Computer Systems Organization

Lecture 16: System-Level I/O

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

Some slides adapted (and slightly modified) from:
• Clark Barrett
• Jinyang Li
• Randy Bryant
• Dave O’Hallaron
I/O Devices

- Very diverse devices
  - behavior (i.e., input vs. output vs. storage)
  - partner (who is at the other end?)
  - data rate

- I/O Design affected by many factors (expandability, resilience)

- Performance:
  - access latency
  - throughput
  - connection between devices and the system
  - the memory hierarchy
  - the operating system

- A variety of different users
Application programs

Language Run-time Systems
high-level facility for I/O (e.g. ANSI C standard I/O)

Kernel – Level I/O system calls

If this is enough

Why bother learning this?
Why Bother?

• Understanding kernel-level I/O will help you understand other systems concepts
  – I/O plays a key role in process creation and execution
  – Process creation plays a key role in how files are shared by different processes

• Sometimes language run-time is not enough to do what you want
Unix I/O

- A file is a sequence of m bytes.
- All I/O devices are modeled as files.
- All I/O is performed by reading and writing the appropriate files.
  - Opening a file: an application wants to use and I/O device. Kernel gives the application a file descriptor (nonnegative integer)
  - Changing the current file position: position is a byte offset from the beginning of the file (kept by kernel)
  - Reading and writing files
  - Closing files
Unix I/O

• UNIX abstracts many things into files
  – E.g. regular files, devices (/dev/sda2), FIFO pipes, sockets

• Allow a common set of syscalls for handling I/O
  – E.g. reading and writing to files/pipes/sockets: read and write
Overview of File System implementation in UNIX

- **Inodes** contain meta-data about files/directories
  - Last modification time, size, user id ...
- **Hard links**: multiple names for the same file (/home/f1.txt and /usr/f2.txt refer to the same file)
UNIX I/O (i.e. I/O related syscalls)

• Getting meta-data (info maintained in i-nodes)
  – Stat

• Directory operations
  – opendir, readdir, rmdir

• Open/close files
  – Open, close

• Read/write files
  – read/write
File Metadata

- Access file meta-data using `stat` syscall

Example: rkmatch.c

```c
void read_file(const char *fname, char **doc, int *doc_len)
{
    struct stat st;
    ...
    if (stat(fd, &st) != 0) {
        perror("read_file: fstat ");
        exit(1);
    }
    
    *doc = (char *)malloc(st.st_size);
    ...
}
```
File Metadata

• Access file meta-data using `stat` syscall

```c
struct stat {
    dev_t st_dev;  /* ID of device containing file */
    ino_t st_ino;  /* inode number */
    mode_t st_mode; /* protection */
    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 ID (if special file) */
    off_t st_size; /* total size, in bytes */
    blksize_t st_blksize; /* block size for file system I/O */
    blkcnt_t st_blocks; /* number of 512B 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 status change */
};
```
Opening Files

• **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
Closing Files

• Closing a file informs kernel that you are finished accessing that file

```c
int fd;    /* file descriptor */
if (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(STDIN_FILENO, &c, 1) == 1){
        write(STDOUT_FILENO, &c, 1);
    }
    exit(0);
}
```

Returns # of bytes read, -1 for error
Returns # of bytes written, -1 for error
Kernel Presentation of Open Files

- Kernel uses 3 related data structures to represent open files
  - **Descriptor table:**
    - per process
    - Indexed by the process open file descriptor
    - Each entry points to an entry in the file table
  - **File table:**
    - Shared by all processes
    - Each entry contains info about file position, reference count, ..., and a pointer to an entry in the v-node table
  - **v-node table:**
    - Shared by all processes
    - contains info that can be read by stat syscall
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]

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

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

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

- File access
- File size
- File type
- Info in stat struct
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]

  ![Diagram showing file descriptors and file access details](image-url)
Child process inherits its parent’s open files

- **Before** `fork()` call:

**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 (terminal)
- 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

- **After fork()**: Child’s descriptor table same as parent’s, and +1 to each refcnt

**Diagram**

- **Descriptor table** [one table per process]
- **Open file table** [shared by all processes]
- **v-node table** [shared by all processes]
What would this program print for file containing "abcde"?

```c
#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;
}
```

Solution:

Parent: c1 = a, c2 = b
Child: c1 = a, c2 = c
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;
}
```

Solution:

c1 = a, c2 = a, c3 = b

- 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`

<table>
<thead>
<tr>
<th>Descriptor table before <code>dup2(4, 1)</code></th>
<th>Descriptor table after <code>dup2(4, 1)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>fd 0</td>
<td>fd 0</td>
</tr>
<tr>
<td>fd 1</td>
<td>fd 1</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>fd 2</td>
<td>fd 2</td>
</tr>
<tr>
<td>fd 3</td>
<td>fd 3</td>
</tr>
<tr>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>fd 4</td>
<td>fd 4</td>
</tr>
</tbody>
</table>
I/O Redirection Example

- Step #1: open output file to which stdout should be redirected

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

<table>
<thead>
<tr>
<th>stdin</th>
<th>fd 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>stdout</td>
<td>fd 1</td>
</tr>
<tr>
<td>stderr</td>
<td>fd 2, fd 3, fd 4</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Opened file has fd=4

- File A
  - File pos
  - refcnt=1
  - ...

- File B
  - File pos
  - refcnt=1
  - ...

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

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

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

- 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]
  - stdin: fd 0
  - stdout: fd 1
  - stderr: fd 2
  - fd 3
  - fd 4

- **Open file table**
  - [shared by all processes]
  - File A
    - File pos
    - refcnt=0
    - ...
  - File B
    - File pos
    - refcnt=2
    - ...

- **v-node table**
  - [shared by all processes]
  - File access
  - File size
  - File type
    - ...

- **Nodes:**
  - File pos table
  - refcnt table
  - Other file attributes
Standard I/O Functions

- The C library (`libc.so`) contains a collection of higher-level *standard I/O* functions:
  - `fopen`, `fdopen`, `fread`, `fwrite`, `fscanf`, `fprintf`, `sscanf`, `sprintf`, `fgets`, `fputs`, `fflush`, `fseek`, `fclose`

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(stderr, "Hello, world\n");
}
```
Unix I/O vs. standard I/O

- **Unix I/O:**
  - **Pros**
    - most general, lowest overhead.
    - All other I/O packages are implemented using Unix I/O functions.
    - Provides functions for accessing file metadata.
    - async-signal-safe and can be used safely in signal handlers.
  - **Cons**
    - Efficient reading/writing may require some form of buffering
Unix I/O vs. Standard I/O:

• Standard I/O:
  – Pros:
    • Buffering increases efficiency by reducing # of read and write system calls
  – Cons:
    • Provides no function for accessing file metadata
    • Not async-signal-safe, and 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 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
  – Inside signal handlers, because Unix I/O is async-signal-safe.
  – When working with network sockets
  – In rare cases when you want to tune for absolute highest performance.
You can see this buffering in action for yourself – use `strace` to monitor a program's syscall invocation:

```c
#include <stdio.h>
void main()
{
    char c;
    while ((c = getc(stdin))!='\n') {
        printf("%c",c);
    }
    printf("\n");
}
```

```
linux% strace ./a.out
execve("./a.out", ["./a.out"], [/* ... */]).
... 
read(0,"hello\n", 1024) = 6
write(1, "hello\n", 6) = 6
... 
exit_group(0) = ?
```
Conclusions

• UNIX/LINUX use files to abstract many I/O devices
• Accessing files can be done either by standard I/O or UNIX I/O