System-Level I/O

15-213 / 18-213: Introduction to Computer Systems
15th Lecture, Mar. 8, 2012

Instructors:
Todd C. Mowry & Anthony Rowe

---

Today

- Unix I/O
- Metadata, sharing, and redirection
- Standard I/O
- RIO (robust I/O) package
- Closing remarks

---

Unix Files

- A Unix file is a sequence of $m$ bytes:
  - $B_0, B_1, \ldots, B_k, \ldots, B_{m-1}$

- All I/O devices are represented as files:
  - `/dev/sda2` (/usr disk partition)
  - `/dev/tty2` (terminal)

- Even the kernel is represented as a file:
  - `/dev/kmem` (kernel memory image)
  - `/proc` (kernel data structures)

---

Unix File Types

- Regular file
  - File containing user/app data (binary, text, whatever)
  - OS does not know anything about the format
    - other than "sequence of bytes", akin to main memory

- Directory file
  - A file that contains the names and locations of other files

- Character special and block special files
  - Terminals (character special) and disks (block special)

- FIFO (named pipe)
  - A file type used for inter-process communication

- Socket
  - A file type used for network communication between processes
Unix I/O

- **Key Features**
  - Elegant mapping of files to devices allows kernel to export simple interface called Unix I/O
  - Important idea: All input and output is handled in a consistent and uniform way

- **Basic Unix I/O operations (system calls):**
  - Opening and closing files
    - `open()` and `close()`
  - Reading and writing a file
    - `read()` and `write()`
  - Changing the current file position (seek)
    - Indicates next offset into file to read or write
    - `lseek()`

### Opening Files

- **Opening a file informs the kernel that you are getting ready to access that file**

```
int fd; // file descriptor */
if ((fd = open("/etc/hosts", O_RDONLY)) < 0) {
    perror("open");
    exit(1);
}
```

- **Returns a small identifying integer file descriptor**
  - `fd == -1` indicates that an error occurred

- **Each process created by a Unix shell begins life with three open files associated with a terminal:**
  - 0: standard input
  - 1: standard output
  - 2: standard error

### Closing Files

- **Closing a file informs the kernel that you are finished accessing that file**

```
int fd; /* file descriptor */
int retval; /* return value */
if ((retval = close(fd)) < 0) {
    perror("close");
    exit(1);
}
```

- **Closing an already closed file is a recipe for disaster in threaded programs (more on this later)**
- **Moral:** Always check return codes, even for seemingly benign functions such as `close()`

### Reading Files

- **Reading a file copies bytes from the current file position to memory, and then updates file position**

```
char buf[512];
int fd; /* file descriptor */
int nbytes; /* number of bytes read */
/* Open file fd ... */
if ((nbytes = read(fd, buf, sizeof(buf))) < 0) {
    perror("read");
    exit(1);
}
```

- **Returns number of bytes read from file fd into buf**
  - Return type `ssize_t` is signed integer
  - `nbytes < 0` indicates that an error occurred
  - **Short counts** (`nbytes < sizeof(buf)`) are possible and are not errors!
Writing Files

- Writing a file copies bytes from memory to the current file position, and then updates current file position

```c
char buf[512];
int fd; /* file descriptor */
int nbytes; /* number of bytes read */

/* Open the file fd ... */
/* Then write up to 512 bytes from buf to file fd */
if ((nbytes = write(fd, buf, sizeof(buf)) < 0) {  
    perror("write");
    exit(1);
}
```

- Returns number of bytes written from `buf` to file `fd`
  - `nbytes < 0` indicates that an error occurred
  - As with reads, short counts are possible and are not errors!

