CSE 344 System Programming

Week 3

Files and low level I/O in the UNIX world

- *open/close/read/write/lseek/fcnt/ioctl*
- *Redirection*
- *File implementation in UNIX*
- *File stats*
- *Temporary files*
- *Relationship between file descriptors and open files*
- *Buffering and stdio*

Files

"On a UNIX system, everything is a file; if something is not a file, it is a process."*

Directories: **files** containing the names of files in them

Programs, texts, images, videos: **files;** either binary or ASCII

Devices, e.g. monitor, cpu, gpu, printer: all represented as **files**

Consequently, it is highly important to know how to handle them!

* minor exceptions apply

File types

Regular

```
$ ls -l *
 -rw-r--r-- 1 greys greys        1024 Mar 29 06:31 text
```

Directory

```
$ ls -ld *
 -rw-r--r-- 1 greys greys  1024 Mar 29 06:31 text
 drwxr-xr-x 2 greys greys  4096 Aug 21 11:00 mydir
```

Device file

```
$ ls -al /dev/loop0 /dev/ttys0
brw-rw---- 1 root disk 7,   0 Sep  7 05:03 /dev/loop0
crw-rw-rw- 1 root tty  3, 48 Sep  7 05:04 /dev/ttys0
```

File types

Named Pipe (IPC)

```
$ ls -al /dev/xconsole
prw-r----- 1 root adm 0 Sep 25 08:58 /dev/xconsole
```

Symbolic link

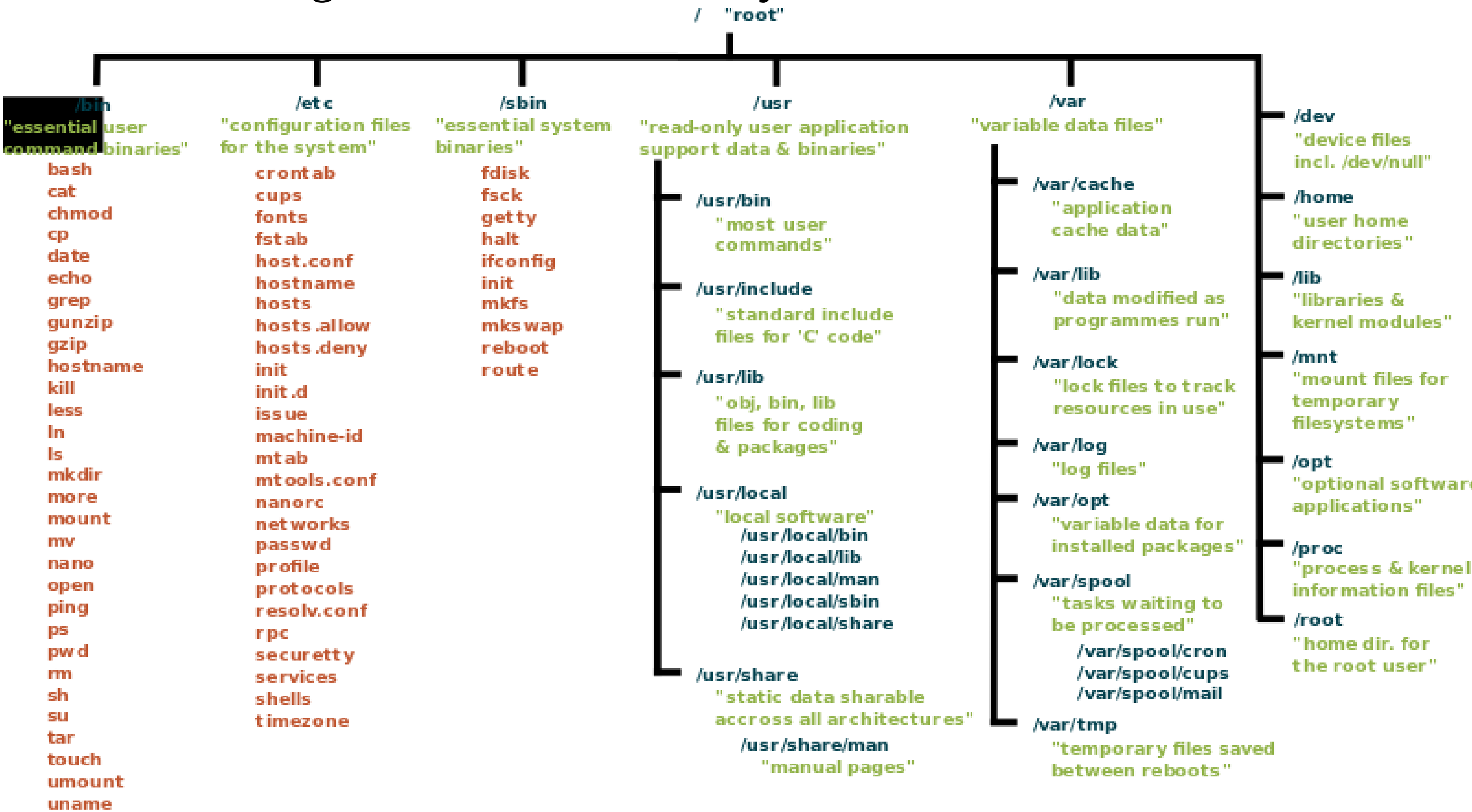
```
$ ls -al hosts
lrwxrwxrwx 1 greys www-data 10 Sep 25 09:06 hosts -> /etc/host
```

Socket (IPC)

```
$ ls -al /dev/log
srw-rw-rw- 1 root root 0 Sep  7 05:04 /dev/log0
```

Files

All files are organized within a **file system**; a rooted tree of directories.



