15-213
“The course that gives CMU its Zip!”

System-Level I/O
November 4, 2003

Topics
- Unix I/O
- Robust reading and writing
- Reading file metadata
- Sharing files
- I/O redirection
- Standard I/O
Unix I/O Key Characteristics

Classic Unix/Linux I/O:
I/O operates on linear streams of Bytes
- Can reposition insertion point and extend file at end
I/O tends to be synchronous
- Read or write operation block until data has been transferred
Fine grained I/O
- One key-stroke at a time
- Each I/O event is handled by the kernel and an appropriate process

Mainframe I/O:
I/O operates on structured records
- Functions to locate, insert, remove, update records
I/O tends to be asynchronous
- Overlap I/O and computation within a process

Coarse grained I/O
- Process writes “channel programs” to be executed by the I/O hardware
- Many I/O operations are performed autonomously with one interrupt at completion
A Typical Hardware System

- CPU chip
  - register file
  - ALU
  - bus interface
  - I/O bridge
  - system bus
  - memory bus

- I/O bus
- Expansion slots for other devices such as network adapters.

- main memory
- USB controller
- mouse/keyboard

- graphics adapter
- monitor

- disk controller
- disk
Reading a Disk Sector: Step 1

CPU initiates a disk read by writing a command, logical block number, and destination memory address to a port (address) associated with disk controller.
Reading a Disk Sector: Step 2

CPU chip

register file

ALU

bus interface

Disk controller reads the sector and performs a direct memory access (DMA) transfer into main memory.

main memory

I/O bus

USB controller

mouse keyboard

graphics adapter

monitor

disk controller

disk
When the DMA transfer completes, the disk controller notifies the CPU with an interrupt (i.e., asserts a special “interrupt” pin on the CPU)
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
- Binary or text file.
- Unix does not know the difference!

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 interprocess communication

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

The elegant mapping of files to devices allows kernel to export simple interface called Unix I/O.

Key Unix 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()`
- Changing the current file position (seek)
  - `lseek` (not discussed)
- Reading and writing a file
  - `read()` and `write()`
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

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.

```c
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.

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

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

Returns number of bytes read from file `fd` into `buf`

- `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!

Transfers up to 512 bytes from address `buf` to file `fd`
Unix I/O Example

Copying standard input to standard output one byte at a time.

```
#include "csapp.h"

int main(void)
{
    char c;

    while(Read(STDIN_FILENO, &c, 1) != 0)
        Write(STDOUT_FILENO, &c, 1);

    exit(0);
}
```

Note the use of error handling wrappers for read and write (Appendix B).
Dealing with 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.

How should you deal with short counts in your code?

- Use the RIO (Robust I/O) package from your textbook’s csapp.c file (Appendix B).
The RIO Package

RIO is a set of wrappers that provide efficient and robust I/O in applications 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`
  - Cleans up some problems with Stevens’s `readline` and `readn` functions.
  - Unlike the Stevens routines, the buffered RIO routines are *thread-safe* and can be interleaved arbitrarily on the same descriptor.

Download from

csapp.cs.cmu.edu/public/ics/code/src/csapp.c
csapp.cs.cmu.edu/public/ics/code/include/csapp.h
Unbuffered RIO Input and Output

Same interface as Unix read and write

Especially useful for transferring data on network sockets

```c
#include "csapp.h"

ssize_t rio_readn(int fd, void *usrbuf, size_t n);
ssize_t rio_writen(nt fd, void *usrbuf, size_t n);

Return: num. bytes transferred if OK, 0 on EOF (rio_readn only), -1 on error
```

- **rio_readn** returns short count only 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.
/*  * rio_readn - robustly read n bytes (unbuffered)  */  ssize_t rio_readn(int fd, void *usrbuf, size_t n)  {  size_t nleft = n;  ssize_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 */  }
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.
- `rio_readnb` reads up to `n` bytes from file `fd`.
- 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.

```
#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);
}
```
**File Metadata**

**Metadata** is data about data, in this case file data.

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 */
};
```
/* statcheck.c - Querying and manipulating a file’s meta data */
#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)) /* file type*/
        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);
}
Metadata as File (Plan 9, ReiserFS 4)

Access to metadata requires a different API and is not easily extensible. The file notation can be used as a uniform assess mechanism in future file systems:

- **Files as directories:**

```bash
Bass> ls -l
-rw-r--r-- 1 bovik users 120 Nov 3 04:33 bar.c
-rw-r--r-- 1 agn users 727 Nov 3 04:35 foo.c
Bass> cat bar.c/..rwx
-rw-r--r--
Bass> echo 0777 > bar.c/..rwx
Bass> ls -l bar.c
-rwxrwxrwx 1 bovik users 120 Nov 3 04:33 bar.c
Bass> cp bar.c/..uid foo.c/..uid
Bass> ls -l
-rw-r--r-- 1 bovik users 120 Nov 3 04:33 bar.c
-rwxrwxrwx 1 bovik users 727 Nov 3 04:35 foo.c
Bass>
```
Accessing Directories