On Short Counts

- Short counts can occur in these situations:
  - Encountering (end-of-file) EOF on reads
  - Reading text lines from a terminal
  - Reading and writing network sockets or Unix pipes

- Short counts never occur in these situations:
  - Reading from disk files (except for EOF)
  - Writing to disk files

Simple Unix I/O example

- Copying standard in to standard out, one byte at a time

```c
int main(void)
{
    char c;
    int len;

    while ((len = read(0 /* stdin */, &c, 1)) == 1) {
        if (write(1 /* stdout */, &c, 1) != 1) {
            exit(20);
        }
    }

    if (len < 0) {
        printf("read from stdin failed");
        exit (10);
    }
    exit(0);
}
```

Today

- Unix I/O
- Metadata, sharing, and redirection
- Standard I/O
- RIO (robust I/O) package
- Closing remarks
**File Metadata**

- **Metadata** is data about data, in this case file data.
- Per-file metadata maintained by kernel
  - accessed by users with the `stat` and `fstat` functions.

```c
/* Metadata returned by the stat and fstat functions */
struct stat {
  dev_t         st_dev; /* device */
  ino_t         st_ino; /* inode */
  mode_t        st_mode; /* protection and file type */
  nlink_t       st_nlink; /* number of hard links */
  uid_t         st_uid; /* user ID of owner */
  gid_t         st_gid; /* group ID of owner */
  dev_t         st_rdev; /* device type (if inode device) */
  off_t         st_size; /* total size, in bytes */
  unsigned long st_blksize; /* blocksize for filesystem I/O */
  unsigned long st_blocks; /* number of blocks allocated */
  time_t        st_atime; /* time of last access */
  time_t        st_mtime; /* time of last modification */
  time_t        st_ctime; /* time of last change */
};
```

**Example of Accessing File Metadata**

```c
#include "csapp.h"

int main (int argc, char **argv) {
  struct stat stat;
  char *type, *readok;

  Stat(argv[1], &stat);
  if (S_ISREG(stat.st_mode))
    type = "regular";
  else if (S_ISDIR(stat.st_mode))
    type = "directory";
  else
    type = "other";

  if ((stat.st_mode & S_IRUSR))
    /* OK to read?*/
    readok = "yes";
  else
    readok = "no";

  printf("type: %s, read: %s\n", type, readok);
  exit(0);
}
```

**Repeated Slide: Opening Files**

- Opening a file informs the kernel that you are getting ready to access that file.

```c
int fd; /* file descriptor */

if ((fd = open("/etc/hosts", O_RDONLY)) < 0) {
  perror("open");
  exit(1);
}
```

- Returns a small identifying integer **file descriptor**
  - `fd` == -1 indicates that an error occurred

**How the Unix Kernel Represents Open Files**

- Two descriptors referencing two distinct open disk files.
  - Descriptor 1 (stdout) points to terminal, and descriptor 4 points to open disk file.

```c
int fd;

if (fd = open("/etc/hosts", O_RDONLY)) < 0) {
  perror("open");
  exit(1);
}
```

- **Descriptor table** [one table per process]
- **Open file table** [shared by all processes]
- **V-node table** [shared by all processes]
### File Sharing

- Two distinct descriptors sharing the same disk file through two distinct open file table entries
  - E.g., Calling `open` twice with the same `filename` argument

### How Processes Share Files: Fork()

- A child process inherits its parent's open files
  - Note: situation unchanged by `exec` functions (use `fcntl` to change)
- **Before** `fork()` call:

### I/O Redirection

- **Question:** How does a shell implement I/O redirection?  
  `unix> ls > foo.txt`
- **Answer:** By calling the `dup2(oldfd, newfd)` function
  - Copies (per-process) descriptor table entry `oldfd` to entry `newfd`
**I/O Redirection Example**

- **Step #1:** open file to which stdout should be redirected
  - Happens in child executing shell code, before `exec`

**Diagram**

- **Descriptor table**
  - [one table per process]
  - `fd 0` for `stdin`,
  - `fd 1` for `stdout`,
  - `fd 2` for `stderr`

- **Open file table**
  - [shared by all processes]
  - File A
    - `refcnt=1`
    - `File access`
    - `File size`
    - `File type`

- **v-node table**
  - [shared by all processes]
  - File B
    - `refcnt=1`
    - `File access`
    - `File size`
    - `File type`

