UNIT-5—File I/O-File I/O Basics

File System

UNIX File System (UFS):

UFS is used by many UNIX and UNIX-like operating systems.
Proprietary UNIX systems (e.g., SunOS/Solaris, HP-UX, Tru64 UNIX) have adapted UFS with their own extensions.

Linux File System Support:
Linux supports a wide range of file systems, including:

- Older file systems (MINIX, MS-DOS, ext2).
- Newer journaling file systems (ext4, JFS, ReiserFS).
- Specialized file systems like the Cryptographic File System (CFS) and the virtual file system /proc.

Mac OS X File System Support:
Mac OS X supports several file systems, such as:

- HFS+ (Mac OS file system).
- UFS (BSD standard).
- NFS, SMB, AFP, ISO 9660, UDF (for different purposes like file sharing and CD-ROM support).

Distributed File Systems:
Some file systems (e.g., NFS, AFS, DFS) are distributed, allowing users to access files on remote machines as if they were local.

POSIX and Directory Systems:
POSIX (a standard for UNIX-like systems) does not define directories or hierarchical file systems.

UNIT-5—File I/O-File I/O Basics

File Systems:

Hierarchical File System:

Files are organized in a tree-like structure with a root directory.
Directories are non-leaf nodes, and files are leaf nodes.
Multiple directory entries can refer to the same file, making it a directed acyclic graph (DAG).

i-node:

Each file has an associated i-node that stores file attributes and addresses of file blocks.
The i-node also includes a pointer to another block of addresses.

Directories:

Directories are special files containing a list of directory entries.
Each entry includes a file name and its associated i-node number.

File Naming and Path Names:

Files have names, and path names are used to locate files.
Path names can be absolute (starting with a separator) or relative (starting from the current working directory).

Path Name Structure:

Absolute path names begin with a separator (e.g., /).
Nonterminal names in the path refer to directories, and the terminal name refers to a file or directory.

UNIT-5—File I/O-File I/O Basics

File System

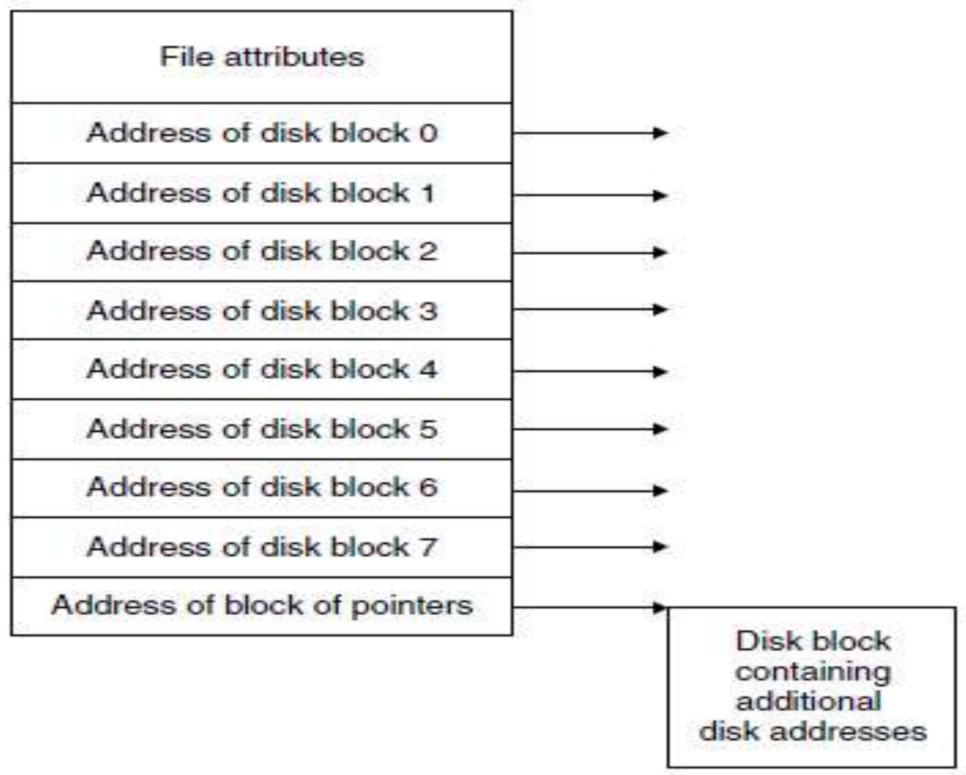


Figure 8.1 Sample i-node

UNIT-5—File I/O-File I/O Basics

File System

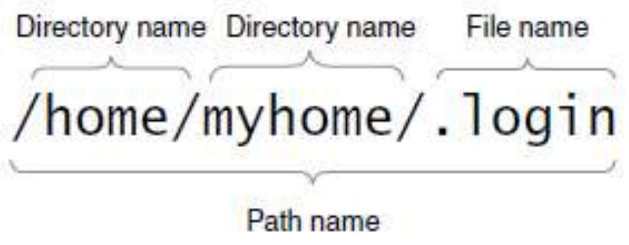


Figure 8.2 Path name components

UNIT-5—File I/O—File I/O Basics

Special Files

- Directories contain only a list of other files (the contents of a directory). They are marked with a d as the first letter of the permissions field when viewed with the ls –l command:
  ```
  drwxr-xr-x /
  ```

- Symbolic links are references to other files. Such a reference is stored as a textual representation of the file's path. Symbolic links are indicated by an l in the permissions string:
  ```
  lrwxrwxrwx termcap -> /usr/share/misc/termcap
  ```

- Named pipes enable different processes to communicate and can exist anywhere in the file system. Named pipes are created with the command mkfifo, as in mkfifo mypipe. They are indicated by a p as the first letter of the permissions string:
  ```
  prw-rw---- mypipe
  ```

- Sockets allow communication between two processes running on the same machine. They are indicated by an s as the first letter of the permissions string:
  ```
  srwxrwxrwx X0
  ```

- Device files are used to apply access rights and to direct operations on the files to the appropriate device drivers. Character devices provide only a serial stream of input or output (indicated by a c as the first letter of the permissions string):
  ```
  crw------- /dev/kbd
  ```

- Block devices are randomly accessible (indicated by a b):
  ```
  brw-rw---- /dev/hda
  ```

UNIT-5—File I/O—File I/O Interfaces

File I/O in C:

- File I/O in C includes functions defined in <stdio.h>.
- Security of I/O operations depends on the compiler, operating system, and file system.
- Older libraries are more prone to security flaws than newer ones.

Byte Input Functions:
Functions for reading byte characters or byte strings: fgetc(), fgets(), getc(), getchar(), fscanf(), scanf(), vfscanf(), and vscanf().

Byte Output Functions:
Functions for writing byte characters or byte strings: fputc(), fputs(), putc(), putchar(), fprintf(), vfprintf(), vprintf().

Byte I/O Functions:
The ungetc() function and all byte input/output functions form the set of byte I/O functions.

Wide-Character Input Functions:
Functions for reading wide characters or wide strings: fgetwc(), fgetws(), getwc(), getwchar(), fwscanf(), wscanf(), vfwscanf(), and vwscanf().

Wide-Character Output Functions:
Functions for writing wide characters or wide strings: fputwc(), fputws(), putwc(), putwchar(), fwprintf(), wprintf(), vfwprintf(), and vwprintf().

UNIT-5—File I/O—File I/O Interfaces

Data Streams

- stdin: standard input (for reading conventional input)
- stdout: standard output (for writing conventional output)
- stderr: standard error (for writing diagnostic output)

UNIT-5—File I/O—File I/O Interfaces

Opening and Closing Files

fopen() Function:

Opens a file and associates a stream with it.
Takes two arguments: the filename and the mode (how the file should be opened).
The mode argument defines how the file will be accessed (e.g., read, write, append).

```
FILE *fopen(
const char * restrict filename,
const char * restrict mode
);
```

UNIT-5—File I/O—File I/O Interfaces

Opening and Closing Files

The argument mode points to a string. If the string is valid, the file is open in the indicated mode; otherwise, the behavior is undefined.
C99 supported the following modes:

- r: open text file for reading
- w: truncate to zero length or create text file for writing
- a: append; open or create text file for writing at end-of-file
- rb: open binary file for reading
- wb: truncate to zero length or create binary file for writing
- ab: append; open or create binary file for writing at end-of-file
- r+: open text file for update (reading and writing)
- w+: truncate to zero length or create text file for update
- a+: append; open or create text file for update, writing at end-of-file
- r+b or rb+: open binary file for update (reading and writing)
- w+b or wb+: truncate to zero length or create binary file for update
- a+b or ab+: append; open or create binary file for update, writing at end-of-file

UNIT-5—File I/O—File I/O Interfaces

Opening and Closing Files

Opening a file with exclusive mode (x as the last character in the mode argument) fails if the file already exists or cannot be created. Otherwise, the file is created with exclusive (also known as nonshared) access to the extent that the underlying system supports exclusive access.

- wx: create exclusive text file for writing
- wbx: create exclusive binary file for writing
- w+x: create exclusive text file for update
- w+bx or wb+x: create exclusive binary file for update

UNIT-5—File I/O—File I/O Interfaces

Closing a File (fclose()):

- The fclose() function disassociates the file from its stream.
- Unwritten data is flushed to the file, and unread data is discarded.
- The value of the FILE pointer becomes indeterminate after closing the file.

File Persistence:

- Closed files may be reopened later for reading or modification.
- On normal program termination (e.g., main() returns or exit() is called), all open files are closed and buffered data is flushed.
- Abnormal program termination (e.g., abort()) might not flush all files, though Linux ensures buffered data is flushed even in such cases.

UNIT-5—File I/O—File I/O Interfaces

POSIX

```
int open(const char *path, int oflag, ...);
int close(int fildes);
```

- Open() function creates an open file description refers to a file and file descriptor that refers to that open file description. The file descriptor is used by other I/O functions such as close().

- A file descriptor is a per-process, unique, nonnegative integer used to identify an open file for the purpose of file access. The value of a file descriptor is from 0 to OPEN_MAX.
- A process can have no more than OPEN_MAX file descriptors open simultaneously.
- A common exploit is to exhaust the number of available file descriptors to launch a denial-of-service (DoS) attack.
- An open file description is a record of how a process or group of processes is accessing a file. A file descriptor is just an identifier or handle;
- Figure 8.3 shows an example of two separate processes that are opening the same file or i-node. The information

stored in the open file description is different for each process, whereas the information stored in the i-node is associated with the file and is the same for each process.