Files

Unlike windows where each drive has a letter that's the root of its own FS, in UNIX, drives, partitions, removable media and even network shares can be **mounted**, and thus the entire volume's FS appears as a directory. The root is always denoted by **/** ("slash")

/bin: binaries, ls, cp, etc.

/home : user directories

/boot: files needed for booting the system

/etc: system-wide configuration files

/dev: file representations of devices

/lib: libraries

/proc: files representing runtime system information

/usr: non-system critical binaries, libraries, resources

/var: logs, temporary files, mail, print jobs, etc.

Files

An arbitrary location or address within this tree structure is known as a **path**, e.g.: /home/ezerger/cse344/week3.pdf

Every directory contains the files: “.” and “..” that represent respectively the current and parent directories.

If a path starts with / then it’s an **absolute or fully qualified path,** otherwise, the program pretends the absolute path of the current working directory;

e.g, you are located at: /home/ezerger/cse344

../cse685/midterm.pdf -> /home/ezerger/cse685/midterm.pdf

Files

All system calls that deal with files (of any type) refer to them through **file descriptors**; i.e. a small non-negative integer.

All programs start with 3 open files that are opened on their behalf by the shell:

File descriptor	Purpose	POSIX name	*stdio* stream
0	standard input	STDIN_FILENO	*stdin*
1	standard output	STDOUT_FILENO	*stdout*
2	standard error	STDERR_FILENO	*stderr*

Files

There are four key system calls upon which programming libraries (fopen, fclose, fwrite, etc) rely for file I/O:

- `fd = open(pathname, flags, mode)`

Opens the file *pathname* and returns its file descriptor

- `numread= read(fd, buffer, count)`

Read at most count bytes from the open file fd and stores in buffer

- `numwritten=write(fd, buffer, count)`

Writes up to count bytes from buffer into the open file fd

- `status=close(fd)`

Is called after all I/O operations are completed, and releases resources

open

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
    int open(const char *pathname, int flags);
    int open(const char *pathname, int flags, mode_t mode);
```

Flag examples:

O_RDONLY: read only; O_WRONLY: write only

O_RDWR: read and write

O_CREAT: create a new file

O_APPEND: any data written to the file will be appended to its end

O_TRUNC:discard previous content

O_EXCL: used together with O_CREAT, returns error if the file already exists

O_NONBLOCK: if the file cannot be opened, instead of blocking it returns an error

...and more..

Returns the fd or -1 in case of error.

open

mode: is an octal number specifying the permissions of the newly created file (in conjunction with umask and the access permissions of the parent directory).

POSIX defines symbolic names for the permission masks so that you can specify them independently of the underlying implementation (defined in `sys/stat.h`)

S_I(R|W|X)(USR|GRP|OTH)

e.g.

S_IRUSR: read access for user

S_IWGRP: write access for group

S_IXOTH: execution permission for others

open

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
  /* The path at which to create the new file.  */
  char* path = argv[1];
  /* The permissions for the new file.  */
  mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH;

  /* Create the file.  */
  int fd = open (path, O_WRONLY | O_EXCL | O_CREAT, mode);
  if (fd == -1) {
    /* An error occurred.  Print an error message and bail.  */
    perror ("open");
    return 1;
  }

  return 0;
}
```

close

Even though when a process terminates the OS closes all open fd's associated with that process, it is good practice to `close` a file once you are done with it.

```
#include <unistd.h>
   int close(int fd);
```

Returns zero on success and -1 on error.

Open file descriptors use kernel resources (every process needs a table to keep track of its open files). The typical limit is (used to be) 1024 file descriptors per process. You can adjust this limit through the `getrlimit` and `setrlimit` system calls.

write

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

It writes up to count bytes from the buffer pointed by `buf` **to the current offset** of the file referred to by the file descriptor fd. It might write less than count due to a signal interruption, etc. The data to write need not be a character string; it works with arbitrary bytes.

Returns the number of bytes written or -1 on error.

Error examples:

EBADF: fd is not a valid file descriptor or is not open for writing.

ENOSPC: the device containing the file referred to by fd has no room for the data.

write

example on how to append a timestamp to a file

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>

/* Return a character string representing the current date and time.  */

char* get_timestamp ()
{
  time_t now = time (NULL);
  return asctime (localtime (&now));
}

int main (int argc, char* argv[])
{
  /* The file to which to append the timestamp.  */
  char* filename = argv[1];
  /* Get the current timestamp.  */
  char* timestamp = get_timestamp ();
  /* Open the file for writing.  If it exists, append to it;
     otherwise, create a new file.  */
  int fd = open (filename, O_WRONLY | O_CREAT | O_APPEND, 0666);
  /* Compute the length of the timestamp string.  */
  size_t length = strlen (timestamp);
  /* Write the timestamp to the file.  */
  write (fd, timestamp, length);
  /* All done.  */
  close (fd);
  return 0;
}
```

```
% ./timestamp tsfile
% cat tsfile
Thu Feb  1 23:25:20 2001
% ./timestamp tsfile
% cat tsfile
Thu Feb  1 23:25:20 2001
Thu Feb  1 23:25:47 2001
```

read

