System Programming

Prof. Chuan-Ju Wang
Dept. of Computer Science
University of Taipei

File Sharing

- UNIX is a multi-user/multi-tasking system.
  - It is conceivable (and useful) if more than one process can act on a single file simultaneously.
- In order to understand how this is accomplished, we need to examine some kernel data structures which relate to files.

File Sharing

- Each process table entry has a table of file descriptors, which contain
  - the file descriptor flags (`fcntl(2)`)
  - a pointer to a file table entry
- The kernel maintains a file table; each entry contains
  - file status flags (`O_APPEND`, `O_SYNC`, `O_RDONLY`, etc.)
  - current offset
  - pointer to a vnode table entry
- A vnode structure contains
  - vnode information
  - inode information (such as current file size)

File Sharing

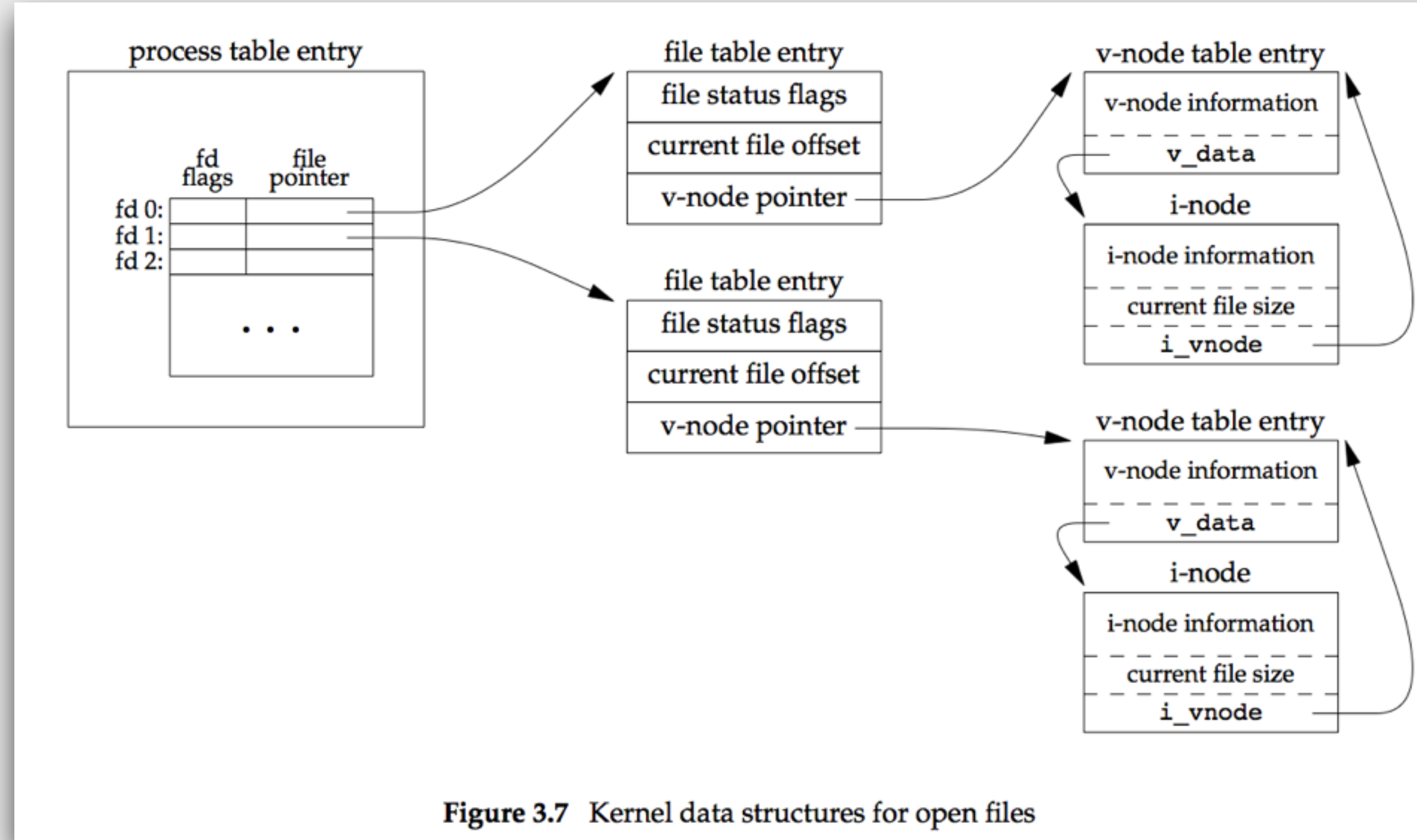


Figure 3.7 Kernel data structures for open files

Data Structures for Files

- The operating system (OS) has three kinds of data structures for files:
  - File table entry: It has one of these for each file descriptor for a user. It contains information like the current `lseek` pointer, and a pointer to a "vnode" (see below). So, when you start your program, the OS has three file table entries for you --- one each for `stdin`, `stdout`, `stderr`. Each time you call `open()`, a new file table entry is created for you in the OS.
  - vnode: There is one of these for each physical file that has been opened. It contains a pointer to the file's inode, the file's size, etc.
  - inode: There is one of these for each file on disk. It contains all the information returned by **`stat()`**.

Data Structures for Files

- The difference between a vnode and an inode is where it's located and when it's valid.
  - Inodes are located on disk and are always valid because they contain information that is always needed such as ownership and protection.
  - Vnodes are located in the operating system's memory, and only exist when a file is opened. However, just one vnode exists for every physical file that is opened.

