((dp = readdir(dirp)) != NULL)
     if (strcmp(dp->d_name, name) == 0) {
          closedir(dirp);
          return FOUND;
     }
closedir(dirp);
return NOT_FOUND;
```

3. **Code Example** – *using* `stat()` ..

```
#define #_POSIX_SOURCE 1

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

char *rfn= “myfile.txt”;
extern void print_statinfo(struct stat *statinfo);

int main(int argc, char **argv) {
   struct stat statinfo;
   char fn[1024];

   fprintf(stderr, “Enter File Name:”);
   while (gets(fn)) {
      if(stat(fn, &statinfo) != 0) {
         fprintf(stderr, “stat() failed: %s\n”, strerror(errno));
      else {
         fprintf(stderr, “stat() info: %s\n”, fn);
         print_statinfo(&statinfo);
      }
   }

   exit(0);
}

void print_statinfo(struct stat *statinfo) {
   printf(“\tino_t  st_ino ==%10lu\n”, statinfo->st_ino);
   printf(“\tmode_t st_mode==%07o\n”,  statinfo->st_mode);
   printf(“\toff_t  st_size==%10lu\n”, statinfo->st_size);
   printf(“\ttime_t st_atime==%10d\n”, statinfo->st_atime);
   printf(“\ttime_t st_mtime==%10d \n”,statinfo->st_mtime);
   printf(“\ttime_t st_ctime==%10d \n”,statinfo->st_ctime);
   printf(“\tlong st_blksize==%9ld\n”, statinfo->st_blksize);
   printf(“\tlong st_blocks==%9ld\n”,  statinfo->st_blocks);
}
```

Alternatively, use the ctime() function to get time output in a human readable format