```
#include <unistd.h>
    ssize_t read(int fd, void *buf, size_t count)
```

Similar to write in principle. Returns the number of bytes read, 0 on EOF, -1 on error.

Warning: in the UNIX world file lines are separated by the newline character ‘\n’ (ASCII 10). In the windows world, lines are separated by two characters: a carriage return ‘\r’ (ASCII 13) **and** a newline character.

So if your file was saved in a windows environment, and you read it in a UNIX environment, do not be alarmed when you see the **^M** expression (corresponding to the carriage return character) at the end of every line.

read

print the hexadecimal dump of a file

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
  unsigned char buffer[16];
  size_t offset = 0;
  size_t bytes_read;
  int i;

  /* Open the file for reading.  */
  int fd = open (argv[1], O_RDONLY);

  /* Read from the file, one chunk at a time.  Continue until read
     "comes up short", that is, reads less than we asked for.
     This indicates that we've hit the end of the file.  */
  do {
    /* Read the next line's worth of bytes.  */
    bytes_read = read (fd, buffer, sizeof (buffer));
    /* Print the offset in the file, followed by the bytes themselves.  */
    printf ("0x%06x : ", offset);
    for (i = 0; i < bytes_read; ++i)
      printf ("%02x ", buffer[i]);
    printf ("\n");
    /* Keep count of our position in the file.  */
    offset += bytes_read;
  }
  while (bytes_read == sizeof (buffer));

  /* All done.  */
  close (fd);
  return 0;
}
```

```
% ./hexdump hexdump
0x000000 : 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
0x000010 : 02 00 03 00 01 00 00 00 c0 83 04 08 34 00 00 00
0x000020 : e8 23 00 00 00 00 00 00 34 00 20 00 06 00 28 00
0x000030 : 1d 00 1a 00 06 00 00 00 34 00 00 00 34 80 04 08
...
```

example

example: copy a file

```
#include <errno.h>
#include <unistd.h>
#define BLKSIZE 1024

int copyfile(int fromfd, int tofd) {
   char *bp;
   char buf[BLKSIZE];
   int bytesread;
   int byteswritten = 0;
   int totalbytes = 0;

   for (  ;  ;  ) {
      while (((bytesread = read(fromfd, buf, BLKSIZE)) == -1) &&
             (errno == EINTR)) ;         /* handle interruption by signal */
      if (bytesread <= 0)          /* real error or end-of-file on fromfd */
         break;
      bp = buf;
      while (bytesread > 0) {
         while(((byteswritten = write(tofd, bp, bytesread)) == -1 ) &&
              (errno == EINTR)) ;        /* handle interruption by signal */
         if (byteswritten < 0)                         /* real error on tofd */
            break;
         totalbytes += byteswritten;
         bytesread -= byteswritten;
         bp += byteswritten;
      }
      if (byteswritten == -1)                          /* real error on tofd */
          break;
   }
   return totalbytes;
}
```

example continued

example: copy a file

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>

#define READ_FLAGS O_RDONLY
#define WRITE_FLAGS (O_WRONLY | O_CREAT | O_EXCL)
#define WRITE_PERMS (S_IRUSR | S_IWUSR)

/* function definitions */
int copyfile(int fromfd, int tofd);

int main(int argc, char *argv[]) {
   int bytes;
   int fromfd, tofd;

   if (argc != 3) {
      fprintf(stderr, "Usage: %s from_file to_file\n", argv[0]);
      return 1;
   }

   if ((fromfd = open(argv[1], READ_FLAGS)) == -1) {
      perror("Failed to open input file");
      return 1;
   }

   if ((tofd = open(argv[2], WRITE_FLAGS, WRITE_PERMS)) == -1) {
      perror("Failed to create output file");
      return 1;
   }

   bytes = copyfile(fromfd, tofd);
   printf("%d bytes copied from %s to %s\n", bytes, argv[1], argv[2]);
   return 0;                                  /* the return closes the files */
}
```

lseek

A file descriptor remembers its position in a file. As you read or write the position advances depending on the number of bytes read or written. If you want to move arbitrarily within a file then:

```
#include <sys/types.h>
#include <unistd.h>
    off_t lseek(int fd, off_t offset, int whence);
```

offset: new position. The third argument determines how to interpret the second arg.

SEEK_SET: number of bytes from the start of the file (only positive)

SEEK_CUR:number of bytes from the current position (positive or negative)

SEEK_END: number of bytes from the end of the file (positive or negative)

Returns the new position from the beginning. Cannot be used with sockets.

lseek

```
lseek(fd, 0, SEEK_SET);         /* Start of file */
lseek(fd, 0, SEEK_END);         /* Next byte after the end of the file */
lseek(fd, -1, SEEK_END);        /* Last byte of file */
lseek(fd, -10, SEEK_CUR);       /* Ten bytes prior to current location */
lseek(fd, 10000, SEEK_END);     /* 10001 bytes past last byte of file */
```

Listing B.5 (*lseek-huge.c*) **Create Large Files with *lseek***

```
#include <fcntl.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
  int zero = 0;
  const int megabyte = 1024 * 1024;

  char* filename = argv[1];
  size_t length = (size_t) atoi (argv[2]) * megabyte;

  /* Open a new file.  */
  int fd = open (filename, O_WRONLY | O_CREAT | O_EXCL, 0666);
  /* Jump to 1 byte short of where we want the file to end.  */
  lseek (fd, length - 1, SEEK_SET);
  /* Write a single 0 byte.  */
  write (fd, &zero, 1);
  /* All done.  */
  close (fd);

  return 0;
}
```

lseek

Using `lseek-huge`, we'll make a 1GB (1024MB) file. Note the free space on the drive before and after the operation.

```
% df -h .
Filesystem            Size  Used Avail Use% Mounted on
/dev/hda5             2.9G  2.1G  655M  76% /
% ./lseek-huge bigfile 1024
% ls -l bigfile
-rw-r-----    1 samuel   samuel   1073741824 Feb  5 16:29 bigfile
% df -h .
Filesystem            Size  Used Avail Use% Mounted on
/dev/hda5             2.9G  2.1G  655M  76% /
```

No appreciable disk space is consumed, despite the enormous size of `bigfile`. Still, if we open `bigfile` and read from it, it appears to be filled with 1GB worth of 0s. For instance, we can examine its contents with the `hexdump` program of Listing B.4.

