

# File I/O and Standard I/O

---

Advanced Programming in the UNIX Environment

Chun-Ying Huang <chuang@cs.nctu.edu.tw>

# Outline

---

Introduction

File I/O Functions

File I/O Issues

Standard I/O Functions

# Introduction

---

# File Descriptors

---

In the kernel, all opened files are referred to by file descriptors

It is a non-negative integer

A convention for shells and many applications

- File descriptor 0, 1, and 2 refers to standard input, output, and error, respectively
- `STDIN_FILENO (0)`, `STDOUT_FILENO (1)`, `STDERR_FILENO (2)`  
-- defined in the header file `unistd.h`

The file descriptors can be used in a process is ranged from 0 to `OPEN_MAX-1`

- Can be changed using the `setrlimit(2)` function
- But it requires root permissions

# Standard Input, Output, and Error

---

A shell creates standard input, standard output, and standard error when it runs a program

Standard input, output, and error can be *piped* and/or *redirected*

## Examples – The “cat” program

- `$ cat /etc/passwd`
- `$ cat < /etc/passwd`
- `$ cat /etc/passwd | cat | cat`
- `$ cat /etc/passwd | cat | cat > /tmp/p.txt`

# File I/O VS Standard I/O

---

## File I/O (Unbuffered I/O)

- Access via file descriptors
- Talk to the kernel directly

## Standard I/O (Buffered I/O)

- Access via wrapped file descriptors, i.e., the FILE data structure
- Default wrappers for standard input, output, and errors
- stdin, stdout, and stderr
- The fileno function

# Unbuffered I/O

---

## Definition

- Each `read` or `write` invokes a system call in the kernel
- Not buffered in user space programs and libraries

Usually can be performed by using only the five functions

- `open`, `read`, `write`, `lseek`, and `close`

# File I/O Functions

---



# The open(2) Function

---

Open a file

## Synopsis



- `int open(const char *pathname, int flags, mode_t mode);`
- Returns: file descriptor opened for write-only if OK, -1 on error

## Mandatory flags

- `O_RDONLY`
- `O_WRONLY`
- `O_RDWR`



## Common optional flags

- `O_APPEND`
- `O_CREAT`
- `O_EXCL`
- `O_TRUNC`
- `O_SYNC`

# The creat(2) Function

---

Open a file for write only

## Synopsis

- `int creat(const char *pathname, mode_t mode);`
- Returns: file descriptor if OK, -1 on error

It is equivalent to

- `open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);`

# The close(2) Function

---

Close a opened file

## Synopsis

- `int close(int fildes);`
- Returns: 0 if OK, -1 on error

When a process terminates, all of its open files are closed automatically

# The lseek(2) Function

---

Move the “file pointer” position

## Synopsis

- `off_t lseek(int fd, off_t offset, int whence);`
- Returns: new file offset if OK, -1 on error
- Usually `off_t` is 32-bit long
- Consider the `lseek64()` function using `off64_t`

## Choices of *whence*

- `SEEK_SET, SEEK_CUR, SEEK_END`

Can be used to determine if a file is seekable

# Large File Support in Linux

---

Handle file sizes that are greater than 2GB

- `_LARGEFILE64_SOURCE` and `_LARGEFILE_SOURCE` must be defined
- Open a file with the option `O_LARGEFILE`
- Data type `off_t` is 32-bit signed integer on 32-bit architecture
- `#define _FILE_OFFSET_BITS 64` // ensure that `off_t` is 64-bit
- `lseek64()` and `off64_t` are available when `_LARGEFILE64_SOURCE` is defined

The macro(s) must be defined before including any other C library headers!