UNIT-5—File I/O—File I/O Interfaces

POSIX

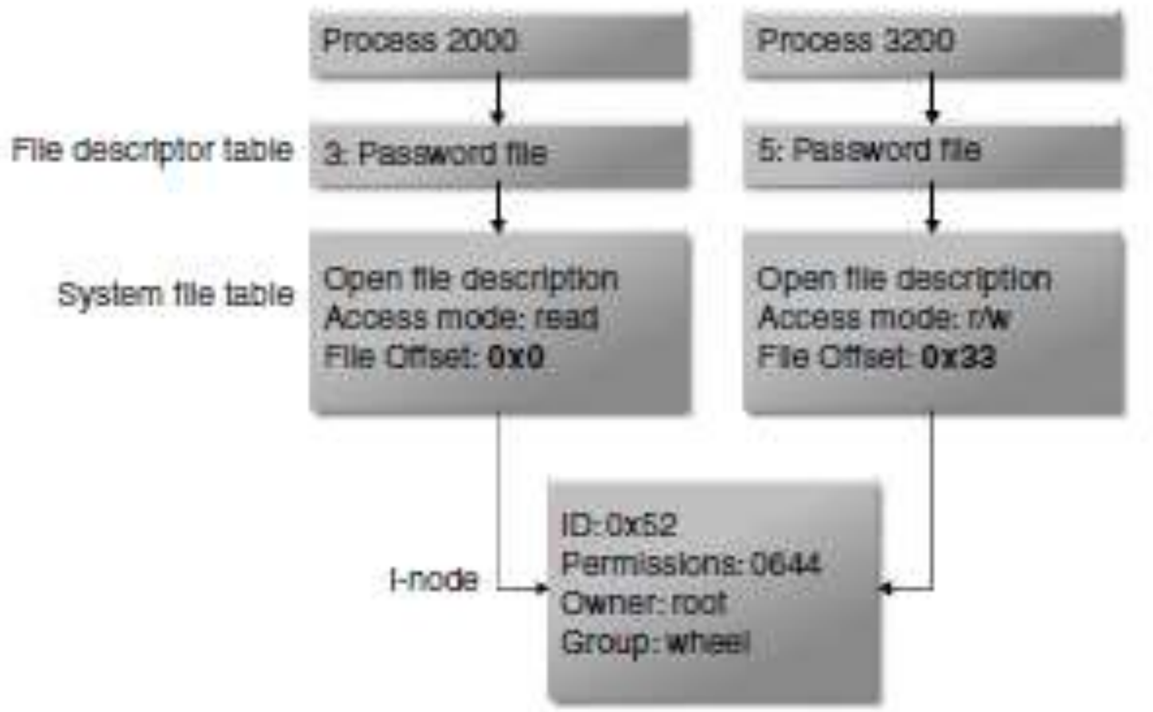


Figure 8.3 Independent opens of the same file

UNIT-5—File I/O—File I/O Interfaces

POSIX

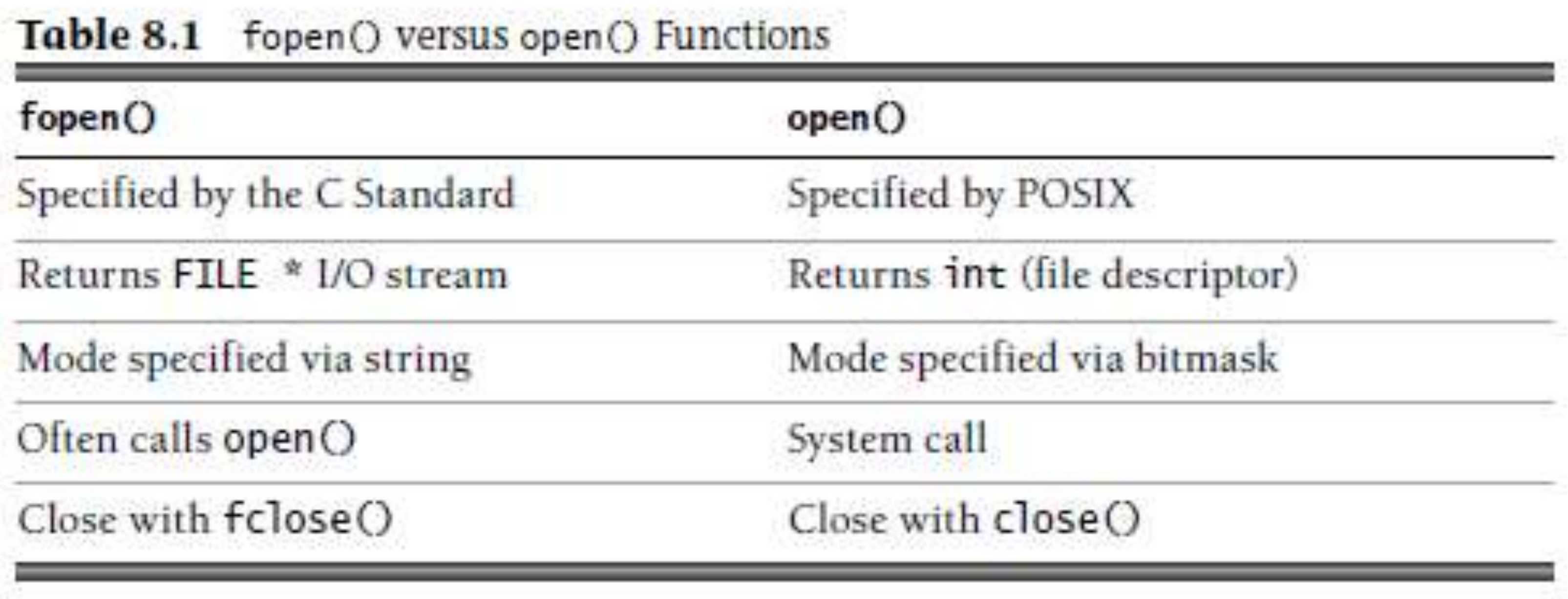


Table 8.1 fopen() versus open() Functions

fopen()	open()
Specified by the C Standard	Specified by POSIX
Returns FILE * I/O stream	Returns int (file descriptor)
Mode specified via string	Mode specified via bitmask
Often calls open()	System call
Close with fclose()	Close with close()

UNIT-5—File I/O—File I/O Interfaces

File I/O in C++

In C++, the I/O system is similar to C but with different syntax. The C++ <iostream> library includes <cstdio>, so C I/O functions are also available in C++. However, C++ uses specific classes for file handling:

ifstream: For input (reading from files).
ofstream: For output (writing to files).
iofstream: For both input and output (reading and writing to files).
wifstream, wofstream, wiofstream, wfstream: For wide-character I/O using wchar_t.

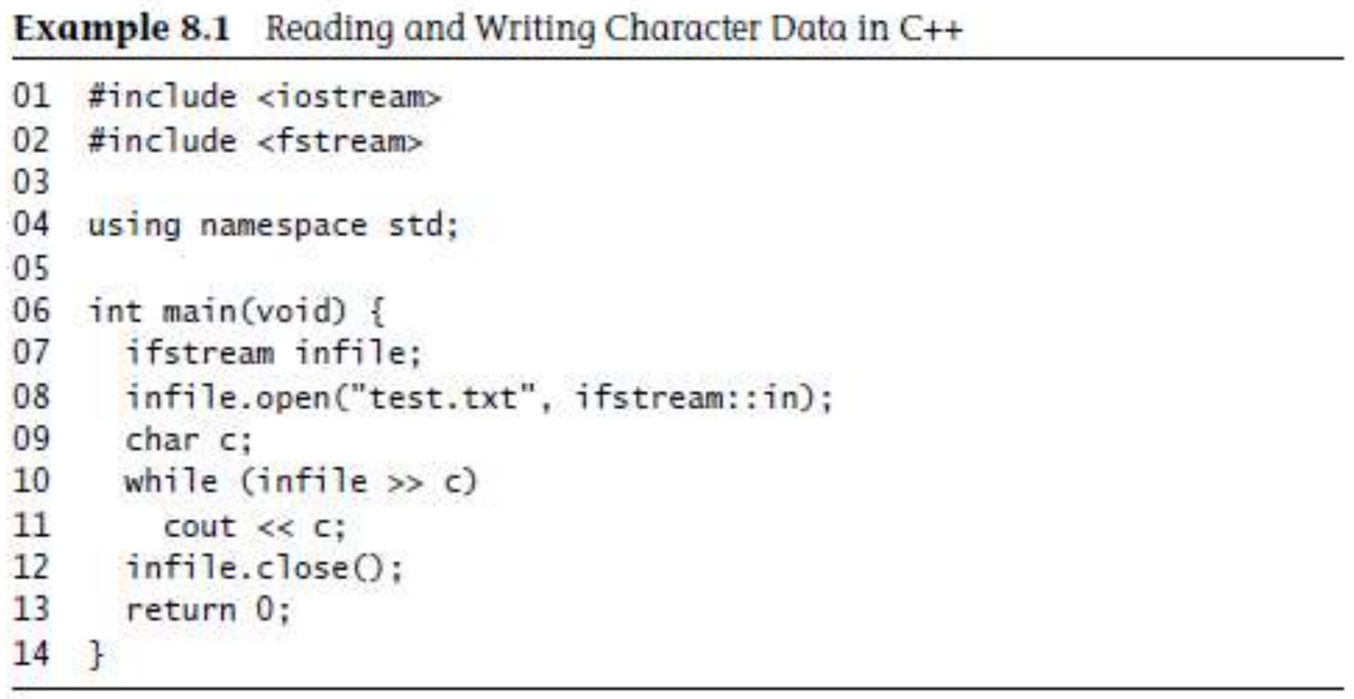
Additionally, C++ provides predefined streams for handling input and output:

cin: Standard input (replaces stdin).
cout: Standard output (replaces stdout).
cerr: Unbuffered standard error (replaces stderr).
clog: Buffered standard error (useful for logging).

For wide-character streams, use wcin, wcout, wcerr, and wclog.

UNIT-5—File I/O—File I/O Interfaces

File I/O in C++



Example 8.1 Reading and Writing Character Data in C++

```
#include <iostream>
#include <fstream>

using namespace std;

int main(void) {
  ifstream infile;
  infile.open("test.txt", ifstream::in);
  char c;
  while (infile >> c)
    cout << c;
  infile.close();
  return 0;
}
```

Example 8.1 is a simple C++ program that reads character data from a file named test.txt and writes it to standard output.

UNIT-5—File I/O—Access Control

Imagine a large organization with many employees using a powerful computer system, a UNIX-based system, to manage their tasks. Each user in this system has a unique identity, so no one can accidentally or maliciously interfere with someone else’s work. The system is designed to keep things secure while allowing users to do their jobs.

The User Logs In: All the login details of the user are stored in a special file called /etc/passwd , which stores user credentials.

The System Checks user’s Identity: The system now knows user’s user ID (UID) and group ID (GID) from the /etc/passwd file. This helps the system understand what resources user can access. If the user belongs to a group, say the "Marketing" group, (the superuser with UID 0, who has access to everything).

Access Control: user needs to access a file to work on a report. Permissions in UNIX are set to control who can read, write, or execute a file. The file might have different permissions for the owner, the group (Marketing), and everyone else.

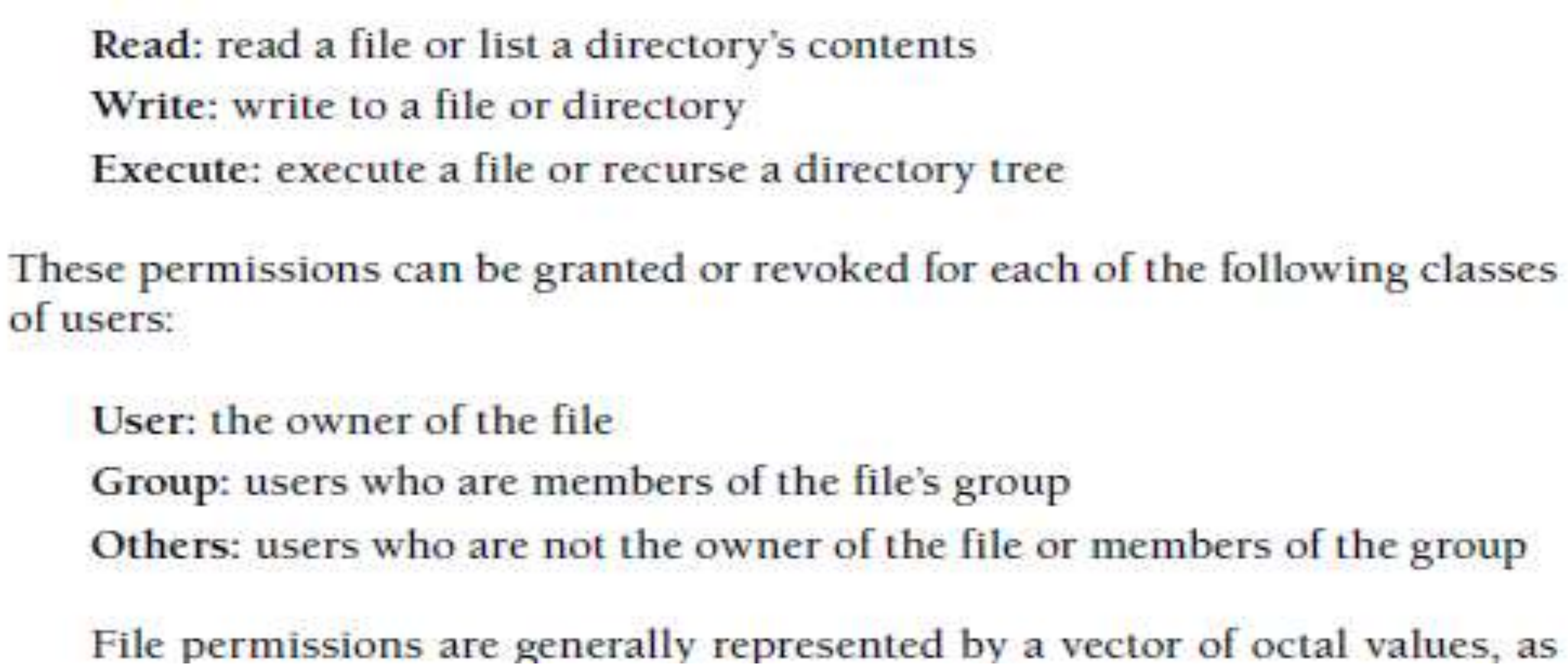
Preventing Unauthorized Access: Now, imagine that user’s coworker, user1, tries to access a file that belongs to user. The system will check the permissions of the file and see that user1 doesn’t have the proper permission to modify it. Therefore, user1 won’t be able to make any changes to user’s file, preventing accidental or malicious data corruption.

