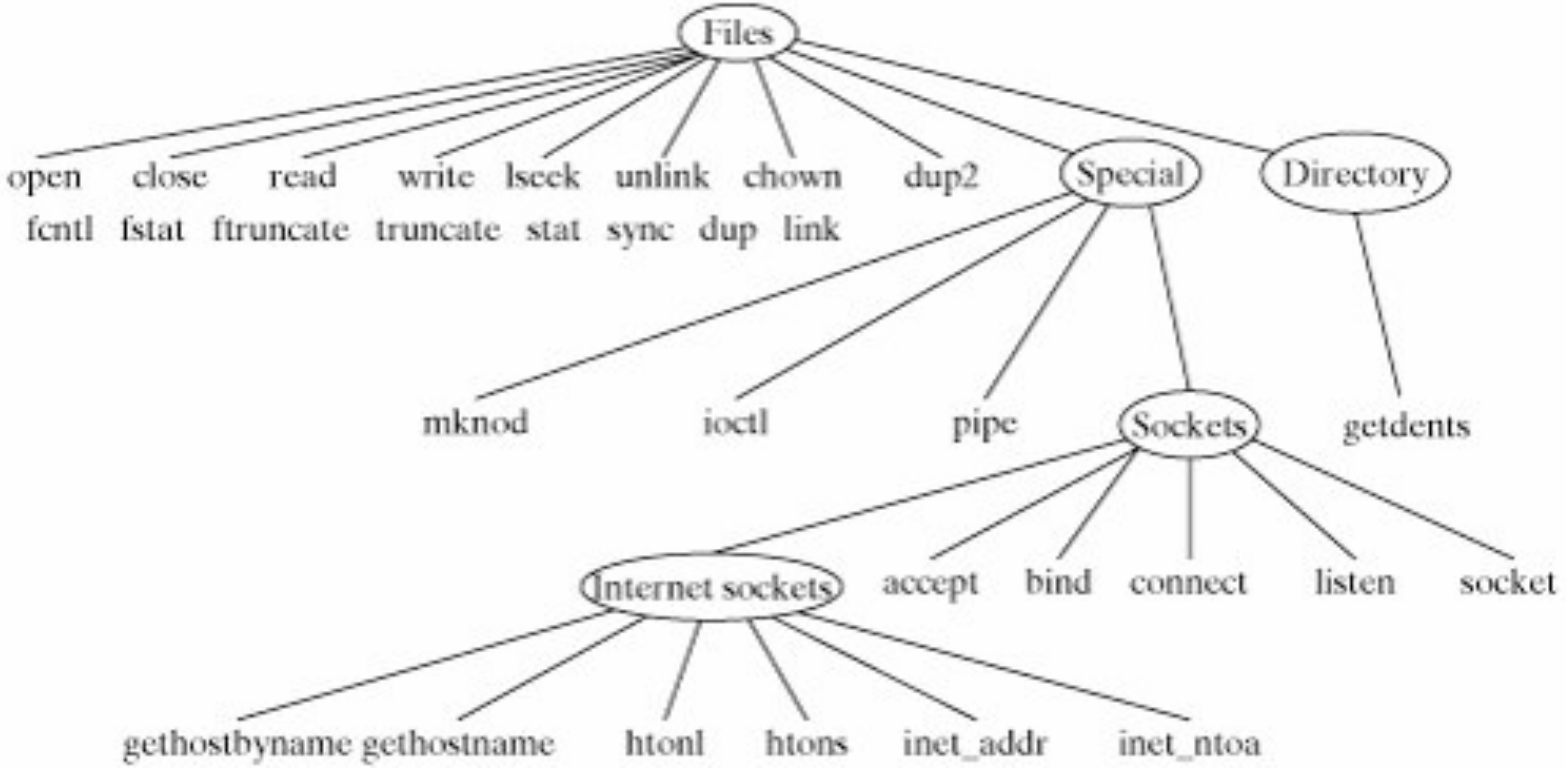
System calls

- We will investigate several issues related to system calls.
  - Read chapter 12 of the book
- Linux system call categories
  - file management
  - process management
  - error handling
  - note that these categories are loosely defined and much is behind included, e.g. communication. Why?