Data Structures for Files

- The OS has created two file table entries, one for each `open( )` call, but only one vnode.
  - This is because there is only one file. Both file table entries point to the same vnode, but they each have different seek pointers.

```
main()
{
  int fd1, fd2;

  fd1 = open("file1", O_WRONLY | O_CREAT | O_TRUNC, 0644);
  fd2 = open("file1", O_WRONLY);

}
```

Data Structures for Files

See `fs1.c fs1a.c`

```
main()
{
  int fd1, fd2;

  fd1 = open("file1", O_WRONLY | O_CREAT | O_TRUNC, 0644);
  fd2 = open("file1", O_WRONLY);

  write(fd1, "Jim\n", strlen("Jim\n"));
  write(fd2, "Plank\n", strlen("Plank\n"));

  close(fd1);
  close(fd2);
}
```

- The first `write()` call will write the string "Jim\n" into file1.
- Then the second `write()` call will overwrite it with"Plank \n".
  - This is because each `fd` points to its own file table entry, which has its own lseek pointer, and thus the first `write()` does not update the `lseek` pointer of the `fd2`.

http://web.eecs.utk.edu/~huangj/cs360/360/notes/Dup/lecture.html

File Sharing

- Knowing this, here's what happens with each of the calls we discussed earlier:
  - after each `write` completes, the current file offset in the file table entry is incremented. (If current file offset > current file size, change current file size in i-node table entry.)
  - If file was opened `O_APPEND` set corresponding flag in file status flags in file table. For each `write`, current file offset is first set to current file size from the i-node entry.
  - `lseek` simply adjusts current file offset in file table entry
  - to `lseek` to the end of a file, just copy current file size into current file offset.