**I/O Redirection Example (cont.)**

- **Step #2:** call `dup2 (4 , 1)`
  - `cause fd=1 (stdout) to refer to disk file pointed at by fd=4`

**Diagram**

- **Descriptor table**
  - [one table per process]
  - `fd 0` for `stdin`,
  - `fd 1` for `stdout`,
  - `fd 2` for `stderr`

- **Open file table**
  - [shared by all processes]
  - File A
    - `refcnt=0`
    - `File access`
    - `File size`
    - `File type`

- **v-node table**
  - [shared by all processes]
  - File B
    - `refcnt=2`
    - `File access`
    - `File size`
    - `File type`

---

**Today**

- Unix I/O
- Metadata, sharing, and redirection
- Standard I/O
- RIO (robust I/O) package
- Closing remarks

**Standard I/O Functions**

- The C standard library (`libc.so`) contains a collection of higher-level standard I/O functions
  - Documented in Appendix B of K&R

- Examples of standard I/O functions:
  - Opening and closing files (`fopen` and `fclose`)
  - Reading and writing bytes (`fread` and `fwrite`)
  - Reading and writing text lines (`fgets` and `fputs`)
  - Formatted reading and writing (`fscanf` and `fprintf`)
Standard I/O Streams

- Standard I/O models open files as *streams*
  - Abstraction for a file descriptor and a buffer in memory

- C programs begin life with three open streams (defined in stdio.h)
  - stdin (standard input)
  - stdout (standard output)
  - stderr (standard error)

```c
#include <stdio.h>
extern FILE *stdin; /* standard input (descriptor 0) */
extern FILE *stdout; /* standard output (descriptor 1) */
extern FILE *stderr; /* standard error (descriptor 2) */

int main() {
    fprintf(stdout, "Hello, world\n");
}
```

Buffered I/O: Motivation

- Applications often read/write one character at a time
  - `getc`, `putc`, `ungetc`
  - `gets`, `fgets`
    - Read line of text on character at a time, stopping at newline

- Implementing as Unix I/O calls expensive
  - `read` and `write` require Unix kernel calls
    - > 10,000 clock cycles

- Solution: Buffered read
  - Use Unix read to grab block of bytes
  - User input functions take one byte at a time from buffer
    - Refill buffer when empty

Buffering in Standard I/O

- Standard I/O functions use buffered I/O

```c
#include <stdio.h>
int main() {
    printf("h");
    printf("e");
    printf("l");
    printf("l");
    printf("o");
    printf("n");
    fflush(stdout);

    return 0;
}
```

Standard I/O Buffering in Action

- You can see this buffering in action for yourself, using the always fascinating Unix `strace` program:
Today

- Unix I/O
- Metadata, sharing, and redirection
- Standard I/O
- RIO (robust I/O) package
- Closing remarks

The RIO Package

- 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](http://csapp.cs.cmu.edu/public/code.html) → `src/csapp.c` and `include/csapp.h`

Implementation of `rio_readn`

```c
/*
 * rio_readn - robustly read n bytes (unbuffered)
 */
ssize_t rio_readn(int fd, void *usrbuf, size_t n)
{
    size_t nleft = n;
    size_t nread;
    char *bufp = usrbuf;
    while (nleft > 0) {
        if ((nread = read(fd, bufp, nleft)) < 0) {
            if (errno == EINTR) /* interrupted by sig handler return */
                nread = 0; /* and call read() again */
            else
                return -1; /* errno set by read() */
        } else if (nread == 0)
            break; /* EOF */
        nleft -= nread;
        bufp += nread;
    }
    return (n - nleft); /* return >= 0 */
}
```

Today

- Unix I/O
- Metadata, sharing, and redirection
- Standard I/O
- RIO (robust I/O) package
- Closing remarks
Unix I/O vs. Standard I/O vs. RIO

- Standard I/O and RIO are implemented using low-level Unix I/O.
- Which ones should you use in your programs?