Privileges vs Permissions: These privileges are granted to special users, like the root user, who has full access to everything on the system.

UNIT-5—File I/O—Access Control

Unix File Permissions:

- Each file in a UNIX file system has an owner (UID) and a group (GID). Ownership determines which users and processes can access files.
- Only the owner of the file or root can change permissions. This privilege cannot be delegated or shared. The permissions are



Read: read a file or list a directory's contents

Write: write to a file or directory

Execute: execute a file or recurse a directory tree

These permissions can be granted or revoked for each of the following classes of users:

User: the owner of the file

Group: users who are members of the file's group

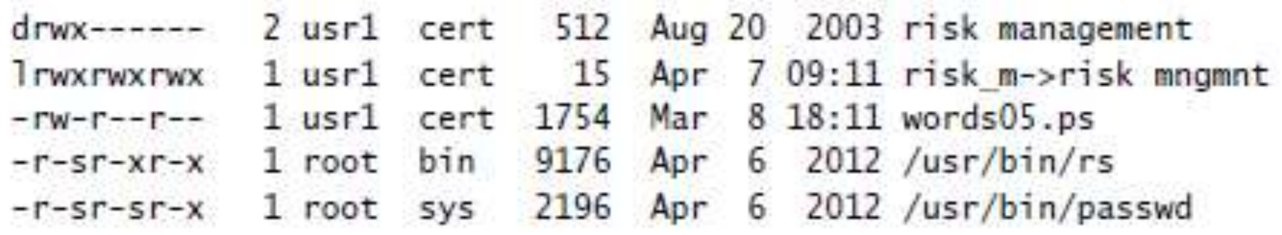
Others: users who are not the owner of the file or members of the group

File permissions are generally represented by a vector of octal values, as

UNIT-5—File I/O—Access Control

Unix File Permissions:

ls -l



```
drwx------   2 usr1  cert    512  Aug 20  2003 risk management
lrwxrwxrwx   1 usr1  cert     15  Apr  7 09:11 risk_m->risk mngmnt
-rw-r--r--   1 usr1  cert   1754  Mar  8 18:11 words05.ps
-r-sr-xr-x   1 root  bin    9176  Apr  6  2012 /usr/bin/rs
-r-sr-sr-x   1 root  sys    2196  Apr  6  2012 /usr/bin/passwd
```

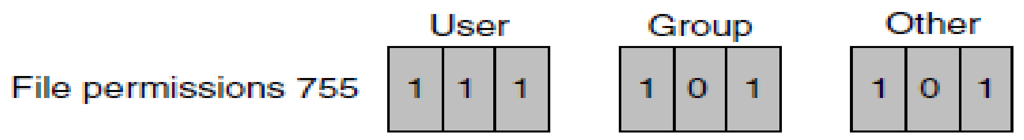


Figure 8.4 File permission represented by vector of octal values

UNIT-5—File I/O—Access Control

Process Privileges:

1. **Real User ID (RUID):** This is the ID of the user who started the process. It typically reflects the identity of the user that invoked the process.
If the process is a child of another process, the RUID of the child process will be the same as the RUID of the parent process, unless the RUID is changed by a special system call.

2**. Effective User ID (EUID):**
The EUID determines which permissions are used by the operating system when the process accesses resources (like files).
If a program is executed with the set-user-ID (SUID) bit set (which is a special permission on executable files), the EUID of the process will be set to the user ID (UID) of the file owner, not necessarily the user who launched the process. This allows the program to run with elevated privileges, according to the file owner's permissions, rather than the user's permissions.

3. **Saved Set-User-ID (SSUID):**
The SSUID is a backup of the EUID that is stored when the program executes with special privileges (like the SUID bit). It allows the program to restore the EUID to the original value after completing certain privileged tasks.

4. **Effective Group ID (EGID):**
The EGID is used by the kernel when checking a process's permissions to access group-related resources. Like the EUID, the EGID is used when determining access permissions to resources owned by the group. The EGID can be modified in a way similar to the EUID if the set-group-ID (SGID) bit is set on a program.

6. **Saved Set-Group-ID (SSGID):**
This is the backup of the EGID and is used when the process executes with SGID privileges.

UNIT-5—File I/O—Access Control

Process Privileges:

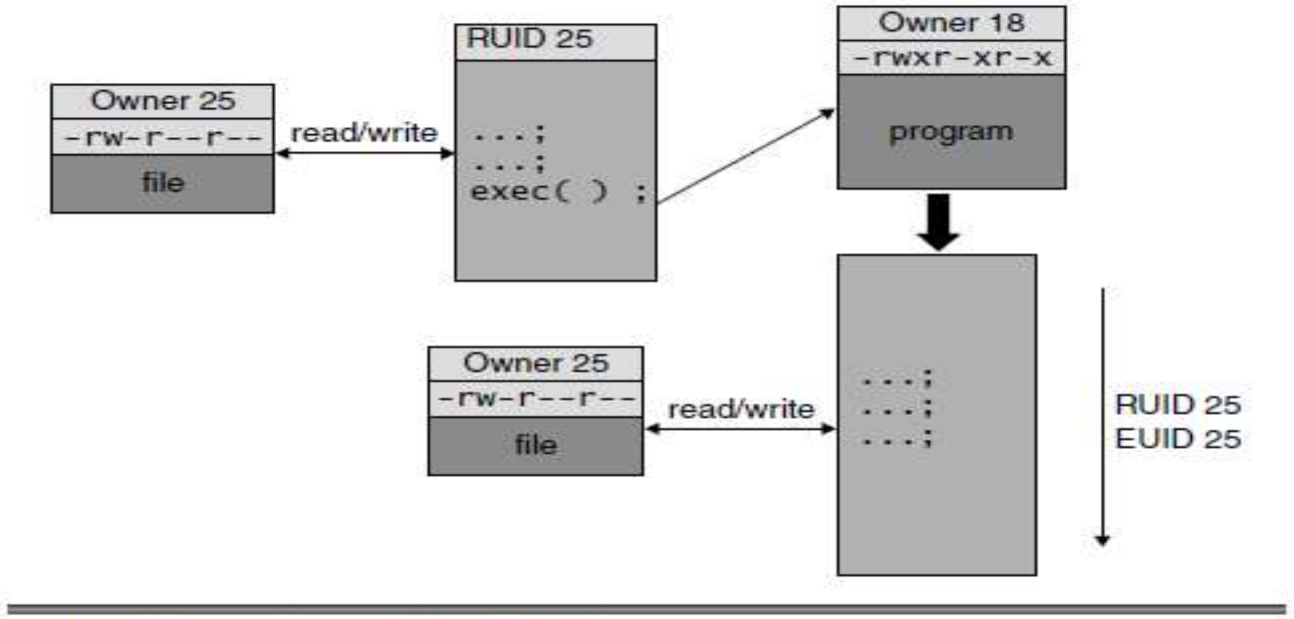


Figure 8.5 Executing a non-setuid program

In the example shown in Figure 8.5, file is owned by UID 25. A process running with an RUID of 25 executes the process image stored in the file program. The program file is owned by UID 18. However, when the program executes, it executes with the permissions of the parent process. Consequently, the program runs with an RUID and EUID of 25 and is able to access files owned by that UID.

UNIT-5—File I/O—Access Control

Process Privileges:

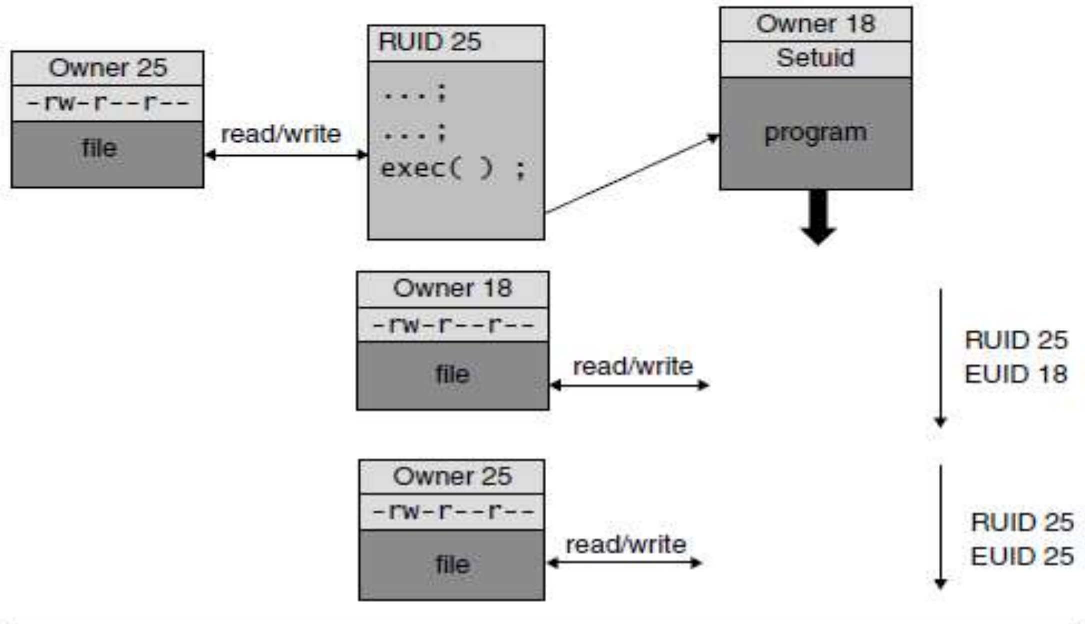


Figure 8.6 Executing a setuid program

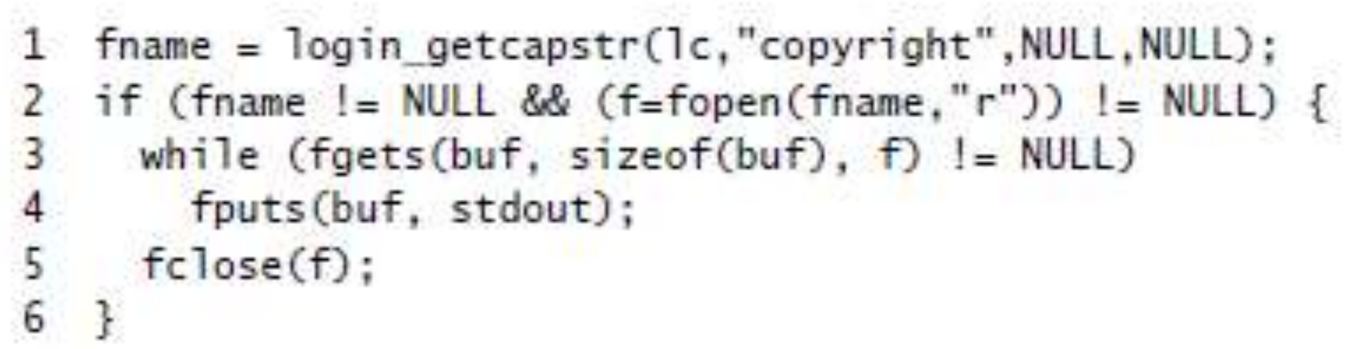
```
sudo chmod u+s setuid_test
```

```
-rwsr-xr-x 1 root root 12345 Dec 30 12:00 setuid_test
```

UNIT-5—File I/O—Access Control

Changing Privileges:

If the process is running with elevated privileges and accessing files in shared directories or user directories, there is chance that the program might be exploited to perform an operation on a file for which the user of your program does not have appropriate privileges.



```
fname = login_getcapstr(lc,"copyright",NULL,NULL);
if (fname != NULL && (f=fopen(fname,"r")) != NULL) {
  while (fgets(buf, sizeof(buf), f) != NULL)
    fputs(buf, stdout);
  fclose(f);
}
```

This vulnearability allows an attacker to read any file system by specifying the configuration optin in the user's ~/.login_conf file:

copyright=/etc/shadow

Dropping privileges is an effective mitigation strategy can be incorporated however complete risk cannot be eliminated.

