MODULE 3

CHAPTER 1

FILES & DIRECTORIES

The four stat functions in UNIX system programming are part of the sys/stat.h library and are used to obtain information about files and their attributes. Here's a detailed breakdown of these functions:

1. stat

Prototype:

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

Description:

- Retrieves information about the file or directory specified by pathname and stores it in the structure pointed to by buf.
- It follows symbolic links. If pathname is a symbolic link, it retrieves information about the target of the link.

Use Case: To get file metadata when you know the path to the file.

2. fstat

Prototype:

```
int fstat(int fd, struct stat *buf);
```

Description:

- Retrieves information about a file descriptor fd.
- Since it works with an open file descriptor, it can be used for files, sockets, pipes, and other file-like objects.

Use Case: To get metadata of a file already opened via a file descriptor.

3. lstat

Prototype:

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

Description:

- Similar to stat, but if pathname refers to a symbolic link, lstat returns information about the link itself, not its target.

Use case: To get metadata of a symbolic link, rather than the file it points to.

4. fstatat

Prototype:

```
int fstatat(int fd, const char *restrict pathname, struct stat *restrict buf, int flag);
```

Description:

- A more flexible version of stat and lstat.
- Allows the specification of a file descriptor fd as a starting point for relative path resolution.
- The flag parameter modifies behavior:
  - AT_SYMLINK_NOFOLLOW: Do not follow symbolic links (like lstat).
  - AT_EMPTY_PATH: If pathname is an empty string, retrieve information about the file referred to by fd.

Use case: To perform file metadata queries with additional control over relative paths and symbolic link handling.

Return Value

- All four functions return:
  - 0: Success. File information is stored in the struct stat pointed to by buf.
  - -1: Error. errno is set to indicate the error type.

The buf argument is a pointer to a structure that we must supply. The functions fill in the structure. The definition of the structure can differ among implementations, but it could look like

```
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 */
  struct timespec st_atim;    /* time of last access */
  struct timespec st_mtim;    /* time of last modification */
  struct timespec st_ctim;    /* time of last file status change */
  blksize_t       st_blksize; /* best I/O block size */
  blkcnt_t        st_blocks;  /* number of disk blocks allocated */
};
```

The timespec structure type defines time in terms of seconds and nanoseconds. It includes at least the following fields: time_t tv_sec; long tv_nsec;

time_t tv_sec;

long tv_nsec;

FILE TYPES:

Most files on a UNIX system are either regular files or directories, but there are additional types of files. The types are

1. Regular file: The most common type of file, which contains data of some form. There is no distinction to the UNIX kernel whether this data is text or binary. Any interpretation of the contents of a regular file is left to the application processing the file. One notable exception to this is with binary executable files. To execute a program, the kernel must understand its format. All binary executable files conform to a format that allows the kernel to identify where to load a program's text and data.

2. Directory file: A file that contains the names of other files and pointers to information on these files. Any process that has read permission for a directory file can read the contents of

the directory, but only the kernel can write directly to a directory file. Processes must use the functions described in this chapter to make changes to a directory.

3. Block special file: A type of file providing buffered I/O access in fixed-size units to devices such as disk drives.

4. Character special file: A type of file providing unbuffered I/O access in variable-sized units to devices. All devices on a system are either block special files or character special files.

5. FIFO: A type of file used for communication between processes. It's sometimes called a named pipe. We describe FIFOs in Section 15.5. 6.

6. Socket: A type of file used for network communication between processes. A socket can also be used for non-network communication between processes on a single host.

7. Symbolic link: A type of file that points to another file.

The type of a file is encoded in the st_mode member of the stat structure. The argument to each of these macros is the st_mode member from the stat structure.

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

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

POSIX.1 allows implementations to represent interprocess communication (IPC) objects, such as message queues and semaphores, as files. The macros shown in Figure 4.2 allow us to determine the type of IPC object from the stat structure. Instead of taking the st_mode member as an argument, these macros differ from those in Figure 4.1 in that their argument is a pointer to the stat structure.

Macro	Type of object
S_TYPEISMQ( )	message queue
S_TYPEISSEM( )	semaphore
S_TYPEISSHM( )	shared memory object

Figure 4.2 IPC type macros in <sys/stat.h>

The program prints the type of file for each command-line argument.

```
#include "apue.h"
#include <sys/stat.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
   int i;
   struct stat buf;
   char *ptr;

   if (argc < 2)
   {
      fprintf(stderr, "Usage: %s <file>...\n", argv[0]);
      return 1;
   }

   for (i = 1; i < argc; i++)
   {
      printf("%s: ", argv[i]);

      if (lstat(argv[i], &buf) < 0)
      {
         err_ret("lstat error");                    // Function from `apue.h` to print an 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))
            ptr = "socket";
        else
            ptr = "** unknown mode **";

        printf("%s\n", ptr);
    }
    return 0;
}
```

OUTPUT:

```
$ ./a.out /etc/passwd /etc /dev/log /dev/tty \
> /var/lib/oprofile/opd_pipe /dev/sr0 /dev/cdrom
/etc/passwd: regular
/etc: directory
/dev/log: socket
/dev/tty: character special
/var/lib/oprofile/opd_pipe: fifo
/dev/sr0: block special
/dev/cdrom: symbolic link
```

Set-User-ID and Set-Group-ID

Every process has six or more IDs associated with it. These are shown in below figure

real user ID real group ID	who we really are
effective user ID effective group ID supplementary group IDs	used for file access permission checks
saved set-user-ID saved set-group-ID	saved by `exec` functions

Figure 4.5 User IDs and group IDs associated with each process

The real user ID and real group ID identify who we really are. These two fields are taken from our entry in the password file when we log in. Normally, these values don't change during a login session, although there are ways for a superuser process to change them.

The effective user ID, effective group ID, and supplementary group IDs determine our file access permissions.

The saved set-user-ID and saved set-group-ID contain copies of the effective user ID and the effective group ID, respectively, when a program is executed. We describe the function of these two saved values when

File Acess Permission:

There are nine permission bits for each file, divided into three categories. They are shown in figure below-

st_mode mask	Meaning
S_IRUSR	user-read
S_IWUSR	user-write
S_IXUSR	user-execute
S_IRGRP	group-read
S_IWGRP	group-write
S_IXGRP	group-execute
S_IROTH	other-read
S_IWOTH	other-write
S_IXOTH	other-execute

The term user in the first three rows in above figure refers to the owner of the file. The chmod(1) command, which is typically used to modify these nine permission bits, allows us to specify u for user (owner), g for group, and o for other. The three categories in above figure — read, write, and execute—are used in various ways by different functions. We'll summarize them here-

- The first rule is that whenever we want to open any type of file by name, we must have execute permission in each directory mentioned in the name, including the current directory, if it is implied. This is why the execute permission bit for a directory is often called the search bit.

  For example, to open the file /usr/include/stdio.h, we need execute permission in the directory /, execute permission in the directory /usr, and execute permission in the directory /usr/include. We then need appropriate permission for the file itself, depending on how we're trying to open it: read-only, read–write, and so on.
- The read permission for a file determines whether we can open an existing file for reading: the O_RDONLY and O_RDWR flags for the open function.
- The write permission for a file determines whether we can open an existing file for writing: the O_WRONLY and O_RDWR flags for the open function.
- We must have write permission for a file to specify the O_TRUNC flag in the open function. We cannot create a new file in a directory unless we have write permission and execute permission in the directory.
- To delete an existing file, we need write permission and execute permission in the directory containing the file. We do not need read permission or write permission for the file itself.

- Execute permission for a file must be on if we want to execute the file using any of the seven exec function. The file also has to be a regular file.

The file access tests that the kernel performs each time a process opens, creates, or deletes a file depend on the owners of the file (st_uid and st_gid), the effective IDs of the process (effective user ID and effective group ID), and the supplementary group IDs of the process, if supported. The two owner IDs are properties of the file, whereas the two effective IDs and the supplementary group IDs are properties of the process.

The tests performed by the kernel are as follows:

1. If the effective user ID of the process is 0 (the superuser), access is allowed. This gives the superuser free rein throughout the entire file system.

2. If the effective user ID of the process equals the owner ID of the file (i.e., the process owns the file), access is allowed if the appropriate user access permission bit is set. Otherwise, permission is denied. By appropriate access permission bit, we mean that if the process is opening the file for reading, the user-read bit must be on. If the process is opening the file for writing, the user-write bit must be on. If the process is executing the file, the user-execute bit must be on.

3. If the effective group ID of the process or one of the supplementary group IDs of the process equals the group ID of the file, access is allowed if the appropriate group access permission bit is set. Otherwise, permission is denied. 4. If the appropriate other access permission bit is set, access is allowed. Otherwise, permission is denied.

Ownership of New Files and Directories:

The user ID of a new file is set to the effective user ID of the process. POSIX.1 allows an implementation to choose one of the following options to determine the group ID of a new file:

- The group ID of a new file can be the effective group ID of the process.
- The group ID of a new file can be the group ID of the directory in which the file is being created.

- Using the second option—inheriting the directory's group ID—assures us that all files and directories created in that directory will have the same group ID as the directory. This group ownership of files and directories will then propagate down the hierarchy from that point.

access and faccessat Functions:

As we described earlier, when we open a file, the kernel performs its access tests based on the effective user and group IDs. Sometimes, however, a process wants to test accessibility based on the real user and group IDs. This is useful when a process is running as someone else, using either the set-user-ID or the set-group-ID feature. Even though a process might be set-user-ID to root, it might still want to verify that the real user can access a given file. The access and faccessat functions base their tests on the real user and group IDs. (Replace effective with real in the four steps at the end)

```
#include <unistd.h>
int access(const char *pathname, int mode);
int faccessat(int fd, const char *pathname, int mode, int flag);
                                                    Both return: 0 if OK, –1 on error
```

The mode is either the value F_OK to test if a file exists, or the bitwise OR of any of the flags shown in Figure

mode	Description
R_OK	test for read permission
W_OK	test for write permission
X_OK	test for execute permission

The faccessat function behaves like access when the pathname argument is absolute or when the fd argument has the value AT_FDCWD and the pathname argument is relative. Otherwise, faccessat evaluates the pathname relative to the open directory referenced by the fd argument.

The flag argument can be used to change the behavior of faccessat. If the AT_EACCESS flag is set, the access checks are made using the effective user and group IDs of the calling process instead of the real user and group IDs.

```
#include "apue.h"
#include <fcntl.h>

int
main(int argc, char *argv[])
{
    if (argc != 2)
        err_quit("usage: a.out <pathname>");
    if (access(argv[1], R_OK) < 0)
        err_ret("access error for %s", argv[1]);
    else
        printf("read access OK\n");
    if (open(argv[1], O_RDONLY) < 0)
        err_ret("open error for %s", argv[1]);
    else
        printf("open for reading OK\n");
    exit(0);
}
```

Figure 4.8 Example of access function

Umask Function

The umask function sets the file mode creation mask for the process and returns the previous value.