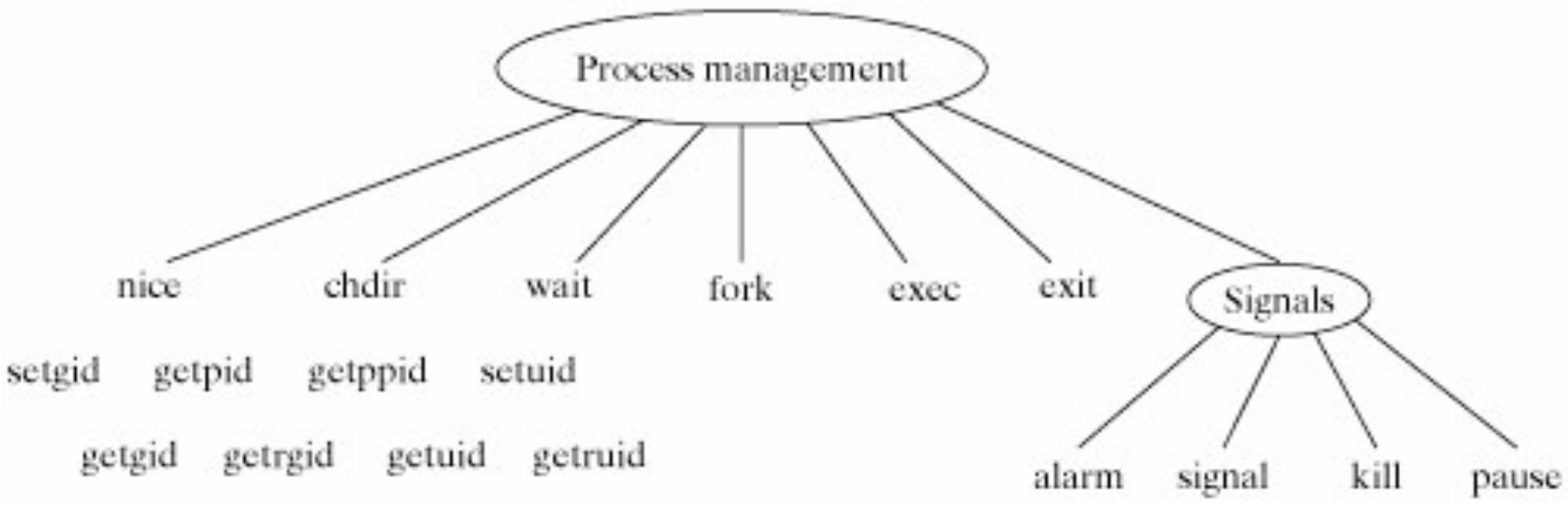
System calls

- File management system call hierarchy
  - you may not see some topics as part of “file management”, e.g., sockets



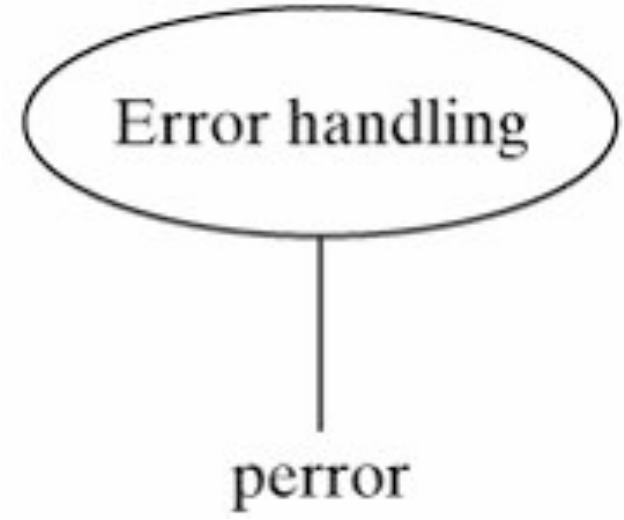
System calls

- Process management system call hierarchy



System calls

- Error handling hierarchy



Error Handling

- Anything can fail!
  - System calls are no exception
  - Try to read a file that does not exist!
- Error number: **errno**
  - every process contains a global variable *errno*
  - *errno* is set to 0 when process is created
  - when error occurs *errno* is set to a specific code associated with the error cause
    - trying to open file that does not exist sets *errno* to 2

Error Handling

- error constants are defined in errno.h
  - here are the first few of errno.h on OS X 10.6.4

```
#define EPERM     1     /* Operation not permitted */
#define ENOENT    2     /* No such file or directory */
#define ESRCH     3     /* No such process */
#define EINTR     4     /* Interrupted system call */
#define EIO       5     /* Input/output error */
#define ENXIO     6     /* Device not configured */
#define E2BIG     7     /* Argument list too long */
#define ENOEXEC   8     /* Exec format error */
#define EBADF     9     /* Bad file descriptor */
#define ECHILD    10    /* No child processes */
#define EDEADLK   11    /* Resource deadlock avoided */
```