UNIT-5—File I/O—Access Control

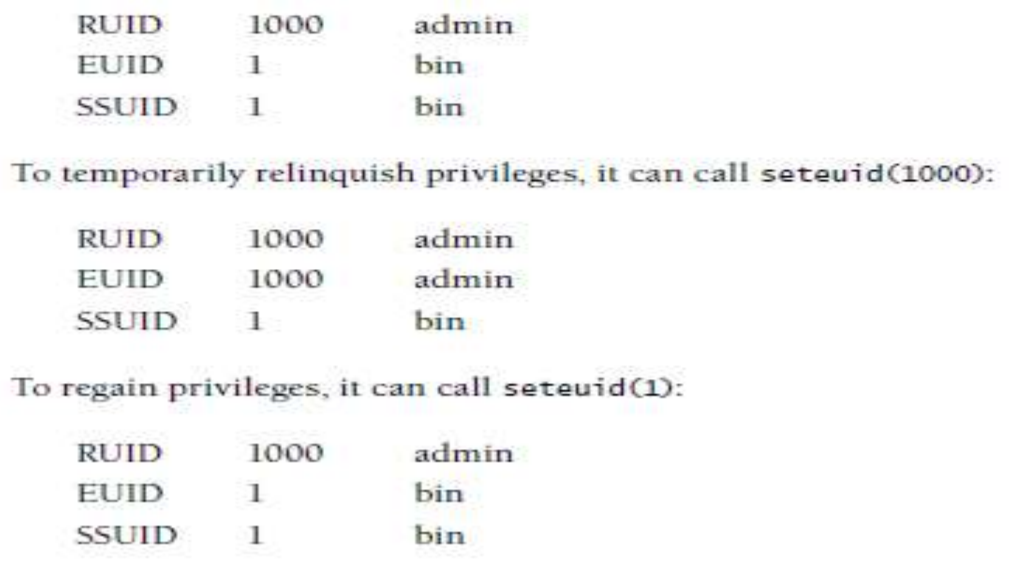
Privilege Management Functions:

In general, a process is allowed to change its EUID to its RUID (the user who started the program) and the saved set-user-ID, which allows a process to toggle effective privileges. Processes with root privileges can, of course, do anything.

The seteuid() function changes the EUID associated with a process and has the following signature:

int seteuid(uid_t euid);

Unprivileged user processes can only set the EUID to the RUID or the SSUID. Processes running with root privileges can set the EUID to any value. The setegid() function behaves the same for groups.



RUID	1000	admin
EUID	1	bin
SSUID	1	bin

To temporarily relinquish privileges, it can call seteuid(1000):

RUID	1000	admin
EUID	1000	admin
SSUID	1	bin

To regain privileges, it can call seteuid(1):

RUID	1000	admin
EUID	1	bin
SSUID	1	bin

UNIT-5—File I/O—Access Control

Privilege Management Functions:

The setresuid() function is used to explicitly set the RUID, EUID, and SSUID:

```
int setresuid(
uid_t ruid, uid_t euid, uid_t siud
);
```

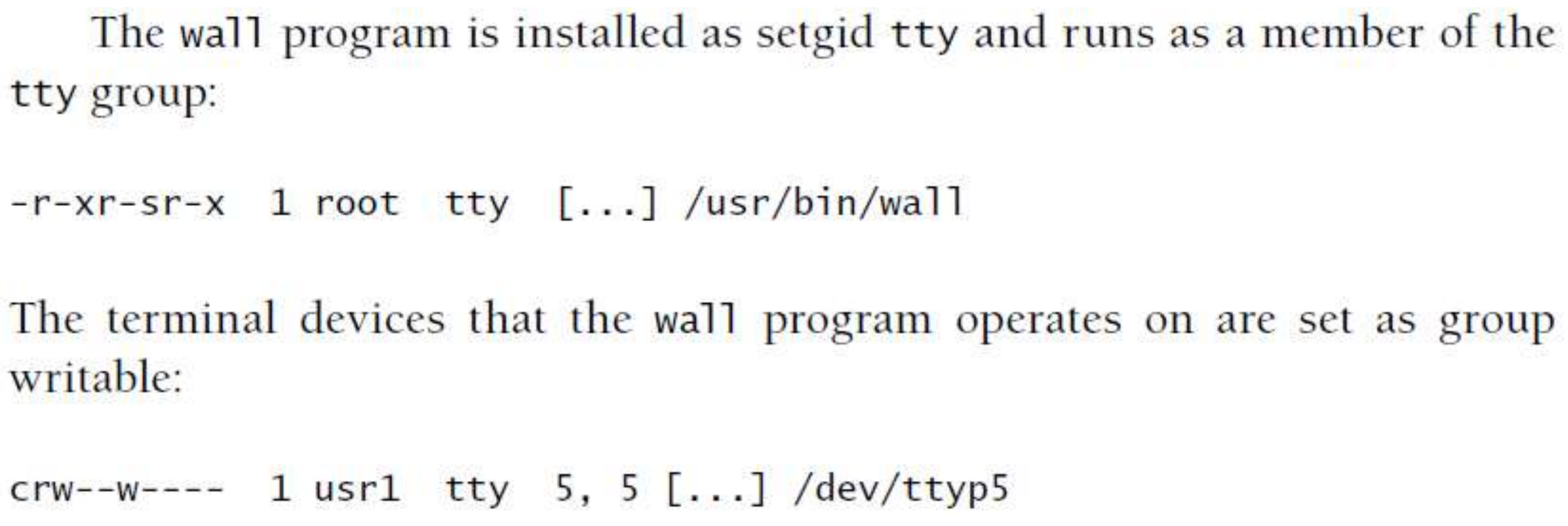
The setresuid() function sets all three UIDs and returns 0 if successful or −1 if an error occurs. If any of the ruid, euid, or siud arguments is -1.

UNIT-5—File I/O—Access Control

Managing Privileges:

The best example of setgid program is Wall Program. The wall command is used in Unix-based operating systems (like Linux) to send a message to all users currently logged in.

Regular (nonroot) user cannot write directly to another user’s terminal device, as it would allow users to spy on each other or to interfere with one another’s terminal sessions



The `wall` program is installed as setgid `tty` and runs as a member of the `tty` group:

```
-r-xr-sr-x  1 root  tty  [...] /usr/bin/wall
```

The terminal devices that the `wall` program operates on are set as group writable:

```
crw--w----  1 usr1  tty  5, 5 [...] /dev/ttyp5
```

UNIT-5—File I/O—Access Control

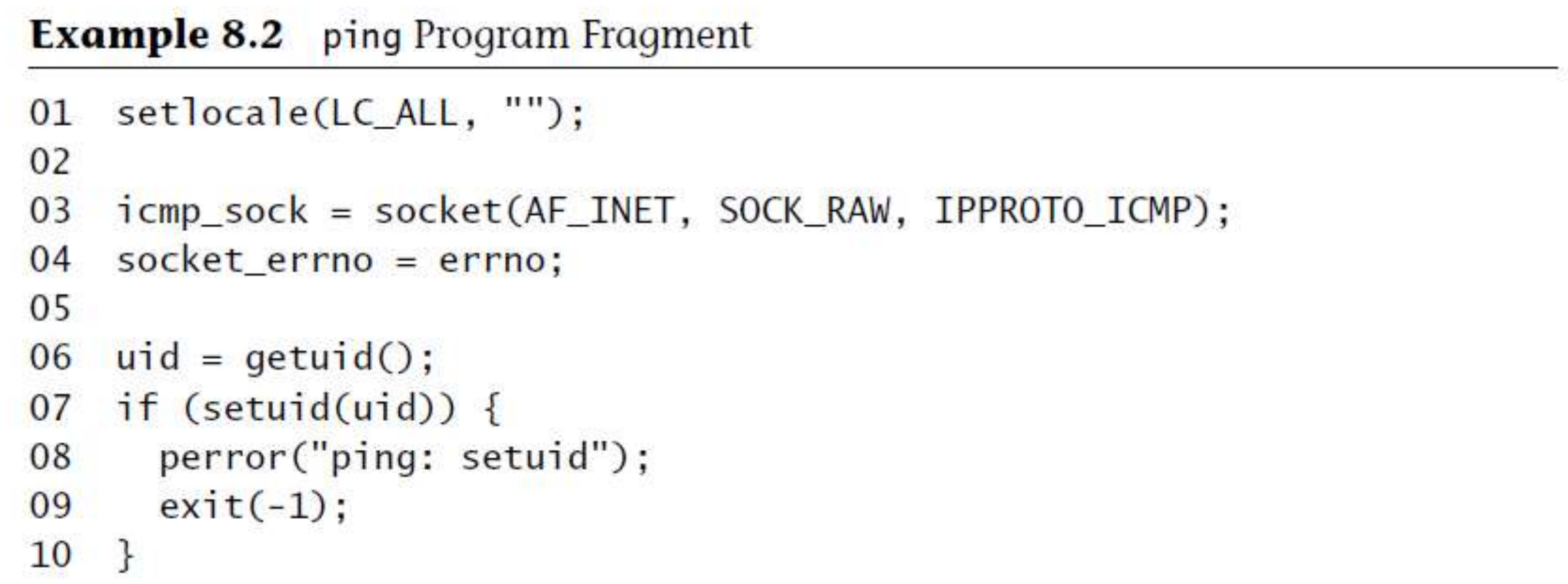
Managing Privileges:

```
$ ls -l /usr/bin/passwd
-r-sr-xr-x  1 root  bin  [...] /usr/bin/passwd
```

```
$ ls -l /sbin/ping
-r-sr-xr-x  1 root  bin  [...] /sbin/ping
```

UNIT-5—File I/O—Access Control

Managing Privileges:

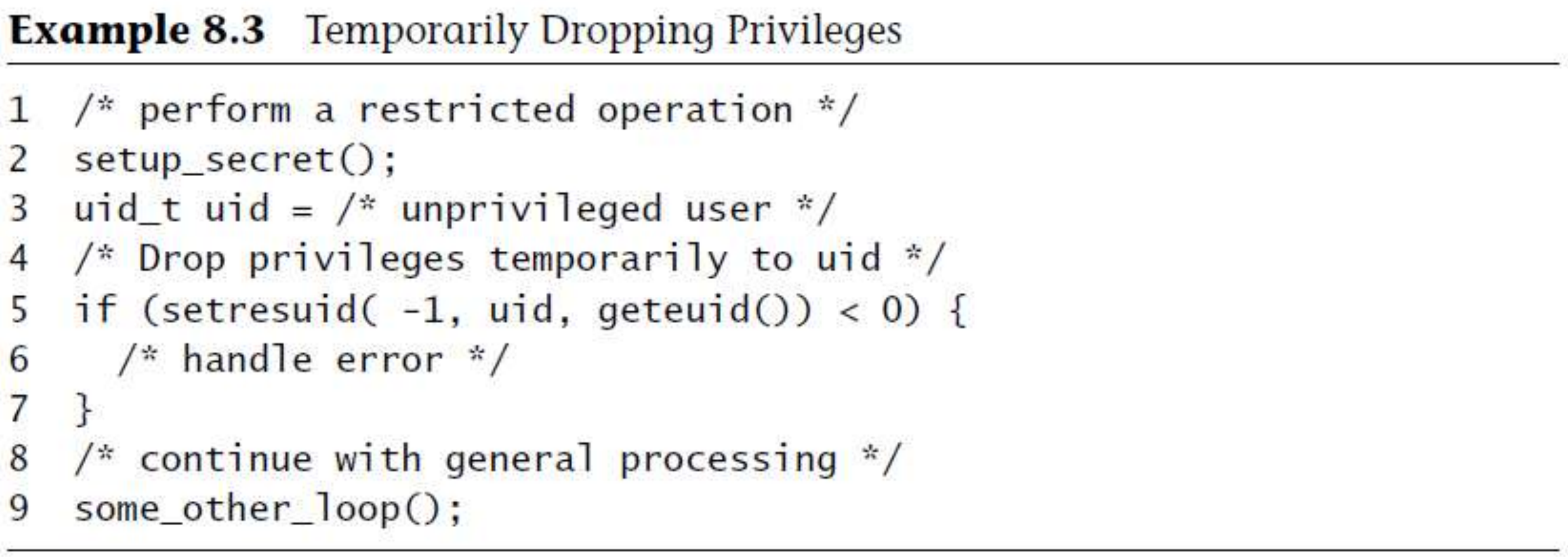


Example 8.2 ping Program Fragment

```
setlocale(LC_ALL, "");

icmp_sock = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
socket_errno = errno;

uid = getuid();
if (setuid(uid)) {
  perror("ping: setuid");
  exit(-1);
}
```