```
% ./hexdump bigfile | head -10
0x000000 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000010 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000020 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000030 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000040 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000050 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
...
```

These "magic" file holes are a nice property of UNIX file systems. In windows environments this would lead to an actual 1GB file.

ioctl

Non standard I/O operations: `ioclt`

```
#include <sys/ioctl.h>
    int ioctl(int d, int request, ...);
```

It manipulates the underlying device parameters of special files.
It requires detailed understanding of the device represented by the fd.
It's beyond our scope.

```
#include <fcntl.h>
#include <linux/cdrom.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
  /* Open a file descriptor to the device specified on the command line.  */
  int fd = open (argv[1], O_RDONLY);
  /* Eject the CD-ROM.  */
  ioctl (fd, CDROMEJECT);
  /* Close the file descriptor.  */
  close (fd);

  return 0;
}
```

Example: ejecting a cdrom.

fcntl

Advanced file operations: `fcnlt`

```
#include <unistd.h>
#include <fcntl.h>
    int fcntl(int fd, int cmd, ... /* arg */ );
```

- It can manipulate the flags associated with a fd (same ones used during opening).
- It can duplicate file descriptors
- It can lock/unlock files - very useful for inter process communication...

To place a lock on a file, first create and zero out a struct flock variable. Set the `l_type` field of the structure to `F_RDLCK` for a read lock or `F_WRLCK` for a write lock.

Then call `fcntl`, passing a file descriptor to the file, the `F_SETLCKW` operation code, and a pointer to the `struct flock` variable. If another process holds a lock that prevents a new lock from being acquired, `fcntl` blocks until that lock is released.

fcntl

Listing 8.2 **(*lock-file.c*) Create a Write Lock with *fcntl***

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
  char* file = argv[1];
  int fd;
  struct flock lock;

  printf ("opening %s\n", file);
  /* Open a file descriptor to the file.  */
  fd = open (file, O_WRONLY);
  printf ("locking\n");
  /* Initialize the flock structure.  */
  memset (&lock, 0, sizeof(lock));
  lock.l_type = F_WRLCK;
  /* Place a write lock on the file.  */
  fcntl (fd, F_SETLKW, &lock);

  printf ("locked; hit Enter to unlock... ");
  /* Wait for the user to hit Enter.  */
  getchar ();

  printf ("unlocking\n");
  /* Release the lock.  */
  lock.l_type = F_UNLCK;
  fcntl (fd, F_SETLKW, &lock);

  close (fd);
  return 0;
}
```

Only one process can hold a write-lock for a given fd.

Many can hold a read-lock.

In case of an already acquired lock,

the call to fcntl will block the process.

Redirection

Under normal circumstances a process reads from standard input, outputs its result to standard output and in case of error, it is sent to standard error.

We can however redirect them!

In Bourne-style shells: `$ ./myscript > results.log 2>&1`

i.e. redirect stdout to results.log and redirect `stderr` to wherever stdout points to; so both `stdout` and `stderr` end at results.log

```
$ ./myscript 2>&1 | less
```

Both stderr and stdout of “myscript” become stdin of “less”

Redirection

How does redirection work behind the scenes?

```
#include <unistd.h>
    int dup2(int oldfd , int newfd );
```

It makes a duplicate of the file descriptor given in `oldfd` using the descriptor number supplied in `newfd`. If the file descriptor specified in `newfd` is already open, it closes it first; e.g.

```
dup2(1,2)
```

first closes `stderr`, then replaces it with a copy of `stdout`.

Or you can use `fcntl` instead of `dup2`:

```
newfd = fcntl(oldfd, F_DUPFD, startfd);
```

Uses as `newfd` the lowest unused file descriptor greater than or equal to `startfd`

File implementation

The designers of UNIX have separated file data and meta-data.

All the meta-data concerning a given file reside in a fixed-length structure called **inode** (short for index node).

The inode contains information about the file size, the file location, the owner of the file, the time of creation, time of last access, time of last modification, permissions and so on.

In addition to descriptive information about the file, the inode contains pointers to the first few data blocks of the file. If the file is large, the indirect pointer is a pointer to a block of pointers that point to additional data blocks. If the file is still larger, the double indirect pointer is a pointer to a block of indirect pointers. If the file is really huge, the triple indirect pointer contains a pointer to a block of double indirect pointers.