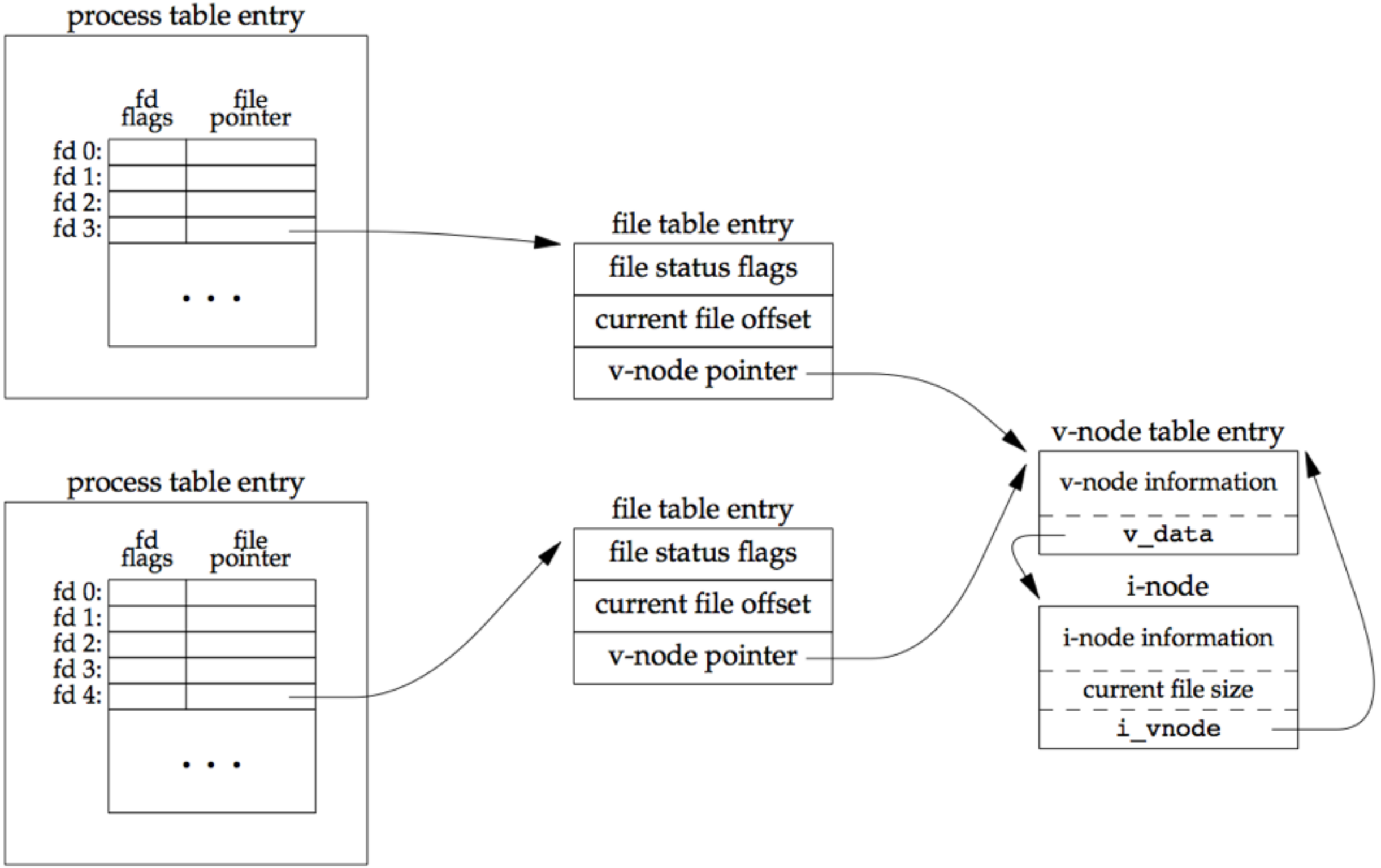


Figure 3.8 Two independent processes with the same file open

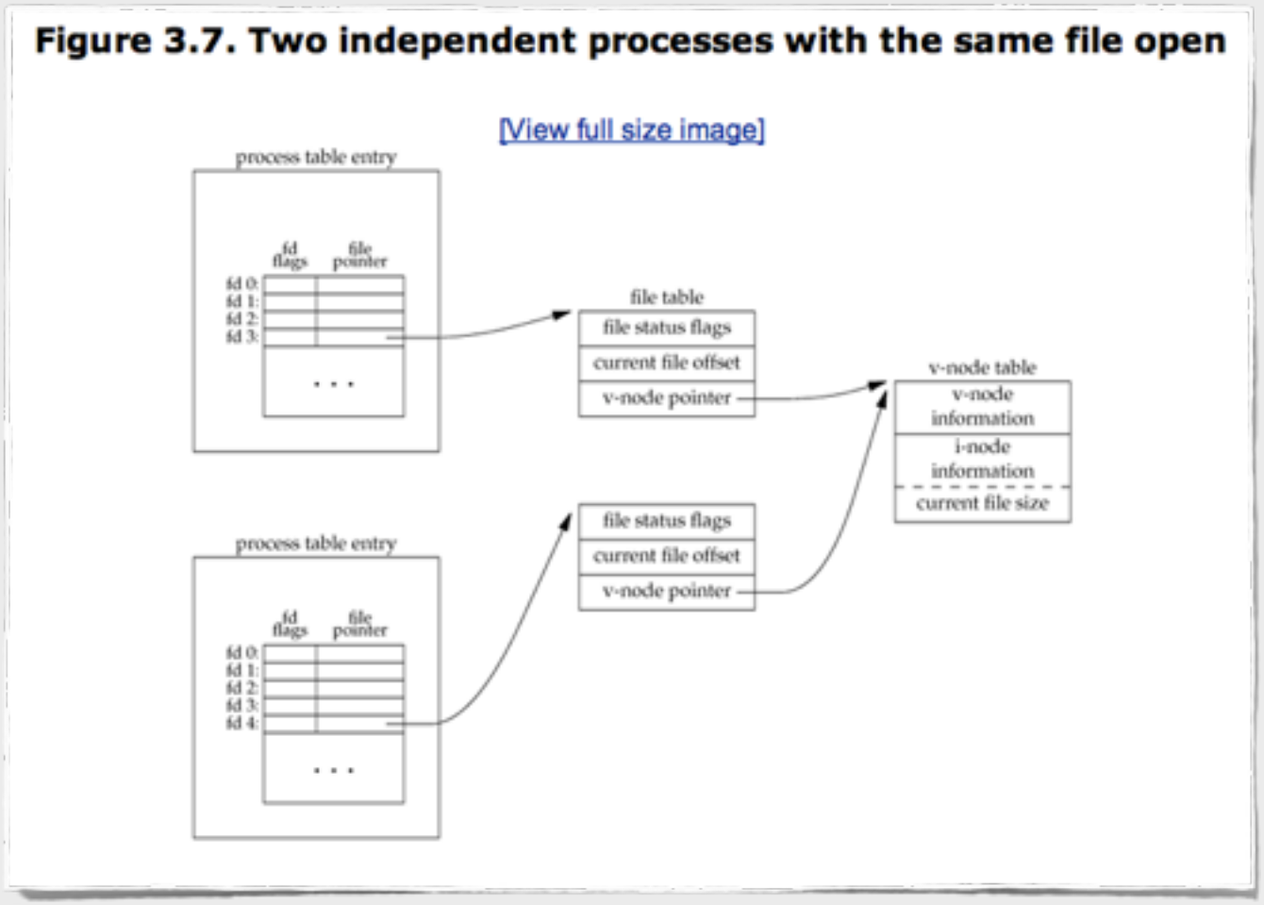
Atomic Operations

- In order to ensure consistency across multiple writes, we require atomicity in some operations.
- An operation is atomic if either all of the steps are performed or none of the steps are performed.
- Suppose UNIX didn't have `O_APPEND` (early versions didn't).
- To append, you'd have to do this:

```
if (lseek(fd, 0L, 2) < 0)                    /* position to EOF */
   err_sys("lseek error");
if (write(fd, buf, 100) != 100)              /* and write */
   err_sys("write error");
```

- What if another process was doing the same thing to the same file?

Atomic Operations



- Process A does the `lseek`, which sets the current offset for the file for process A to byte offset 1,500.
- Then the kernel switches processes, and B continues running.
- Process B then does the `lseek`, which sets the current offset for the file for process B to byte offset 1,500 also
- Then B calls write.
  - Increments B's current file offset for the file to 1,600.
  - The kernel also updates the current file size in the v-node to 1,600.
- Then the kernel switches to process A.
- A calls write.
  - the data is written starting at the current file offset for A, which is byte offset 1,500.
  - This overwrites the data that B wrote to the file.

Atomic Operations

- The problem: Our logical operation of "position to the end of file and write" requires two separate function calls.
- The solution: To have the positioning to the current end of file and the write be an atomic operation with regard to other processes.
- The UNIX System provides an atomic way to do this operation if we set the `O_APPEND` flag when a file is opened.
  - This causes the kernel to position the file to its current end of file before each write.