#include<sys/stat.h>

mode_t umask(mode_t cmask);

Returns: previous file mode creation mask

The cmask argument is formed as the bitwise OR of any of the nine constants from S_IRUSR, S_IWUSR, and so on. The file mode creation mask is used whenever the process creates a new file or a new directory. Any bits that are on in the file mode creation mask are turned off in the file's mode.

```
#include "apue.h"
#include <fcntl.h>
#define RWRWRW (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)
int
main(void)
{
    umask(0);
    if (creat("foo", RWRWRW) < 0)
        err_sys("creat error for foo");
    umask(S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
    if (creat("bar", RWRWRW) < 0)
        err_sys("creat error for bar");
    exit(0);
}
```

Output:

```
$ umask                                  first print the current file mode creation mask
002
$ ./a.out
$ ls -l foo bar
-rw-------  1 sar                         0 Dec  7 21:20 bar
-rw-rw-rw-  1 sar                         0 Dec  7 21:20 foo
$ umask                                  see if the file mode creation mask changed
002
```

Users can set the umask value to control the default permissions on the files they create. This value is expressed in octal, with one bit representing one permission to be masked off, Permissions can be denied by setting the corresponding bits. Some common umask values are 002 to prevent others from writing your files, 022 to prevent group members and others from writing your files, and 027 to prevent group members from writing your files and others from reading, writing, or executing your files.

Mask bit	Meaning
0400	user-read
0200	user-write
0100	user-execute
0040	group-read
0020	group-write
0010	group-execute
0004	other-read
0002	other-write
0001	other-execute

Figure 4.10 The umask file access permission bits

chmod, fchmod and fchmodat Functions

The chmod, fchmod, and fchmodat functions allow us to change the file access permissions for an existing file.

```
#include <sys/stat.h>
int chmod(const char *pathname, mode_t mode);
int fchmod(int fd, mode_t mode);
int fchmodat(int fd, const char *pathname, mode_t mode, int flag);
```

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

The chmod function operates on the specified file, whereas the fchmod function operates on a file that has already been opened. The fchmodat function behaves like chmod when the pathname argument is absolute or when the fd argument has the value AT_FDCWD and the pathname argument is relative. Otherwise, fchmodat evaluates the pathname relative to the open directory referenced by the fd argument. The flag argument can be used to change the behavior of fchmodat—when the AT_SYMLINK_NOFOLLOW flag is set, fchmodat doesn't follow symbolic links.

To change the permission bits of a file, the effective user ID of the process must be equal to the owner ID of the file, or the process must have superuser permissions. The mode is specified as the bitwise OR of the constants shown in below figure-

mode	Description
S_ISUID	set-user-ID on execution
S_ISGID	set-group-ID on execution
S_ISVTX	saved-text (sticky bit)
S_IRWXU	read, write, and execute by user (owner)
S_IRUSR	read by user (owner)
S_IWUSR	write by user (owner)
S_IXUSR	execute by user (owner)
S_IRWXG	read, write, and execute by group
S_IRGRP	read by group
S_IWGRP	write by group
S_IXGRP	execute by group
S_IRWXO	read, write, and execute by other (world)
S_IROTH	read by other (world)
S_IWOTH	write by other (world)
S_IXOTH	execute by other (world)

Figure 4.11 The *mode* constants for chmod functions, from <sys/stat.h>

chown, fchown, fchownat, and lchown functions

The chown functions allow us to change a file's user ID and group ID, but if either of the arguments owner or group is −1, the corresponding ID is left unchanged.

```
#include <unistd.h>
int chown(const char *pathname, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
int fchownat(int fd, const char *pathname, uid_t owner, gid_t group,
             int flag);
int lchown(const char *pathname, uid_t owner, gid_t group);
```

All four return: 0 if OK, –1 on error

These four functions operate similarly unless the referenced file is a symbolic link. In that case, lchown and fchownat (with the AT_SYMLINK_NOFOLLOW flag set) change the owners of the symbolic link itself, not the file pointed to by the symbolic link.

The **fchown** function changes the ownership of the open file referenced by the fd argument. Since it operates on a file that is already open, it can't be used to change the ownership of a symbolic link.

The **fchownat** function behaves like either chown or lchown when the pathname argument is absolute or when the fd argument has the value AT_FDCWD and the pathname argument is relative. In these cases, fchownat acts like lchown if the AT_SYMLINK_NOFOLLOW flag is set in the flag argument, or it acts like chown if the AT_SYMLINK_NOFOLLOW flag is clear. When the fd argument is set to the file descriptor of an open directory and the pathname argument is a relative pathname, **fchownat** evaluates the pathname relative to the open directory.

If _POSIX_CHOWN_RESTRICTED is in effect for the specified file, then

1. Only a superuser process can change the user ID of the file.

2. A nonsuperuser process can change the group ID of the file if the process owns the file (the effective user ID equals the user ID of the file), owner is specified as −1 or equals the user ID of the file, and group equals either the effective group ID of the process or one of the process's supplementary group IDs.

File Size:

The st_size member of the stat structure contains the size of the file in bytes. This field is meaningful only for regular files, directories, and symbolic links.

For a regular file, a file size of 0 is allowed. We'll get an end-of-file indication on the first read of the file. For a directory, the file size is usually a multiple of a number, such as 16 or 512. For a symbolic link, the file size is the number of bytes in the filename.

For example, in the following case, the file size of 7 is the length of the pathname usr/lib: lrwxrwxrwx 1 root 7 Sep 25 07:14 lib -> usr/lib

File Truncation:

Sometimes we would like to truncate a file by chopping off data at the end of the file. Emptying a file, which we can do with the O_TRUNC flag to open, is a special case of truncation.

```
#include <unistd.h>
int truncate(const char *pathname, off_t length);
int ftruncate(int fd, off_t length);
```

Both return: 0 if OK, –1 on error

These two functions truncate an existing file to length bytes. If the previous size of the file was greater than length, the data beyond length is no longer accessible. Otherwise, if the previous size was less than length, the file size will increase and the data between the old end of file and the new end of file will read as 0.

File System:

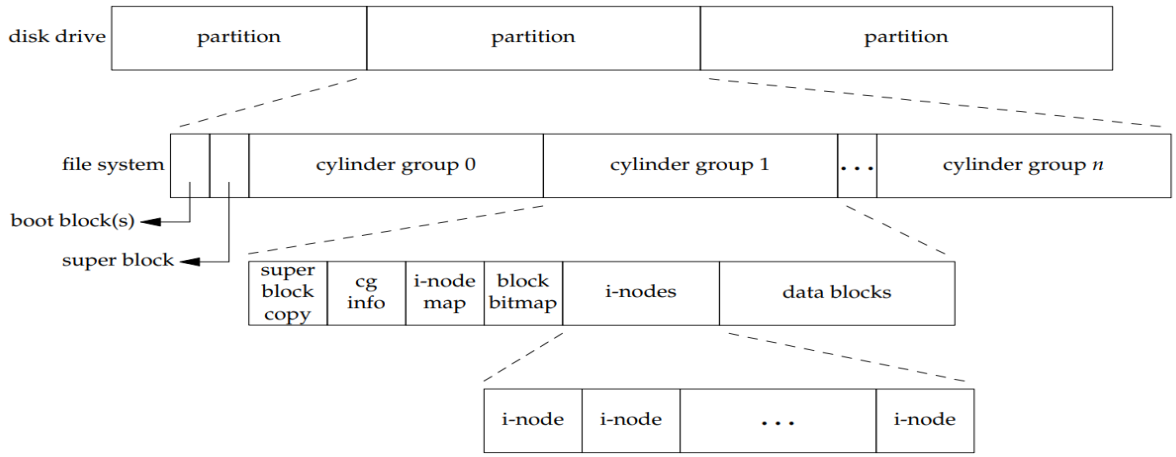


Figure 4.13 Disk drive, partitions, and a file system

If we examine the i-node and data block portion of a cylinder group in more detail, we could have the arrangement shown in Figure 4.14.

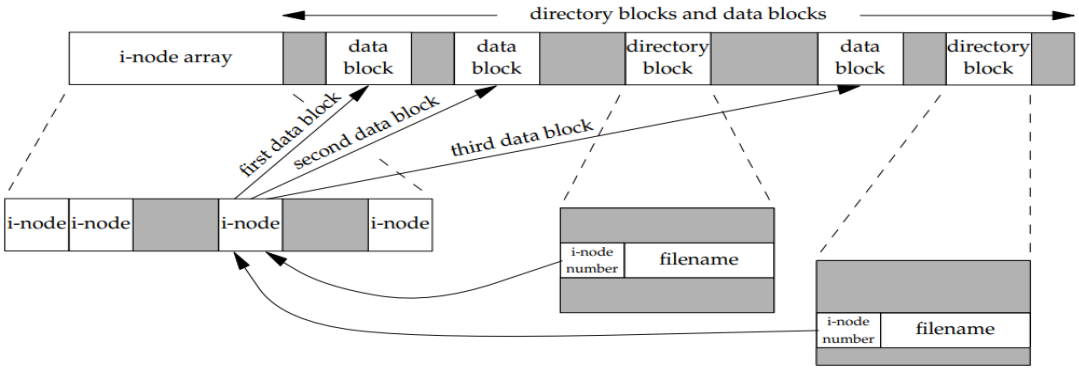


Figure 4.14 Cylinder group's i-nodes and data blocks in more detail

Note the following points from Figure 4.14.

• Two directory entries point to the same i-node entry. Every i-node has a link count that contains the number of directory entries that point to it. Only when the link count goes to 0 can

the file be deleted (thereby releasing the data blocks associated with the file). This is why the operation of ‘‘unlinking a file’’ does not always mean ‘‘deleting the blocks associated with the file.’’ This is why the function that removes a directory entry is called unlink, not delete. In the stat structure, the link count is contained in the st_nlink member. Its primitive system data type is nlink_t. These types of links are called hard links. Recall from Section 2.5.2 that the POSIX.1 constant LINK_MAX specifies the maximum value for a file’s link count.

• The other type of link is called a symbolic link. With a symbolic link, the actual contents of the file—the data blocks—store the name of the file that the symbolic link points to. In the following example, the filename in the directory entry is the three-character string lib and the 7 bytes of data in the file are usr/lib: lrwxrwxrwx 1 root 7 Sep 25 07:14 lib -> usr/lib The file type in the i-node would be S_IFLNK so that the system knows that this is a symbolic link.

• The i-node contains all the information about the file: the file type, the file’s access permission bits, the size of the file, pointers to the file’s data blocks, and so on. Most of the information in the stat structure is obtained from the i-node. Only two items of interest are stored in the directory entry: the filename and the i-node number. The other items—the length of the filename and the length of the directory record—are not of interest to this discussion. The data type for the i-node number is ino_t.

• Because the i-node number in the directory entry points to an i-node in the same file system, a directory entry can’t refer to an i-node in a different file system. This is why the ln(1) command (make a new directory entry that points to an existing file) can’t cross file systems. We describe the link function in the next section.