Pros and Cons of Unix I/O

- Pros:
  - Unix I/O is the most general and lowest overhead form of I/O.
  - All other I/O packages are implemented using Unix I/O functions.
  - Unix I/O provides functions for accessing file metadata.
  - Unix I/O functions are async-signal-safe and can be used safely in signal handlers.

- Cons:
  - Dealing with short counts is tricky and error prone.
  - Efficient reading of text lines requires some form of buffering, also tricky and error prone.
  - Both of these issues are addressed by the standard I/O and RIO packages.

Pros and Cons of Standard I/O

- Pros:
  - Buffering increases efficiency by decreasing the number of `read` and `write` system calls.
  - Short counts are handled automatically.

- Cons:
  - Provides no function for accessing file metadata.
  - Standard I/O functions are not async-signal-safe, and not appropriate for signal handlers.
  - Standard I/O is not appropriate for input and output on network sockets.
    - There are poorly documented restrictions on streams that interact badly with restrictions on sockets (CS:APP2e, Sec 10.9)

Choosing I/O Functions

- General rule: use the highest-level I/O functions you can.
  - Many C programmers are able to do all of their work using the standard I/O functions.
  - But, be sure to understand the functions you use!

- When to use standard I/O
  - When working with disk or terminal files.

- When to use raw Unix I/O
  - Inside signal handlers, because Unix I/O is async-signal-safe.
  - In rare cases when you need absolute highest performance.

- When to use RIO
  - When you are reading and writing network sockets.
  - Avoid using standard I/O on sockets.
Aside: Working with Binary Files

- **Binary File Examples**
  - Object code, Images (JPEG, GIF).

- **Functions you shouldn’t use on binary files**
  - Line-oriented I/O such as fgets, scanf, printf, rio_readlineb
    - Different systems interpret 0x0A (‘\n’) (newline) differently:
      - Linux and Mac OS X: LF (0x0a) [‘\n’]
      - HTTP servers & Windows: CR+LF (0x0d 0x0a) [‘\r\n’]
    - Use things like rio_readn or rio_readnb instead

- **String functions**
  - strlen, strcpy
  - Interprets byte value 0 (end of string) as special

For Further Information

- **The Unix bible:**
    - Updated from Stevens’s 1993 classic text.

- **Stevens is arguably the best technical writer ever.**
  - Produced authoritative works in:
    - Unix programming
    - TCP/IP (the protocol that makes the Internet work)
    - Unix network programming
    - Unix IPC programming

- **Tragically, Stevens died Sept. 1, 1999**
  - But others have taken up his legacy

Fun with File Descriptors (1)

```c
#include "csapp.h"
int main(int argc, char *argv[]) {
    int fd1, fd2, fd3;
    char c1, c2, c3;
    char *fname = argv[1];
    fd1 = Open(fname, O_RDONLY, 0);
    fd2 = Open(fname, O_RDONLY, 0);
    fd3 = Open(fname, O_RDONLY, 0);
    Dup2(fd2, fd3);
    Read(fd1, &c1, 1);
    Read(fd2, &c2, 1);
    Read(fd3, &c3, 1);
    printf("c1 = %c, c2 = %c, c3 = %c\n", c1, c2, c3);
    return 0;
}
```

What would this program print for file containing “abcde”? 

Fun with File Descriptors (2)

```c
#include "csapp.h"
int main(int argc, char *argv[]) {
    int fd1; int s = getpid() & 0x1;
    char c1, c2, c3;
    char *fname = argv[1];
    fd1 = Open(fname, O_RDONLY, 0);
    Read(fd1, &c1, 1);
    if (fork()) { /* Parent */
        sleep(s);
        Read(fd1, &c2, 1);
        printf("Parent: c1 = %c, c2 = %c\n", c1, c2);
    } else { /* Child */
        sleep(1-s);
        Read(fd1, &c2, 1);
        printf("Child: c1 = %c, c2 = %c\n", c1, c2);
    }
    return 0;
}
```