Atomic Operations

- Another example of an atomic operation
  - The `O_CREAT` and `O_EXCL` options for the `open` function
  - The `open` will fail if the file already exists.
  - The check for the existence of the file and the creation of the file was performed as an atomic operation.

```
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");
    }
}
```

If that other process creates the file and writes something to the file, that data is erased when this `creat` is executed.

  - What will happen if the file is created by another process between the `open` and the `creat`?

Atomic Operations

- Atomic operation refers to an operation that might be composed of multiple steps.
- If the operation is performed atomically,
  1) either all the steps are performed, or
  2) none are performed.
- It must not be possible for a subset of the steps to be performed!!

dup(2) and dup2(2) Functions

- An existing file descriptor is duplicated by either of the following functions.

```
#include <unistd.h>

int dup(int filedes);

int dup2(int filedes, int filedes2);


                Both return: new file descriptor if OK, 1 on error
```

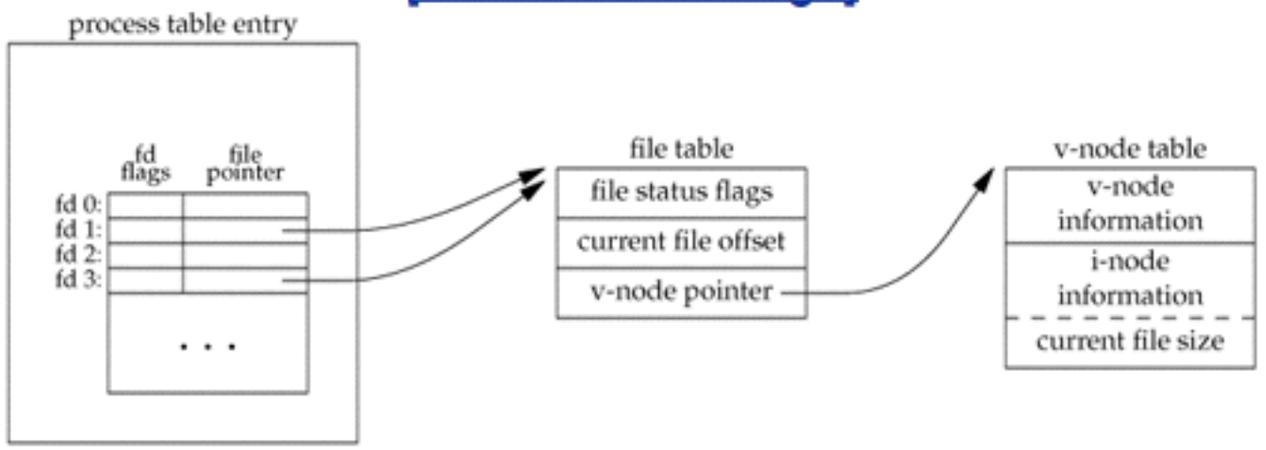
  - The new file descriptor returned by `dup(2)` is guaranteed to be the lowest-numbered available file descriptor.
  - With `dup2(2)`, we specify the value of the new descriptor with the `filedes2` argument.
    - If `filedes2` is already open, it is first closed.
    - If `filedes` equals `filedes2`, then `dup2(2)` returns `filedes2` without closing it.

`dup(2)` and `dup2(2)` Functions

- The new file descriptor that is returned as the value of the functions shares the same file table entry as the *`filedes`* argument.

Figure 3.8. Kernel data structures after `dup(1)`

[View full size image]



`fcntl(2)` Function

- The `fcntl` function can change the properties of a file that is already open.

```
#include <fcntl.h>

int fcntl(int filedes, int cmd, ... /* int arg */ );


    Returns: depends on cmd if OK (see following), 1 on
                                                    error
```

cmd	effect	return value
`F_DUPFD`	duplicate *filedes*	new *filedes*
`F_GETFD`	get the file descriptor flags for *filedes*	descriptor flags
`F_SETFD`	set the file descriptor flags to the value of the third argument	not 1
`F_GETFL`	get the file status flags	status flags
`F_SETFL`	set the file status flags	not 1

Figure 3.10. Print file flags for specified descriptor

Figure 3.11 in edition 3

```
#include "apue.h"
#include <fcntl.h>
int
main(int argc, char *argv[])
{

    int       val;

    if (argc != 2)
        err_quit("usage: a.out <descriptor#>");

    if ((val = fcntl(atoi(argv[1]), F_GETFL, 0)) < 0)
        err_sys("fcntl error for fd %d", atoi(argv[1]));

    switch (val & O_ACCMODE) {
    case O_RDONLY:
        printf("read only");
        break;

    case O_WRONLY:
        printf("write only");
        break;

    case O_RDWR:
        printf("read write");
        break;

    default:
        err_dump("unknown access m
    }
```

```
    if (val & O_APPEND)
        printf(", append");
    if (val & O_NONBLOCK)
        printf(", nonblocking");
#if defined(O_SYNC)
    if (val & O_SYNC)
        printf(", synchronous writes");
#endif
#if !defined(_POSIX_C_SOURCE) && defined(O_FSYNC)
    if (val & O_FSYNC)
        printf(", synchronous writes");
#endif
    putchar('\n');
    exit(0);
}
```