- These macros belong to feature test macros defined by the C library
- See `feature_test_macros(7)` manual page
- Compiler options to enable large file support

```
gcc -D_LARGEFILE64_SOURCE -D_LARGEFILE_SOURCE -D_FILE_OFFSET_BITS=64 ...
```

# Detect Capability of Seeking

---

```
#include "apue.h"    /* fig 3.1 */
int main(void) {
    if (lseek(STDIN_FILENO, 0, SEEK_CUR) == -1)
        printf("cannot seek\n");
    else
        printf("seek OK\n");
    exit(0);
}
```

```
$ ./a.out < /etc/motd
seek OK
$ cat < /etc/motd | ./a.out
cannot seek
$ ./a.out < /var/spool/cron/FIFO
cannot seek
```



# File Hole

---

## The `lseek(2)` and the `write(2)` function

```
#include "apue.h"      /* fig 3.2 */
#include <fcntl.h>
char    buf1[] = "abcdefghij", buf2[] = "ABCDEFGHIJ";
int main(void) {      int fd;
    if ((fd = creat("file.hole", FILE_MODE)) < 0)
        err_sys("creat error");
    if (write(fd, buf1, 10) != 10)
        err_sys("buf1 write error");
    /* offset now = 10 */
    if (lseek(fd, 16384, SEEK_SET) == -1)
        err_sys("lseek error");
    /* offset now = 16384 */
    if (write(fd, buf2, 10) != 10)
        err_sys("buf2 write error");
    /* offset now = 16394 */
    exit(0);
}
```

# File Hole (Cont'd)

---

The resulted file with a hole

```
$ ./fig3.2-hole
$ ls -l file.hole
-rw-r--r-- 1 chuang chuang 16394 2009-01-01 11:45 file.hole
$ hexdump -C file.hole
00000000  61 62 63 64 65 66 67 68  69 6a 00 00 00 00 00 00 |abcdefghij.....|
00000010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 |.....|
*
00004000  41 42 43 44 45 46 47 48  49 4a                                |ABCDEFGH IJ|
0000400a
```

Compare with a no-hole file

```
$ ls -ls file.*
 8 -rw-r--r-- 1 chuang chuang 16394 2009-01-01 11:45 file.hole
20 -rw-r--r-- 1 chuang chuang 16394 2009-01-01 11:49 file.nohole
```



# The read(2) Function

---

Read from an opened file

## Synopsis

- `ssize_t read(int fd, void *buf, size_t nbytes);`
- Returns: number of bytes read, 0 if EOF, -1 on error

File offset moves forward after read

Read bytes are usually less than requested

- Regular file: A special case when *read* encounters EOF
- Network: Depends on network buffering state
- Pipe or FIFO: Read all available data
- The read operation may be interrupt by signals

# The write(2) Function

---

Write data to an opened file

## Synopsis

- `ssize_t write(int fd, const void *buf, size_t nbytes);`
- Returns: number of bytes written if OK, -1 on error

File offset moves forward after write

# File I/O: Other Issues

---

# I/O Efficiency

---

## A simple “cat” program

- How to choose the size of a buffer?

```
#include "apue.h"
#define      BUFFSIZE      4096
int main(void) {
    int          n;
    char  buf[BUFFSIZE];
    while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("write error");
    if (n < 0)
        err_sys("read error");
    exit(0);
}
```

# I/O Efficiency (Cont'd)

---

Test with the command

- `$ ./fig3.4-mycat < /path/to/a/file > /dev/null`

BUFFSIZE	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)	# of loops
1	11.425	81.993	93.412	134,217,728
4	2.680	20.965	23.645	33,554,432
16	0.628	5.216	5.845	8,388,608
64	0.148	1.344	1.492	2,097,152
256	0.044	0.340	0.384	524,288
1024	0.012	0.108	0.118	131,072
4096	0.000	0.052	0.052	32,768
16384	0.000	0.040	0.040	8,192
65536	0.000	0.036	0.036	2,048
262144	0.000	0.032	0.034	512

