CS354: Machine Organization and Programming

Lecture 25
Friday the October 30th 2015

Section 2
Instructor: Leo Arulraj

Class Announcements

1. Assignment 3 is due coming Wednesday (11/04) by 9 AM. It is high time you started working on it if you have not already.
2. Review class for Midterm Exam 2 on 11/09 (Monday). Use piazza, office hours and the review class to clarify your doubts.

Lecture Overview

1. Unix I/O
2. Sharing files
3. I/O redirection
4. Standard I/O
5. RIO (Our library package for reliable I/O)

Unix I/O

Unix file is a sequence of bytes.

All I/O devices including networks, terminals, disks are modeled as files. E.g. /dev/sda for disk, /dev/tty for terminal

This way, programmers use a simple low-level interface called the Unix I/O to interact with all I/O devices

Some examples of operations on files:

- Opening and Closing files
- Reading and writing files
- Changing the current file offset
- Read metadata about a file

Unix File Types

Regular file: contains user supplied data. OS does not know anything about the contents in the file.

Directory file: contains names and locations of files.

Character special file, block special file: Terminals are character special files and disks are block special files.

FIFO (named pipe): used for first in first out inter process communication.

Socket: used for network communication between processes.

Unix I/O APIs

int open(char *filename, int flags, mode_t mode);

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

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

off_t lseek(int fd, off_t offset, int whence);

int fstat(int fd, struct stat *buf);

Each process created by a Unix shell begins life with three open files: standard input (descriptor 0), standard output (descriptor 1), and standard error (descriptor 2)

Problem of ShortCounts

Read and Write return fewer bytes than requested.

This does not indicate an error scenario.

Can happen due to many reasons:

- Encountering EOF on reads
- Reading text lines from terminal
- Reading and Writing network sockets

Can handle it using our own library(RIO) or Standard I/O

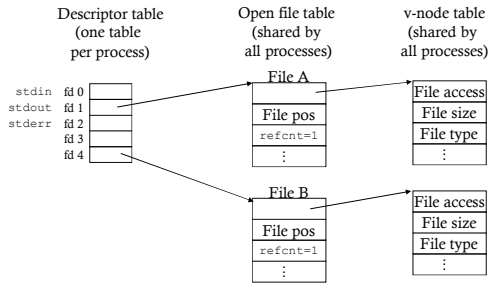
Sharing Files
(Descriptions from CSAPP Text)

Descriptor table: Each process has its own separate descriptor table whose entries are indexed by the process's open file descriptors. Each open descriptor entry points to an entry in the file table.

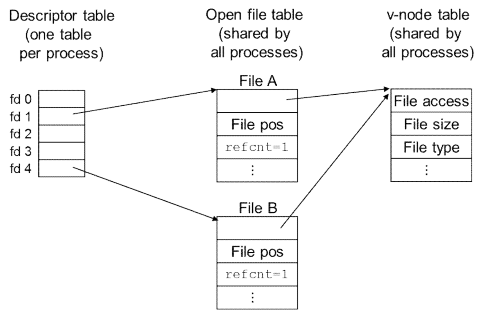
File table: The set of open files is represented by a file table that is shared by all processes. Each file table entry consists of (for our purposes) the current file position, a reference count of the number of descriptor entries that currently point to it, and a pointer to an entry in the v-node table. Closing a descriptor decrements the reference count in the associated file table entry. The kernel will not delete the file table entry until its reference count is zero.

v-node table: Like the file table, the v-node table is shared by all processes. Each entry contains most of the information in the stat structure, including the st_mode and st_size members. v-node table and the related VFS(Virtual File System) interface is the separation between specific file system implementations and the generic file system operations.

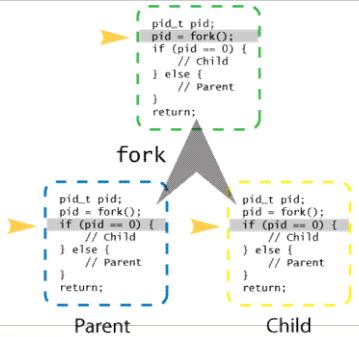
Typical OS kernel data structures for open files.



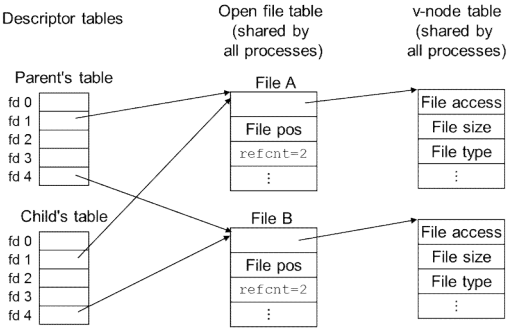
File sharing: This example shows two descriptors sharing the same disk file through two open file table entries.



Forking new child processes



How a child process inherits the parent's open files.



I/O Redirection

In a Unix shell, you can use "<" for input redirection , ">" for output redirection.

Stderr redirection example:

grep searchword filepaths 2> grep-errors.txt

Redirect stdout to stderr:

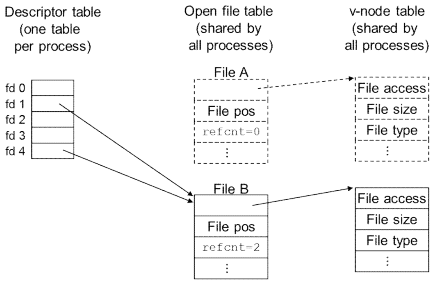
grep searchword filepaths 1>&2

Redirect both stdout and stdout to a file:

rm -f $(find / -name core) &> /dev/null

Redirection is internally implemented using:

int dup2(int oldfd, int newfd);



Standard I/O

C Standard library (libc) provides:

fopen, fclose

fread, fwrite

fgets, fputs

fscanf, fprintf

Standard I/O models files as streams and buffers I/O

Standard I/O

```
#include <stdio.h>

int main(){
   printf("h");
   printf("e");
   printf("l");
   printf("l");
   printf("o");
   printf("\n");
   fflush(stdout);
   exit(0);
}
```

Using the strace linux command on this program, we can see the buffering in action.

```
linux> strace ./hello
execve("./hello", ["hello"], [/*
... */]).
...
write(1, "hello\n", 6)
= 6
...
exit_group(0)
= ?
```

RIO

- RIO is a set of wrappers that provide efficient and robust I/O in apps, such as network programs that are subject to short counts
- RIO provides two different kinds of functions
  - Unbuffered input and output of binary data
    - **`rio_readn`** and **`rio_writen`**
  - Buffered input of binary data and text lines
    - **`rio_readlineb`** and **`rio_readnb`**
    - Buffered RIO routines are thread-safe and can be interleaved arbitrarily on the same descriptor
- Download from http://csapp.cs.cmu.edu/public/code.html
  → **`src/csapp.c`** and **`include/csapp.h`**

Unix I/O vs Standard I/O vs RIO

Unix I/O is the lowest form of interface and provides basis for both Standard I/O and RIO

Unix I/O can be used in signal handlers

Unix I/O does not deal with short counts like Standard I/O or RIO

Unix I/O does not have buffering and hence is inefficient.

Standard I/O solves both short counts and buffering issues of Unix I/O but cannot be used for networking applications due to poorly documented restrictions hence the need for RIO.

Go over RIO source code in class

Different I/O interfaces in perspective