File I/O

CST 357/457 – Systems Programming
Michael Ruth, Ph.D.
Associate Professor
Computer Science & I.T.
mruth@roosevelt.edu

Objectives

- Discuss file I/O concepts including file descriptors and file tables and their use
- Explain the open and creat system calls and the flags that govern their usage
- Discuss the read and write system calls including blocking and non-blocking
- Explain synchronized I/O and its use
- Discuss closing files, seeking, reading/writing by position, and truncating files
- Explain unix kernel internals

File I/O Concepts

- If we're to read/write, we need to **open** the file
  - System manages a per-process list of open files (***file table***)
    - Indexed using positive **int**s called file descriptors (**fds**)
    - Entry in list contains info about open files
      - Copy of inode included
    - Generally, we'll use these fds as cookies

Special File Descriptors

- Unless process explicitly closes them, every process has 3 open
  - 0 stdin STDIN_FILENO
  - 1 stdout STDOUT_FILENO
  - 2 stderr STDERR_FILENO
- We generally don't use the numbers…

open() System Call

- `int open (const char *name, int flags)`
- `int open (const char *name, int flags, mode_t mode)`
  - These map the name (pathname) to a file descriptor
    - Which is returned!

Flags for Open

- Flags argument is the bitwise OR of one or more flags
  - O_RDONLY, O_WRONLY, O_RDRW
  - O_APPEND
  - O_ASYNC, O_SYNC
  - O_CLOEXEC
  - O_CREAT
  - O_DIRECT
  - O_DIRECTORY
  - O_EXCL
  - O_LARGEFILE
  - O_NATIME+
  - O_NOCITY
  - O_NOFOLLOW
  - O_NONBLOCK
  - O_TRUNC
  - If we need more than one, we OR them:
    - fd = open(<pathname>, O_WRONLY | O_TRUNC);

Permission Rules

- Owners:
  - UID of file's owner is the effective UID of the process creating the file
  - Owning group is more complicated…
- Permissions:
  - Mode argument specifies the permissions of the newly created file
  - Set of permission constants that can be bitwise OR'd together
    - S_IRWXU <= rwx owner
    - S_IRUSR <= read owner
    - S_IWUSR <= write owner
    - S_IRWXG <= rwx group

More Permission Rules

- If you don't specify, the umask of the system controls it…
  - Can be modified with a call to umask()
  - However, generally 022
    - So the permissions would be 777-022
      - 755
    - You can test this by creating a new file
      - touch newfile
        - » Creates a blank file

creat() function

- The combination of O_WRONLY | O_CREAT | O_TRUNC is so common, there is a system call for that
  - `int creat(const char *name, mode_t mode)`

Return value codes?

- Return file descriptors on success
- On error, both return -1
- They also set errno to an appropriate value
  - There is a list in Chapter 1
  - We can use that errno to correct the error

Reading via Read()

- Now that we have a file open, we can read
  - `ssize_t read(int fd, void *buff, size_t len)`
    - Each call reads up to len bytes into the memory buff from the current offset of fd

Simple Example

```
unsigned long word
ssize_t nr;

nr = read(fd,&word, sizeof (unsigned long));
If (nr == -1)
     /*error*/
```

Return values (1)

- It is legal for read to return a positive non-zero value less than len
- It could also return 0
  - Usually to indicate EOF
- Finally, if a call to read len bytes and there are no bytes available
  - The read blocks!

Return Values (2)

- Read() can result in many possibilities:
  - The call returns a val equal to len
  - The call returns a val less than len
  - The call returns 0 (EOF)
  - The call blocks
  - The call returns -1
    - errorno => EINTR
      - signal received before bytes were read (reissue call)
  - The call returns -1
    - errorno is not EINTR (or EAGAIN)

Reading all the bytes

```
ssize_t ret;
while (len !=0 && (ret = read(fd,buff,len) !=0) {
   if (ret = -1) {
      if (errno == EINTR)
         continue;
         perror("read");
         break;
   }
   len -= ret;
   buf += ret;
}
```

Nonblocking reads?

