CS 3411 Systems Programming

Department of Computer Science
Michigan Technological University

File Systems (cont.)

Today's Topics

- File manipulation via kernel calls
- File locking and synchronious writing

open() Kernel Call

- Read the manual!
- We have to tell the function what file we want opened, and with what intent.
- Some of the possible flags are:
  - `O_RDONLY` - Open for reading only
  - `O_WRONLY` - Open for writing only
  - `O_RDWR` - Open for reading and writing
  - `O_APPEND` - If set, the seek pointer will be set to the end of the file prior to each write.
  - `O_CREAT` - If the file exists, this flag has no effect. Otherwise, the file is created, owner ID of the file set to the *effective* user ID of the process
  - `O_TRUNC` - If the file exists and is a regular file, and the file is successfully opened via `O_RDWR` or `O_WRONLY`, its length is truncated to zero but the mode and owner are unchanged.

open() Kernel Call

- A successful open call returns a *file descriptor*
- File descriptors, when used to do I/O, access a *currency indicator* stored by the OS to figure out where to start the next I/O operation
- If you open the same file multiple times, you will end up with multiple file descriptors and currency indicators!
- Some sample calls:
- ```
  datafile = open("mydata", O_RDWR);

  newfile = open("mydata", O_RDWR|O_CREAT, 0644);
  ```
  ```

open() vs. fopen()

- `open()` is a Unix kernel call; `fopen()` is a standard I/O library function
- `open()` returns an int (file descriptor); `fopen()` returns a `FILE *` (an stdio stream)
- Unix I/O kernel calls `[read(),write(),lseek()]` operate on file descriptors; standard I/O functions `[fscanf(),fprintf(),fread(),fwrite(),fseek()]` operate on stdio streams
- User level application programs should use `fopen()` for conventional files; systems programs should use `open()`.
- Never mix and match the two calls!

Default File Descriptors

- 0 is stdin
- 1 is stdout
- 2 is stderr
- You can use these as integer values directly without opening them
- Opened on process creation

read() Kernel Call

- Manual page!
- `ssize_t read(int fd, void *buf, size_t count);`
- Attempts to read up to count bytes from file descriptor fd into the buffer starting at buf
- Normal use:

```
int datafile;
ssize_t num;
char buff[100];
datafile = open("mydata", O_RDWR);
num = read(datafile, buff, 100);
```

read() Kernel Call

- Use sizeof when reading into variables
- Check the output to confirm read was done as you wanted:
  - 0 means end of file
  - -1 means an error
  - Anything else is the number of bytes read
- We could also read() to read saved structs from disk

write() Kernel Call

- Manual page!
- `ssize_t write(int fd, const void *buf, size_t count);`
- Writes up to count bytes to the file referenced by the file descriptor from the buffer starting at buf
- Normal use:

```
int datafile;
ssize_t num;
char buff[100];
buff = "Some string...";
datafile = open("mydata", O_RDWR);
num = write(datafile, buff, 100);
```

write() Kernel Call

- Use sizeof when writing variables to files
- Check the output to confirm read was done as you wanted:
  - 0 means nothing was written
  - -1 means an error
  - Anything else is the number of bytes written
- We could also write() to write structs to disk

lseek() Kernel Call

- Manual page!
- `off_t lseek(int fd, off_t offset, int whence);`
- Repositions the currency indicator of the open file fd to the argument offset according to whence
- If whence is SEEK_SET, the currency indicator is set to offset bytes
- If whence is SEEK_CUR, the currency indicator is set to current location plus offset bytes
- If whence is SEEK_END, the currency indicator is set to the size of the file plus offset bytes

lseek() Kernel Call

- Moves currency indicator without doing any I/O

```
n = lseek(fd, (off_t)100, SEEK_SET);
n = lseek(fd, (off_t)100, SEEK_CUR);
n = lseek(fd, (off_t)-100, SEEK_END);
n = lseek(fd, (off_t)100, SEEK_END);
```

File Locking

- Each process has it's own pointer to the current position and file status flags, possibly to the same file
- read() and write() are atomic from perspective of other processes
- When file is opened for append, position pointer moved to end of file prior to write()
- While read() and write() are atomic, concurrent operations on shared files can have unexpected results:

Examples

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

main() {
  char buf[] = "aaa aaa aaa aaa aaa aaa aaa aaa aaa\n";
  int sharedFd, i, j, x;

  sharedFd = open("test.out",O_RDWR|O_APPEND|O_CREAT, 0644);
  for (i=0; i < strlen(buf); i++) {
    write(sharedFd, &buf[i], 1);
    for (j = 0; j < 100000; j++) {
      x = 0;
    }
  }
}
```