File implementation

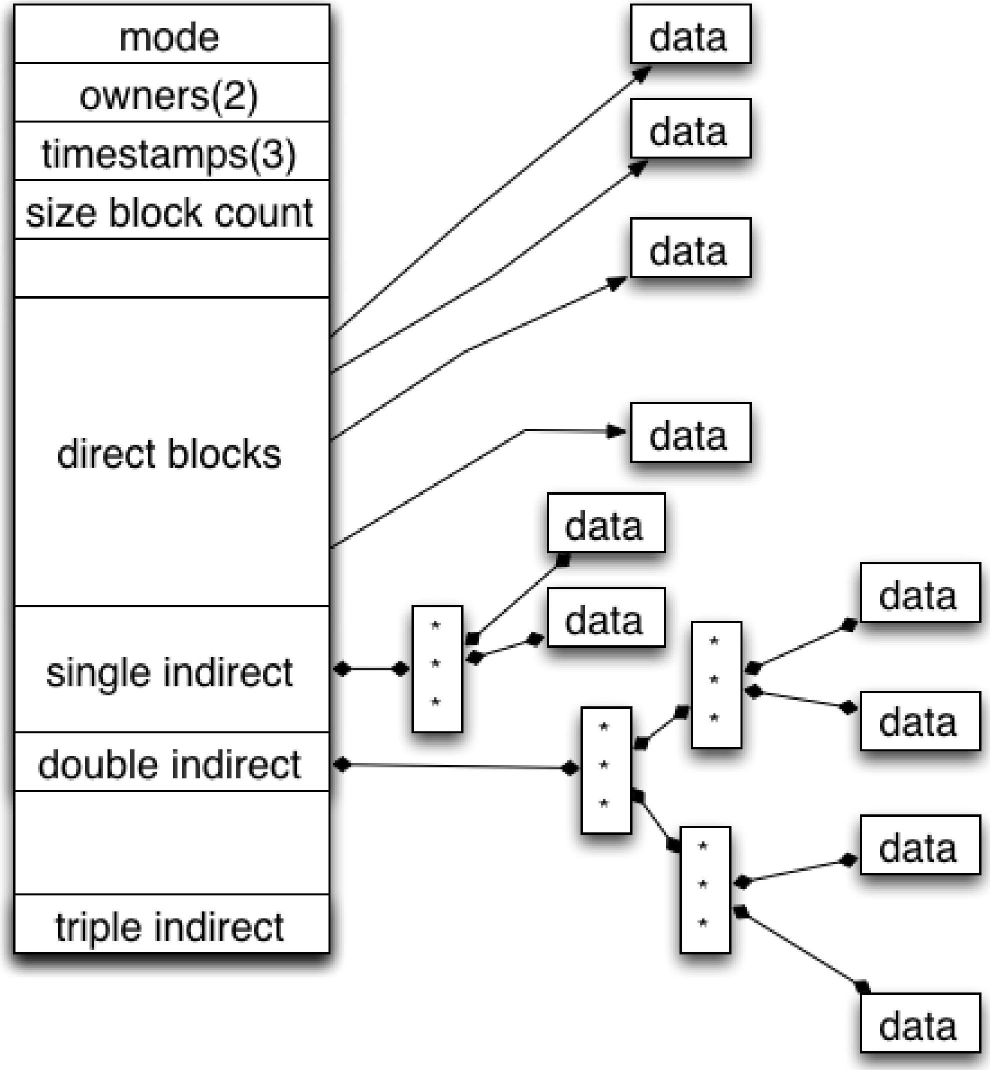


Figure 12.9 UNIX inode

Directory implementation

Directories in UNIX are basically associate arrays of filenames and inode numbers; e.g.

```
1167010 .
1158721 ..
1167626 subdir
132651 barfile
132650 bazfile
```

The inode itself does not contain the filename. When a program references a file by pathname, the operating system traverses the file system tree to find the filename and inode number in the appropriate directory.

Once it has the inode number, the operating system can determine other information about the file by accessing the inode.

Directory implementation

A directory implementation that contains only names and inode numbers has the following advantages.

1. Changing the filename requires changing only the directory entry. A file can be moved from one directory to another just by moving the directory entry, as long as the move keeps the file on the same partition.

2. Only one physical copy of the file needs to exist on disk, but the file may have several names or the same name in different directories. Again, all of these references must be on the same physical partition.

3. Directory entries are of variable length because the filename is of variable length. Directory entries are small, since most of the information about each file is kept in its inode. Manipulating small variable-length structures can be done efficiently. The larger inode structures are of fixed length.

Links

UNIX directories have two types of links—links and symbolic links

– A link is an association between a filename and an inode, sometimes called a hard link,

– A symbolic link, sometimes called a soft link, is a file that stores a string used to modify the pathname when it is encountered during pathname resolution

• Each inode contains a count of the number of hard links to the inode.

• When a file is created, a new directory entry is created an a new inode is assigned.

• Additional hard links can be created with

```
ln newname oldname
```

or with

```
#include <unistd.h>
   int link(const char *oldpath, const char *newpath);
```

Links

A new hard link to an existing file creates a new directory entry but assigns no other additional disk space.

- A new hard link increments the link count in the inode.
- A hard link can be removed with the rm command or the unlink system call:

```
#include <unistd.h>
    int unlink(const char *pathname);
```

- These decrement the link count.
- The inode and associated disk space are freed when the count is decremented to 0.

Links

A symbolic link is a special type of file that contains the name of another file.

- A reference to the name of a symbolic link causes the operating system to use the name stored in the file, rather than the name itself.
- Symbolic links are created with the command:

`ln -s newname oldname` or

```
#include <unistd.h>
   int symlink(const char *target, const char *linkpath);
```

- Symbolic links do not affect the link count in the inode.
- Unlink hard links, symbolic links can span filesystems.

Temporary files

Some programs need to create temporary files that are used only while the program is running, and these files should be removed when the program terminates.

The `mkstemp` call generates a unique filename based on a template supplied by the caller and opens the file, returning a file descriptor that can be used with I/O system calls.

```
#include <stdlib.h>
    int mkstemp(char * template);
```

Returns the file descriptor on success, or -1 on error

Typically, a temporary file is unlinked (deleted) soon after it is opened, using the `unlink` system call

Temporary files

```
int fd;
char template[] = "/tmp/somestringXXXXXX";

fd = mkstemp(template);
if (fd == -1)
    errExit("mkstemp");
printf("Generated filename was: %s\n", template);
unlink(template);     /* Name disappears immediately, but the file
                         is removed only after close() */

/* Use file I/O system calls - read(), write(), and so on */

if (close(fd) == -1)
    errExit("close");
```

File stats

You can access the inode information through the `stat` calls

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

int stat(const char *pathname, struct stat *statbuf);
int lstat(const char *pathname, struct stat *statbuf);
int fstat(int fd, struct stat *statbuf);
```

All return 0 on success, or −1 on error

`stat` works with filenames

`fstat` works with file descriptors

`lstat` is similar to `stat`, except that if the named file is a symbolic link, information about the link itself is returned, rather than the file to which the link points

File stats