The only recommended operation on directories is to read its entries.

```c
#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);
}
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.
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

A child process inherits its parent’s open files. Here is the situation immediately after a fork.

Descriptor tables

Parent's table
- fd 0
- fd 1
- fd 2
- fd 3
- fd 4

Child's table
- fd 0
- fd 1
- fd 2
- fd 3
- fd 4

Open file table (shared by all processes)
- File A
  - File pos
  - refcnt=2
  - :

v-node table (shared by all processes)
- File B
  - File pos
  - refcnt=2
  - :

- File access
  - File size
  - File type
  - :

- File access
  - File size
  - File type
  - :

File A
- File pos
- refcnt=2
- :

File B
- File pos
- refcnt=2
- :

File access
- File size
- File type
- :
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

Before calling dup2 (4, 1), stdout (descriptor 1) points to a terminal and descriptor 4 points to an open disk file.
I/O Redirection Example (cont)

After calling `dup2 (4, 1)` , stdout is now redirected to the disk file pointed at by descriptor 4.

Descriptor table
(one table per process)

Open file table
(shared by all processes)

v-node table
(shared by all processes)

File A

File B

File access

File size

File type

File pos

refcnt=0

File pos

refcnt=2
Standard I/O Functions

The C standard library (libc.a) 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 \textit{streams}

- Abstraction for a file descriptor and a buffer in memory.

\textbf{C programs begin life with three open streams (defined in \texttt{stdio.h})}

- \texttt{stdin} (standard input)
- \texttt{stdout} (standard output)
- \texttt{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”);
}
```
Buffering in Standard I/O

Standard I/O functions use buffered I/O

```c
printf("h");
printf("e");
printf("l");
printf("l");
printf("o");
printf("n");
fflush(stdout);
write(1, buf += 6, 6);
```
Standard I/O Buffering in Action

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

```c
#include <stdio.h>

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

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

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 is not appropriate for input and output on network sockets
- There are poorly documented restrictions on streams that interact badly with restrictions on sockets
Pros and Cons of Standard I/O (cont)

Restrictions on streams:
- Restriction 1: input function cannot follow output function without intervening call to fflush, fseek, fsetpos, or rewind.
  - Latter three functions all use lseek to change file position.
- Restriction 2: output function cannot follow an input function with intervening call to fseek, fsetpos, or rewind.

Restriction on sockets:
- You are not allowed to change the file position of a socket.
Pros and Cons of Standard I/O (cont)

Workaround for restriction 1:
- Flush stream after every output.

Workaround for restriction 2:
- Open two streams on the same descriptor, one for reading and one for writing:

```c
FILE *fpin, *fpout;
fpin = fdopen(sockfd, "r");
fpout = fdopen(sockfd, "w");
```

- However, this requires you to close the same descriptor twice:

```c
fclose(fpin);
fclose(fpout);
```

- Creates a deadly race in concurrent threaded programs!
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.

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

When to use raw Unix I/O
- When you need to fetch file metadata.
- In rare cases when you need absolute highest performance.

When to use RIO?
- When you are reading and writing network sockets or pipes.
- Never use standard I/O or raw Unix I/O on sockets or pipes.
Asynchronous I/O

How to deal with multiple I/O operations concurrently?

For example: wait for a keyboard input, a mouse click and input from a network connection.

- Select system call

```
int select(int n, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

- Poll system call (same idea, different implementation)

```
int poll(struct pollfd *ufds, unsigned int nfds, int timeout);

struct pollfd { int fd;    /* file descriptor */
                short events; /* requested events */
                short revents; /* returned events */
};
```

- /dev/poll (Solaris, being considered for Linux)
- Posix real-time signals + sigtimedwait()
- Native Posix Threads Library (NPTL)

For more info see [http://www.kegel.com/c10k.html](http://www.kegel.com/c10k.html)
Asynchronous I/O (cont.)

POSIX P1003.4 Asynchronous I/O interface functions:
-available in Solaris, AIX, Tru64 Unix,...

- `aio_cancel`
  - cancel asynchronous read and/or write requests
- `aio_error`
  - retrieve Asynchronous I/O error status
- `aio_fsync`
  - asynchronously force I/O completion, and sets errno to ENOSYS
- `aio_read`
  - begin asynchronous read
- `aio_return`
  - retrieve return status of Asynchronous I/O operation
- `aio_suspend`
  - suspend until Asynchronous I/O Completes
- `aio_write`
  - begin asynchronous write
- `lio_listio`
  - issue list of I/O requests
For Further Information

The Unix bible:

  Somewhat dated, but still useful.

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.