Error Handling

- common mistake for displaying errno
  - from Linux errno man page:

```
if (somecall() == -1) {
    printf("somecall() failed\n");
    if (errno == ...) { ... }
}
```

where errno no longer needs to have the value it had upon return from somecall() (i.e., it may have been changed by the **printf**()). If the value of errno should be preserved across a library call, it must be saved:

```
if (somecall() == -1) {
    int errsv = errno;
    printf("somecall() failed\n");
    if (errsv == ...) { ... }
}
```

Error Handling

- Description of the perror () system call.
  - Library Function: void *perror* (char* *str*)
    - perror () displays the string *str*, followed by a colon, followed by a description of the last system call error.
      - If there is no error to report, it displays the string "Error 0." Actually, perror () isn't a system call, it is a standard C library function.

example from text

```
$ cat showErrno.c
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
main ()
{
 int fd;
  /* Open a nonexistent file to cause an error */
 fd = open ("nonexist.txt", O_RDONLY);
 if (fd == -1) /* fd == -1 =, an error occurred */
   {
    printf ("errno = %d\n", errno);
    perror ("main");
   }
 fd = open ("/", O_WRONLY); /* Force a different error */
 if (fd == -1)
   {
    printf ("errno = %d\n", errno);
    perror ("main");
   }
 /* Execute a successful system call */
 fd = open ("nonexist.txt", O_RDONLY | O_CREAT, 0644);
 printf ("errno = %d\n", errno); /* Display after successful call */
 perror ("main");
 errno = 0; /* Manually reset error variable */
 perror ("main");
}
```

- output from example above

```
$ ./showErrno                  ...run the program.
errno = 2
main: No such file or directory
errno = 21
main: Is a directory
errno = 29        ...even after a successful call
main: Illegal seek
main: Success    ...after we reset manually.
$ _
```

File Management

- What is a file?
  - In unix file types go beyond just your regular files on disk
  - The file types (and symbols) are:
    - Regular files ( )
    - Directories (d)
    - Links (l)
    - Special files (c)
    - Sockets (s)
    - Named pipes (p)

File Management

- Examples
  - file is opened, a *file descriptor* is returned, after certain operations the file is closed

```
int fd; /* File descriptor */
...
fd = open (fileName, ...); /* Open file, return file descriptor */
if (fd == -1) {  /* deal with error condition */ }
...
fcntl (fd, ...); /* Set some I/O flags if necessary */
...
read (fd, ...);  /* Read from file */
...
write (fd, ...); /* Write to file */
...
lseek (fd, ...); /* Seek within file*/
...
close (fd); /* Close the file, freeing file descriptor */
```

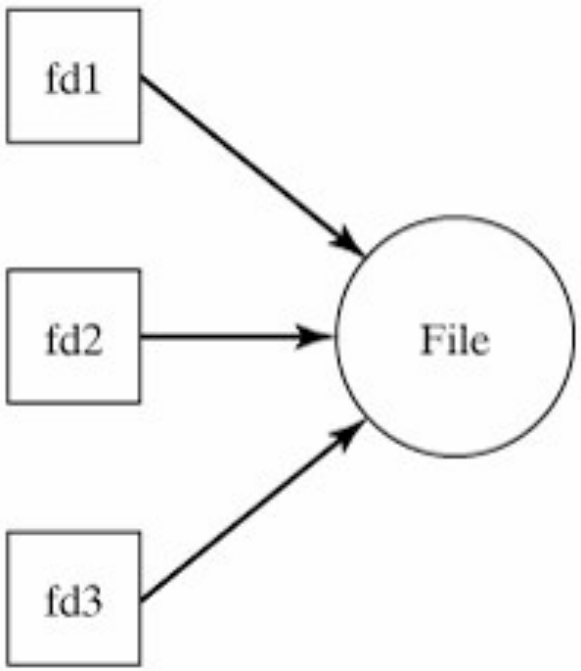
  - close the file, even though you know....

File Management

- File descriptors
  - sequential numbers, starting with 0
  - first three descriptors are
    - 0 = stdin
    - 1 = stdout
    - 2 = stderr
  - when reference to file is closed, the fd is freed to be reassigned

File Management

- A file may have multiple file discriptors



File Management

- File descriptor properties include:
  - file pointer indicating the offset in the file where it is reading/ writing
  - flag that indicates whether the descriptor should be automatically closed if the process calls *exec*()
  - flag that indicates whether output should be appended to the end of file

File Management

- File descriptor properties for special files include:
  - flag that indicates whether a process should block on input from a file if it does not currently contain any input
  - A number that indicates a process ID or process group that should be sent a SIGIO signal if input becomes available
    - a SIGIO signal indicates that I/O is now possible

File Management

- Linux basic I/O operations
  - open Opens/creates a file.
  - read Reads bytes from a file into a buffer.
  - write Writes bytes from a buffer to a file.
  - lseek Moves to a particular offset in a file.
  - close Closes a file.
  - unlink Removes a file.