```
struct stat {
    dev_t     st_dev;       /* IDs of device on which file resides */
    ino_t     st_ino;       /* I-node number of file */
    mode_t    st_mode;      /* File type and permissions */
    nlink_t   st_nlink;     /* Number of (hard) links to file */
    uid_t     st_uid;       /* User ID of file owner */
    gid_t     st_gid;       /* Group ID of file owner */
    dev_t     st_rdev;      /* IDs for device special files */
    off_t     st_size;      /* Total file size (bytes) */
    blksize_t st_blksize;   /* Optimal block size for I/O (bytes) */
    blkcnt_t  st_blocks;    /* Number of (512B) blocks allocated */
    time_t    st_atime;     /* Time of last file access */
    time_t    st_mtime;     /* Time of last file modification */
    time_t    st_ctime;     /* Time of last status change */
};
```

File stats

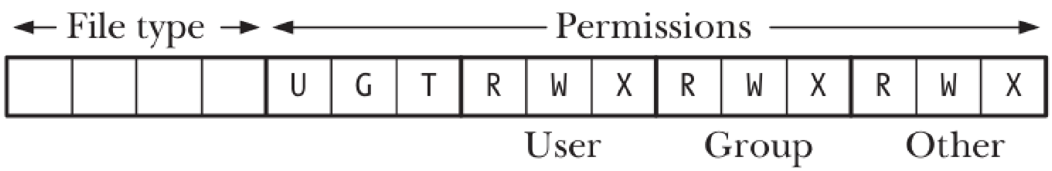


Figure 15-1: Layout of *st_mode* bit mask

The file type can be extracted by AND'ing (&) with the constant S_IFMT.

```
if ((statbuf.st_mode & S_IFMT) == S_IFREG)
printf("regular file\n")
```

But since this is a common operation there are macros for it.

File stats

Table 15-1: Macros for checking file types in the *st_mode* field of the *stat* structure

Constant	Test macro	File type
S_IFREG	S_ISREG()	Regular file
S_IFDIR	S_ISDIR()	Directory
S_IFCHR	S_ISCHR()	Character device
S_IFBLK	S_ISBLK()	Block device
S_IFIFO	S_ISFIFO()	FIFO or pipe
S_IFSOCK	S_ISSOCK()	Socket
S_IFLNK	S_ISLNK()	Symbolic link