• When renaming a file without changing file systems, the actual contents of the file need not be moved—all that needs to be done is to add a new directory entry that points to the existing i-node and then unlink the old directory entry. The link count will remain the same. For example, to rename the file /usr/lib/foo to /usr/foo, the contents of the file foo need not be moved if the directories /usr/lib and /usr are on the same file system. This is how the mv(1) command usually operates.

Figure 4.15 shows the result. Note that in this figure, we explicitly show the entries for dot and dot-dot.

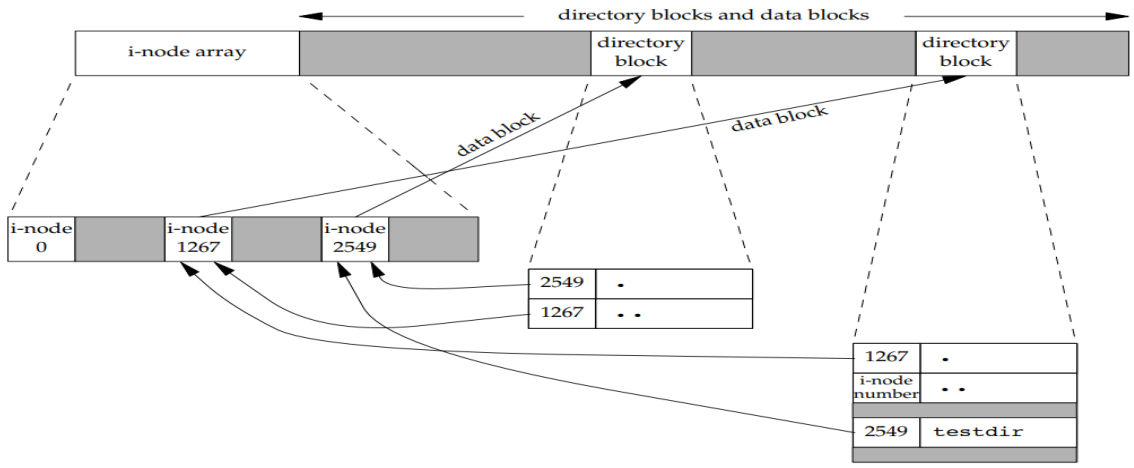


Figure 4.15 Sample cylinder group after creating the directory `testdir`

link, linkat, unlink, unlinkat and remove Functions

We can use either the link function or the linkat function to create a link to an existing file.

```
#include <unistd.h>

int link(const char *existingpath, const char *newpath);

int linkat(int efd, const char *existingpath, int nfd, const char *newpath,
           int flag);
```

Both return: 0 if OK, –1 on error

These functions create a new directory entry, newpath, that references the existing file existingpath. If the newpath already exists, an error is returned. Only the last component of the newpath is created. The rest of the path must already exist.

With the **linkat function**, the existing file is specified by both the efd and existingpath arguments, and the new pathname is specified by both the nfd and newpath arguments. By default, if either pathname is relative, it is evaluated relative to the corresponding file descriptor. If either file descriptor is set to AT_FDCWD, then the corresponding pathname, if it is a relative pathname, is evaluated relative to the current directory. If either pathname is absolute, then the corresponding file descriptor argument is ignored.

When the existing file is a **symbolic link**, the flag argument controls whether the linkat function creates a link to the symbolic link or to the file to which the symbolic link points. If the AT_SYMLINK_FOLLOW flag is set in the flag argument, then a link is created to the target of the symbolic link. If this flag is clear, then a link is created to the symbolic link itself.

To remove an existing directory entry, we call the unlink function.

```
#include <unistd.h>

int unlink(const char *pathname);

int unlinkat(int fd, const char *pathname, int flag);
```

Both return: 0 if OK, –1 on error

As mentioned earlier, to unlink a file, we must have write permission and execute permission in the directory containing the directory entry, as it is the directory entry that we will be removing. If the sticky bit is set in this directory we must have write permission for the directory and meet one of the following criteria:

- Own the file
- Own the directory
- Have superuser privileges

If the pathname argument is a relative pathname, then the unlinkat function evaluates the pathname relative to the directory represented by the fd file descriptor argument. If the pathname argument is an absolute pathname, then the fd argument is ignored.

```
#include "apue.h"
#include <fcntl.h>

int
main(void)
{
    if (open("tempfile", O_RDWR) < 0)
        err_sys("open error");
    if (unlink("tempfile") < 0)
        err_sys("unlink error");
    printf("file unlinked\n");
    sleep(15);
    printf("done\n");
    exit(0);
}
```

rename and renameat Functions

1. A file or a directory is renamed with either the rename or renameat function. There are several conditions to describe for these functions, depending on whether oldname refers to a file, a directory, or a symbolic link. We must also describe what happens if newname already exists. If oldname specifies a file that is not a directory, then we are renaming a file or a symbolic link. In this case, if newname exists, it cannot refer to a directory. If newname exists and is not a directory, it is removed, and oldname is renamed to newname. We must have write permission for the directory containing oldname and the directory containing newname, since we are changing both directories.

2. If oldname specifies a directory, then we are renaming a directory. If newname exists, it must refer to a directory, and that directory must be empty. (When we say that a directory is empty, we mean that the only entries in the directory are dot and dot-dot.) If newname exists and is an empty directory, it is removed, and oldname is renamed to newname. Additionally, when we're renaming a directory, newname cannot contain a path prefix that names oldname. For example, we can't rename /usr/foo to /usr/foo/testdir, because the old name (/usr/foo) is a path prefix of the new name and cannot be removed.

3. If either oldname or newname refers to a symbolic link, then the link itself is processed, not the file to which it resolves.

4. We can't rename dot or dot-dot. More precisely, neither dot nor dot-dot can appear as the last component of oldname or newname.

5. As a special case, if oldname and newname refer to the same file, the function returns successfully without changing anything.

Symbolic Links

A symbolic link is an indirect pointer to a file, unlike the hard links described in the previous section, which pointed directly to the i-node of the file. Symbolic links were introduced to get around the limitations of hard links.

• Hard links normally require that the link and the file reside in the same file system.

• Only the superuser can create a hard link to a directory (when supported by the underlying file system).

One exception to the behavior summarized in Figure 4.17 occurs when the open function is called with both O_CREAT and O_EXCL set. In this case, if the pathname refers to a symbolic link, open will fail with errno set to EEXIST. This behavior is intended to close a security hole so that privileged processes can't be fooled into writing to the wrong files.

This creates a directory foo that contains the file a and a symbolic link that points to foo. We show this arrangement in Figure 4.18, drawing a directory as a circle and a file as a square.

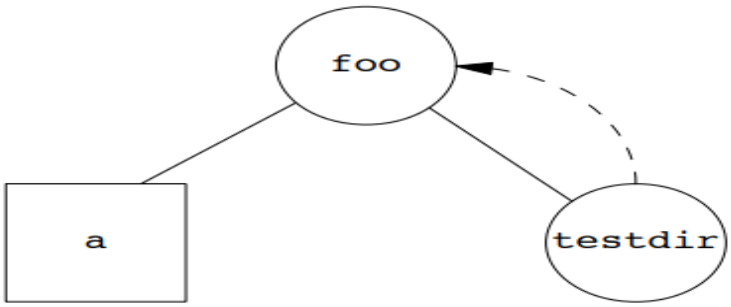


Figure 4.18 Symbolic link `testdir` that creates a loop

Creating and Reading Symbolic Links:

A symbolic link is created with either the symlink or symlinkat function.

```
#include <unistd.h>
int symlink(const char *actualpath, const char *sympath);
int symlinkat(const char *actualpath, int fd, const char *sympath);
                                        Both return: 0 if OK, –1 on error
```

The symlinkat function is similar to symlink, but the sympath argument is evaluated relative to the directory referenced by the open file descriptor for that directory (specified by the fd argument). If the sympath argument specifies an absolute pathname or if the fd argument has the special value AT_FDCWD, then symlinkat behaves the same way as symlink.

Because the open function follows a symbolic link, we need a way to open the link itself and read the name in the link. The readlink and readlinkat functions do this.

```
#include <unistd.h>

ssize_t readlink(const char* restrict pathname, char *restrict buf,
                 size_t bufsize);

ssize_t readlinkat(int fd, const char* restrict pathname,
                   char *restrict buf, size_t bufsize);
```

Both return: number of bytes read if OK, –1 on error

File Times:

File systems that store timestamps in a resolution higher than seconds, the partial seconds value will be converted into nanoseconds and returned in the nanoseconds fields. Three time fields are maintained for each file. Their purpose is summarized in Figure 4.19.

Field	Description	Example	ls(1) option
st_atim	last-access time of file data	read	-u
st_mtim	last-modification time of file data	write	default
st_ctim	last-change time of i-node status	chmod, chown	-c

Figure 4.19 The three time values associated with each file

Figure 4.20 summarizes the effects of the various functions that we've described on these three times. Recall from Section 4.14 that a directory is simply a file containing directory entries: filenames and associated i-node numbers. Adding, deleting, or modifying these directory entries can affect the three times associated with that directory. This is why Figure 4.20 contains one column for the three times associated with the file or directory and another column for the three times associated with the parent directory of the referenced file or directory. For example, creating a new file affects the directory that contains the new file, and it affects the i-node for the new file. Reading or writing a file, however, affects only the i-node of the file and has no effect on the directory.

Function	Referenced file or directory			Parent directory of referenced file or directory			Section	Note
	a	m	c	a	m	c		
chmod, fchmod			•				4.9	
chown, fchown			•				4.11	
creat	•	•	•		•	•	3.4	O_CREAT new file
creat		•	•				3.4	O_TRUNC existing file
exec	•						8.10	
lchown			•				4.11	
link			•		•	•	4.15	parent of second argument
mkdir	•	•	•		•	•	4.21	
mkfifo	•	•	•		•	•	15.5	
open	•	•	•		•	•	3.3	O_CREAT new file
open		•	•				3.3	O_TRUNC existing file
pipe	•	•	•				15.2	
read	•						3.7	
remove			•		•	•	4.15	remove file = unlink
remove					•	•	4.15	remove directory = rmdir
rename			•		•	•	4.16	for both arguments
rmdir					•	•	4.21	
truncate, ftruncate		•	•				4.13	
unlink			•		•	•	4.15	
utimes, utimensat, futimens	•	•	•				4.20	
write		•	•				3.8	

Figure 4.20 Effect of various functions on the access, modification, and changed-status times

Futimens, utimensat and utimes Function:

Several functions are available to change the access time and the modification time of a file. The futimens and utimensat functions provide nanosecond granularity for specifying timestamps, using the timespec structure.