- If you want to read only what is available, you use nonblocking reads
  - Now, we care about EAGAIN
    - errno will be set to EAGAIN if there is no data to read
  - So, when doing it this way, we have to check for EAGAIN

Non-blocking Example

```
char buf[BUFSIZ]
ssize_t ret;

nr = read(fd,buff,BUFSIZE);
if (nr == -1)
     if (errno == EINTR)
          //go back
     if (errno == EAGAIN)
          //resubmit later
     else
          //error
```


Size Limits

- size_t and ssize_t are mandated by POSIX
  - size_t is used for storing values used to measure bytes
  - ssize_t is a signed version of size_t
  - On 32-bit systems, the size is generally unsigned int and int respectively
    - So, there are maximums
      - size_t => given by SIZE_MAX
      - ssize_t => given by SSIZE_MAX
        - » On most systems, it's the same size as LONG_MAX

```
if (len > SSIZE_MAX)
   len = SSIZE_MAX;
```

Writing with write()

- The most basic and common system call for writing is:
  - `#include <unistd.h>`
  - `ssize_t write(int fd, const void *buf, size_t count)`
- As with read, the most basic usage is simple:

```
const char *buf = "My ship is solid!";
ssize_t nr;

nr = write(fd, buf, strlen(buf));
if (nr == -1)
      //error
```


Partial Writes

- The write() call is much less likely to return a partial write than a read() call
- Generally, for regular files, there is no need to use loops to ensure writing
  - We'll need to do this with sockets!

Size Limits (Revisited)

- If count is larger than SSIZE_MAX, the results of the call are undefined…
  - A call to write with a count of zero results in the call returning immediately with 0

Write Behavior

- Generally speaking, when writes occur, there is buffering going on
  - The write to the actual disk will happen but at a time when the system is not doing as much
  - Later in the background, the kernel collects all the dirty buffers, sorts them, and writes them out to disk (writeback)
  - Issues with this:
    - write-ordering…
    - error reporting on the writeback

Synchronized I/O

- Buffering writes provides a significant performance improvement
- However, when you wish to control the writes to the disk, there is a way
  - You should ONLY do this if absolutely, positively, necessary!

fsync()

- The simplest method is to use the fsync() call
  - `#include <unistd.h>`
  - `int fsync(int fd)`
  - Flushes the write!
    - Writes the data & metadata

fdatasync()

- Another way to do it:
  - `#include <unistd.h>`
  - `int fdatasync(int fd)`
  - Flushes the write!
    - Writes the data & metadata, but only the metadata required to access the file in the future

Old School sync!

- Another way to do it:
  - `#include <unistd.h>`
  - `void sync(void)`
  - Flushes all buffers
    - No params, no return
    - Always works!

Open with O_SYNC

- If you open it with O_SYNC, all reads/writes are synchronous…
  - Imagine a fsync() call after every read/write…

Closing Files

- After we're finished with a file, we need to close it
  - `#include <unistd.h>`
  - `int close(int fd)`

Seek!

- We can move around the file using seek
  - #include <sys/types.h>
  - #include <unistd.h>
  - off_t lseek(int fd, off_t pos, int origin)
  - Origin:
    - SEEK_CUR
      - Current file position is set to current value + pos
    - SEEK_END
      - Current file position is set to current length of the file + pos
        - » Can be negative, 0, or positive
    - SEEK_SET
      - Current file position is set to pos (0 is the beginning)
  - The call returns the new file position on success

Seek past the file?

- It is possible to seek past the end of the file
  - On it's own, no biggie…
    - next read will be EOF
    - Next write will create space between the real end of the file and the position and write!
      - This is zero padding and called a hole
        - » Holes do not actually occupy space
          - This is the how files can occupy more space than the disk can hold…
        - » Files with holes are called sparse files

Positional Reads/Writes

- Instead of two-stepping – seek and then action, you can read/write from position
  - pread/pwrite
  - Work basically the same as using the individual

Truncating Files

- You can truncate a file to a given length using the truncate system calls
  - `#include <sys/types.h>`
  - `#include <unistd.h>`
  - `int ftruncate(int fd, off_t len)`

Kernel Internals

- There are three primary subsystems of the kernel:
  - Virtual filesystem (VFS)
  - Page cache
    - Exploits ***temporal locality***
    - Dynamic in size
    - Heuristics can be tuned… ***swappiness***
    - Exploits ***sequential locality*** using ***readahead***
      - Sequential file I/O takes advantage of this
  - Page writeback
    - Carried out by a gang of flusher threads
      - Utilize congestion avoidance

Summary

- Discussed file I/O concepts including file descriptors and file tables and their use
- Explained the open and creat system calls and the flags that govern their usage
- Discussed the read and write system calls including blocking and non-blocking
- Explained synchronized I/O and its use
- Discussed closing files, seeking, reading/writing by position, and truncating files
- Explained unix kernel internals

Questions?