```
if (S_ISREG(statbuf.st_mode))
   printf("regular file\n");
```

File stats

```
[root@desktop /root] # ls – l
total 558414
d rwxr-xr-x  5 root  root      1024 Dec 23 13:48 GNUstep
- rw-r--r--  1 root  root       331 Feb 11 10:19 Xrootenv.0
- rw-rw-r--  1 root  root       490 Jan  6 15:07 audio.cddb
- rw-r--r--  1 root  root  45254876 Jan  6 15:08 audio.wav
d rwxr-xr-x  2 root  root      1024 Feb 20 16:41 axhome
- rw-r--r--  1 root  root       900 Jan 18 20:15 conf
d rwxr-xr-x  2 root  root      1024 Dec 25 10:03 corel
- rw-r--r--  1 root  root       915 Jan 18 20:57 firewall
d rwxrwxr-x  2 root  root      1024 Jan  6 15:42 linux
d rwx------  2 root  root      1024 Jan  4 02:19 mail
d rwxr-xr-x  3 root  root      1024 Jan  4 01:49 mirror
- rwxr--r--  1 root  root        29 Dec 27 15:07 openn
d rwxr-xr-x  3 root  root      1024 Dec 26 13:24 scan
d rwxrwxr-x  3 root  root      1024 Jan  4 02:34 sniff
```

type | access modes | # of links | owner | group | size (bytes) | modification date and time | name

Relationship between open files and processes

There is no one-to-one correspondence between file descriptors and open files. It is possible and useful to have multiple descriptors referring to the same open file. These file descriptors may be open in the same process or in different processes.

The kernel maintains 3 data structures

- the per-process file descriptor table (with fd flags);
- the system-wide table of open file descriptions (offset, inode, signal settings, etc)
- the file system i-node table (file type, owner, location, etc)

Relationship between open files and processes

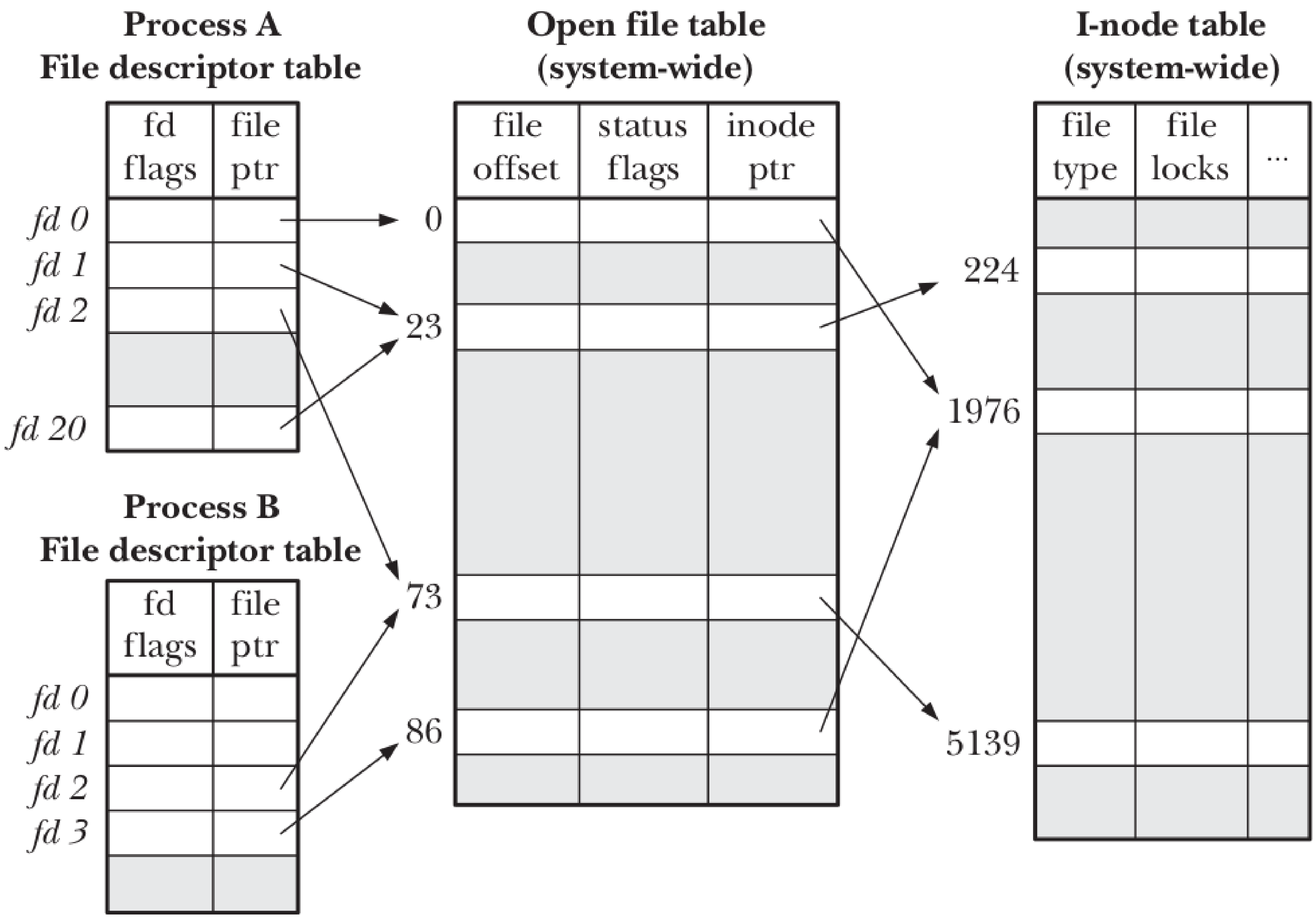


Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

Relationship between open files and processes

In process A, descriptors 1 and 20 both refer to the same open file description (labeled 23). This situation may arise as a result of a call to `dup2`, or `fcntl`

Descriptor 2 of process A and descriptor 2 of process B refer to a single open file description (73). This scenario could occur e.g. if process A is the parent of process B, or vice versa), or if one process passed an open descriptor.

Finally, we see that descriptor 0 of process A and descriptor 3 of process B refer to different open file descriptions, but that these descriptions refer to the same i-node table entry (1976)-i.e. to the same file. This occurs because each process independently called open() for the same file. A similar situation could occur if a single process opened the same file twice.

- Two distinct fd's of the same file share the same offset
- File descriptor flags (i.e., the close-on-exec flag) are private to the process and file descriptor

Buffering

When working with disk files, the `read()` and `write()` system calls don't directly initiate disk access. Instead, they simply copy data between a user-space buffer and a buffer in the kernel buffer cache. For example, the following call transfers 3 bytes of data from a buffer in user-space memory to a buffer in kernel space:

```
write(fd, "abc", 3);
```

At this point, `write()` returns. At some later point, the kernel writes (flushes) its buffer to the disk. (Hence, we say that the system call is not synchronized with the disk operation.) If, in the interim, another process attempts to read these bytes of the file, then the kernel automatically supplies the data from the buffer cache, rather than from (the outdated contents of) the file (a similar scenario is valid for `read`).

Buffering

The kernel performs the same number of disk accesses, regardless of whether we perform 1000 writes of a single byte or a single write of a 1000 bytes. However, the latter is preferable, since it requires a single system call, while the former requires 1000. Although much faster than disk operations, system calls nevertheless take an appreciable amount of time, since the kernel must trap the call, check the validity of the system call arguments, and transfer data between user space and kernel space.

Let’s look at the effect of the buffer size on duplicating a large file.

Buffering

Table 13-1: Time required to duplicate a file of 100 million bytes

BUF_SIZE	Time (seconds)			
	Elapsed	Total CPU	User CPU	System CPU
1	107.43	107.32	8.20	99.12
2	54.16	53.89	4.13	49.76
4	31.72	30.96	2.30	28.66
8	15.59	14.34	1.08	13.26
16	7.50	7.14	0.51	6.63
32	3.76	3.68	0.26	3.41
64	2.19	2.04	0.13	1.91
128	2.16	1.59	0.11	1.48
256	2.06	1.75	0.10	1.65
512	2.06	1.03	0.05	0.98
1024	2.05	0.65	0.02	0.63
4096	2.05	0.38	0.01	0.38
16384	2.05	0.34	0.00	0.33
65536	2.06	0.32	0.00	0.32

Buffering and File pointers

Buffering of data into large blocks to reduce system calls is exactly what is done by the C library I/O functions (e.g., `fprintf()`, `fscanf()`, `fgets()`, `fputs()`, `fputc()`, `fgetc()`) when operating on disk files. Thus, using the stdio library relieves us of the task of buffering data for output with `write()` or input via `read()`.

Be careful as these higher level functions handle files in terms of pointers to FILE structures.

the FILE structure

```
typedef struct {
    char *fpos; // Current position of file pointer (absolute address)
    void *base; /* Pointer to the base of the file */
    unsigned short handle; /* File handle */
    short flags; /* Flags (see FileFlags) */
    short unget; /* 1-byte buffer for ungetc (b15=1 if non-empty) */
    unsigned long alloc; // # of currently allocated bytes for the file
    unsigned short buffincrement; /* # of bytes allocated at once */
} FILE;
```

Buffering

The `setvbuf()` function controls the form of buffering employed by the stdio library.