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

int futimens(int fd, const struct timespec times[2]);

int utimensat(int fd, const char *path, const struct timespec times[2],
              int flag);
```

Both return: 0 if OK, –1 on error

In both functions, the first element of the times array argument contains the access time, and the second element contains the modification time.. Partial seconds are expressed in nanoseconds.

Timestamps can be specified in one of four ways:

1. The times argument is a null pointer. In this case, both timestamps are set to the current time.

2. The times argument points to an array of two timespec structures. If either tv_nsec field has the special value UTIME_NOW, the corresponding timestamp is set to the current time. The corresponding tv_sec field is ignored.

3. The times argument points to an array of two timespec structures. If either tv_nsec field has the special value UTIME_OMIT, then the corresponding timestamp is unchanged. The corresponding tv_sec field is ignored.

4. The times argument points to an array of two timespec structures and the tv_nsec field contains a value other than UTIME_NOW or UTIME_OMIT. In this case, the corresponding timestamp is set to the value specified by the corresponding tv_sec and tv_nsec fields.

The privileges required to execute these functions depend on the value of the times argument.

• If times is a null pointer or if either tv_nsec field is set to UTIME_NOW, either the effective user ID of the process must equal the owner ID of the file, the process must have write permission for the file, or the process must be a superuser process.

• If times is a non-null pointer and either tv_nsec field has a value other than UTIME_NOW or UTIME_OMIT, the effective user ID of the process must equal the owner ID of the file, or the process must be a superuser process. Merely having write permission for the file is not adequate.

• If times is a non-null pointer and both tv_nsec fields are set to UTIME_OMIT, no permissions checks are performed.

Both futimens and utimensat are included in POSIX.1. A third function, utimes, is included in the Single UNIX Specification as part of the XSI option.

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

int utimes(const char *pathname, const struct timeval times[2]);
                                                    Returns: 0 if OK, –1 on error
```

The utimes function operates on a pathname. The times argument is a pointer to an array of two timestamps—access time and modification time—but they are expressed in seconds and microseconds:

```
struct timeval {
        time_t tv_sec;      /* seconds */
        long   tv_usec;     /* microseconds */
};
```

mkdir, mkdirat, and rmdir Functions:

Directories are created with the mkdir and mkdirat functions, and deleted with the rmdir function.

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

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

int mkdirat(int fd, const char *pathname, mode_t mode);
```

Both return: 0 if OK, –1 on error

These functions create a new, empty directory. The entries for dot and dot-dot are created automatically. The specified file access permissions, mode, are modified by the file mode creation mask of the process.

The mkdirat function is similar to the mkdir function. When the fd argument has the special value AT_FDCWD, or when the pathname argument specifies an absolute pathname, mkdirat behaves exactly like mkdir. Otherwise, the fd argument is an open directory from which relative pathnames will be evaluated.

```
#include <unistd.h>

int rmdir(const char *pathname);
```

Returns: 0 if OK, –1 on error

If the link count of the directory becomes 0 with this call, and if no other process has the directory open, then the space occupied by the directory is freed. If one or more processes have the directory open when the link count reaches 0, the last link is removed and the dot and dot-

dot entries are removed before this function returns. Additionally, no new files can be created in the directory.

Reading Directories:

```
#include <dirent.h>
DIR *opendir(const char *pathname);
DIR *fdopendir(int fd);
                                        Both return: 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
long telldir(DIR *dp);
                              Returns: current location in directory associated with dp
void seekdir(DIR *dp, long loc);
```

The **fdopendir** function first appeared in version 4 of the Single UNIX Specification. It provides a way to convert an open file descriptor into a DIR structure for use by the directory handling functions.

The **telldir** and seekdir functions are not part of the base POSIX.1 standard. They are included in the XSI option in the Single UNIX Specification, so all conforming UNIX System implementations are expected to provide them.

The dirent structure defined in is implementation dependent.

Implementations define the structure to contain at least the following two members:

ino_t d_ino; /* i-node number */

char d_name[]; /* null-terminated filename */

The pointer to a DIR structure returned by **opendir** and **fdopendir** is then used with the other five functions. The opendir function initializes things so that the first readdir returns the first entry in the directory. When the DIR structure is created by fdopendir, the first entry returned by readdir depends on the file offset associated with the file descriptor passed to fdopendir..

Chdir, fchdir and getcwd Functions :

Every process has a current working directory. This directory is where the search for all relative pathnames starts When a user logs in to a UNIX system, the current working directory normally starts at the directory specified by the sixth field in the /etc/passwd file — the user 's home directory.

The current working directory is an attribute of a process; the home directory is an attribute of a login name. We can change the current working directory of the calling process by calling the chdir or fchdir function.

```
#include <unistd.h>

int chdir(const char *pathname);

int fchdir(int fd);

                                        Both return: 0 if OK, -1 on error
```

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);
}
```

Output:

```
$ pwd
/usr/lib
$ mycd
chdir to /tmp succeeded
$ pwd
/usr/lib
```

The current working directory for the shell that executed the mycd program didn't change. This is a side effect of the way that the shell executes programs. Each program is run in a separate process, so the current working directory of the shell is unaffected by the call to chdir in the program. For this reason, the chdir function has to be called directly from the shell, so the cd command is built into the shells.

getcwd()

```
#include <unistd.h>
char *getcwd(char *buf, size_t size);
                                        Returns: buf if OK, NULL on error
```

We must pass to this function the address of a buffer, buf, and its size (in bytes). The buffer must be large enough to accommodate the absolute pathname plus a terminating null byte, or else an error will be returned.

```
#include "apue.h"

int
main(void)
{
    char    *ptr;
    size_t      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);
}
```

Device Special Files:

The two fields st_dev and st_rdev are often confused. The rules for their use are simple.

- Every file system is known by its major and minor device numbers, which are encoded in the primitive system data type dev_t. The major number identifies the device driver and sometimes encodes which peripheral board to communicate with; the minor

number identifies the specific subdevice. Recall from Figure 4.13 that a disk drive often contains several file systems. Each file system on the same disk drive would usually have the same major number, but a different minor number.

- We can usually access the major and minor device numbers through two macros defined by most implementations: major and minor. Consequently, we don't care how the two numbers are stored in a dev_t object.
- The st_dev value for every filename on a system is the device number of the file system containing that filename and its corresponding i-node.
- Only character special files and block special files have an st_rdev value. This value contains the device number for the actual device.

CHAPTER 2

FILE INPUT

open and openat Functions

A file is opened or created by calling either the open function or the openat function.

```
#include <fcntl.h>

int open(const char *path, int oflag, ... /* mode_t mode */ );

int openat(int fd, const char *path, int oflag, ... /* mode_t mode */ );
```

Both return: file descriptor if OK, –1 on error

We show the last argument as ..., which is the ISO C way to specify that the number and types of the remaining arguments may vary. For these functions, the last argument is used only when a new file is being created.

The path parameter is the name of the file to open or create. This function has a multitude of options, which are specified by the oflag argument. This argument is formed by ORing together one or more of the following constants from the header:

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing.

O_EXEC Open for execute only.

O_SEARCH Open for search only

One and only one of the previous five constants must be specified. The following constants are optional:

1. **O_APPEND**
Ensures all writes append data to the end of the file, regardless of the file offset.

2. **O_CLOEXEC**
Automatically closes the file descriptor when an exec family function is called. Discussed in

3. **O_CREAT**
Creates the file if it doesn't exist. This requires a third argument (mode), which defines the file's access permission bits. The behavior of mode is influenced by the process's umask value.

4. **O_DIRECTORY**
Ensures that the path being opened is a directory. Returns an error if it is not.

5. O_EXCL
Used in conjunction with O_CREAT to ensure the file does not already exist. The file creation and existence check is performed atomically.

6. O_NOCTTY
Prevents the terminal device referenced by path from becoming the controlling terminal of the process.

7. O_NOFOLLOW
Prevents following symbolic links. If the path refers to a symbolic link, an error is generated.

8. O_NONBLOCK

If path refers to a FIFO, a block special file, or a character special file, this option sets the nonblocking mode for both the opening of the file and subsequent I/O.

9. **O_SYNC** Have each write wait for physical I/O to complete, including I/O necessary to update file attributes modified as a result of the write.

10. **O_TRUNC** If the file exists and if it is successfully opened for either write-only or read–write, truncate its length to 0.

11. **O_TTY_INIT** When opening a terminal device that is not already open, set the nonstandard termios parameters to values that result in behavior that conforms to the Single UNIX Specification.

The following two flags are also optional. They are part of the synchronized input and output option of the Single UNIX Specification

1. O_DSYNC

- **Behavior**: Ensures that each write operation waits for the physical I/O of the data to complete but doesn't wait for file attribute updates unless those attributes are necessary for accessing the written data.
- **Use Case**: Efficient for applications that require data consistency but do not need frequent updates to metadata such as file modification times.
- **Example**: Writing additional data to a file may not immediately update the file's timestamps but ensures that the data is fully written to storage.

2. O_SYNC

- **Behavior**: Ensures that both the file's data and its attributes are updated synchronously on every write. This means every write operation waits until data and metadata changes (like size or modification times) are physically written to storage before returning.
- **Comparison with O_DSYNC**: Unlike O_DSYNC, O_SYNC guarantees that metadata (e.g., timestamps) is also immediately consistent, even when overwriting existing data.

- **Behavior**: Ensures that a read operation waits until all pending write operations for the same file region are complete.
- **Use Case**: Useful for applications that require strict read-after-write consistency in multi-process or multi-threaded environments.

The fd parameter distinguishes the openat function from the open function. There are three possibilities:

1. The path parameter specifies an absolute pathname. In this case, the fd parameter is ignored and the openat function behaves like the open function.

2. The path parameter specifies a relative pathname and the fd parameter is a file descriptor that specifies the starting location in the file system where the relative pathname is to be evaluated. The fd parameter is obtained by opening the directory where the relative pathname is to be evaluated.

3. The path parameter specifies a relative pathname and the fd parameter has the special value AT_FDCWD. In this case, the pathname is evaluated starting in the current working directory and the openat function behaves like the open function.

create Function

A new file can also be created by calling the creat function.

```
#include <fcntl.h>
int creat(const char *path, mode_t mode);
                    Returns: file descriptor opened for write-only if OK, –1 on error
```

Note that this function is equivalent to

open(path, O_WRONLY | O_CREAT | O_TRUNC, mode);

One deficiency with creat is that the file is opened only for writing. Before the new version of open was provided, if we were creating a temporary file that we wanted to write and then read back, we had to call creat, close, and then open. A better way is to use the open function, as in

open(path, O_RDWR | O_CREAT | O_TRUNC, mode);

close Function

An open file is closed by calling the close function

```
#include <unistd.h>
int close(int fd);
                                    Returns: 0 if OK, –1 on error
```

Closing a file also releases any record locks that the process may have on the file. When a process terminates, all of its open files are closed automatically by the kernel. Many programs take advantage of this fact and don't explicitly close open files.

read Function

Data is read from an open file with the read function.