Example: Reverse

- Write a Utility: *reverse* -c [fileName]
  - reverse reverses the lines of its input and displays them to standard output.
  - If no file name is specified, reverse reverses its standard input.
  - When the -**c** option is used, reverse also reverses the characters in each line.

Example: Reverse

- Examples of its application

```
$ gcc reverse.c -o reverse    ...compile the program.
$ cat test                    ...list the test file.
Christmas is coming,
The days that grow shorter,
Remind me of seasons I knew in the past.
$ ./reverse test              ...reverse the file.
Remind me of seasons I knew in the past.
The days that grow shorter,
Christmas is coming,
$ ./reverse -c test           ...reverse the lines too.
.tsap eht ni wenk I snosaes fo em dnimeR
,retrohs worg taht syad ehT
,gnimoc si samtsirhC
$ cat test | ./reverse        ...pipe output to "reverse".
Remind me of seasons I knew in the past.
The days that grow shorter,
Christmas is coming,
$ _
```

Example: Reverse

- How reverse works
  - it makes two passes over its input.
    - During the first pass, it notes the starting offset of each line in the file and stores this information in an array.
    - During the second pass, it jumps to the start of each line in reverse order, copying it from the original input file to its standard output.
  - If no file name is specified on the command line, reverse reads from its standard input during the first pass and copies it into a temporary file for the second pass.
  - When the program is finished, the temporary file is removed.

Figure 12-9. Description of algorithm used in reverse.c.