# File Sharing – An Overview

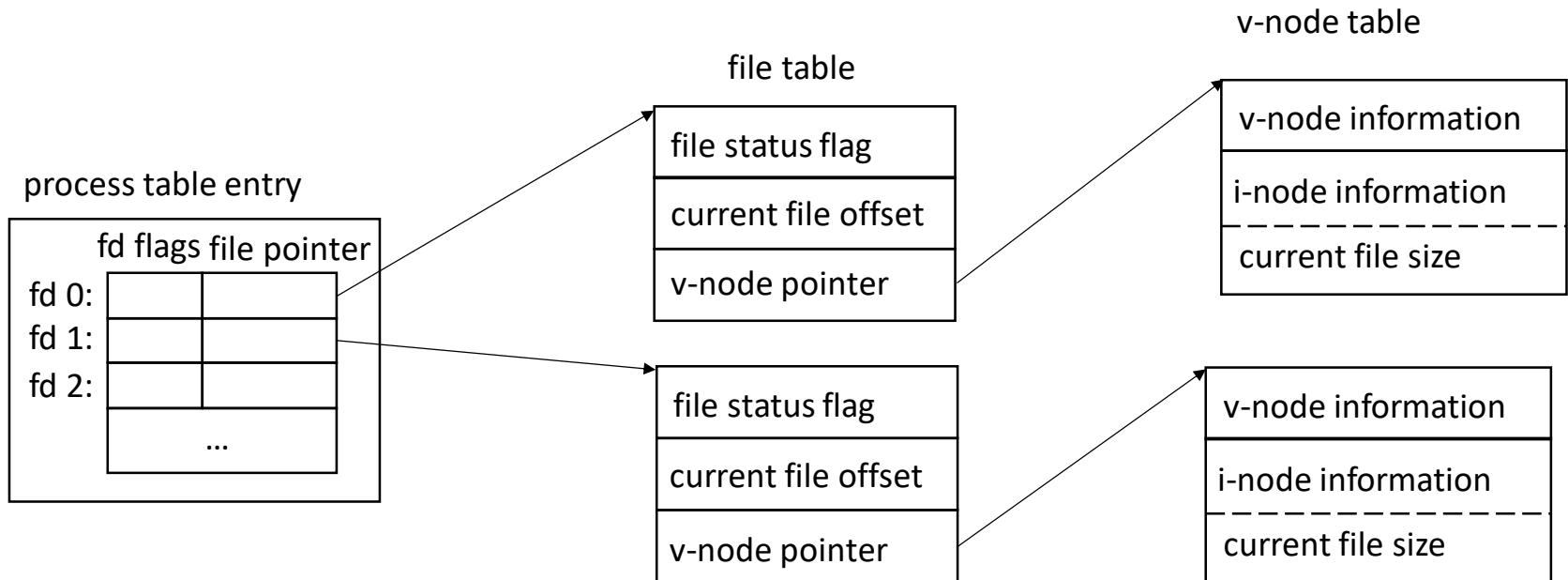
---

An opened file can be shared among different processes

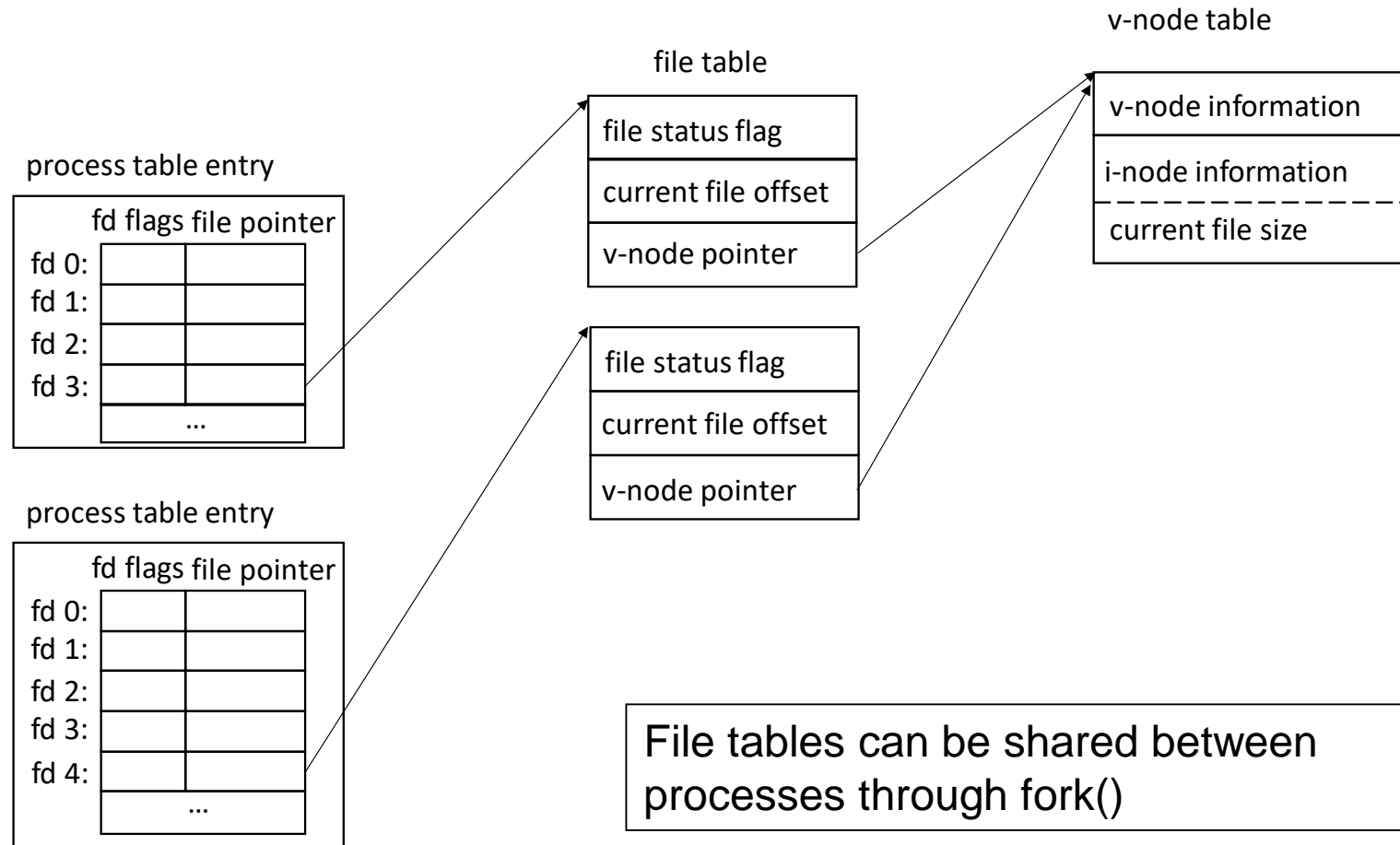
The kernel maintains several different data structures for opened files

- Each process has an entry in the *process table*
- Each process table entry contains *a table of opened file descriptors*
- *A file table* for all opened files
- Each file table is associated with a *v-node structure*