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t nbytes);

                    Returns: number of bytes read, 0 if end of file, -1 on error
```

If the read is successful, the number of bytes read is returned. If the end of file is encountered, 0 is returned.

There are several cases in which the number of bytes actually read is less than the amount requested:

• When reading from a regular file, if the end of file is reached before the requested number of bytes has been read. For example, if 30 bytes remain until the end of file and we try to read 100 bytes, read returns 30. The next time we call read, it will return 0 (end of file).

• When reading from a terminal device. Normally, up to one line is read at a time.

• When reading from a network. Buffering within the network may cause less than the requested amount to be returned.

• When reading from a pipe or FIFO. If the pipe contains fewer bytes than requested, read will return only what is available.

• When reading from a record-oriented device. Some record-oriented devices, such as magnetic tape, can return up to a single record at a time.

• When interrupted by a signal and a partial amount of data has already been read.

write Function

Data is written to an open file with the write function.

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t nbytes);

                    Returns: number of bytes written if OK, -1 on error
```

The return value is usually equal to the nbytes argument; otherwise, an error has occurred. A common cause for a write error is either filling up a disk or exceeding the file size limit for a given proce. For a regular file, the write operation starts at the file's current offset. If the O_APPEND option was specified when the file was opened, the file's offset is set to the current end of file before each write operation. After a successful write, the file's offset is incremented by the number of bytes actually written.

lseek Function

An open file's offset can be set explicitly by calling lseek.

```
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
                                    Returns: new file offset if OK, –1 on error
```

The interpretation of the offset depends on the value of the whence argument.

• If whence is SEEK_SET, the file's offset is set to offset bytes from the beginning of the file.

• If whence is SEEK_CUR, the file's offset is set to its current value plus the offset. The offset can be positive or negative.

• If whence is SEEK_END, the file's offset is set to the size of the file plus the offset. The offset can be positive or negative.

Because a successful call to lseek returns the new file offset, we can seek zero bytes from the current position to determine the current offset:

```
off_t currpos;

currpos = lseek(fd, 0, SEEK_CUR);
```

This technique can also be used to determine if a file is capable of seeking. If the file descriptor refers to a pipe, FIFO, or socket, lseek sets errno to ESPIPE and returns −1.

```
#include "apue.h"

int
main(void)
{
    if (lseek(STDIN_FILENO, 0, SEEK_CUR) == -1)
        printf("cannot seek\n");
    else
        printf("seek OK\n");
    exit(0);
}
```

Figure 3.1 Test whether standard input is capable of seeking

OUTPUT:

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

File Sharing

The UNIX System supports the sharing of open files among different processes

The kernel uses three data structures to represent an open file, and the relationships among them determine the effect one process has on another with regard to file sharing.

1. Every process has an entry in the process table. Within each process table entry is a table of open file descriptors, which we can think of as a vector, with one entry per descriptor. Associated with each file descriptor are

(a) The file descriptor flags (close-on-exec)

(b) A pointer to a file table entry

2. The kernel maintains a file table for all open files. Each file table entry contains

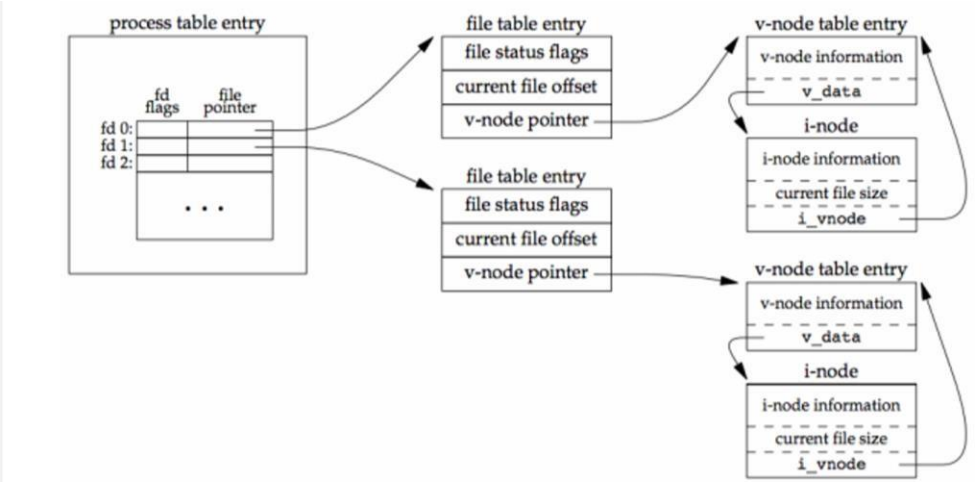
(a) The file status flags for the file, such as read, write, append, sync, and nonblocking

(b) The current file offset

(c) A pointer to the v-node table entry for the file

3. Each open file (or device) has a v-node structure that contains information about the type of file and pointers to functions that operate on the file. For most files, the v-node also contains the i-node for the file. This information is read from disk when the file is opened, so that all the pertinent information about the file is readily available. For example, the i-node contains the owner of the file, the size of the file, pointers to where the actual data blocks for the file are located on disk, and so on.

Figure belwo shows a pictorial arrangement of these three tables for a single process that has two different files open: one file is open on standard input (file descriptor 0), and the other is open on standard output (file descriptor 1).



If two independent processes have the same file open, we could have the arrangement shown below

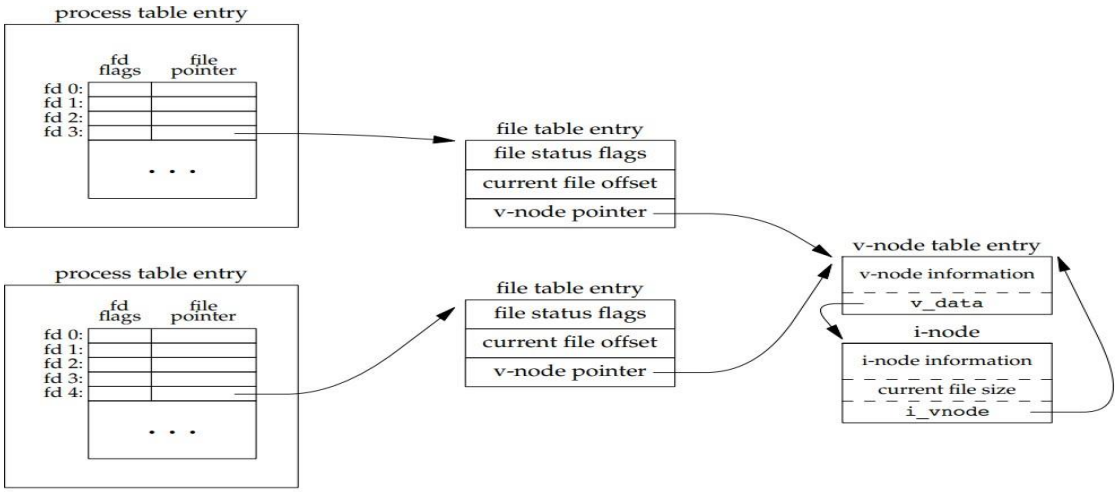


Figure 3.8 Two independent processes with the same file open

We assume here that the first process has the file open on descriptor 3 and that the second process has that same file open on descriptor 4. Each process that opens the file gets its own file table entry, but only a single v-node table entry is required for a given file. One reason each process gets its own file table entry is so that each process has its own current offset for the file.

Atomic Operation

Consider a single process that wants to append to the end of a file. Older versions of the UNIX System didn't support the O_APPEND option to open, so the program was coded as follows:

```
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");
```

The UNIX System provides an atomic way to do this operation if we set the O_APPEND flag when a file is opened. As we described in the previous section, this causes the kernel to position the file to its current end of file before each write. We no longer have to call lseek before each write.

pread and pwrite Operations

The Single UNIX Specification includes two functions that allow applications to seek and perform I/O atomically: pread and pwrite.

```
#include <unistd.h>
ssize_t pread(int fd, void *buf, size_t nbytes, off_t offset);
                        Returns: number of bytes read, 0 if end of file, –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
```

Calling pread is equivalent to calling lseek followed by a call to read, with the following exceptions.

• There is no way to interrupt the two operations that occur when we call pread.

• The current file offset is not updated.

Calling pwrite is equivalent to calling lseek followed by a call to write, with similar exceptions.

dup and dup2 Functions

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

```
#include <unistd.h>
int dup(int fd);
int dup2(int fd, int fd2);
                        Both return: new file descriptor if OK, –1 on error
```

The new file descriptor returned by dup is guaranteed to be the lowest-numbered available file descriptor. With dup2, we specify the value of the new descriptor with the fd2 argument. If fd2 is already open, it is first closed. If fd equals fd2, then dup2 returns fd2 without closing it. Otherwise, the FD_CLOEXEC file descriptor flag is cleared for fd2, so that fd2 is left open if the process calls exec.

The new file descriptor that is returned as the value of the functions shares the same file table entry as the fd argument. We show this in Figure 3.9.

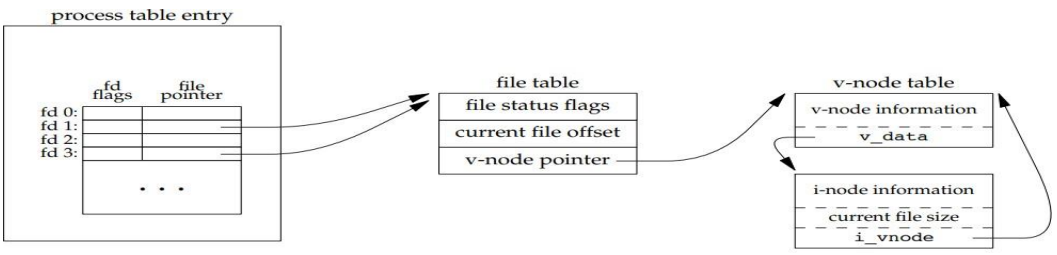


Figure 3.9 Kernel data structures after dup(1)

In this figure, we assume that when it's started, the process executes newfd = dup(1);

We assume that the next available descriptor is 3 (which it probably is, since 0, 1, and 2 are opened by the shell). Because both descriptors point to the same file table entry, they share the same file status flags—read, write, append, and so on—and the same current file offset. Each descriptor has its own set of file descriptor flags. As we describe in Section 3.14, the close-on-exec file descriptor flag for the new descriptor is always cleared by the dup functions. Another way to duplicate a descriptor is with the fcntl function, Indeed, the call

dup(fd); is equivalent to fcntl(fd, F_DUPFD, 0);

Similarly, the call

dup2(fd, fd2);

is equivalent to

close(fd2);

fcntl(fd, F_DUPFD, fd2);

In this last case, the dup2 is not exactly the same as a close followed by an fcntl.

The differences are as follows:

1. dup2 is an atomic operation, whereas the alternate form involves two function calls. It is possible in the latter case to have a signal catcher called between the close and the fcntl that could modify the file descriptors. The same problem could occur if a different thread changes the file descriptors.
2. There are some errno differences between dup2 and fcntl.

Sync, fsync, and fdatasync Functions

```
#include <unistd.h>
int fsync(int fd);
int fdatasync(int fd);
                                                    Returns: 0 if OK, –1 on error

void sync(void);
```

The sync function simply queues all the modified block buffers for writing and returns; it does not wait for the disk writes to take place. The function sync is normally called periodically

(usually every 30 seconds) from a system daemon, often called update. This guarantees regular flushing of the kernel's block buffers. The command sync(1) also calls the sync function.