This macro stands for a mask that can be bitwise-ANDed with the file status flag value to produce a value representing the file access mode.#

```
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e] cp fig3.10 ./test/fig3-10.c
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e] cd test
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] make fig3-10
gcc -DLINUX -ansi -I/home/jere/SystemProgramming/apue.2e/include -Wall -D_GNU_SO
 -lapue  -o fig3-10
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig3-10 0 < /dev/tty
read only
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig3-10 1 > temp.foo
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] cat temp.foo
write only
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig3-10 2 2>> temp.foo
write only, append
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig3-10 5 5<> temp.foo
read write
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test]
```

* http://linux.vbird.org/linux_basic/0320bash.php
# http://www.gnu.org/software/libc/manual/html_node/Access-Modes.html

- When we modify either the file descriptor flags or the file status flags, we must be careful to fetch the existing flag value, modify it as desired, and then set the new flag value.
- We can't simply do an `F_SETFD` or an `F_SETFL`, as this could turn off flag bits that were previously set.
- If we change the middle statement to

```
val &= ~flags;   /* turn flags off */
```

Figure 3.11. Turn on one or more of the file status flags for a descriptor

```
#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");
}
```

`sync(2)`, `fsync(2)`, and `fdatasync(2)` Functions

- Delayed write
  - The data written to a file is normally copied by the kernel into one of its buffers and queued for writing to disk at later time.
  - The kernel writes all the delayed-write blocks to disk when it needs to reuse the buffer for other disk block.
  - `sync`, `fsync`, and `fdatasync` are used to ensure consistency of the file system on disk with the contents of the buffer cache.

sync(2), fsync(2), and fdatasync(2) Functions

```
#include <unistd.h>

int fsync(int filedes);

int fdatasync(int filedes);

                                    Returns: 0 if OK, 1 on error
void sync(void);
```

- sync
  - Queues all the modified block buffers for writing and returns.
  - It does not wait for the disk writes to take place.
  - Called periodically (30s) from a system daemon (update).
- fsync
  - Waits for the disk writes to complete before returning.
  - Data and file attributes are updated.
- fdatasync
  - Similar to fsync but affects only the data portion of a file.

/dev/fd

- /dev/fd for new UNIX systems
- /dev/fd: directory whose entries are files named 0, 1, 2 …
- Open the file /dev/fd/n is equivalent to duplicating descriptor n (n is open),
  - fd = open("/dev/fd/0", mode) is equivalent to fd=dup(0)
    - The descriptors 0 and fd share the same file table entry.
    - If descriptor 0 was opened read-only, we can only read on fd.

/dev/fd

- Some systems provide the pathnames /dev/stdin, /dev/stdout, and /dev/stderr.
  - Equivalent to /dev/fd/0, /dev/fd/1, and /dev/fd/2
- Main use of /dev/fd files is from the shell.
  - It allows programs that use pathname arguments to handle standard input and standard output in the same manner as other pathnames.
- $> head -3 txt1.txt | cat txt2.txt /dev/fd/0 txt3.txt

```
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] cd /dev
jere@VirtualBox-MBP [/dev] ls
autofs           fd       mapper              ram11   rtc0      tty0
block            full     mcelog              ram12   scd0      tty1
bsg              fuse     mem                 ram13   sda       tty10
btrfs-control    hidraw0  net                 ram14   sda1      tty11
bus              hpet     network_latency     ram15   sda2      tty12
cdrom            input    network_throughput  ram2    sda5      tty13
char             kmsg     null                ram3    sg0       tty14
console          log      oldmem              ram4    sg1       tty15
core             loop0    port                ram5    shm       tty16
cpu              loop1    ppp                 ram6    snapshot  tty17
cpu_dma_latency  loop2    psaux               ram7    snd       tty18
disk             loop3    ptmx                ram8    sr0       tty19
dri              loop4    pts                 ram9    stderr    tty2
dvd              loop5    ram0                random  stdin     tty20
ecryptfs         loop6    ram1                rfkill  stdout    tty21
fb0              loop7    ram10               rtc     tty       tty22
```

Assignment

- Reading (do it at home):
  - Manual pages for the functions covered
  - Stevens Chap. 3

Assignment

- Assignment 4
  1. Finding and thinking:
     - What does the bourne shell syntax "> &" and "<<" and "2>>"mean?
     - ➡ A brief report (upload via lms system) should be handed in.
  2. Coding:
     - `tcp(1)` (see the html file)
     - ➡ A program (submit via TA's server) should be handed in.
     - ➡ A brief report (upload via lms system) should be handed in.
       - How to compile the source file to the execution file called `tcp`?
       - To execute "tcp a.txt b.txt" in shell, where should you put your executable file?
       - Any details about your program you want to mention

Advanced Utilities for Development

A Tool: `cscope(1)`

- What is it?
  - A developer's tool for browsing source code
- How to install?
  - `sudo apt-get install cscope`
- How to use?
  - Use as a command
  - Use in vim

* http://cscope.sourceforge.net/
* http://manpages.ubuntu.com/manpages/lucid/man1/cscope-indexer.1.html
* http://stackoverflow.com/questions/2188405/how-to-let-cscope-use-absolute-path-in-cscope-out-file

A Tool: `cscope(1)`

- Use as a command
  - Type `cscope` in command line
    - Build the symbol cross-reference the first time it is used on the source files for the program being browsed.
      - `-R/-b/-k` options (see man page of `cscope`)
      - Will create a symbol cross-reference file (default: `cscope.out`)
  - Cscope's curses-based GUI
    - Use the arrow keys to move around between search types
    - Use tab to switch between the search types and your search results
    - Exit: Ctrl+d