```
printf(“\ttime_t st_atime==%.24s\n”, ctime(&statinfo->st_atime));
printf(“\ttime_t st_ctime==%.24s\n”, ctime(&statinfo->st_ctime));
printf(“\ttime_t st_mtime==%.24s\n”, time(&statinfo->st_mtime));
```

- Chapter 2

File and File IO

Terms & Concepts Worksheet

Table #1 ("Basic")

1. Files.	Named "Sequence" of bits. Storage location. On disk.
2. Task Structure (u_area.)	Kernel-Maintained Area for info .. in process context ..
3. File Table	In kernel. Resource table .. currently open in <u>system</u> .. "System Context".
4. fd table	In u_area.Usage table .. currently for a <u>process</u> .. "Process Context".

Table #2 ("FILE * vs *fd*")

5. fd	a. fd .. file descriptor ..datatype '`int`' .. index# of the element. b. .. each `fd` => element contains reference to entry in Kernel File Table - if same file opened twice in same process .. two *fd*'s => two file table entries .. c. .. init by `open()`, and used by calls like `read(), write(), close()`

6. FILE *	FILE IO Interface -- FILE .. <u>user-space</u> structure. Allocated on heap by `fopen()`
	a. FILE struct **may** be declared as ..
	`typedef struct`
	`{ int cnt;` /* number of available characters in buffer */
	`unsigned char *ptr;` /* next character from/to here in buffer */
	`unsigned char *base;`/* the buffer */
	`unsigned char flag;` /* the state of the stream */
	`unsigned char fd;` /* UNIX System file descriptor */
	`} FILE;`
	b.init by `fopen();fgets(),getchar();putc(),puts(),putchar()`
	c. At a high level, `fopen()` does the following:
	- `malloc()` for struct FILE
	- prepares flags and calls `open()`
	- if open() successful, populate `fd` member in FILE struct
	- else, deallocate .. FILE struct
	- return .. FILE *

Table #3 ("System call `open()`")

7. `open()`	a. `int fd=open(path, flags, mode)` .. takes 3 parameters. `fd` .. returns index# of *lowest available* element in u_area *fd* table. `path` .. full name for file, includes absolute/relative path `flag` .. indicates file characteristics .. being opened for read/write etc .. e.g. O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_TRUNC, O_CREAT `mode` .. for file creation file ONLY.. refers to **permission modes** for file, at creation.

8. `open()` flags vs. `fopen()`	`fopen("myfl","r") .. open("myfl",O_RDONLY);`
	`fopen("myfl","w") .. open("myfl",O_WRONLY|O_CREAT|O_TRUNC)`

9. `mode_t`	a. perm. bits .. single int .. each bit=>special perms. `drwxrwxrwx ..` b. `POSIX` data type .. similar to other "`_t`" data types, belong to "`int`" class c. can change permissions .. `chmod` is a wrapper on **`chmod()`** ..

	d. **chmod()** args e.g. S_I*mabc* .. *m*=>R,W or X; *abc*=>USR,GRP or OTH
10. umask()call	a. .. **bits masked out** of permission bits for all files, created by the process. **turnoff bits**. b. .. Every process has a umask int .. stored in the u_area. default 022.

Table #4 (“read(), write() and lseek() call”)

11. read()call	a. ssize_t read(*fd*,void *buf,size_t nbytes) .. 3 parameters. b. Binary IO. Lowest-level system call. Reads bytes from a file (.. or any device). c. Returns #bytes read. ‘0’ on End-Of-File (.. sets error status.)
12. write()call	a. ssize_t write(*fd*,void *buf,size_t nbytes) .. 3 parameters. b. Binary IO. Like read(). Lowest-level system call. Writes bytes from file (..or device). c. Returns #bytes written. Usually same as requested. ‘-1’ on error ..
13. lseek()call	a. off_t lseek(*fd*,off_t offset,int whence) .. 3 parameters. b. position IO ptr .. from whence .. SEEK_SET, SEEK_CUR, SEEK_END c. offset can be -ve.
14. dup()call	a. int dup(*fd*) /* creates new *fd*. Returns lowest avail. *fd*. **<u>same</u>** Kernel File Entry */ b. int dup2(*oldfd*,*newfd*) /* closes old *fd*. Returns new *fd*. **<u>same</u>** Kernel File Entry */

Table #5 (“More on FILE datatype”)

15. stdin stdout and stderr	a. stdin,stdout and stderr are FILE * structs; correspond to fd’s 0,1,2 b. 0,1,2 .. opened automatically; assigned to global vars stdin, stdout, stderr c. redirection .. close(0); open(fn,O_RDONLY); open() returns lowest fd .. 0 d. default buffering .. FILE (1 blkbuf); stdin,stdout(linebuf); stderr(No Buf)
16. Conversion	a. rfp=fdopen(fd, mode); /* allocate a new FILE * over an existing fd ..*/ b. fileno(rfp); /* macro to obtain corresponding fd ..*/
17. fflush() and setvbuf()	a. int fflush(FILE *fp); /* flush FILE * buffer ..*/ b. int setvbuf(FILE *fp, char *buf, int mode, size_t size); /*called before any IO. To change buf size or mode _IOFBF,_IOLBF,_IONBF ..*/

Table #6 (“blocking IO -- open(), fcntl() and select() calls”)

18. file attributes	a. blocking IO vs non-blocking IO .. O_NONBLOCK flag set when open()’ing file. b. locking files .. fcntl() also flock() <u>in BSD</u> vs lockf() <u>in SYSV</u>.
19. fcntl()	a. change attributes on fd’s already opened. e.g. blocking/non-blocking.
20. select()	a. For polling multiple *fd*’s. select() blocks until IO is possible on *fdset*. Then, chk each *fd* b. select() are implemented as layer over poll() in BSD. Vice-Versa in SYS V.

Table #7 (“stat(), opendir() and readdir() calls”)

21. directory files	a. directories are files .. binary files .. consisting of filename and inode number pairs.
22. opendir()	a. open()’s directory files; returns DIR *; used by readdir();
23. readdir()	a. reads DIR *; returns ptr to struct direct (BSD) or struct dirent (POSIX)
24. stat()	a. prints inode information .. creation/modification/last-access dates, size.

Further Reading
Robbins and Robbins
Chapters 2 and 3
http://www.linux-tutorial.info/modules.php?name=Tutorial&pageid=260

http://www.faqs.org/docs/kernel_2_4/lki-3.html
http://jan.netcomp.monash.edu.au/OS/l8_2.html
http://linux.about.com/od/commands/l/blcmdl_2a.htm
http://www.linuxjournal.com/node/6908
http://openit.disco.unimib.it/lxrfarina/source/include/sys/ioctl.h

Chapter 2

File and File IO

Assignment Questions

Questions:

2.1 Write a simple program to count the number of chars, words and lines in an ASCII text file – as a simple implementation of the 'wc' utility

2.2 Write a simple program to read the contents of one or more ASCII text file(s), and print it out – as a simple implementation of the 'cat' utility.

b) modify above to take the '-l' argument, that prints out each line prefixed by line# in the file.

2.3 Determine the limit of maximum number of open files, using sysconf().

2.4 Write a simple program to redirect both -- stdin and stdout, to files

a) stdin to read from a file,

b) stdout "echoes" stdin, and,

c) stderr to display #lines copied.

2.5 a) Write a simple program that is a simple version of the 'ls' utility. It will list the names of all files in "." -- current working directory.

Extra Credit

b) Modify 2.5 to take arguments for, filename or directory name, and, depending on whether the argument is a file or directory, it will list only the filename, or, the directory name followed by the contents of the directory.

c) '-l' .. long listing, and combines with 2.5.b, to provide your own implementation of 'ls –l <filenames>'

Assignment Hints

#2.1 Hint: counters for chars, words & lines.
words are white space delimited, lines are '\n' counted.

#2.2 Hint: a) use either fgets() or read() b)use counters for lines. lines are '\n' counted.

#2.4. Hint: Per Example .. use close() or dup() or dup2()

#2.5. Hint: Per Example .. use opendir() ..

Chapter 2

File and File IO

Useful Links

3. Linux Documentation and Links

1.4. Journalling File Systems .. http://www.linux-mag.com/id/1180

1.5. POSIX File IO .. http://lca2007.linux.org.au/talk/278

1.6. The 2006 Linux File Systems Workshop .. http://lwn.net/Articles/190222/

1.7. **Excellent Resource for Linux**.. http://www.linuxlinks.com/

1.8. **Kernel Source Browser** http://lxr.linux.no/source

1.9. **Getting a Linux Distribution** .. http://www.linux.org.uk/LinuxFTP.html

---

1.10. **Some Vendor Docs …**

1.10.1. http://www.intel.com/cd/ids/developer/asmo-na/eng/os/linux/index.htm

1.10.2. http://www.kernelhacking.org/links/index.htm

1.11. **Some More …**

1.11.1. http://www.linux.org/docs/index.html

1.11.2. http://www.kernelhacking.org/links/index.htm

1.11.3. http://www.faqs.org/faqs/linux/howto/index/

1.11.4. http://www.linux-tutorial.info/modules.php?name=Tutorial&pageid=260

1.11.5. http://www.faqs.org/docs/kernel_2_4/lki-3.html

1.11.6. http://jan.netcomp.monash.edu.au/OS/l8_2.html

1.11.7. http://linux.about.com/od/commands/l/blcmdl_2a.htm

1.11.8. http://www.linuxjournal.com/node/6908

1.11.9. http://openit.disco.unimib.it/lxrfarina/source/include/sys/ioctl.h