What would this program print for file containing “abcde”?
Fun with File Descriptors (3)

```c
#include "csapp.h"
int main(int argc, char *argv[])
{
    int fd1, fd2, fd3;
    char *fname = argv[1];
    fd1 = Open(fname, O_CREAT|O_TRUNC|O_RDWR, S_IRUSR|S_IWUSR);
    Write(fd1, "pqrs", 4);
    fd3 = Open(fname, O_APPEND|O_WRONLY, 0);
    Write(fd3, "jklmn", 5);
    fd2 = dup(fd1); /* Allocates descriptor */
    Write(fd2, "wxyz", 4);
    Write(fd3, "ef", 2);
    return 0;
}  
```

What would be the contents of the resulting file?

Unbuffered RIO Input and Output

- Same interface as Unix `read` and `write`
- Especially useful for transferring data on network sockets

```c
#define "csapp.h"
ssize_t rio_readn(int fd, void *usrbuf, size_t n);
ssize_t rio_writen(int fd, void *usrbuf, size_t n);
```

- `rio_readn` returns short count only if it encounters EOF
- `rio_writen` never returns a short count
- Calls to `rio_readn` and `rio_writen` can be interleaved arbitrarily on the same descriptor

Accessing Directories

- Only recommended operation on a directory: read its entries
- `dirent` structure contains information about a directory entry
- `DIR` structure contains information about directory while stepping through its entries

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

DIR *directory;
struct dirent *de;
...
if (!(directory = opendir(dir_name)))
    error("Failed to open directory");
...
while (0 != (de = readdir(directory))) {
    printf("Found file: %s\n", de->d_name);
...
    closedir(directory);
}
```

Buffered I/O: Implementation

- For reading from file
- File has associated buffer to hold bytes that have been read from file but not yet read by user code

```
Buffer
\|-- rio_buf
\|-- already read
  \|-- unread
\|-- rio_cnt
```

- Layered on Unix file:

```
Buffered Portion
\|-- not in buffer
\|-- already read
\|-- unread
\|-- unseen
```

Current File Position
**Buffered I/O: Declaration**

- All information contained in `struct`.

  ```c
  typedef struct {
    int rio_fd; /* descriptor for this internal buf */
    int rio_cnt; /* unread bytes in internal buf */
    char *rio_bufptr; /* next unread byte in internal buf */
    char rio_buf[RIO_BUFSIZE]; /* internal buffer */
  } rio_t;
  ```

**Buffered RIO Input Functions**

- Efficiently read text lines and binary data from a file partially cached in an internal memory buffer.

  ```c
  #include "csapp.h"
  
  void rio_readinitb(rio_t *rp, int fd);
  ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen);
  ssize_t rio_readnb(rio_t *rp, void *usrbuf, size_t n);
  
  Return: num. bytes read if OK, 0 on EOF, -1 on error
  ```

- `rio_readlineb` reads a text line of up to `maxlen` bytes from file `fd` and stores the line in `usrbuf`.
  - Especially useful for reading text lines from network sockets.
- Stopping conditions:
  - `maxlen` bytes read
  - EOF encountered
  - Newline ('\n') encountered

**Buffered RIO Input Functions (cont)**

- `rio_readnb` reads up to `n` bytes from file `fd`.
- Stopping conditions:
  - `maxlen` bytes read
  - EOF encountered
- Calls to `rio_readlineb` and `rio_readnb` can be interleaved arbitrarily on the same descriptor
  - Warning: Don’t interleave with calls to `rio_readn`

**RIO Example**

- Copying the lines of a text file from standard input to standard output.

  ```c
  $include "csapp.h"
  int main(int argc, char **argv)
  {
    int n;
    rio_t rio;
    char buf[MAXLINE];
    
    Rio_readinitb(&rio, STDIN_FILENO);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0)
      Rio_writen(STDOUT_FILENO, buf, n);
    exit(0);
  }
  ```

  `cpfile.c`