Examples

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

main() {
  char buf[] = "bbb bbb bbb bbb bbb bbb bbb bbb bbb\n";
  int sharedFd, i, j, x;

  sharedFd = open("test.out",O_RDWR|O_APPEND|O_CREAT, 0644);
  for (i=0; i < strlen(buf); i++) {
    write(sharedFd, &buf[i], 1);
  }
}
```

Examples

Append is not the only problem!

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

main() {
  char buf[] = "aaa aaa aaa aaa aaa aaa aaa aaa aaa\n";
  int sharedFd, i, j, x;

  sharedFd = open("test.out",O_RDWR|O_CREAT, 0644);
  srandom(getpid());
  for (i=0; i < strlen(buf); i++) {
    x = random() % 3;
    sleep(x);
    write(sharedFd, &buf[i], 1);
  }
}
```

Examples

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

main() {
  char buf[] = "bbb bbb bbb bbb bbb bbb bbb bbb bbb\n";
  int sharedFd, i, j, x;

  sharedFd = open("test.out",O_RDWR|O_CREAT, 0644);
  srandom(getpid());
  for (i=0; i < strlen(buf); i++) {
    x = random() % 3;
    sleep(x);
    write(sharedFd, &buf[i], 1);
  }
}
```

Locking Portions of Files

- `int fcntl(int fd, int cmd, ... /* arg */);`
- fcntl() performs an operation described by the cmd argument on the open file descriptor fd
- cmd options for locking
  - F_GETLK returns information about the lock if lock is held, otherwise returns F_UNLCK in the appropriate field of the struct (next slide)
  - This does not acquire the lock!
  - F_SETLK sets the lock as described
  - F_SETLKW tries to set the lock as described, but will put the calling process to sleep if lock can't be granted

The flock Struct

```
struct flock {
  short l_type; /* Type of lock: F_RDLCK, F_WRLCK, F_UNLCK */
  short l_whence; /* How to interpret l_start: SEEK_SET,
  SEEK_CUR, SEEK_END */
  off_t l_start; /* Starting offset for lock */
  off_t l_len; /* Number of bytes to lock */
  pid_t l_pid; /* PID of process blocking our lock,
  returned with F_GETLCK */
}
```

These are advisory locks! They will only work if all processes honor lock status.

Examples I

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

main() {
  char buf[] = "aaa aaa aaa aaa aaa aaa aaa aaa aaa\n";
  int sharedFd, i, j, x;
  struct flock sharedFdLock;

  sharedFd = open("test.out",O_RDWR|O_APPEND|O_CREAT, 0644);
  sharedFdLock.l_type=F_WRLCK;
  sharedFdLock.l_start=0;
  sharedFdLock.l_whence=SEEK_SET;
  sharedFdLock.l_len=0;

  fcntl(sharedFd,F_SETLKW, &sharedFdLock);
  write(2, "Writer 1 beginning write.\n",26);
  for (i=0; i < strlen(buf); i++) {
    write(sharedFd, &buf[i], 1);
    for (j = 0; j < 100000; j++) {
      x = 0;
    }
  }
```

Examples II

```
    sharedFdLock.l_type=F_UNLCK;
    sharedFdLock.l_start=0;
    sharedFdLock.l_whence=SEEK_SET;
    sharedFdLock.l_len=0;

    write(2, "Writer 1 done writing.\n", 23);
    fcntl(sharedFd,F_SETLK, &sharedFdLock);
    sleep(2);
}
```

Examples I

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

main() {
  char buf[] = "bbb bbb bbb bbb bbb bbb bbb bbb bbb\n";
  int sharedFd, i, j, x;
  struct flock sharedFdLock;

  sharedFd = open("test.out",O_RDWR|O_APPEND|O_CREAT, 0644);
  sharedFdLock.l_type=F_WRLCK;
  sharedFdLock.l_start=0;
  sharedFdLock.l_whence=SEEK_SET;
  sharedFdLock.l_len=0;

  fcntl(sharedFd,F_SETLKW, &sharedFdLock);
  write(2, "Writer 2 beginning write.\n",26);

  for (i=0; i < strlen(buf); i++) {
    write(sharedFd, &buf[i], 1);
  }

  sharedFdLock.l_type=F_UNLCK;
```

Examples II

```
    sharedFdLock.l_start=0;
    sharedFdLock.l_whence=SEEK_SET;
    sharedFdLock.l_len=0;

    write(2, "Writer 2 done writing.\n", 23);
    fcntl(sharedFd,F_SETLK, &sharedFdLock);
    sleep(2);
}
```