UNIT-5—File I/O—Access Control

Managing Privileges:

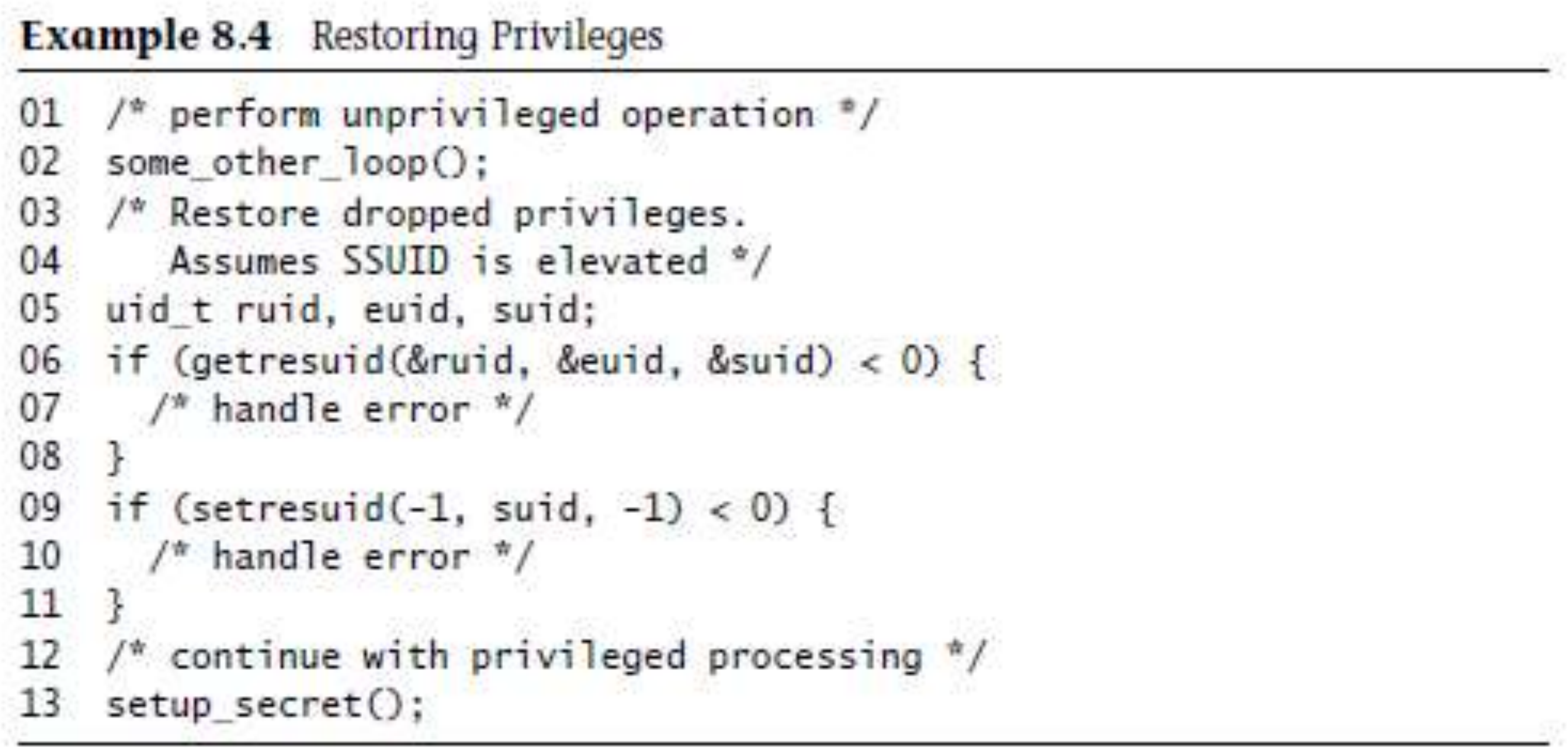


Example 8.3 Temporarily Dropping Privileges

```
/* perform a restricted operation */
setup_secret();
uid_t uid = /* unprivileged user */
/* Drop privileges temporarily to uid */
if (setresuid( -1, uid, geteuid()) < 0) {
  /* handle error */
}
/* continue with general processing */
some_other_loop();
```

UNIT-5—File I/O—Access Control

Managing Privileges:

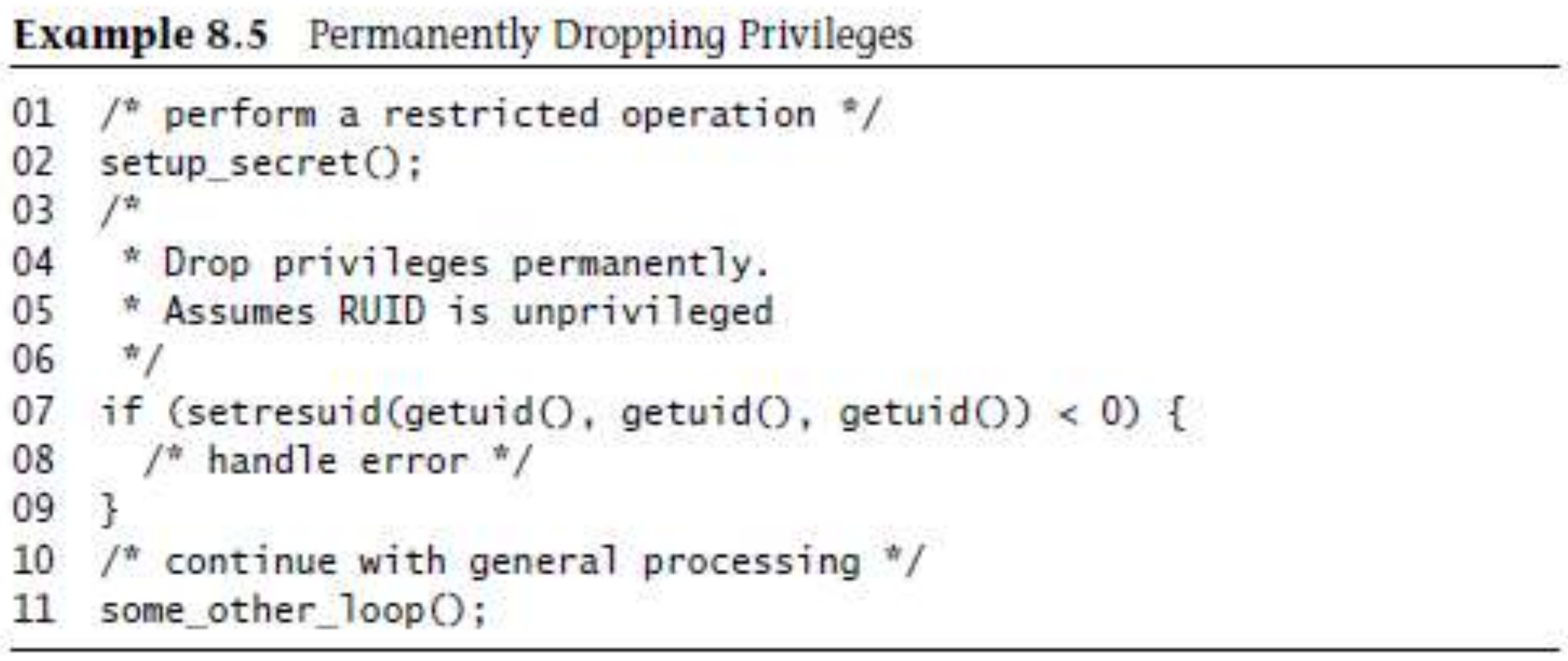


Example 8.4 Restoring Privileges

```
/* perform unprivileged operation */
some_other_loop();
/* Restore dropped privileges.
   Assumes SSUID is elevated */
uid_t ruid, euid, suid;
if (getresuid(&ruid, &euid, &suid) < 0) {
  /* handle error */
}
if (setresuid(-1, suid, -1) < 0) {
  /* handle error */
}
/* continue with privileged processing */
setup_secret();
```

UNIT-5—File I/O—Access Control

Managing Privileges:



Example 8.5 Permanently Dropping Privileges

```
/* perform a restricted operation */
setup_secret();
/*
 * Drop privileges permanently.
 * Assumes RUID is unprivileged
 */
if (setresuid(getuid(), getuid(), getuid()) < 0) {
  /* handle error */
}
/* continue with general processing */
some_other_loop();
```

UNIT-5—File I/O—Access Control

Managing Permissions

Secure Directories

Imagine you have sensitive data stored in a file inside a directory like **/home/myhome/stuff/securestuff**. If the parent directory **(/home/myhome/stuff)** is writable by another user (someone who isn't the owner of the file), that user could rename securestuff. As a result, your program or system may no longer be able to locate the file. Even though your file is still there, its name has changed, making it hard to find or access.

How to secure a directory:
Ownership: The directory and all its parent directories should be owned by the user (or the root user) who owns the sensitive files. This ensures that only the owner or the administrator can make changes to the directory structure.

No write permissions for others: The directory should not be writable by any other user. If other users can write to the directory, they could create, rename, or delete files, which could affect the security of your files.

Prevent deletion or renaming: The directory should be set up so that other users cannot delete or rename files that they do not own. This prevents attackers from tampering with files they shouldn't have access to.

UNIT-5—File I/O—Access Control

Managing Permissions

Permissions on newly created files

- When a file is created, permissions should be restricted exclusively to the owner.
- On POSIX, the operating system stores a value known as the umask for each process it uses when creating new files on behalf of the process.

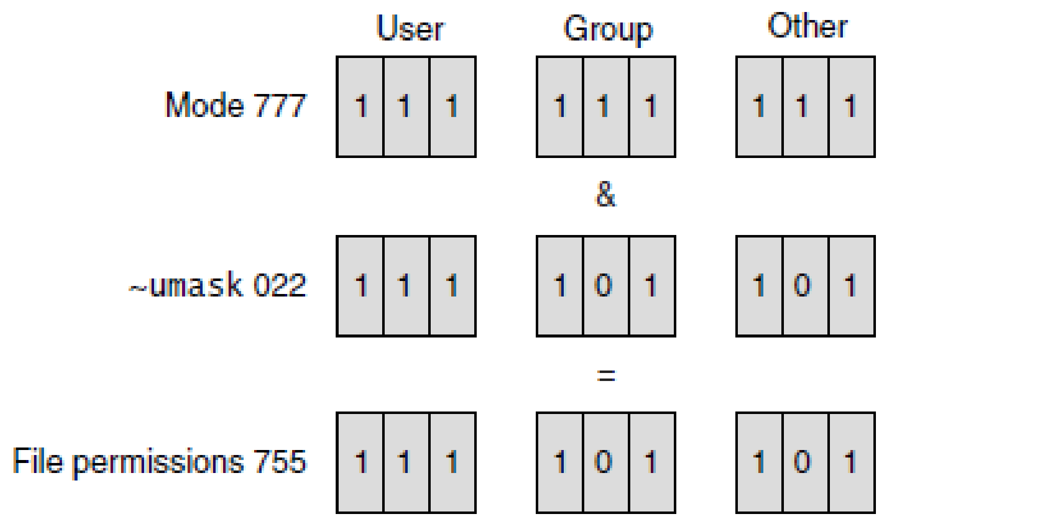


Figure 8.8 Restricting file permissions using umask

UNIT-5—File I/O—Access Control

Managing Permissions

Permissions on newly created files

The umask of 022 is inversed and then ANDed with the mode. The result is that the permission bits specified by the umask in the original mode are turned off, resulting in file permissions of 755 in this case and disallowing “group” or “other” from writing the file.

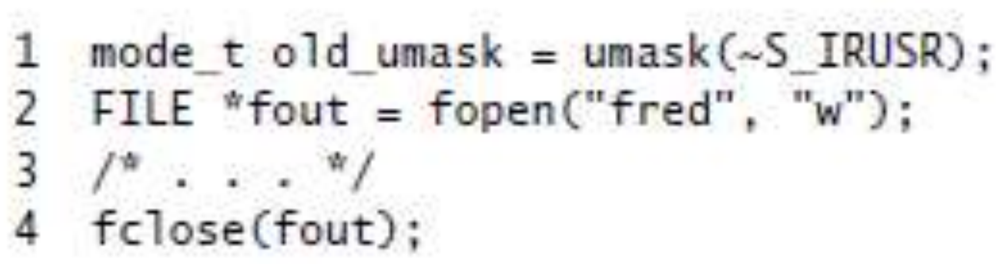
A process inherits the value of its umask from its parent process when the process is created. Normally, when a user logs in, the shell sets a default umask of

- 022 (disable group- and world-writable bits), or
- 02 (disable world-writable bits)

UNIT-5—File I/O—Access Control

Managing Permissions

Permissions on newly created files



```
mode_t old_umask = umask(~S_IRUSR);
FILE *fout = fopen("fred", "w");
/* . . . */
fclose(fout);
```

```
fd = open("fred", O_RDWR|O_CREAT|O_EXCL, S_IRUSR);
```

creates the file fred for writing with user read permission. The open() function returns a file descriptor for the named file that is the lowest file descriptor not currently open for that process.