The function fsync refers only to a single file, specified by the file descriptor fd, and waits for the disk writes to complete before returning. This function is used when an application, such as a database, needs to be sure that the modified blocks have been written to the disk.

The fdatasync function is similar to fsync, but it affects only the data portions of a file. With fsync, the file's attributes are also updated synchronously.

fcntl Function

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

In the examples in this section, the third argument is always an integer, corresponding to the comment in the function prototype just shown. However, the third argument becomes a pointer to a structure.

The fcntl function is used for five different purposes.

1. Duplicate an existing descriptor (cmd = F_DUPFD or F_DUPFD_CLOEXEC)

2. Get/set file descriptor flags (cmd = F_GETFD or F_SETFD)

3. Get/set file status flags (cmd = F_GETFL or F_SETFL)

4. Get/set asynchronous I/O ownership (cmd = F_GETOWN or F_SETOWN)

5. Get/set record locks (cmd = F_GETLK, F_SETLK, or F_SETLKW)

We'll now describe the first 8 of these 11 cmd values.

F_DUPFD Duplicate the file descriptor fd. The new file descriptor is returned as the value of the function. It is the lowest-numbered descriptor that is not already open, and that is greater than or equal to the third argument (taken as an integer). The new descriptor shares the same file table entry as fd. But the new descriptor has its own set of file descriptor flags, and its FD_CLOEXEC file descriptor flag is cleared.

F_DUPFD_CLOEXEC Duplicate the file descriptor and set the FD_CLOEXEC file descriptor flag associated with the new descriptor. Returns the new file descriptor. F_GETFD Return the file descriptor flags for fd as the value of the function. Currently, only one file descriptor flag is defined: the FD_CLOEXEC flag.

F_SETFD Set the file descriptor flags for fd. The new flag value is set from the third argument (taken as an integer).

F_GETFL Return the file status flags for fd as the value of the function.

File status flag	Description
O_RDONLY	open for reading only
O_WRONLY	open for writing only
O_RDWR	open for reading and writing
O_EXEC	open for execute only
O_SEARCH	open directory for searching only
O_APPEND	append on each write
O_NONBLOCK	nonblocking mode
O_SYNC	wait for writes to complete (data and attributes)
O_DSYNC	wait for writes to complete (data only)
O_RSYNC	synchronize reads and writes
O_FSYNC	wait for writes to complete (FreeBSD and Mac OS X only)
O_ASYNC	asynchronous I/O (FreeBSD and Mac OS X only)

Figure 3.10 File status flags for `fcntl`

Unfortunately, the five access-mode flags—O_RDONLY, O_WRONLY, O_RDWR, O_EXEC, and O_SEARCH—are not separate bits that can be tested. (As we mentioned earlier, the first three often have the values 0, 1, and 2, respectively, for historical reasons. Also, these five values are mutually exclusive; a file can have only one of them enabled.) Therefore, we must first use the O_ACCMODE mask to obtain the access-mode bits and then compare the result against any of the five values.

F_SETFL Set the file status flags to the value of the third argument (taken as an integer). The only flags that can be changed are O_APPEND, O_NONBLOCK, O_SYNC, O_DSYNC, O_RSYNC, O_FSYNC, and O_ASYNC.

F_GETOWN Get the process ID or process group ID currently receiving the SIGIO and SIGURG signals.

F_SETOWN Set the process ID or process group ID to receive the SIGIO and SIGURG signals. A positive arg specifies a process ID. A negative arg implies a process group ID equal to the absolute value of argument.

The return value from fcntl depends on the command. All commands return −1 on an error or some other value if OK. The following four commands have special return values: F_DUPFD, F_GETFD, F_GETFL, and F_GETOWN. The first command returns the new file descriptor, the next two return the corresponding flags, and the final command returns a positive process ID or a negative process group ID.

ioct Function

The ioctl function has always been the catchall for I/O operations. Anything that couldn't be expressed using one of the other functions in this chapter usually ended up being specified with an ioctl.

```
#include <unistd.h>     /* System V */
#include <sys/ioctl.h>  /* BSD and Linux */

int ioctl(int fd, int request, ...);
```

Returns: –1 on error, something else if OK

Each device driver can define its own set of ioctl commands. The system, however, provides generic ioctl commands for different classes of devices. Examples of some of the categories for these generic ioctl commands supported in FreeBSD are summarized in Figure

Category	Constant names	Header	Number of ioctls
disk labels	DIOxxx	<sys/disklabel.h>	4
file I/O	FIOxxx	<sys/filio.h>	14
mag tape I/O	MTIOxxx	<sys/mtio.h>	11
socket I/O	SIOxxx	<sys/sockio.h>	73
terminal I/O	TIOxxx	<sys/ttycom.h>	43

Figure 3.15 Common FreeBSD ioctl operations

The mag tape operations allow us to write end-of-file marks on a tape, rewind a tape, space forward over a specified number of files or records, and the like. None of these operations is easily expressed in terms of the other functions in the chapter (read, write, lseek, and so on), so the easiest way to handle these devices has always been to access their operations using ioctl.

/dev/fd

Newer systems provide a directory named /dev/fd whose entries are files named 0, 1, 2, and so on. Opening the file /dev/fd/n is equivalent to duplicating descriptor n, assuming that descriptor n is open.

In the function call

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

most systems ignore the specified mode, whereas others require that it be a subset of the mode used when the referenced file (standard input, in this case) was originally opened. Because the previous open is equivalent to

```
fd = dup(0);
```

the descriptors 0 and fd share the same file table entry. For example, if descriptor 0 was opened read-only, we can only read on fd. Even if the system ignores the open mode and the call

```
fd = open("/dev/fd/0", O_RDWR);
```

succeeds, we still can't write to fd.

We can also call creat with a /dev/fd pathname argument as well as specify O_CREAT in a call to open. This allows a program that calls creat to still work if the pathname argument is /dev/fd/1, for example.

The main use of the /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.

For example, the cat(1) program specifically looks for an input filename of - and uses it to mean standard input. The command

filter file2 | cat file1 - file3 | lpr

is an example. First, cat reads file1, then its standard input (the output of the filter program on file2), and then file3. If /dev/fd is supported, the special handling of - can be removed from cat, and we can enter filter

file2 | cat file1 /dev/fd/0 file3 | lpr

CHAPTER 4

PROCESS ENVIRONMENT

main Function

A C program starts execution with a function called main.

The prototype for the main function is int main(int argc, char *argv[]);

where argc is the number of command-line arguments, and argv is an array of pointers to the arguments

When a C program is executed by the kernel—by one of the exec functions, a special start-up routine is called before the main function is called. The executable program file specifies this routine as the starting address for the program; this is set up by the link editor when it is invoked by the C compiler. This start-up routine takes values from the kernel—the command-line arguments and the environment — and sets things up so that the main function is called.

Process Termination

There are eight ways for a process to terminate.

Normal termination occurs in five ways:

1. Return from main
2. Calling exit
3. Calling _exit or _Exit
4. Return of the last thread from its start routine.
5. Calling pthread_exit from the last thread

Abnormal termination occurs in three ways:

6. Calling abort
7. Receipt of a signal
8. Response of the last thread to a cancellation request

exit Functions

Three functions terminate a program normally: _exit and _Exit, which return to the kernel immediately, and exit, which performs certain cleanup processing and then returns to the kernel.

```
#include <stdlib.h>

void exit(int status);

void _Exit(int status);

#include <unistd.h>

void _exit(int status);
```

the exit function has always performed a clean shutdown of the standard I/O library: the fclose function is called for all open streams. All three exit functions expect a single integer argument, which we call the exit status.

Most UNIX System shells provide a way to examine the exit status of a process. If (a) any of these functions is called without an exit status, (b) main does a return without a return value, or (c) the main function is not declared to return an integer, the exit status of the process is undefined.

Returning an integer value from the main function is equivalent to calling exit with the same value. Thus exit(0); is the same as return(0); from the main function.

```
#include    <stdio.h>

main()
{
    printf("hello, world\n");
}
```

Figure 7.1 Classic C program

When we compile and run the program in Figure 7.1, we see that the exit code is random. If we compile the same program on different systems, we are likely to get different exit codes, depending on the contents of the stack and register contents at the time that the main function returns:

```
$ gcc hello.c
$ ./a.out
hello, world
$ echo $?                          print the exit status
13
```

atexit Function

With ISO C, a process can register at least 32 functions that are automatically called by exit. These are called exit handlers and are registered by calling the atexit function.

```
#include <stdlib.h>

int atexit(void (*func)(void));
```

Returns: 0 if OK, nonzero on error

The exit function calls these functions in reverse order of their registration. Each function is called as many times as it was registered.

POSIX.1 extends the ISO C standard by specifying that any exit handlers installed will be cleared if the program calls any of the exec family of functions, figure below summarizes

how a C program is started and the various ways it can terminate.

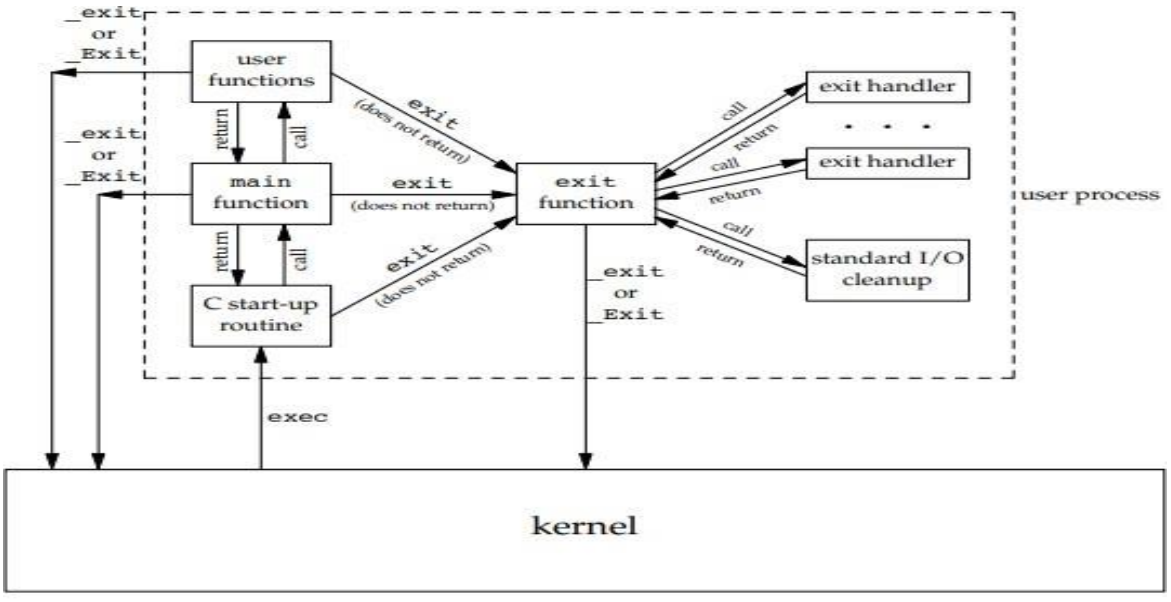


Figure 7.2 How a C program is started and how it terminates

The only way a program can be executed by the kernel is if one of the exec functions is called. The only way a process can voluntarily terminate is if _exit or _Exit is called, either explicitly or implicitly (by calling exit).

```
#include "apue.h"

static void my_exit1(void);
static void my_exit2(void);

int
main(void)
{
    if (atexit(my_exit2) != 0)
        err_sys("can't register my_exit2");

    if (atexit(my_exit1) != 0)
        err_sys("can't register my_exit1");
    if (atexit(my_exit1) != 0)
        err_sys("can't register my_exit1");

    printf("main is done\n");
    return(0);
}

static void
my_exit1(void)
{
    printf("first exit handler\n");
}

static void
my_exit2(void)
{
    printf("second exit handler\n");
}
```

Figure 7.3 Example of exit handlers

OUTPUT:

Executing the program in Figure 7.3 yields

```
$ ./a.out
main is done
first exit handler
first exit handler
second exit handler
```

An exit handler is called once for each time it is registered. In Figure 7.3, the first exit handler is registered twice, so it is called two times. Note that we don't call exit; instead, we return from main.