A Tool: `cscope(1)`

- Use in vim
  - Download the `cscope_maps.vim` file
  - Put this file into ~/.vim/plugin/
  - Basic usages:
    - Put the cursor over a C symbol (keystroke maps from `cscope_maps.vim`)
      - Ctrl+\+s or Ctrl+Space+s or Ctrl+SpaceSpace+s
    - Use Vim's built-in Cscope support
      - `:cscope find symbol foo` (or, more tersely, `:cs f s foo`)

- Use in vim
  - How to use other symbol cross-reference files?
  - If we want to create a symbol cross-reference file for directory `/usr/include` and all its subdirectories,
    - `cd /usr/include`
    - `sudo cscope -R -b -f cscope1.out`
  - Edit `cscope_maps.vim` (`sudo vi cscope_maps.vim`)
    - Add "`cs add /usr/include/cscope1.out /usr/include`"
  - `cd ~/Systemprogramming/apue.2e/test/`
  - `cscope -R -b -k`
    - why should we do this again?

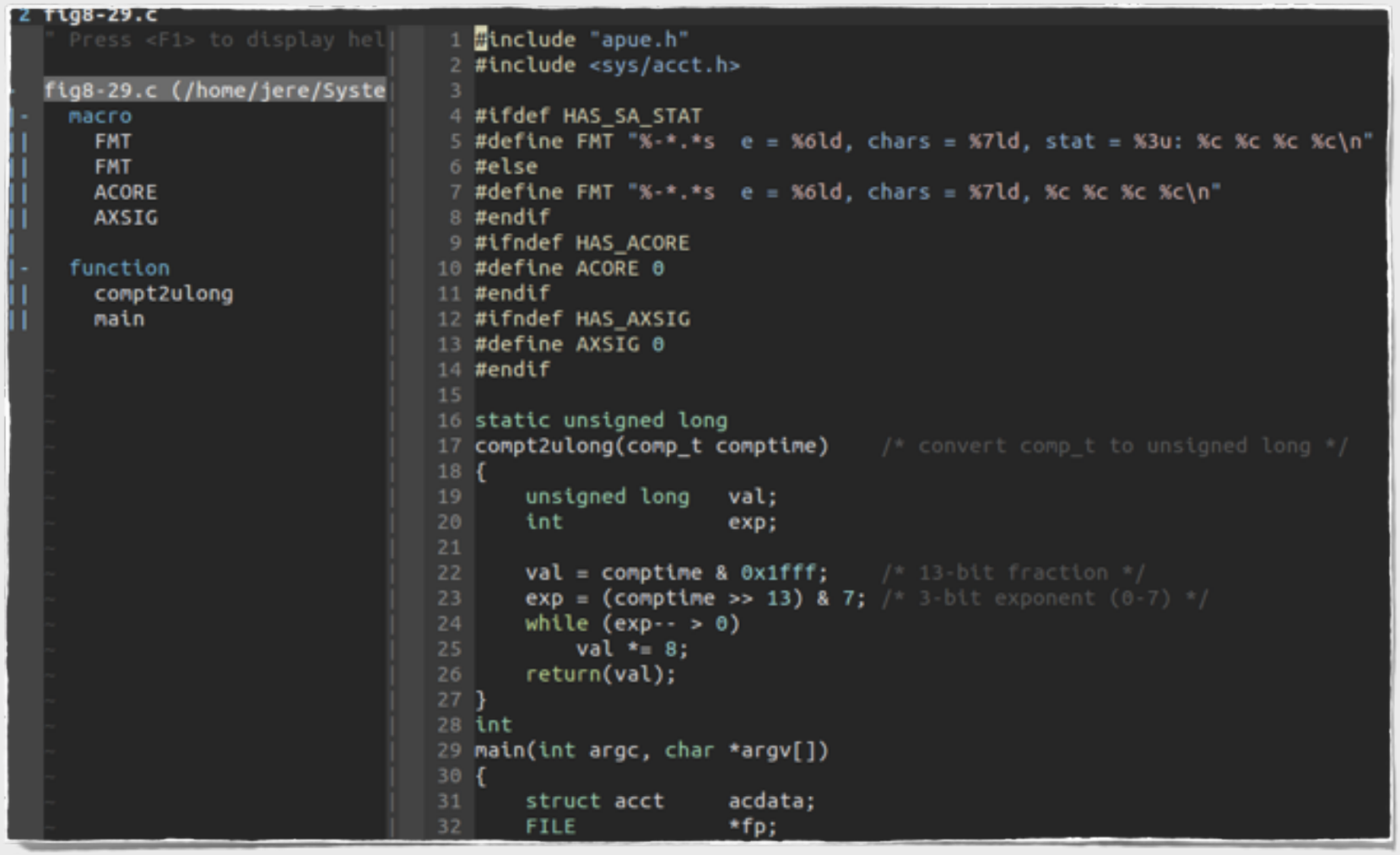
```
"add any cscope database in current directory
cs add /usr/include/cscope1.out /usr/include
if filereadable("cscope.out")
    cs add cscope.out
" else add the database pointed to by environment variable
elseif $CSCOPE_DB != ""
    cs add $CSCOPE_DB
endif
```

A Tool: `ctags(1)`

- Install exuberant-ctags
  - `sudo apt-get install exuberant-ctags`
- You may first check if you already have this package.
  - `dpkg --get-selections | grep ctags`

A Tool: `ctags(1)`

- Use in vim
  - Taglist.vim (Put this file into ~/.vim/plugin/)
  - Normal mode:
    - `:Tlist`



* http://www.vim.org/scripts/script.php?script_id=273
* http://ctags.sourceforge.net/

Tagbar

- Use in vim
  - tagbar.vim (Put this file into ~/.vim/plugin/)

Quickstart

Put something like the following into your ~/.vimrc:

```
nmap <F8> :TagbarToggle<CR>
```

If you do this the F8 key will toggle the Tagbar window. You can of course use any shortcut you want. For more flexible ways to open and close the window (and the rest of the functionality) see the documentation.