UNIT-5—File I/O—File Identification

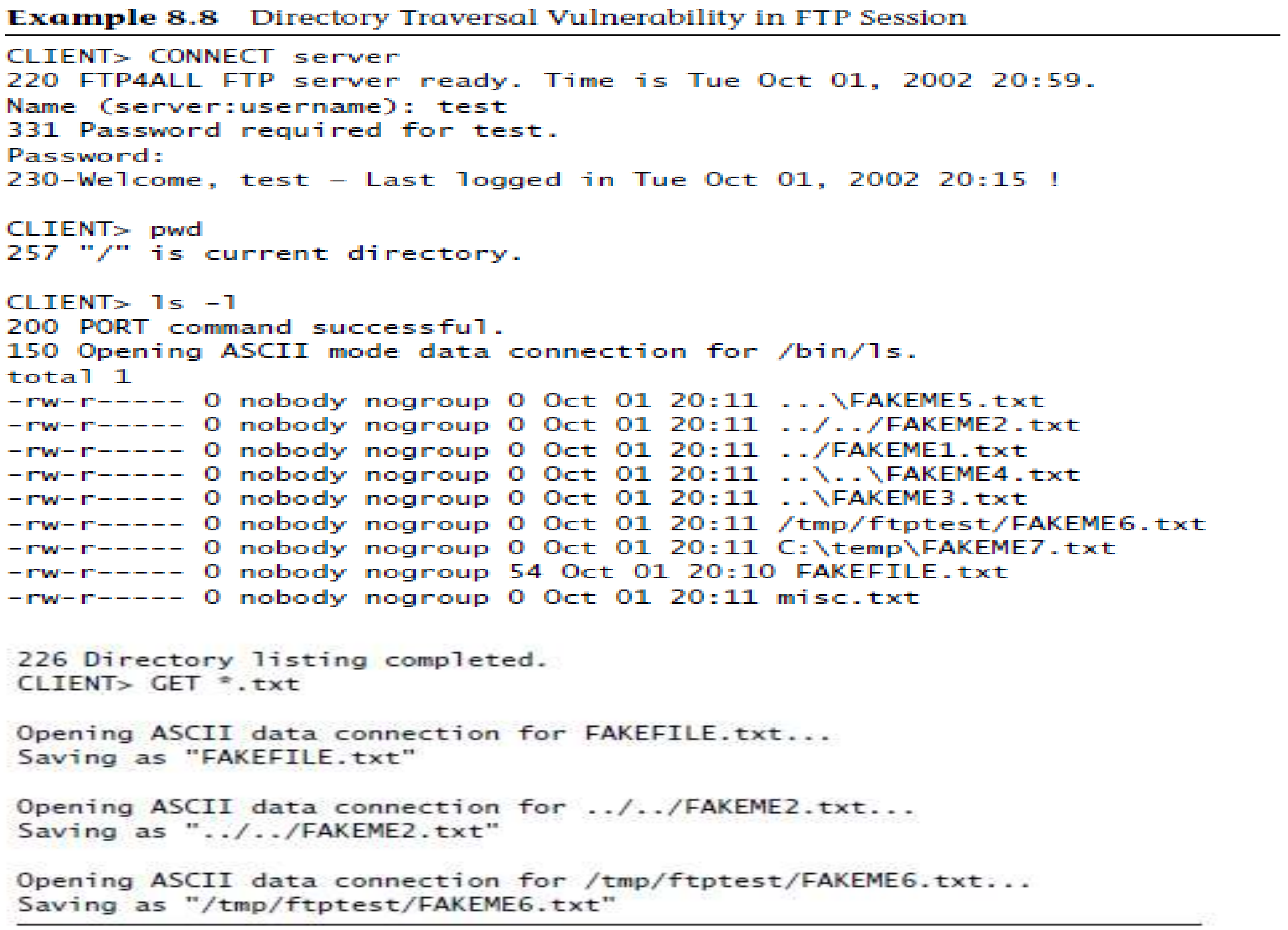
Directory Traversal

Inside a directory, the special file name “.” refers to the directory itself, and “..” refers to the directory’s parent directory. As a special case, in the root directory, “..” may refer to the root directory itself. On Windows systems, drive letters may also be provided (for example, C:), as may other special file
names, such as “...”—which is equivalent to “../.. ”.

A directory traversal vulnerability arises when a program operates on a path name, usually supplied by the user, without sufficient validation.

UNIT-5—File I/O—File Identification

Directory Traversal



Example 8.8 Directory Traversal Vulnerability in FTP Session

```
CLIENT> CONNECT server
220 FTP4ALL FTP server ready. Time is Tue Oct 01, 2002 20:59.
Name (server:username): test
331 Password required for test.
Password:
230-Welcome, test - Last logged in Tue Oct 01, 2002 20:15 !

CLIENT> pwd
257 "/" is current directory.

CLIENT> ls -l
200 PORT command successful.
150 Opening ASCII mode data connection for /bin/ls.
total 1
-rw-r----- 0 nobody nogroup 0 Oct 01 20:11 ...\FAKEME5.txt
-rw-r----- 0 nobody nogroup 0 Oct 01 20:11 ../../FAKEME2.txt
-rw-r----- 0 nobody nogroup 0 Oct 01 20:11 ../FAKEME1.txt
-rw-r----- 0 nobody nogroup 0 Oct 01 20:11 ..\..\FAKEME4.txt
-rw-r----- 0 nobody nogroup 0 Oct 01 20:11 ..\FAKEME3.txt
-rw-r----- 0 nobody nogroup 0 Oct 01 20:11 /tmp/ftptest/FAKEME6.txt
-rw-r----- 0 nobody nogroup 0 Oct 01 20:11 C:\temp\FAKEME7.txt
-rw-r----- 0 nobody nogroup 54 Oct 01 20:10 FAKEFILE.txt
-rw-r----- 0 nobody nogroup 0 Oct 01 20:11 misc.txt

226 Directory listing completed.
CLIENT> GET *.txt

Opening ASCII data connection for FAKEFILE.txt...
Saving as "FAKEFILE.txt"

Opening ASCII data connection for ../../FAKEME2.txt...
Saving as "../../FAKEME2.txt"

Opening ASCII data connection for /tmp/ftptest/FAKEME6.txt...
Saving as "/tmp/ftptest/FAKEME6.txt"
```

UNIT-5—File I/O—File Identification

Directory Traversal

If this program takes the file name argument fn from an untrusted source (such as a user), an attacker can bypass these checks by supplying a file name

such as
/usr/lib/safefiles/../../../etc/shadow

path = replace(path, "../", "");
results in inputs of the form “....//” being converted to “../”. Attempting to strip out “../” and “./” using the following sequence of calls:

```
path = replace(path, "../", "");
path = replace(path, "./", "");
```

results in input of the form “.../....///” being converted to “../”.
A *uniform resource locator* (URL) may contain a host and a path name,

for example:
http://host.name/path/name/file

UNIT-5—File I/O—File Identification

Symbolic links

Symbolic links are a convenient solution to file sharing. Symbolic links are frequently referred to as “symlinks” after the POSIX symlink() system call. Creating a symlink creates a new file with a unique i-node. Symlinks are special files that contain the path name to the actual file.

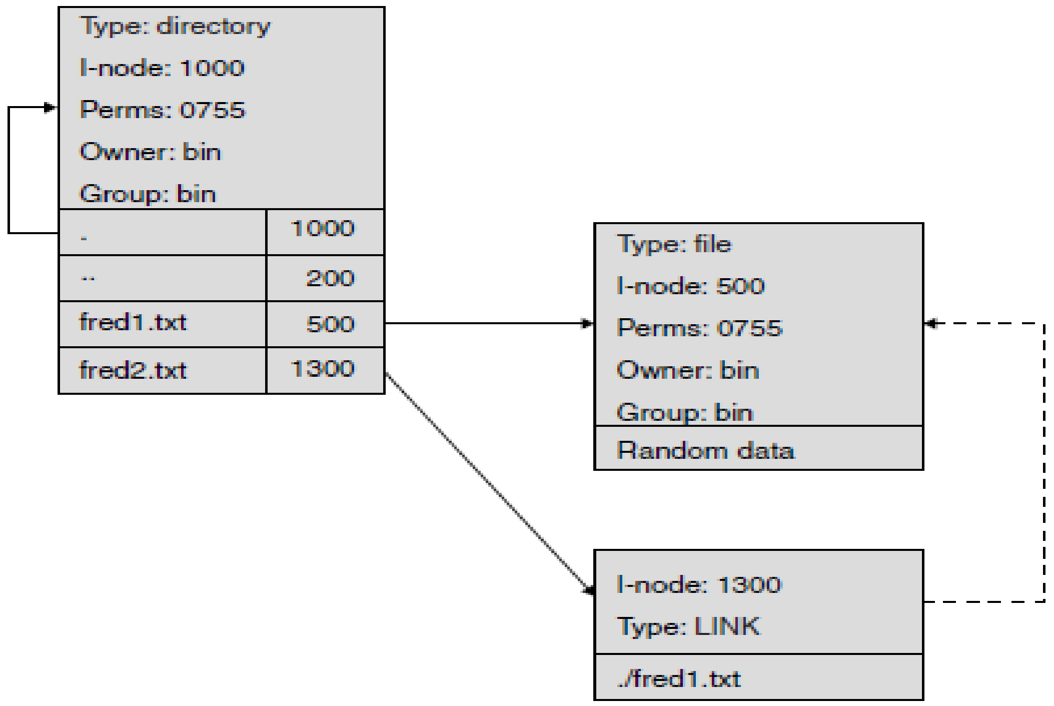
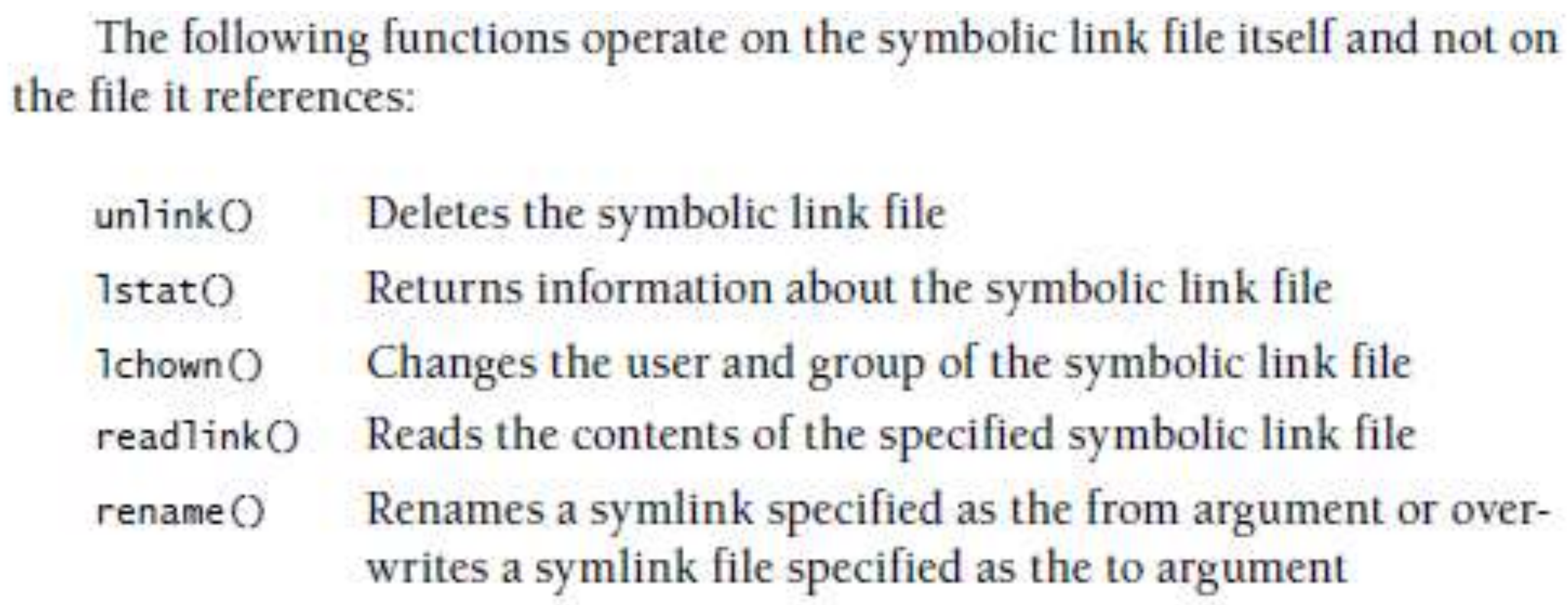