Command Line Arguments:

Key Points:

- Command-line arguments are passed to a program when it is executed.
- These arguments are accessible in the main() function via:

Prototype of main function:

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

- argc: Number of arguments, including the program's name.

- argv: Array of pointers to strings (arguments).

Example Program:

```
int main(int argc, char *argv[])
{
   int i;
   for (i = 0; i < argc; i++)                    // Echo all command-line args
      printf("argv[%d]: %s\n", i, argv[i]);
   exit(0);
}
```

OUTPUT

```
$ ./echoarg arg1 TEST foo
argv[0]: ./echoarg
argv[1]: arg1
argv[2]: TEST
argv[3]: foo
```

Environment List:

The diagram illustrates the **Process Environment** in UNIX systems, focusing on how environment variables are structured and accessed.

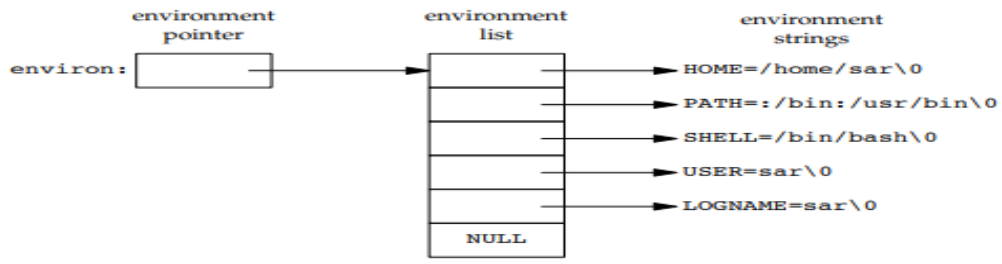


Figure 7.5 Environment consisting of five C character strings

Key Components:

1. **Environment Pointer (environ)**:

- A global pointer in C that points to the **environment list**.
- Defined as: extern char **environ;

2. **Environment List**:
    - An **array of pointers**, where each pointer points to a string in the form of name=value.
    - Ends with a **NULL** pointer, marking the end of the list.
3. **Environment Strings**:
    - Each string represents an environment variable in the name=value format.
    - Examples shown in the diagram:
        - HOME=/home/sar
        - PATH=/bin:/usr/bin
        - SHELL=/bin/bash
        - USER=sar
        - LOGNAME=sar
    - These strings are null-terminated (\0).

Description:

- **Accessing Environment Variables**:
    - Programs can use the global environ pointer or functions like getenv() to access individual variables.
    - Example:

```
char *home = getenv("HOME");
printf("Home directory: %s\n", home);
```

- **Structure**:
    - The **environment list** is essentially a table of pointers pointing to strings.
    - These strings contain information about the environment settings of the process, such as the shell being used, the user's home directory, and the system's path settings.

UNIX systems have provided a third argument to the main function that is the address of the environment list:

```
int main(int argc, char *argv[], char *envp[]);
```

Memory Layout of C Program :

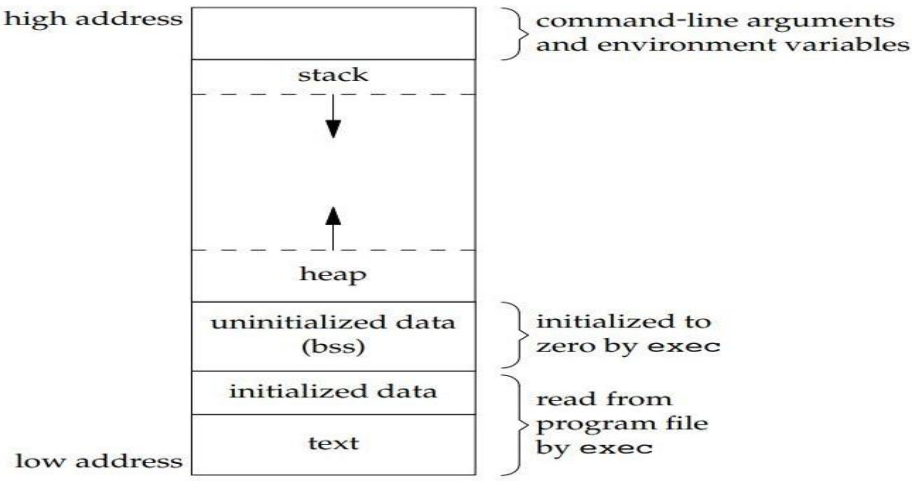


Figure 7.6 Typical memory arrangement

Initialized data segment, usually called simply the data segment, containing variables that are specifically initialized in the program. For example, the C declaration int maxcount = 99; appearing outside any function causes this variable to be stored in the initialized data segment with its initial value.

Uninitialized data segment, often called the ‘‘bss’’ segment, named after an ancient assembler operator that stood for ‘‘block started by symbol.’’ Data in this segment is initialized by the kernel to arithmetic 0 or null pointers before the program starts executing.

The C declaration long sum[1000];

Stack:

Automatic Variables:

- Variables declared inside functions (local variables) are stored in the stack.
- These variables are created when the function is called and destroyed when the function exits.

Function Call Information:

- When a function is called, the stack stores:
  - The return address (where to continue after the function completes).
  - The caller's environment (e.g., machine registers, local variables).
- This allows the program to resume the previous state after the function call finishes.

New Stack Frame:

- Each function call allocates a new "stack frame" to store:
  - Parameters passed to the function.
  - Local variables specific to the function.
  - Temporary data.
- Recursive functions benefit from the stack, as each call creates a new stack frame, preventing interference between instances.

Stack in Recursive Functions:

- When a recursive function calls itself, a separate stack frame is created for each call, maintaining independent variables for each level of recursion.

Dynamic Growth and Cleanup:

- The stack dynamically grows downward in memory when new functions are called.
- It shrinks as functions return, deallocating the memory associated with their stack frames.

Relationship to Other Memory Segments:

- The stack is situated above the heap in memory, and they grow toward each other. This arrangement helps maximize the use of available memory.

Heap, where dynamic memory allocation usually takes place.

Memory Allocation:

ISO C specifies three functions for memory allocation:

1. malloc, which allocates a specified number of bytes of memory. The initial value of the memory is indeterminate.

2. calloc, which allocates space for a specified number of objects of a specified size. The space is initialized to all 0 bits.

3. realloc, which increases or decreases the size of a previously allocated area. When the size increases, it may involve moving the previously allocated area somewhere else, to provide the additional room at the end. Also, when the size increases, the initial value of the space between the old contents and the end of the new area is indeterminate.

```
#include <stdlib.h>
void *malloc(size_t size);
void *calloc(size_t nobj, size_t size);
void *realloc(void *ptr, size_t newsize);
                        All three return: non-null pointer if OK, NULL on error
void free(void *ptr);
```

Alternate Memory Allocators

libmalloc SVR4-based systems, such as Solaris, include the libmalloc library, which provides a set of interfaces matching the ISO C memory allocation functions. The libmalloc library includes mallopt,afunction that allows a process to set certain variables that control the operation of the storage allocator. A function called mallinfo is also available to provide statistics on the memory allocator.

vmalloc Vo [1996] describes a memory allocator that allows processes to allocate memory using different techniques for different regions of memory. In addition to the functions specific to vmalloc, the library provides emulations of the ISO C memory allocation functions.

quick-fit Historically, the standard malloc algorithm used either a best-fit or a first-fit memory allocation strategy. Quick-fit is faster than either, but tends to use more memory. Weinstock and Wulf [1988] describe the algorithm, which is based on splitting up memory into buffers of various sizes and maintaining unused buffers on different free lists, depending on the buffer sizes. Most modern allocators are based on quick-fit.

jemalloc The jemalloc implementation of the malloc family of library functions is the default memory allocator in FreeBSD 8.0. It was designed to scale well when used with multithreaded applications running on multiprocessor systems. Evans [2006] describes the implementation and evaluates its performance.

alloca Function One additional function is also worth mentioning. The function alloca has the same calling sequence as malloc; however, instead of allocating memory from the heap, the memory is allocated from the stack frame of the current function. The advantage is that we don't have to free the space; it goes away automatically when the function returns. The alloca function increases the size of the stack frame. The disadvantage is that some systems can't support alloca, if it's impossible to increase the size of the stack frame after the function has been called. Nevertheless, many software packages use it, and implementations exist for a wide variety of systems.

setjmp and longjmp Functions:

In C, we can't goto a label that's in another function. Instead, we must use the setjmp and longjmp functions to perform this type of branching.

It consists of a main loop that reads lines from standard input and calls the function do_line to process each line. This function then calls get_token to fetch the next token from the input line. The first token of a line is assumed to be a command of some form, and a switch statement selects each command. For the single command shown, the function cmd_add is called. The skeleton in Figure 7.9 is typical for programs that read commands, determine the command type, and then call functions to process each command. Figure 7.10 shows what the stack could look like after cmd_add has been called.

```
#include "apue.h"

#define TOK_ADD    5

void    do_line(char *);
void    cmd_add(void);
int     get_token(void);

int
main(void)
{
    char    line[MAXLINE];

    while (fgets(line, MAXLINE, stdin) != NULL)
        do_line(line);
    exit(0);
}

char    *tok_ptr;       /* global pointer for get_token() */

void
do_line(char *ptr)      /* process one line of input */
{
    int     cmd;

    tok_ptr = ptr;
    while ((cmd = get_token()) > 0) {
        switch (cmd) {  /* one case for each command */
        case TOK_ADD:
                cmd_add();
                break;
        }
    }
}

void
cmd_add(void)
{
    int     token;

    token = get_token();
    /* rest of processing for this command */
}

int
get_token(void)
{
    /* fetch next token from line pointed to by tok_ptr */
}
```

Figure 7.9 Typical program skeleton for command processing

It consists of a main loop that reads lines from standard input and calls the function do_line to process each line. This function then calls get_token to fetch the next token from the input line. The first token of a line is assumed to be a command of some form, and a switch statement selects each command. For the single command shown, the function cmd_add is called. The skeleton in Figure 7.9 is typical for programs that read commands, determine the command type, and then call functions to process each command. Figure 7.10 shows what the stack could look like after cmd_add has been called.