# File Sharing – Kernel Data Structures



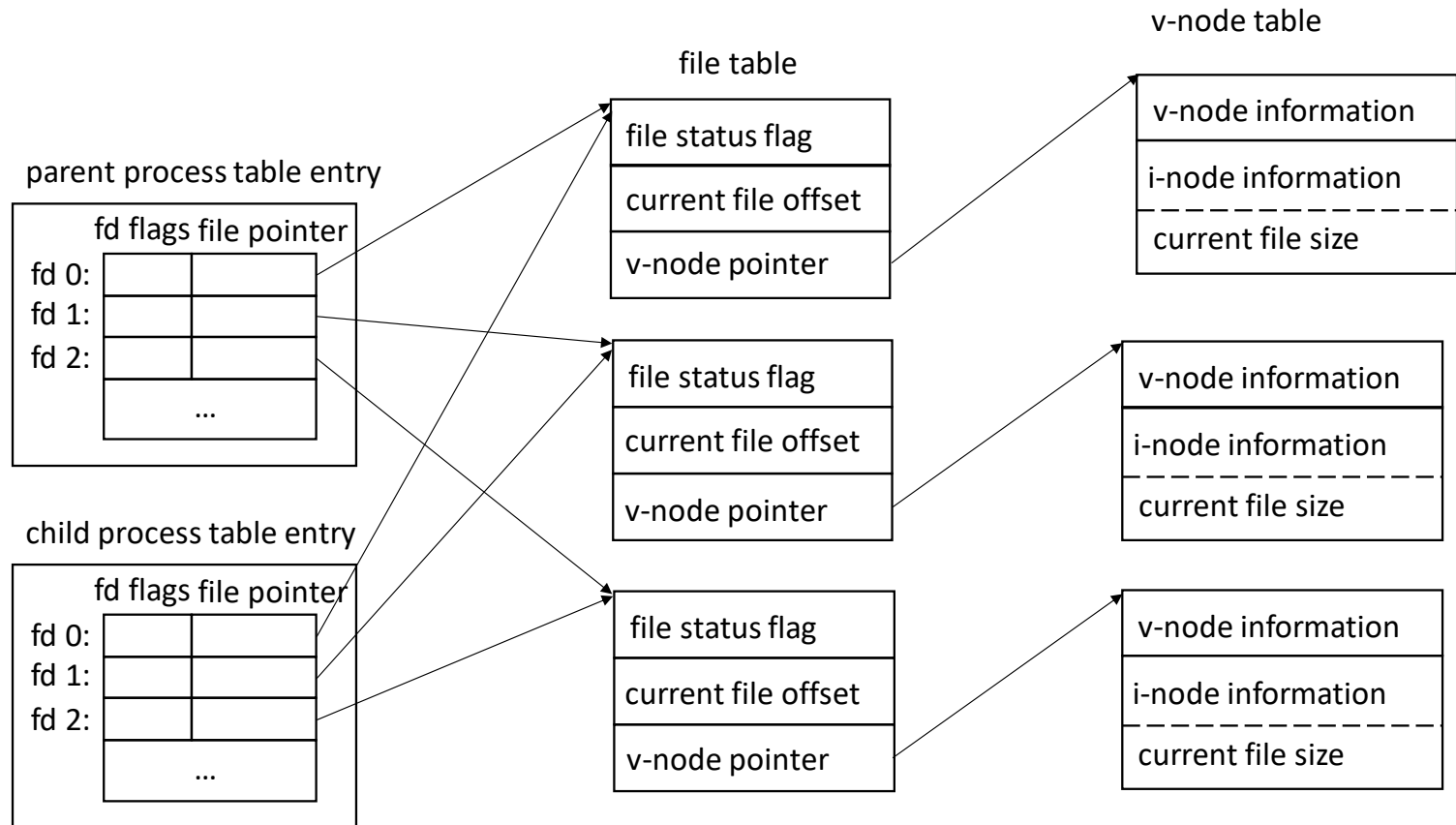
# File Sharing – Two Independent Processes, Open The Same File





# fork and File Sharing

## – from Chapter 8



# Atomic Operation

---

Consider the following program

```
if (lseek(fd, 0, SEEK_END) < 0) /* position to EOF */  
    err_sys("lseek error");  
if (write(fd, buf, 100) != 100) /* and write */  
    err_sys("write error");
```

What happens if *the parent and the child processes* do the same thing to the same file?

Any operation that requires more than one function call cannot be atomic!

# Atomic Operation (Cont'd)

---

Consider the following multi-threaded (-process) program

- Thread A (Process A)

```
if (lseek(fd, 100, SEEK_SET) < 0) /* position to 100 */  
    err_sys("lseek error");  
if (write(fd, "a", 1) != 1)      /* and write */  
    err_sys("write error");
```

- Thread B (or fork of Process A)

```
if (lseek(fd, 200, SEEK_SET) < 0) /* position to 200 */  
    err_sys("lseek error");  
if (write(fd, "b", 1) != 1)      /* and write */  
    err_sys("write error");
```

# pread And pwrite

---

## Atomic seek and read/write

### Synopsis

- `ssize_t pread(int fd, void *buf, size_t nbytes, off_t offset);`
- Returns: number of bytes read, 0 if EOF, -1 on error
- `ssize_t pwrite(int fd, const void *buf, size_t nbytes, off_t offset);`
- Returns: number of bytes written if OK, -1 on error

### Seek first and then read or write

There is no way to interrupt the two operations

The current file offset is not updated

# Atomic Creating of A Non-Existing File

---

Create a file if it does not exist

```
if ((fd = open(pathname, O_WRONLY)) < 0) {  
    if (errno == ENOENT) {  
        if ((fd = creat(pathname, mode)) < 0)  
            err_sys("creat error");  
    } else {  
        err_sys("open error");  
    }  
}
```