Vi Useful Command

- Normal mode
  - Save file --> :w
  - Exit --> :q
  - Copy --> y
  - Paste --> p
  - Delete a line --> dd
  - Delete a letter --> x
  - Undo --> u

Vi Useful Command

- Folding --> zf
- Unfolding --> za
- Tab for multiple lines --> Shift+>/<
    - Switch between windows --> Ctrl+ww
    - Recording macro: Powerful usage!
- Insertion mode
- Visual mode
    - Select a line --> shift + v

To save the folding
: mkview
To reload the folding
:loadview

* http://www.vtk.org/Wiki/VIM_Useful_Commands
* Recording in vim: http://vim.wikia.com/wiki/Recording_keys_for_repeated_jobs

Vi Useful Plugins

- NERD_commenter.vim
  - http://www.vim.org/scripts/script.php?script_id=1218
- comments.vim
  - http://www.vim.org/scripts/script.php?script_id=1528

Files and Directories

`stat(2)` Family of Functions

- All these functions return extended attributes about the referenced file.
  - In the case of symbolic links, `lstat(2)` returns attributes of the link, others return stats of the referenced file.

```
#include <sys/stat.h>

int stat(const char *restrict pathname, struct
➥ stat *restrict buf);

int fstat(int filedes, struct stat *buf);

int lstat(const char *restrict pathname, struct
➥ stat *restrict buf);

                                All three return: 0 if OK, 1 on error
```

* http://en.wikipedia.org/wiki/Restrict

`stat(2)` Family of Functions

```
struct stat {
  mode_t    st_mode;      /* file type & mode (permissions) */
  ino_t     st_ino;       /* i-node number (serial number) */
  dev_t     st_dev;       /* device number (file system) */
  dev_t     st_rdev;      /* device number for special files */
  nlink_t   st_nlink;     /* number of links */
  uid_t     st_uid;       /* user ID of owner */
  gid_t     st_gid;       /* group ID of owner */
  off_t     st_size;      /* size in bytes, for regular files */
  time_t    st_atime;     /* time of last access */
  time_t    st_mtime;     /* time of last modification */
  time_t    st_ctime;     /* time of last file status change */
  blksize_t st_blksize;   /* best I/O block size */
  blkcnt_t  st_blocks;    /* number of disk blocks allocated */
};
```

Figure 2.20. Some common primitive system data types

Type	Description
caddr_t	core address (Section 14.9)
clock_t	counter of clock ticks (process time) (Section 1.10)
comp_t	compressed clock ticks (Section 8.14)
dev_t	device numbers (major and minor) (Section 4.23)
fd_set	file descriptor sets (Section 14.5.1)
fpos_t	file position (Section 5.10)
gid_t	numeric group IDs
ino_t	i-node numbers (Section 4.14)
mode_t	file type, file creation mode (Section 4.5)
nlink_t	link counts for directory entries (Section 4.14)
off_t	file sizes and offsets (signed) (lseek, Section 3.6)
pid_t	process IDs and process group IDs (signed) (Sections 8.2 and 9.4)
ptrdiff_t	result of subtracting two pointers (signed)

Each member is specified by a primitive system data type (see Steven Section 2.8).

File Types

- Encoded in the `st_mode` member of the `stat` structure.
  1. Regular – most common, interpretation of data is up to application
  2. Directory – contains names of other files and pointer to information on those files. Any process can read, only kernel can write.
  3. Character special – used for certain types of devices
  4. Block special – used for disk devices (typically). All devices are either character or block special.
  5. FIFO – used for interprocess communication (sometimes called named pipe)
  6. Socket – used for network communication and non-network communication (same host).
  7. Symbolic link – Points to another file.

The file type can be determined with the macros:

Figure 4.1. File type macros in <sys/stat.h>

Macro	Type of file
S_ISREG()	regular file
S_ISDIR()	directory file
S_ISCHR()	character special file
S_ISBLK()	block special file
S_ISFIFO()	pipe or FIFO
S_ISLNK()	symbolic link
S_ISSOCK()	socket

stat.h = (/usr/include/i386-linux-gnu/sys) - VIM

stat.h

```
126
127 /* Test macros for file types.  */
128
129 #define __S_ISTYPE(mode, mask)  (((mode) & __S_IFMT) == (mask))
130
131 #define S_ISDIR(mode)    __S_ISTYPE((mode), __S_IFDIR)
132 #define S_ISCHR(mode)    __S_ISTYPE((mode), __S_IFCHR)
133 #define S_ISBLK(mode)    __S_ISTYPE((mode), __S_IFBLK)
134 #define S_ISREG(mode)    __S_ISTYPE((mode), __S_IFREG)
135 #ifdef __S_IFIFO
136 # define S_ISFIFO(mode)  __S_ISTYPE((mode), __S_IFIFO)
137 #endif
138 #ifdef __S_IFLNK
139 # define S_ISLNK(mode)   __S_ISTYPE((mode), __S_IFLNK)
140 #endif
141
142 #if defined __USE_BSD && !defined __S_IFLNK
143 # define S_ISLNK(mode)  0
144 #endif
```

