System-Level I/O

Andrew Case

Slides adapted from Jinyang Li, Randy Bryant and Dave O’Hallaron
Unix I/O and Files

- UNIX abstracts many things into files (just a series of bytes)

- 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
- 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()
Open a file before access:
  - Returns a small integer file descriptor (or -1 for error)

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

Why fd?
- Kernel maintains an array of info on currently opened files for a process
- `fd` indexes into this in-kernel array

Each process starts out with three open files
- 0: standard input
- 1: standard output
- 2: standard error

For more info, do “man 2 open”
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);
}
```
Simple read/write example

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

```c
#include <stdio.h>

int main(void) {
    char c;
    while(read(0, &c, 1) == 1) {
        write(1, &c, 1);
    }
    exit(0);
}
```

- Returns # of bytes read, -1 for error
- Returns # of bytes written, -1 for error
Topics

- Unix I/O
- Metadata, sharing, and redirection
- Standard I/O
- Conclusions and examples
File Metadata (data about 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

/* 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))
        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);
}
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

```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);
}```
Kernel tracks user processes’ opened files

- **Descriptor table**
  - [one table per process]

- **Open file table**
  - [shared by all processes]

- **v-node table**
  - [shared by all processes]

- **kernel state**

  - **File A (terminal)**
    - File pos
    - refcnt=1
    - ...

  - **File B (disk)**
    - File pos
    - refcnt=1
    - ...

- **Info in stat struct**

  - File access
  - File size
  - File type
  - ...

- `stdin` fd 0
- `stdout` fd 1
- `stderr` fd 2
- fd 3
- fd 4
Kernel tracks user processes’ opened files

- Calling `open` twice with the same filename

**Descriptor table**
- [one table per process]

**Open file table**
- [shared by all processes]

**v-node table**
- [shared by all processes]

```
stdin  fd 0
stdout fd 1
stderr fd 2
fd     3
fd     4
```

```
File A (disk)

File pos
refcnt=1
```

```
File B (disk)

File pos
refcnt=1
```

```
File access
File size
File type
```
Child process inherits its parent’s open files

- **Before** fork() call:

<table>
<thead>
<tr>
<th>Descriptor table</th>
<th>Open file table</th>
<th>v-node table</th>
</tr>
</thead>
<tbody>
<tr>
<td>[one table per process]</td>
<td>[shared by all processes]</td>
<td>[shared by all processes]</td>
</tr>
</tbody>
</table>

```
stdin  fd 0
stdout fd 1
stderr fd 2
fd 3
fd 4
```

File A (terminal)
- File access
- File size
- File type
- File pos
- refcnt=1
- ...

File B (disk)
- File access
- File size
- File type
- File pos
- refcnt=1
- ...

File pos
refcnt=1
...
Child process inherits its parent’s open files

- **After fork():**
  - Child’s descriptor table same as parent’s, and +1 to each refcnt
#include <stdio.h>
#include <fcntl.h>

int main(int argc, char *argv[]) {
    int fd1;
    char c1, c2;
    char *fname = argv[1];
    fd1 = open(fname, O_RDONLY, 0);
    read(fd1, &c1, 1);
    if (fork()) { /* Parent */
        read(fd1, &c2, 1);
        printf("Parent: c1 = %c, c2 = %c\n", c1, c2);
    } else { /* Child */
        sleep(5);
        read(fd1, &c2, 1);
        printf("Child: c1 = %c, c2 = %c\n", c1, c2);
    }
    return 0;
}

What would this program print for file containing “abcde”?
I/O Redirection

- How does a shell redirect I/O?
  
  unix$ ls > foo.txt

- Use syscall `dup2(oldfd, newfd)`
  - Copies descriptor table entry `oldfd` to entry `newfd`
Fun with File Descriptors (dup2)

```c
#include <stdio.h>
#include <fcntl.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”?

```bash
unix> ./ffiles1 abcde.txt
c1 = a, c2 = a, c3 = b
```
Shell I/O redirection

Exercise: implement shell I/O redirect functionality using dup2

unix$ ls > foo.txt
I/O Redirection Example

- **Step #1: open file to which stdout should be redirected**
  - Do it in parent or child? Before or after calling `exec`?

**Diagram:**
- **Descriptor table**
  - [one table per process]
- **Open file table**
  - [shared by all processes]
- **v-node table**
  - [shared by all processes]

- **Opened file has fd=4**
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

```c
dup2(4, 1)
```

- Descriptor table [one table per process]
- Open file table [shared by all processes]
- v-node table [shared by all processes]

```
fd 0  fd 1  fd 2  fd 3  fd 4
stdin | stdout | stderr
```

```
File A
File pos
refcnt=0
...  
```

```
File B
File pos
refcnt=2
...  
```

```
File access
File size
File type
...  
```
// Parse commandline to obtain “prog”, its arguments
// and redirected to file “out”…

// Here?
if ((pid = fork())==0) {
    // Here?
    if (execve(prog,…) < 0) {
        // Here?
        printf(“%s: command not found\n”, prog);
        exit(0);
    }
}

if (!bg) {
    // Here?
    waitpid(pid,…);
}

// Where does this code go?
fd = open(out, O_WRONLY|O_CREAT);
dup2(fd,1);
close(fd);
// Parse commandline to obtain “prog”, its arguments
// and redirected to file “out”…

if ((pid = fork())==0) {

    fd = open(out, O_WRONLY|O_CREAT);
    dup2(fd, 1);
    close(fd);

    if (execve(prog,...) < 0) {
        printf("%s: command not found\n", prog);
        exit(0);
    }

    // More code...
}

if (!bg) {
    waitpid(pid,...);
}
Topics

- Unix I/O
- Metadata, sharing, and redirection
- Standard I/O
- Conclusions and examples
Standard I/O Functions

- The C library (libc.so) contains a collection of higher-level standard I/O functions

  ```
  fopen  fclose  fdopen
  fread  fwrite
  fscanf  fprintf
  sscanf  sprintf
  fgets  fputs
  fflush
  fseek
  ```

  Internally invokes I/O syscalls

  ```
  open   read  write  lseek
  stat   close
  ```
Standard I/O Streams

- Standard I/O implements *buffered streams*
  - Abstraction for a file descriptor and a buffer in memory.
- C programs begin life with three open streams
  - `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");
}
```
Buffering in Standard I/O

- **Standard I/O functions use buffered I/O**

- **Buffer flushed to output fd on “\n” or fflush() call**

```c
buf = write(1, buf, 6);

fflush(stdout);

printf("h");
printf("e");
printf("l");
printf("l");
printf("o");
printf("\n");
```
Unix I/O vs. standard I/O

- **Unix I/O:**
  - + most general, lowest overhead (no intermediate functions)
  - + provides functions for accessing file metadata
  - + can be used safely in signal handlers
  - - no built-in buffering

- **Standard I/O:**
  - + buffering increases efficiency by reducing # of **read** and **write** system calls
  - - no metadata access
  - - not appropriate for signal handlers.
  - - not appropriate for input and output on network sockets
Choosing I/O Functions

■ General rule: use the highest-level I/O available
  ▪ Many C programs are able to do all of their work using the standard I/O

■ 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
  ▪ When you are reading and writing network sockets.
  ▪ In rare cases when you can tune for absolute highest performance
Standard I/O Buffering in Action

See buffering in using the `strace` program:

```c
#include <stdio.h>

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

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