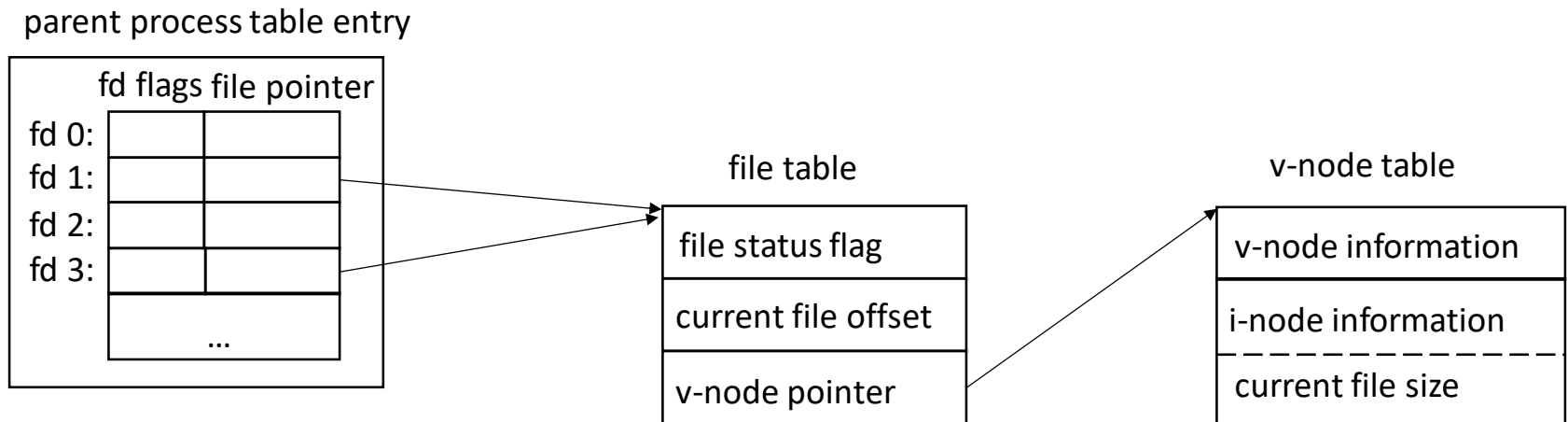
Can be atomically done using open with O\_CREAT and O\_EXCL

- `open(pathname, O_CREAT | O_EXCL, mode)`

# dup And dup2 Functions

dup: Duplicate a file descriptor

dup2: Duplicate a file descriptor to a targeted descriptor



# dup And dup2 Functions (Cont'd)

---

## Synopsis

- `int dup(int fd);`
- `int dup2(int fd, fd2);`
- Returns: both return the new file descriptor if OK, -1 on error

## Equivalent operations

- `dup (fd)`
  - `fcntl(fd, F_DUPFD, 0);`
- `dup2 (fd, fd2)`
  - `close(fd2);`  
`fcntl(fd, F_DUPFD, fd2);`
  - `dup2` is an atomic operation

# sync, fsync, And fdatasync Functions

---

## *Delayed write*

- When data is copied to the kernel, it is queued for writing to disk at some later time

Ask the kernel *starting* to write cached disk blocks

## For a specific file

- `int fsync(int fd); /* filedata + metadata */`
- `int fdatasync(int fd); /* filedata only */`
- Return values for the above two functions: 0 if OK, -1 on error

## For all files

- `void sync(void); /* filedata + metadata */`



# The fcntl Function

---

Change the properties of an opened file

## Synopsis

- `int fcntl(int fd, int cmd, ... /* int arg */);`
- Returns: depends on cmd if OK, -1 on error

## Common commands

- `F_DUPFD` – duplicate the file descriptor
- `F_GETFD/F_SETFD` – get or set the file descriptor flag
  - supports only `FD_CLOEXEC` (close-on-exec)
- `F_GETFL/F_SETFL` – get or set the file status flags
  - `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_APPEND`, `O_NONBLOCK`, `O_SYNC`, ...