Figure 8.9 Symbolic link

UNIT-5—File I/O—File Identification

Symbolic links



The following functions operate on the symbolic link file itself and not on the file it references:

`unlink()`	Deletes the symbolic link file
`lstat()`	Returns information about the symbolic link file
`lchown()`	Changes the user and group of the symbolic link file
`readlink()`	Reads the contents of the specified symbolic link file
`rename()`	Renames a symlink specified as the from argument or over-writes a symlink file specified as the to argument

UNIT-5—File I/O—File Identification

Symbolic links

For example, assume the following code runs as a set-user-ID-root program with effective root privileges:

```
1 fd = open("/home/rcs/.conf", O_RDWR);
2 if (fd < 0) abort();
3 write(fd, userbuf, userlen);
```

Assume also that an attacker can control the data stored in userbuf and written in the call to write(). An attacker creates a symbolic link from .conf to the /etc/shadow authentication file:

```
% cd /home/rcs
% ln –s /etc/shadow .conf
```

and then runs the vulnerable program, which opens the file for writing as root and writes attacker-controlled information to the password file:

```
% runprog
```

This attack can be used, for example, to create a new root account with no password. The attacker can then use the su command to switch to the root account for root access:

```
% su
#
```

UNIT-5—File I/O—File Identification

Canonicalization

Canonicalization is the process of converting a file or directory name into its standard or simplest form, which helps in making file name validation easier and more secure.

For example, a file path like "/usr/../home/rcs" can be simplified to "/home/rcs" by removing unnecessary components like "..", which represent moving up one directory level.

This standardization is important for preventing security issues like directory traversal or file equivalence errors, where attackers might try to exploit different ways of referencing the same file.

By converting paths to their canonical form, including resolving symbolic links, it becomes easier to compare paths and ensure that files are being accessed correctly**.**

However, canonicalization can be complex, especially across different operating systems and file systems. Functions like realpath() can help with this process by resolving symbolic links and eliminating unnecessary path components

UNIT-5—File I/O—File Identification

Hard Links

Hard links can be created using the ln command. For example, the command

```
ln /etc/shadow
```

increments the link counter in the i-node for the shadow file and creates a new directory entry in the current working directory.

Hard links are indistinguishable from original directory entries but cannot refer to directories or span file systems. Ownership and permissions reside with the i-node, so all hard links to the same i-node have the same ownership and permissions.

UNIT-5—File I/O—File Identification

Hard Links

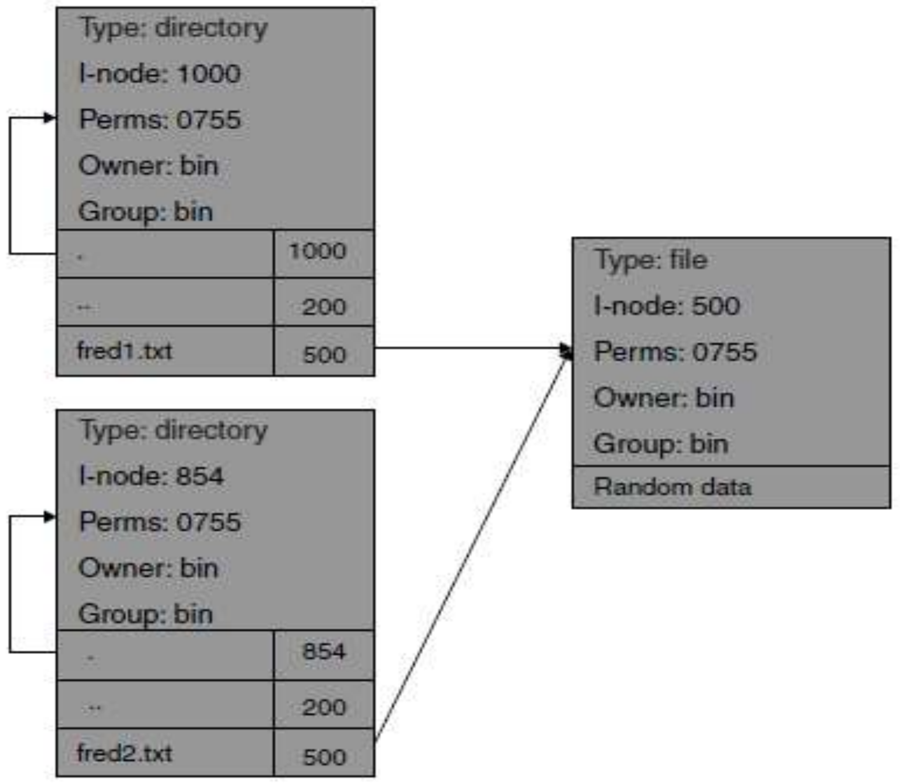


Figure 8.10 Hard link

UNIT-5—File I/O—File Identification

Hard Links

Deleting a hard link doesn’t delete the file unless all references to the file have been deleted. A reference is either a hard link or an open file descriptor; the i-node can be deleted (data addresses cleared) only if its link counter is 0.

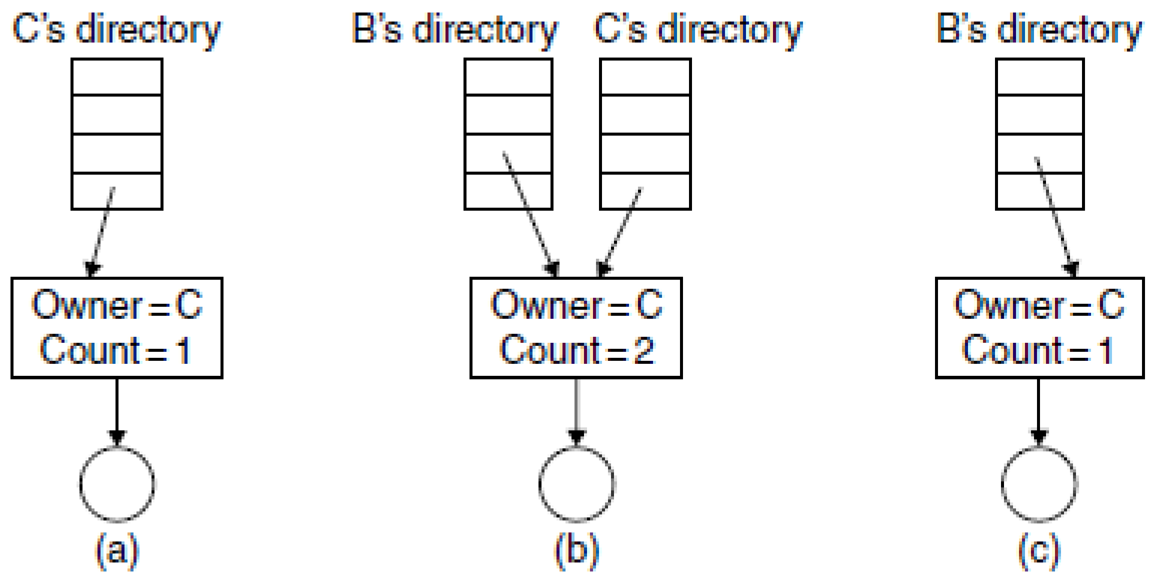


Figure 8.11 Shared file using hard links

UNIT-5—File I/O—File Identification

Hard Links

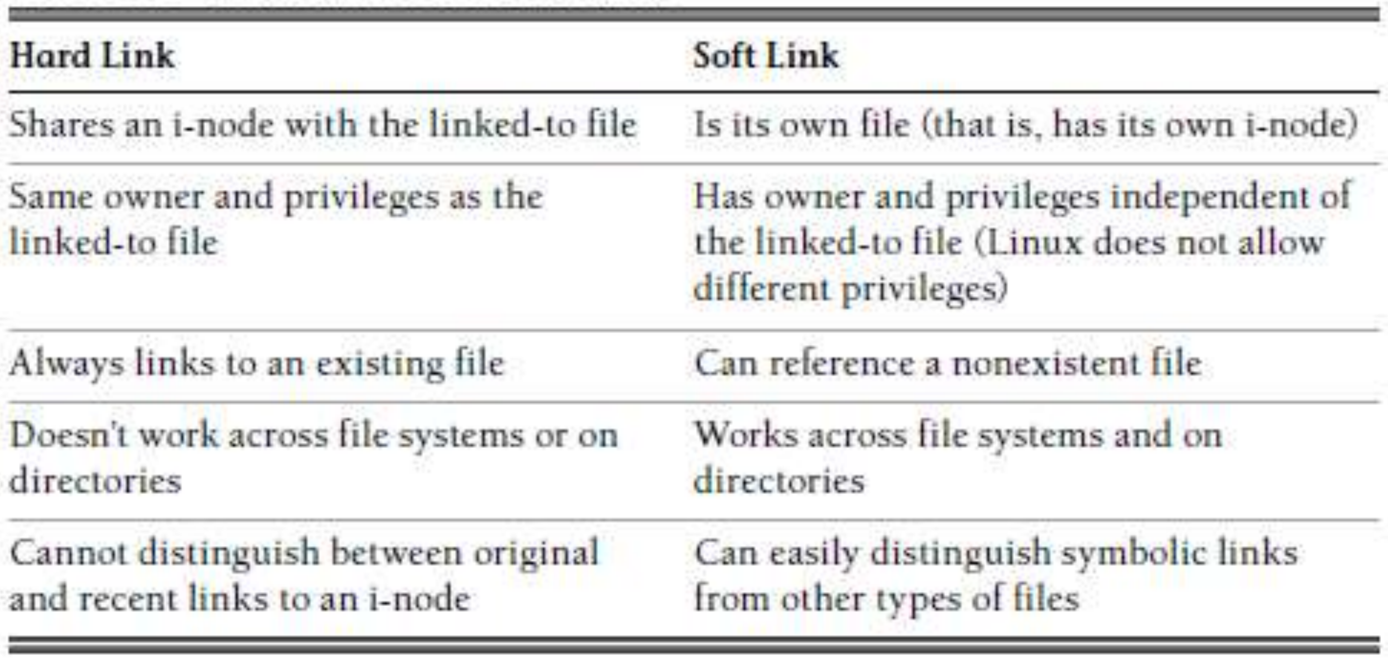


Table 8.3 Hard Links versus Soft Links

Hard Link	Soft Link
Shares an i-node with the linked-to file	Is its own file (that is, has its own i-node)
Same owner and privileges as the linked-to file	Has owner and privileges independent of the linked-to file (Linux does not allow different privileges)
Always links to an existing file	Can reference a nonexistent file
Doesn't work across file systems or on directories	Works across file systems and on directories
Cannot distinguish between original and recent links to an i-node	Can easily distinguish symbolic links from other types of files