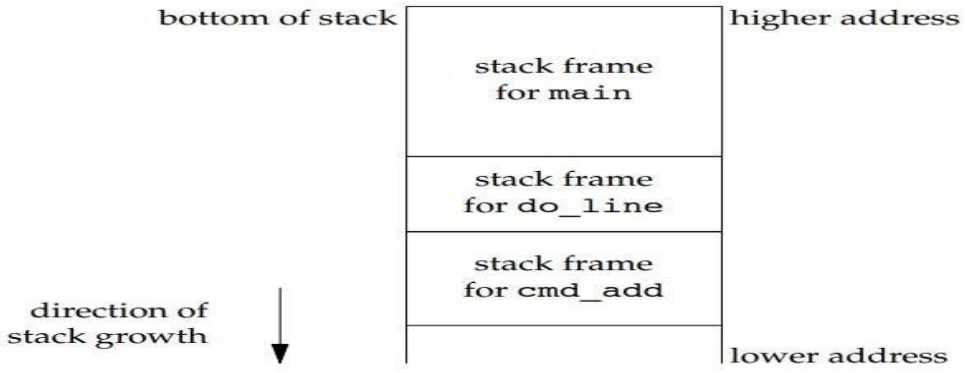


Figure 7.10 Stack frames after cmd_add has been called

The coding problem that's often encountered with programs like the one shown in Figure 7.9

is how to handle nonfatal errors. For example, if the cmd_add function encounters an error — say, an invalid number—it might want to print an error message, ignore the rest of the input line, and return to the main function to read the next input line. But when we're deeply nested numerous levels down from the main function, this is difficult to do in C. (In this example, the cmd_add function is only two levels down from main, but it's not uncommon to be five or more levels down from the point to which we want to return.) It becomes messy if we have to code each function with a special return value that tells it to return one level.

The solution to this problem is to use a nonlocal goto: the setjmp and longjmp functions. The adjective ''nonlocal'' indicates that we're not doing a normal C goto statement within a function; instead, we're branching back through the call frames to a function that is in the call path of the current function.

```
#include <setjmp.h>

int setjmp(jmp_buf env);

                Returns: 0 if called directly, nonzero if returning from a call to longjmp

void longjmp(jmp_buf env, int val);
```

Example Program of setjmp & longjmp

```
#include "apue.h"
#include <setjmp.h>

#define TOK_ADD     5

jmp_buf jmpbuffer;

int
main(void)
{
     char     line[MAXLINE];

     if (setjmp(jmpbuffer) != 0)
          printf("error");
     while (fgets(line, MAXLINE, stdin) != NULL)
          do_line(line);
     exit(0);
}

  . . .

void
cmd_add(void)
{
     int      token;

     token = get_token();
     if (token < 0)       /* an error has occurred */
          longjmp(jmpbuffer, 1);
     /* rest of processing for this command */
}
```

Figure 7.11 Example of setjmp and longjmp

Automatic, Registers and Local Variables:s

We've seen what the stack looks like after calling longjmp. The next question is, ''What are the states of the automatic variables and register variables in the main function?'' When we return to main as a result of the longjmp, do these variables have values corresponding to those when the setjmp was previously called (i.e., are their values rolled back), or are their values

left alone so that their values are whatever they were when do_line was called (which caused cmd_add to be called, which caused longjmp to be called)? Unfortunately, the answer is ‘‘It depends.’’ Most implementations do not try to roll back these automatic variables and register variables, but the standards say only that their values are indeterminate. If you have an automatic variable that you don’t want rolled back, define it with the volatile attribute. Variables that are declared as global or static are left alone when longjmp is executed.

Example : The program in Figure 7.13 demonstrates the different behavior that can be seen with automatic, global, register, static, and volatile variables after calling longjmp.

```
#include "apue.h"
#include <setjmp.h>

static void f1(int, int, int, int);
static void f2(void);

static jmp_buf  jmpbuffer;
static int      globval;

int
main(void)
{
    int             autoval;
    register int    regival;
    volatile int    volaval;
    static int      statval;

    globval = 1; autoval = 2; regival = 3; volaval = 4; statval = 5;

    if (setjmp(jmpbuffer) != 0) {
        printf("after longjmp:\n");
        printf("globval = %d, autoval = %d, regival = %d,"
            " volaval = %d, statval = %d\n",
            globval, autoval, regival, volaval, statval);
        exit(0);
    }
    /*
     * Change variables after setjmp, but before longjmp.
     */
    globval = 95; autoval = 96; regival = 97; volaval = 98;
    statval = 99;

    f1(autoval, regival, volaval, statval); /* never returns */
    exit(0);
}

static void
f1(int i, int j, int k, int l)
{
    printf("in f1():\n");
    printf("globval = %d, autoval = %d, regival = %d,"
        " volaval = %d, statval = %d\n", globval, i, j, k, l);
    f2();
}

static void
f2(void)
{
    longjmp(jmpbuffer, 1);
}
```

Figure 7.13 Effect of longjmp on various types of variables

If we compile and test the program in Figure 7.13, with and without compiler optimizations, the results are different:

```
$ gcc testjmp.c                compile without any optimization
$ ./a.out
in f1():
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
after longjmp:
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
$ gcc -O testjmp.c             compile with full optimization
$ ./a.out
in f1():
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
after longjmp:
globval = 95, autoval = 2, regival = 3, volaval = 98, statval = 99
```

getrlimit and setrlimit Functions:

Every process has a set of resource limits, some of which can be queried and changed by the getrlimit and setrlimit functions.

```
#include <sys/resource.h>
int getrlimit(int resource, struct rlimit *rlptr);
int setrlimit(int resource, const struct rlimit *rlptr);
                                        Both return: 0 if OK, –1 on error
```

Each call to these two functions specifies a single resource and a pointer to the following structure:

```
struct rlimit
{
        rlim_t      rlim_cur;       /* soft limit: current limit */
        rlim_t      rlim_max;       /* hard limit: maximum value for rlim_cur */
};
```

Three rules govern the changing of the resource limits.

1. A process can change its soft limit to a value less than or equal to its hard limit.

2. A process can lower its hard limit to a value greater than or equal to its soft limit. This lowering of the hard limit is irreversible for normal users.

3. Only a superuser process can raise a hard limit.

An infinite limit is specified by the constant RLIM_INFINITY. The resource argument takes on one of the following values. Figure 7.15 shows which limits are defined by the Single UNIX Specification and supported by each implementation.

RLIMIT_AS The maximum size in bytes of a process's total available memory.

RLIMIT_CORE The maximum size in bytes of a core file. A limit of 0 prevents the creation of a core file.

RLIMIT_CPU The maximum amount of CPU time in seconds. When the soft limit is exceeded, the SIGXCPU signal is sent to the process.

RLIMIT_DATA The maximum size in bytes of the data segment: the sum of the initialized data, uninitialized data, and heap.

RLIMIT_FSIZE The maximum size in bytes of a file that may be created. When the soft limit is exceeded, the process is sent the SIGXFSZ signal.

RLIMIT_MEMLOCK The maximum amount of memory in bytes that a process can lock into memory using mlock(2).

RLIMIT_MSGQUEUE The maximum amount of memory in bytes that a process can allocate for POSIX message queues.

RLIMIT_NICE The limit to which a process's nice value can be raised to affect its scheduling priority.

RLIMIT_NOFILE The maximum number of open files per process. Changing this limit affects the value returned by the sysconf function for its _SC_OPEN_MAX argument.

RLIMIT_NPROC The maximum number of child processes per real user ID.

RLIMIT_NPTS The maximum number of pseudo terminals that a user can have open at one time.

RLIMIT_RSS Maximum resident set size (RSS) in bytes. If available physical memory is low, the kernel takes memory from processes that exceed their RSS.

RLIMIT_SBSIZE The maximum size in bytes of socket buffers that a user can consume at any given time. RLIMIT_SIGPENDING The maximum number of signals that can be queued for a process.

RLIMIT_STACK The maximum size in bytes of the stack.

RLIMIT_SWAP The maximum amount of swap space in bytes that a user can consume.

RLIMIT_VMEM This is a synonym for RLIMIT_AS.

Example The program prints out the current soft limit and hard limit for all the resource limits supported on the system.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/resource.h>
#include <string.h>
#include <errno.h>
#define doit(name) pr_limits(#name, name)
static void pr_limits(char *name, int resource);
int main(void)
{
#ifdef RLIMIT_AS
   doit(RLIMIT_AS);
#endif
   doit(RLIMIT_CORE);
```

```
    doit(RLIMIT_CPU);
doit(RLIMIT_DATA);
doit(RLIMIT_FSIZE);
 #ifdef RLIMIT_MEMLOCK
    doit(RLIMIT_MEMLOCK);
 #endif
 #ifdef RLIMIT_MSGQUEUE
    doit(RLIMIT_MSGQUEUE);
 #endif
 #ifdef RLIMIT_NICE
    doit(RLIMIT_NICE);
 #endif
    doit(RLIMIT_NOFILE);
 #ifdef RLIMIT_NPROC
    doit(RLIMIT_NPROC);
 #endif
 #ifdef RLIMIT_NPTS
    doit(RLIMIT_NPTS);
 #endif
 #ifdef RLIMIT_RSS
    doit(RLIMIT_RSS);
 #endif
 #ifdef RLIMIT_SBSIZE
    doit(RLIMIT_SBSIZE);
 #endif
 #ifdef RLIMIT_SIGPENDING
    doit(RLIMIT_SIGPENDING);
 #endif
    doit(RLIMIT_STACK);
 #ifdef RLIMIT_SWAP
    doit(RLIMIT_SWAP);
```

```
#endif
#ifdef RLIMIT_VMEM

    doit(RLIMIT_VMEM);
#endif
    exit(0);
}
static void pr_limits(char *name, int resource)
{
    struct rlimit limit;
    unsigned long long lim;
    if (getrlimit(resource, &limit) < 0) {
        fprintf(stderr, "getrlimit error for %s: %s\n", name, strerror(errno));
        return;
    }

    printf("%-14s ", name);
    if (limit.rlim_cur == RLIM_INFINITY) {
        printf("(infinite) ");
    } else {
        lim = limit.rlim_cur;
        printf("%10llu ", lim);
    }

    if (limit.rlim_max == RLIM_INFINITY) {
        printf("(infinite)");
    } else {
        lim = limit.rlim_max;
        printf("%10llu", lim);
    }
    putchar('\n');
}
```

OUTPUT:

```
RLIMIT_CORE          (infinite) (infinite)
RLIMIT_CPU           3600     (infinite)
RLIMIT_DATA          2147483647 2147483647
RLIMIT_FSIZE         18446744073709551615 (infinite)
RLIMIT_NOFILE        1024       4096
RLIMIT_STACK         8388608    (infinite)
```