# The fcntl Function – Change Status Flags

---

Enable:      `val |= flags;`

Disable:     `val &= ~flags;`

```
#include "apue.h"
#include <fcntl.h>
void set_fl(int fd, int flags)
    /* flags are file status flags to turn on */
{
    int val;

    if ((val = fcntl(fd, F_GETFL, 0)) < 0)
        err_sys("fcntl F_GETFL error");
    val |= flags;          /* turn on flags */
    if (fcntl(fd, F_SETFL, val) < 0)
        err_sys("fcntl F_SETFL error");
}
```

# The ioctl(2) Function

---

The ultimate function to control all I/O operations

## Synopsis

- `int ioctl(int fd, int request, ...);`
- Returns: -1 on error, something else if OK

*There is no standard for the `ioctl` function*

- Each device driver can define its own set of `ioctl` commands, a common method to handle user-kernel interaction

The *request* is a device dependent request code

The third argument is usually an untyped pointer to memory

# ioctl Sample Codes

---

Please install the module first

- Samples: files/module
- You have to ensure that you have kernel header files installed
  - Arch Linux: linux414-headers (for kernel version 4.14.xxx)
  - Ubuntu Linux: linux-headers-4.15.0-33-generic (for kernel 4.15.0-33-generic)
  - Check the version with the command: `$ uname -a`
- The kernel module
  - Please read the codes, and the Linux Cross Reference (LXR)
  - <https://elixir.bootlin.com/linux/latest/source>
  - structu file\_operations:  
<https://elixir.bootlin.com/linux/v4.14/source/include/linux/fs.h#L1692>

The ioctl sample: Please read the codes!

# /dev/fd

---

Modern systems provide a `/dev/fd` directory

It is a virtual file system

Each process has its own view of `/dev/fd`

Assume that descriptor  $n$  has been opened, open the file `/dev/fd/ $n$`  is equivalent to duplicate descriptor  $n$

- `fd = open("/dev/fd/0", mode);`
- `fd = dup(0);`

One use of the `/dev/fd` files is from the shell

Allow programs to handle standard input and standard output as regular files

# /dev/fd: Identify Opened Files

---

Remember: Almost all opened resources have a descriptor

We are looking at opened files

Example: Open a temp file and get its filename

- `tmpnam` – Return a temp file name (and then open by the user) → May conflict with other processes
- `mkstemp` – Create and open a temp file, and then return its descriptor → Will not have a conflict

You can also determine the temp file name from `/dev/fd/ ...`

# /dev/fd: Identify Opened Files

## – Example

---

See new example files: `files/tmpnam.c`

Look at the code

Check the actual value of `/dev/fd` ➔ `/proc/self/fd`

Check the pid of the process

Check `/proc/[pid]/fd`

# /dev/fd – Identify Network Connections – Example

---

This time, we are going to work with the **telnet** program

```
$ ssh linux1.cs.nctu.edu.tw
```

Check the pid of ssh,

Check /proc/[pid]/fd, what do you find?

Check /proc/net/{tcp, udp, raw, unix}, v4 and v6 files – or simply do a 'grep'

The 'netstat -nap' command



# Standard I/O Functions

---

# Standard (Buffered) I/O

---

Standard I/O handles ...

- Buffer allocation
- Perform I/O in optimal-sized chunks

Standard I/O is easier to use

The FILE structure

- Treat all opened files as a stream
- Associate the stream with an underlying file descriptor
- Maintain buffer states

# Buffering

---

## Fully buffered

- Files residing on disk are normally fully buffered by the standard I/O library
- The buffer used is usually obtained by one of the standard I/O functions calling *malloc* the first time I/O is performed on a stream

## Line buffered

- the standard I/O library performs I/O when a newline character is encountered on input or output
- Caveats
  - Buffer size is limited – I/O may perform before seeing a newline
  - Before read, all line-buffered output streams are flushed

## Unbuffered

# Default Buffering Modes

---

## ISO C

- Standard input and standard output are fully buffered, if and only if they *do not refer to an interactive device*
- Standard error is never fully buffered

## Most default implementations

- Standard error is always unbuffered
- All other streams are line buffered if they refer to a terminal device; otherwise, they are fully buffered