Step	Action	Functions	System calls
1	Parse command line.	parseCommandLine, processOptions	open
2	If reading from standard input, create temporary file to store input; otherwise open input file for reading.	pass1	open
3	Read from file in chunks, storing the starting offset of each line in an array. If reading from standard input, copy each chunk to the temporary file.	pass1, trackLines	read, write
4	Read the input file again, backward, copying each line to standard output. Reverse the line if the **-c** option was chosen.	pass2, processLine, reverseLine	lseek
5	Close file, removing it if it was a temporary file.	pass2	close

```
 1  #include <fcntl.h>  /* For file mode definitions */
 2  #include <stdio.h>
 3  #include <stdlib.h>
 4
 5
 6  /* Enumerator */
 7  enum { FALSE, TRUE }; /* Standard false and true values */
 8  enum { STDIN, STDOUT, STDERR }; /* Standard I/O channel indices */
 9
10
11  /* #define Statements */
12  #define BUFFER_SIZE    4096   /* Copy buffer size */
13  #define NAME_SIZE      12
14  #define MAX_LINES      100000 /* Max lines in file */
15
16
17  /* Globals */
18  char *fileName = 0; /* Points to file name */
19  char tmpName [NAME_SIZE];
20  int charOption = FALSE; /* Set to true if -c option is used */
21  int standardInput = FALSE; /* Set to true if reading stdin */
22  int lineCount = 0; /* Total number of lines in input */
23  int lineStart [MAX_LINES]; /* Store offsets of each line */
24  int fileOffset = 0; /* Current position in input */
25  int fd; /* File descriptor of input */
```

```
27  /*************************************/
28
29  main (argc, argv)
30
31  int argc;
32  char* argv [];
33
34  {
35    parseCommandLine (argc,argv); /* Parse command line */
36    pass1 (); /* Perform first pass through input */
37    pass2 (); /* Perform second pass through input */
38    return (/* EXITSUCCESS */ 0); /* Done */
39  }
40
41  /*************************************/
```

```
43  parseCommandLine (argc, argv)
44
45  int argc;
46  char* argv [];
47
48  /* Parse command-line arguments */
49
50  {
51    int i;
52
53    for (i= 1; i < argc; i++)
54      {
55        if(argv[i][0] == '-')
56          processOptions (argv[i]);
57        else if (fileName == 0)
58          fileName= argv[i];
59        else
60          usageError (); /* An error occurred */
61      }
62
63    standardInput = (fileName == 0);
64  }
```

```
68  processOptions (str)
69
70  char* str;
71
72  /* Parse options */
73
74  {
75    int j;
76
77    for (j= 1; str[j] != 0; j++)
78      {
79        switch(str[j]) /* Switch on command-line flag */
80          {
81            case 'c':
82              charOption = TRUE;
83              break;
84
85            default:
86              usageError();
87              break;
88          }
89      }
90  }
```

```
 92  /*****************************************/
 93
 94  usageError ()
 95
 96  {
 97    fprintf (stderr, "Usage: reverse -c [filename]\n");
 98    exit (/* EXITFAILURE */ 1);
 99  }
100
101  /*****************************************/
102
```

```
103  pass1 ()
104
105  /* Perform first scan through file */
106
107  {
108    int tmpfd, charsRead, charsWritten;
109    char buffer [BUFFER_SIZE];
110
111    if (standardInput) /* Read from standard input */
112      {
113        fd = STDIN;
114        sprintf (tmpName, ".rev.%d",getpid ()); /* Random name */
115        /* Create temporary file to store copy of input */
116        tmpfd = open (tmpName, O_CREAT | O_RDWR, 0600);
117        if (tmpfd == -1) fatalError ();
118      }
119    else /* Open named file for reading */
120      {
121        fd = open (fileName, O_RDONLY);
122        if (fd == -1) fatalError ();
123      }
124
125    lineStart[0] = 0; /* Offset of first line */
126
127    while (TRUE) /* Read all input */
128      {
129        /* Fill buffer */
130        charsRead = read (fd, buffer, BUFFER_SIZE);
131        if (charsRead == 0) break; /* EOF */
132        if (charsRead == -1) fatalError (); /* Error */
133        trackLines (buffer, charsRead); /* Process line */
134        /* Copy line to temporary file if reading from stdin */
135        if (standardInput)
136          {
137            charsWritten = write (tmpfd, buffer, charsRead);
138            if(charsWritten != charsRead) fatalError ();
139          }
140      }
141
142    /* Store offset of trailing line, if present */
143    lineStart[lineCount + 1] = fileOffset;
144
145    /* If reading from standard input, prepare fd for pass2 */
146    if (standardInput) fd = tmpfd;
147  }
```

```
103  pass1 ()
104
105  /* Perform first scan through file */
106
107  {
108    int tmpfd, charsRead, charsWritten;
109    char buffer [BUFFER_SIZE];
110
111    if (standardInput) /* Read from standard input */
112      {
113        fd = STDIN;
114        sprintf (tmpName, ".rev.%d",getpid ()); /* Random name*/
115        /* Create temporary file to store copy of input */
116        tmpfd = open (tmpName, O_CREAT | O_RDWR, 0600);
117        if (tmpfd == -1) fatalError ();
118      }
119    else /* Open named file for reading */
120      {
121        fd = open (fileName, O_RDONLY);
122        if (fd == -1) fatalError ();
123      }
124
125    lineStart[0] = 0; /* Offset of first line */
126
```

```
103  pass1 ()
104
105  ...

126
127    while (TRUE) /* Read all input */
128      {
129        /* Fill buffer */
130        charsRead = read (fd, buffer, BUFFER_SIZE);
131        if (charsRead == 0) break; /* EOF */
132        if (charsRead == -1) fatalError (); /* Error */
133        trackLines (buffer, charsRead); /* Process line */
134        /* Copy line to temporary file if reading from stdin */
135        if (standardInput)
136          {
137            charsWritten = write (tmpfd, buffer, charsRead);
138            if(charsWritten != charsRead) fatalError ();
139          }
140      }
141
142    /* Store offset of trailing line, if present */
143    lineStart[lineCount + 1] = fileOffset;
144
145    /* If reading from standard input, prepare fd for pass2 */
146    if (standardInput) fd = tmpfd;
147  }
```

```
151  trackLines (buffer, charsRead)
152
153  char* buffer;
154  int charsRead;
155
156  /* Store offsets of each line start in buffer */
157
158  {
159    int i;
160
161    for (i = 0; i < charsRead; i++)
162      {
163        ++fileOffset; /* Update current file position */
164        if (buffer[i] == '\n') lineStart[++lineCount] = fileOffset;
165      }
166  }
```

```
170  int pass2 ()
171
172  /* Re-Scan input file, display lines in reverse*/
173
174  {
175    int i;
176
177    for (i = lineCount - 1; i >= 0; i--)
178      processLine (i);
179
180    close (fd); /* Close input file */
181    if (standardInput) unlink (tmpName);
                          /* Remove temp file */
182  }
183
```

```
186  processLine (i)
187
188  int i;
189
190  /* Read a line and display it */
191
192  {
193    int charsRead;
194    char buffer [BUFFER_SIZE];
195
196    lseek (fd, lineStart[i], SEEK_SET); /* Find line and read */
197    charsRead = read (fd, buffer, lineStart[i+1] - lineStart[i]);
198    /* Reverse line if -c option was selected */
199    if (charOption) reverseLine (buffer, charsRead);
200    write (1, buffer, charsRead); /* Write it to stdout */
201  }
```

```
205  reverseLine (buffer, size)
206
207  char* buffer;
208  int size;
209
210  /* Reverse all the characters in the buffer */
211
212  {
213    int start = 0, end = size - 1;
214    char tmp;
215
216    if (buffer[end] == '\n') --end; /* Leave trailing newline */
217
218    /* Swap characters in a pairwise fashion */
219    while (start < end)
220      {
221        tmp = buffer[start];
222        buffer[start] = buffer[end];
223        buffer[end] = tmp;
224        ++start; /* Increment start index */
225        --end; /* Decrement end index */
226      }
227  }
```

```
231  fatalError ()
232
233  {
234    perror ("reverse: "); /* Describe error */
235    exit (1);
236  }
```