```
#include <stdio.h>

int setvbuf(FILE * stream , char * buf , int mode , size_t size
);
```

Returns 0 on success, or nonzero on error. Valid for all subsequent stdio operations.

`stream` : the stream upon which the buffering will be applied

`buf`: the buffer

`size`: size of the buffer in bytes

Buffering

`mode` can be one of the following:

_IONBF: no buffering, immediate reads/writes, default of `stderr`

_IOLBF: Employ line-buffered I/O. This flag is the default for streams referring to terminal devices. For output streams, data is buffered until a newline character is output (unless the buffer fills first). For input streams, data is read a line at a time.

_IOFBF: Employ fully buffered I/O. Data is read or written (via calls to `read()` or `write()`) in units equal to the size of the buffer. This mode is the default for streams referring to disk files.

Buffering

```
#define BUF_SIZE 1024
static char buf[BUF_SIZE];

if (setvbuf(stdout, buf, _IOFBF, BUF_SIZE) != 0)
   errExit("setvbuf");
```

Buffering

Regardless of the current buffering mode, at any time, we can force the data in a stdio output stream to be written (i.e., flushed to a kernel buffer via `write()`) using the `fflush()` library function. This function flushes the output buffer for the specified stream.

```
#include <stdio.h>
   int fflush(FILE * stream );
```

Returns 0 on success, EOF on error

Buffering

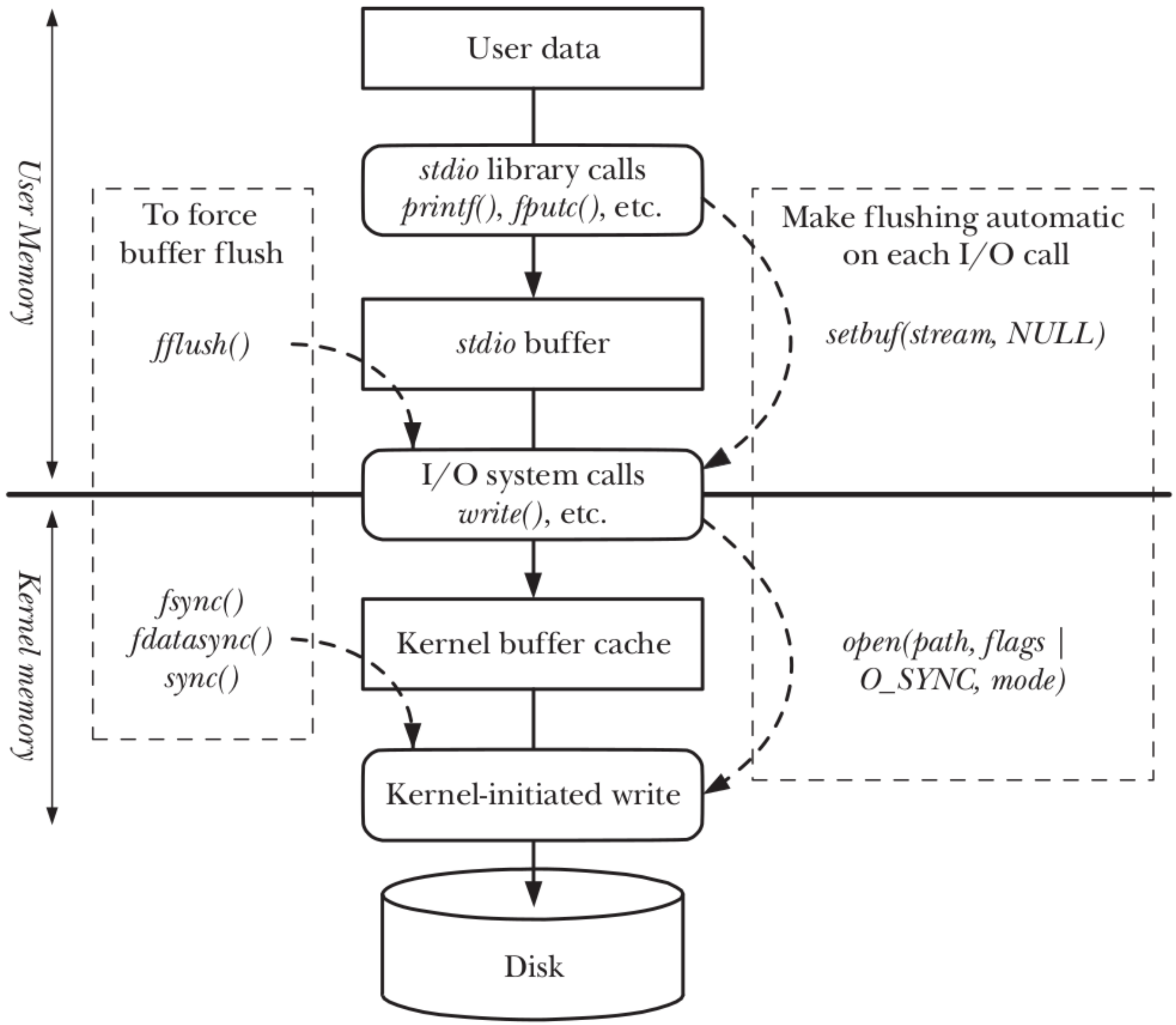


Figure 13-1: Summary of I/O buffering