Find out more in /usr/include/ i386-linux-gnu/sys/stat.h

Figure 4.3. Print type of file for each command-line argument

```
#include "apue.h"

int
main(int argc, char *argv[])
{

    int         i;
    struct stat buf;
    char        *ptr;

    for (i = 1; i < argc; i++) {
        printf("%s: ", argv[i]);
        if (lstat(argv[i], &buf) < 0) {
            err_ret("lstat error");
            continue;

        }
        if (S_ISREG(buf.st_mode))
            ptr = "regular";
        else if (S_ISDIR(buf.st_mode))
            ptr = "directory";
        else if (S_ISCHR(buf.st_mode))
            ptr = "character special";
        else if (S_ISBLK(buf.st_mode))
            ptr = "block special";
        else if (S_ISFIFO(buf.st_mode))
            ptr = "fifo";
        else if (S_ISLNK(buf.st_mode))
            ptr = "symbolic link";
        else if (S_ISSOCK(buf.st_mode))
            ```socket";
        else
            ptr = "** unknown mode **";
        printf("%s\n", ptr);
  }
   exit(0);
}
```

- Define `_GNU_SOURCE` to include the definition of the `S_ISSOCK` macro.
  - glibc* does not make the GNU extensions available automatically.
  - If a program depends on GNU extensions or some other non-standard functionality, it is necessary to
    (1) compile it with the C compiler option -D_GNU_SOURCE
    (2) put #define _GNU_SOURCE at the beginning of your source files, before any C library header files are included.

```
$ ./a.out /etc/passwd /etc /dev/initctl /dev/log /dev/tty \
> /dev/scsi/host0/bus0/target0/lun0/cd /dev/cdrom
/etc/passwd: regular
/etc: directory
/dev/initctl: fifo
/dev/log: socket
/dev/tty: character special
/dev/scsi/host0/bus0/target0/lun0/cd: block special
/dev/cdrom: symbolic link
```

* GLIBC, the GNU C Library: http://www.gnu.org/software/libc/
```

`mkdir(2)` and `rmdir(2)`

- Directories are created with the `mkdir` function and deleted with the `rmdir`.

```
#include <sys/stat.h>

int mkdir(const char *pathname, mode_t mode);

                                Returns: 0 if OK, 1 on error
```

- Creates a new, empty (except for . and .. entries) directory.
- Access permissions specified by *mode*.

```
#include <unistd.h>

int rmdir(const char *pathname);

                                Returns: 0 if OK, 1 on error
```

- If the link count is 0 (after this call), and no other process has the directory open, directory is removed.
- Directory must be empty (only . and .. remaining)

Reading Directories

```
#include <dirent.h>

DIR *opendir(const char *pathname);

        Returns: pointer if OK, NULL on error
struct dirent *readdir(DIR *dp);

        Returns: pointer if OK, NULL at end of
                                directory or error
void rewinddir(DIR *dp);

int closedir(DIR *dp);

                Returns: 0 if OK, 1 on error
```

- `rewinddir` resets an open directory to the beginning so `readdir` will again return the first entry.

```
struct dirent {
  ino_t d_ino;                  /* i-node number */
  char  d_name[NAME_MAX + 1];   /* null-terminated filename */
}
```

Moving Around Directories

- Get the kernel's idea of our process's current working directory.

```
#include <unistd.h>

char *getcwd(char *buf, size_t size);


                Returns: buf if OK, NULL on error
```

Moving Around Directories

- Allows a process to change its current working directory.
- Note that `chdir` and `fchdir` affect only the current process.

```
#include <unistd.h>

int chdir(const char *pathname);

int fchdir(int filedes);
```

Both return: 0 if OK, 1 on error

Figure 4.23. Example of `chdir` function

```
#include "apue.h"

int
main(void)
{

      if (chdir("/tmp") < 0)
          err_sys("chdir failed");
      printf("chdir to /tmp succeeded\n");
      exit(0);
}
```

```
jere@VB(MBA) [~/SystemProgramming/apue.2e/test] make fig4-23
gcc -DLINUX -ansi -I~/SystemProgramming/apue.2e/include -Wall -D_GNU_SOURCE    -L
jere@VB(MBA) [~/SystemProgramming/apue.2e/test] pwd
/home/jere/SystemProgramming/apue.2e/test
jere@VB(MBA) [~/SystemProgramming/apue.2e/test] ./fig4-23
chdir to /tmp succeeded
jere@VB(MBA) [~/SystemProgramming/apue.2e/test] pwd
/home/jere/SystemProgramming/apue.2e/test
```

Figure 4.24. Example of `getcwd` function

```
#include "apue.h"

int
main(void)
{

    char    *ptr;
    int     size;

    if (chdir("/usr/spool/uucppublic") < 0)
        err_sys("chdir failed");

    ptr = path_alloc(&size); /* our own function */
    if (getcwd(ptr, size) == NULL)
        err_sys("getcwd failed");

    printf("cwd = %s\n", ptr);
    exit(0);
}
```

```
$ ./a.out
cwd = /var/spool/uucppublic
$ ls -l /usr/spool
lrwxrwxrwx 1 root 12 Jan 31 07:57 /usr/spool -> ../var/spool
```