System Call: open()

- int open (char* fileName, int mode [, int permissions])
  - open () allows you to open or create a file for reading and/or writing.
  - fileName is an absolute or relative pathname and mode is a bitwise or'ing of a read/write flag together with zero or more miscellaneous flags.
  - permissions is a number that encodes the value of the file's permission flags, and should only be supplied when a file is being created. It is usually written using octal encoding.
    - The permissions value is affected by the process's umask value. The values of the predefined read/write and miscellaneous flags are defined in "/usr/include/fcntl.h". The read/write flags are as follows:

System Call: open()

- Read/write flags:

FLAG	MEANING
O_RDONLY	Open for read-only.
O_WRONLY	Open for write-only.
O_RDWR	Open for read and write.

System Call: open()

- Miscellaneous flags:

  O_APPEND: Position the file pointer at the end of the file before each write ().

  O_CREAT: If the file doesn't exist, create the file, and set the owner ID to the process's effective user ID. The umask value is used when determining the initial permission flag settings.

  O_EXCL: If O_CREAT is set and the file exists, then open () fails.

System Call: open()

- Miscellaneous flags cont.:

  O_NONBLOCK or O_NDELAY: This setting works only for named pipes. If set, an open for read-only will return immediately, regardless of whether the write end is open, and an open for write-only will fail if the read end isn't open. If clear, an open for read-only or write-only will block until the other end is also open.

  O_TRUNC: If the file exists, it is truncated to length zero.

- open () returns a non-negative file descriptor if successful; otherwise, it returns -1.

Creating a file

- Use the O_CREAT flag as part of the mode flags, and supply the initial file permission flag settings as an octal value

```
114   sprintf (tmpName, ".rev.%d", getpid ()); /*Random name*/
115   /* Create temporary file to store copy of input */
116   tmpfd = open (tmpName, O_CREAT | O_RDWR, 0600);
117   if (tmpfd == -1) fatalError ();
```

  - getpid () returns the process's ID number (PID), which is guaranteed to be unique.
  - note the temp file is a hidden file

Opening a file

- Open existing file
  - Specify the mode flags only

```
121     fd = open (fileName, O_RDONLY);
122     if (fd == -1) fatalError ();
```

  - other more complicated flag settings for open (), such as O_NONBLOCK, are intended for use with the pipes, sockets, and STREAMS

System call read()

- ssize_t read (int fd, void* buf, size_t count)
  - [Note: This synopsis describes how read () operates when reading a regular file. Reading from special files comes later]
  - read () copies count bytes from file (referenced by file descriptor fd) into the buffer buf. The bytes are read from the current file position, which is then updated accordingly.
  - read () copies as many bytes from the file as it can, up to the number specified by count, and returns the number of bytes actually copied. If a read () is attempted after the last byte has already been read, it returns 0, which indicates end-of-file.
  - If successful, read () returns the number of bytes that it read; otherwise, it returns -1.

System call read()

- ssize_t read (int fd, void* buf, size_t count)
  - example

```
130     charsRead = read (fd, buffer, BUFFER_SIZE);
131     if (charsRead == 0) break; /* EOF */
132     if (charsRead == -1) fatalError (); /* Error */
```

System call write()

- ssize_t write (int fd, void* buf, size_t count)
  - [Note: This synopsis describes how write () operates when writing to a regular file. Writing to special files comes later]
  - write () copies count bytes from a buffer buf to the file referenced by the file descriptor fd. The bytes are written at the current file position, which is then updated accordingly. If the O_APPEND flag was set for fd, the file position is set to the end of the file before each write.
  - write () copies as many bytes from buffer as it can, up to the number specified by count, and returns the # of bytes actually copied. Always check the return value. If the return value isn't *count*, the disk probably filled up and no space was left.
  - If successful, write () returns the number of bytes that were written; otherwise, it returns -1.