# Functions for Setting Buffer

---

## Synopsis

- *void setbuf(FILE \*fp, char \*buf);*
  - *buf* must be the size of *BUFSIZ*
- *int setvbuf(FILE \*fp, char \*buf, int mode, size\_t size);*
- Returns: 0 if OK, nonzero on error

Buffering is disabled if *buf* is NULL


## Buffering mode

- *\_IOFBF*: fully buffered
- *\_IOLBF*: line buffered
- *\_IONBF*: unbuffered

# Open Files

---

## Open files

- *FILE\** *fopen(char \*pathname, char \*mode);*
- *FILE \*freopen(char \*pathname, char \*mode, FILE \*fp);*
- *FILE \*fdopen(int fd, char \*mode);* 
- Returns: file pointer if OK, NULL on error

## modes

- **r** or **rb**: open for reading
- **w** or **wb**: truncate to 0 length or create for writing
- **a** or **ab**: append, open for writing at EOF or create for writing
- **r+**, **r+b**, or **rb+**: open for reading and writing
- **w+**, **w+b**, or **wb+**: equivalent to w or wb plus reading
- **a+**, **a+b**, or **ab+**: equivalent to a or ab plus reading
- Note: UNIX does not require **t** (text) mode for text files

# Read and Write a String – By Character

---

## Read

- *int getc(FILE \*fp);*                      *int fgetc(FILE \*fp);*
- *int getchar(void);*
- Returns: next character if OK, *EOF* on EOF or error

## How to tell EOF or error?

- *int ferror(FILE \*fp);*      *int feof(FILE \*fp);*
- Returns: nonzero (true) if a condition is true, or zero (false) otherwise

## Write

- *int putc(int c, FILE \*fp);*                      *int fputc(int c, FILE \*fp);*
- *int putchar(int c);*
- Returns: c if OK, *EOF* on error

# Read and Write a String – By Line

---

## Read

- *char \*fgets(char \*buf, int n, FILE \*fp);*
- *char \*gets(char \*buf);*
- Returns: buf if OK, NULL on end of file or error

## Write


- *int fputs(char \*str, FILE \*fp);*
- *int puts(char \*str);*
- Returns: non-negative value if OK, EOF on error



# Standard I/O Efficiency

---

Performance of reading a 98.5MB file from stdin (roughly 3 million lines) and writing to stdout (/dev/null)

Function	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)
Best from unbuffered I/O	0.01	0.18	6.67
fgets, fputs	2.58	0.19	7.15
getc, putc	10.84	0.27	12.07
fgetc, fputc	10.44	0.27 	11.42
single byte from unbuffered I/O	124.89	161.65	288.64

# Binary I/O

---

## Synopsis

- *size\_t fread(void \*ptr, size\_t size, size\_t nobj, FILE \*fp);*
- *size\_t fwrite(void \*ptr, size\_t size, size\_t nobj, FILE \*fp);*
- Returns: number of objects read or written

# Binary I/O – Examples

---

## Example #1

```
float data[10];  
if (fwrite(&data[2], sizeof(float), 1, fp) != 1)  
    err_sys("fwrite error");
```

## Example #2

```
struct {  short    count;  
         long     total;  
         char     name[NAMESIZE];  
} item;  
if (fwrite(&item, sizeof(item), 1, fp) != 1)  
    err_sys("fwrite error");
```

## Notes

- The offset of a member within a structure can differ between compilers and systems
- The binary formats used to store multibyte integers and floating-point values differ among machine architectures

# Positioning a Stream

---

Similar to *seek* ...

- *int fseek(FILE \*fp, long offset, int whence);*
- *long ftell(FILE \*fp);*
- *void rewind(FILE \*fp);*

# Temporary Files

---

## Create a temporary file

### Synopsis

- *char \*tmpnam(char \*ptr);*
- Returns: pointer to unique pathname
- *FILE \*tmpfile(void);*
- Returns: file pointer if OK, NULL on error

### It is not recommend to use *tmpnam*

- It uses a static buffer to store generated filename
- As the generated name are usually /tmp/fileXXXXXX, it might be guessed
- Solutions
  - Use *tmpfile* or *mkstemp* instead
  - Open the temporary file using *open(2)* with the *O\_EXCL* flag

# Q & A

---