System call write()

- ssize_t write (int fd, void* buf, size_t count)
  - Example
    - The write () system call performs low-level output, and has none of the formatting capabilities of printf (). The benefit of write () is that it bypasses the additional layer of buffering supplied by the C library functions, and is therefore very fast.

```
134     /* Copy line to temp file if reading standard input*/
135     if (standardInput)
136       {
137         charsWritten = write (tmpfd, buffer, charsRead);
138         if (charsWritten != charsRead) fatalError ();
139       }
```

System call lseek()

- off_t lseek (int fd, off_t offset, int mode)
  - lseek () allows to move in a file by changing a descriptor's current file position. fd is the file descriptor, offset is a long integer, and mode describes how offset should be interpreted.
  - The three possible values of mode are defined in "/usr/include/stdio.h," and have the following meaning:
    - SEEK_SET: offset is relative to the start of the file.
    - SEEK_CUR: offset is relative to the current file position.
    - SEEK_END: offset is relative to the end of the file.
  - lseek () fails if you try to move before the start of the file.
  - If successful, lseek () returns the current file position; otherwise, it returns -1.

System call lseek()

- off_t lseek (int fd, off_t offset, int mode)
  - example: Lines 196..197 seek to the start of a line and then read in all of its characters. Note that the number of characters to read is calculated by subtracting the start offset of the next line from the start offset of the current line.

```
196  lseek (fd, lineStart[i], SEEK_SET);
                  /* Find line & read it */
197  charsRead = read (fd,buffer,lineStart[i+1]-lineStart[i]);
```

  - If you want to find out your current location without moving, use an offset value of zero relative to the current position:

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

System call close()

- int close (int fd)
  - close () frees the file descriptor fd. If fd is the last file descriptor associated with a particular open file, the kernel resources associated with the file are deallocated.
  - When a process terminates, all of its file descriptors are automatically closed, but it's better programming practice to close a file when you're done with it. If you close a file descriptor that's already closed, an error occurs.
  - If successful, close () returns zero; otherwise, it returns -1.

System call unlink()

- int unlink (const char* fileName)
  - unlink () removes the hard link from the name fileName to its file.
  - If fileName is the last link to the file, the file's resources are deallocated. In this case, if any process's file descriptors are currently associated with the file, the directory entry is removed immediately but the file is only deallocated after all of the file descriptors are closed. This means that an executable file can unlink itself during execution and still continue to completion.
  - If successful, unlink () returns zero; otherwise, it returns -1.

System call stat()

- int stat (const char* *name*, struct stat* *buf*)
  - fills the buffer *buf* with information about the file name. The stat structure is defined in "/usr/include/sys/stat.h".
- int lstat (const char* *name*, struct stat* *buf*)
  - returns information about a symbolic link itself rather than the file it references.
- int fstat (int *fd*, struct stat* *buf*)
  - performs the same function as stat (), except that it takes the file descriptor of the file to be stat'ed as its first parameter.

System call stat()

- Members of structure **stat**:
  - st_dev the device number
  - st_ino the inode number
  - st_mode the permission flags
  - st_nlink the hard link count
  - st_uid the user ID
  - st_gid the group ID
  - st_size the file size
  - st_atime the last access time
  - st_mtime the last modification time
  - st_ctime the last status change time

System call stat()

- some predefined macros defined in "/usr/include/sys/stat.h" that take **st_mode** as their argument and return true (1) for the following file types:
  - **MACRO RETURNS TRUE FOR FILE TYPE**
  - S_ISDIR directory
  - S_ISCHR character-oriented special device
  - S_ISBLK block-oriented special device
  - S_ISREG regular file
  - S_ISFIFO pipe

Directory Information

- Library Function:
  - DIR * opendir (char * fileName)
  - struct dirent * readdir (DIR *dir)
  - int closedir (DIR *dir)
- opendir () opens a directory file for reading and returns a pointer to a stream descriptor which is used as the argument to readdir () and closedir ().
- readdir () returns a pointer to a dirent structure containing information about the next directory entry each time it is called. closedir () is used to close the directory.

Directory Information

- The dirent structure is defined in the system header file "/usr/include/dirent.h"
  - **NAME** **MEANING**
  - d_ino the inode number
  - d_off the offset of the next directory entry
  - d_reclen the length of the directory entry structure
  - d_name the filename
- opendir () returns the directory stream pointer when successful, NULL when not successful. readdir () returns 1 when a directory entry has been successfully read, 0 when the last directory entry has already been read, and -1 in the case of an error. closedir () returns 0 on success, -1 on failure.

Misc. File Management System Calls

Figure 12-20. Linux file management system calls.

Name	Function
chown	Changes a file's owner and/or group.
chmod	Changes a file's permission settings.
dup	Duplicates a file descriptor.
dup2	Similar to dup.
fchown	Works just like chown.
fchmod	Works just like chmod.
fcntl	Gives access to miscellaneous file characteristics.
ftruncate	Works just like truncate.
ioctl	Controls a device.
link	Creates a hard link.
mknod	Creates a special file.
sync	Schedules all file buffers to be flushed to disk.
truncate	Truncates a file.

Changing a File's Owner and/or Group: chown ()

Figure 12-21. Description of the chown () system call.

System Call: int **chown** (const char* *fileName*, uid_t *ownerId*, gid_t *groupId*)

int **lchown** (const char* *fileName*, uid_t *ownerId*, gid_t *groupId*)

int **fchown** (int *fd*, uid_t *ownerId*, gid_t *groupId*)

chown () causes the owner and group IDs of *fileName* to be changed to *ownerId* and *groupId*, respectively. A value of -1 in a particular field means that its associated value should remain unchanged.

Only a super-user can change the ownership of a file, and a user may change the group only to another group that he/she is a member of. If *fileName* is a symbolic link, the owner and group of the link are changed instead of the file that the link is referencing.

fchown () is just like chown () except that it takes an open descriptor as an argument instead of a filename.

lchown () changes the ownership of a symbolic link itself rather than the file the link references.

They both return -1 if unsuccessful, and 0 otherwise.

System call chown()

- example

```
$ cat mychown.c                    ...list the file.
main ()
{
 int flag;
 flag = chown ("test.txt", -1, 62); /* Leave user ID
unchanged */
 if (flag == -1) perror("mychown.c");
}
$ ls -l test.txt                  ...examine file before.
-rw-r--r--  1 glass     music        3 May 25 11:42 test.txt
$ ./mychown                       ...run program.
$ ls -l test.txt                  ...examine file after.
-rw-r--r--  1 glass     cs           3 May 25 11:42 test.txt
$ _
```

- Changing a File's Permissions: chmod ()

Figure 12-22. Description of the chmod () system call.

System Call: int **chmod** (const char* *fileName*, int *mode*)

int **fchmod** (int *fd*, mode_t *mode*);

chmod () changes the mode of *fileName* to *mode*, where *mode* is usually supplied as an octal number as described in Chapter 3, "GNU Utilities for Nonprogrammers." The "set user ID" and "set group ID" flags have the octal values 4000 and 2000, respectively. To change a file's mode, you must either own it or be a super-user.

fchmod () works just like chmod () except that it takes an open file descriptor as an argument instead of a filename.

They both return -1 if unsuccessful, and 0 otherwise.

System call chmod()

- example

```
$ cat mychmod.c                    ...list the file.
main ()
{
 int flag;
 flag = chmod ("test.txt", 0600); /* Use octal encoding */
 if (flag == -1) perror ("mychmod.c");
}
$ ls -lG test.txt                  ...examine file before.
-rw-r--r-- 1 glass                3 May 25 11:42 test.txt
$ ./mychmod                        ...run the program.
$ ls -lG test.txt                  ...examine file after.
-rw------- 1 glass                3 May 25 11:42 test.txt
$ _
```

- Duplicating a File Descriptor: dup ()

Figure 12-23. Description of the dup () system call.

System Call: int **dup** (int *oldFd*)

int **dup2** (int *oldFd*, int *newFd*)

dup () finds the smallest free file descriptor entry and points it to the same file as *oldFd*. dup2 () closes *newFd* if it's currently active and then points it to the same file as *oldFd*. In both cases, the original and copied file descriptors share the same file pointer and access mode.

They both return the index of the new file descriptor if successful, and -1 otherwise.

System call dup()

- example

```
$ cat mydup.c                    ...list the file.
#include <stdio.h>
#include <fcntl.h>
main ()
{
 int fd1, fd2, fd3;
  fd1 = open ("test.txt", 0_RDWR | 0_TRUNC);
 printf ("fd1 = %d\n", fd1);
 write (fd1, "what's", 6);
  fd2 = dup (fd1); /* Make a copy of fd1 */
 printf ("fd2 = %d\n", fd2);
 write (fd2, " up", 3);
  close (0); /* Close standard input */
 fd3 = dup (fd1); /* Make another copy of fd1 */
 printf ("fd3 = %d\n", fd3);
 write (0, " doc", 4);
  dup2 (3, 2); /* Duplicate channel 3 to channel 2 */
 write (2, "?\n", 2);
}
$ ./mydup                ...run the program.
fd1 = 3
fd2 = 4
fd3 = 0
$ cat test.txt           ...list the output file.
what's up doc?
```