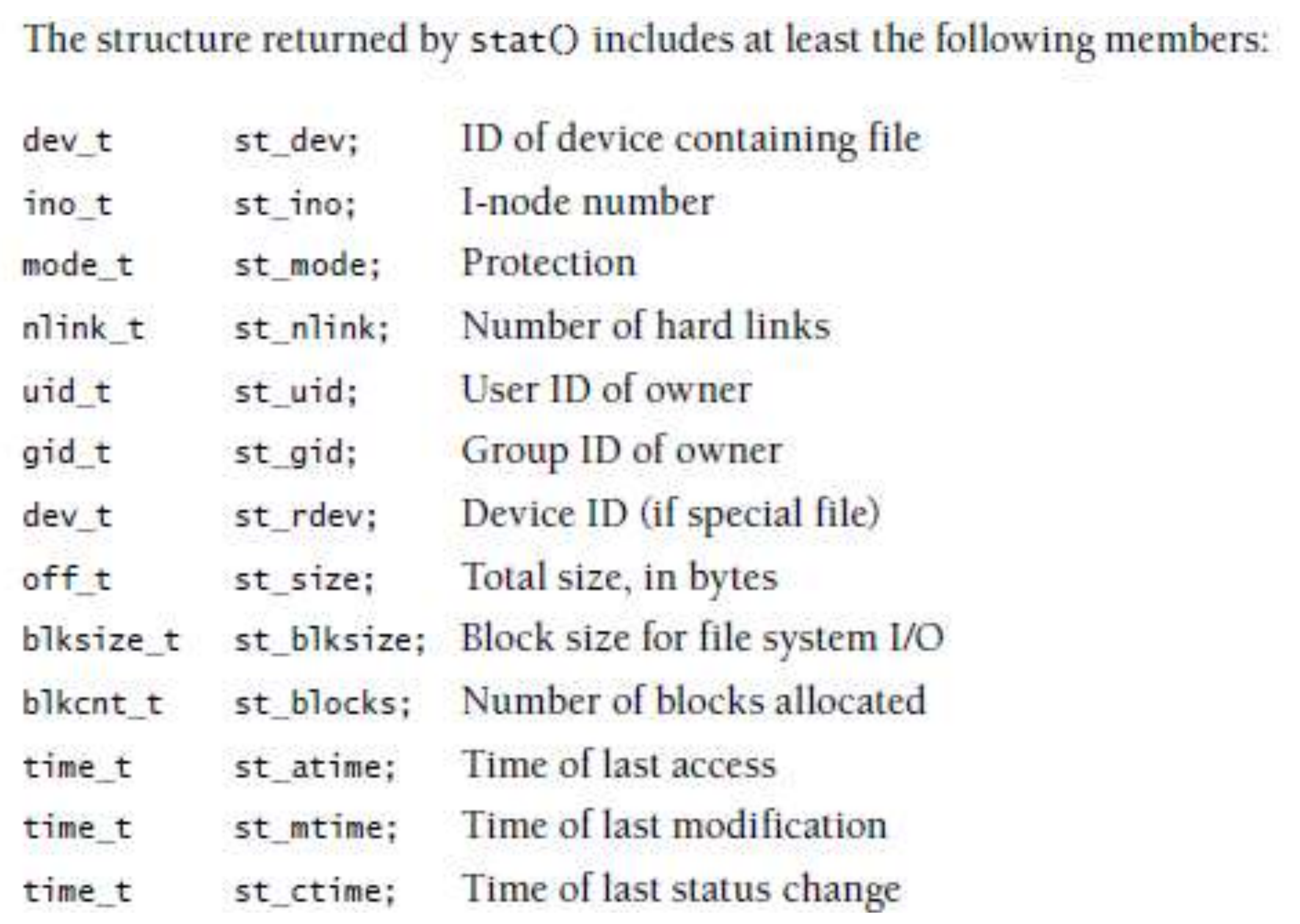
UNIT-5—File I/O—File Identification

Device Files

- Device files are special files in a computer's file system that represent hardware devices or device drivers, allowing programs to interact with physical hardware through standard file operations like reading and writing.
- These files are typically found in specific directories, such as /dev on UNIX-like systems (including Linux), and they serve as a bridge between software and hardware.
- In UNIX-based systems, device files come in two main types: character device files and block device files.
- Character device files, such as /dev/tty (for terminal devices), allow data to be read or written one character at a time.
- Block device files, such as /dev/sda (for hard drives), interact with hardware in fixed-size blocks, making them suitable for storing data.
- Device files can be used to interact with various hardware components, like disk drives, printers, mice, or network interfaces.

UNIT-5—File I/O—File Identification

File Attributes

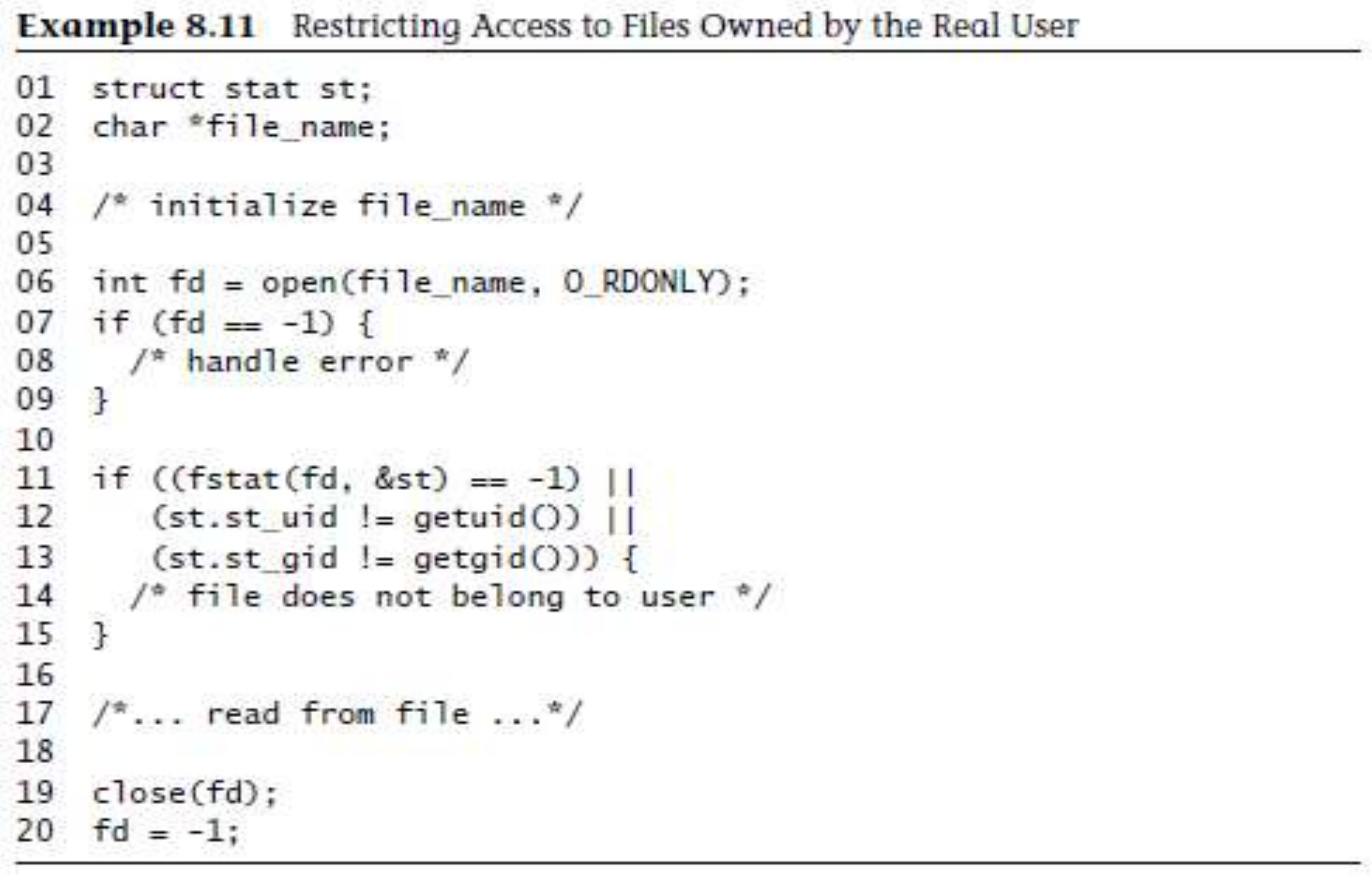


The structure returned by stat() includes at least the following members:

dev_t	st_dev;	ID of device containing file
ino_t	st_ino;	I-node number
mode_t	st_mode;	Protection
nlink_t	st_nlink;	Number of hard links
uid_t	st_uid;	User ID of owner
gid_t	st_gid;	Group ID of owner
dev_t	st_rdev;	Device ID (if special file)
off_t	st_size;	Total size, in bytes
blksize_t	st_blksize;	Block size for file system I/O
blkcnt_t	st_blocks;	Number of blocks allocated
time_t	st_atime;	Time of last access
time_t	st_mtime;	Time of last modification
time_t	st_ctime;	Time of last status change

UNIT-5—File I/O—File Identification

File Attributes



Example 8.11 Restricting Access to Files Owned by the Real User

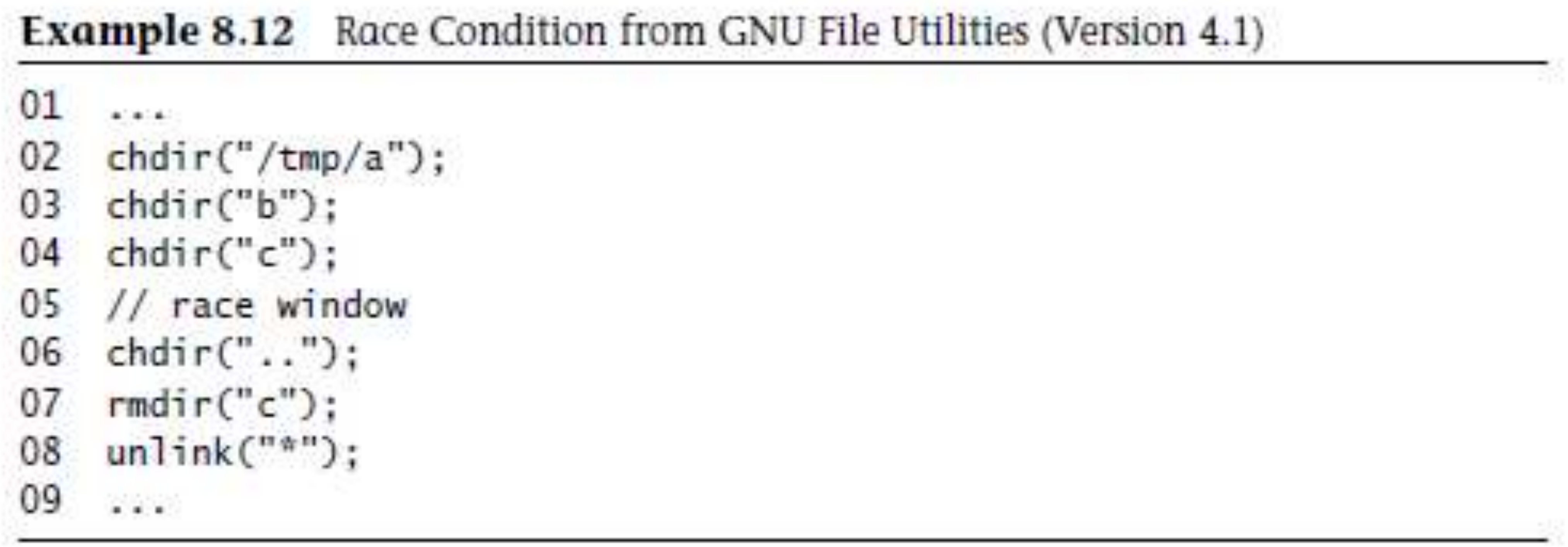
```
struct stat st;
char *file_name;

/* initialize file_name */

int fd = open(file_name, O_RDONLY);
if (fd == -1) {
  /* handle error */
}

if ((fstat(fd, &st) == -1) ||
   (st.st_uid != getuid()) ||
   (st.st_gid != getgid())) {
  /* file does not belong to user */
}

/*... read from file ...*/

close(fd);
fd = -1;
```

UNIT-5—File I/O—Race Conditions

- A race condition occurs when multiple processes or threads try to access shared resources, like files or directories, at the same time, leading to unpredictable behavior.
- This can happen when one part of a program expects a certain file or directory to remain unchanged while another part, or an external process, modifies it concurrently.
- For example, in the GNU file utilities, a race condition was discovered where a directory path was expected to exist, but an attacker could exploit the timing between two steps in the program.
- If the attacker moves the directory during this "race window," it could cause the program to delete unintended files, especially if it's running with elevated privileges like root, which can have serious security consequences.

UNIT-5—File I/O—Race Conditions

mv /tmp/a/b/c /tmp/c



Example 8.12 Race Condition from GNU File Utilities (Version 4.1)

```
...
chdir("/tmp/a");
chdir("b");
chdir("c");
// race window
chdir("..");
rmdir("c");
unlink("*");
...
```

UNIT-5—File I/O—Race Conditions

Time of Check and Time of Use(TOCTOU)

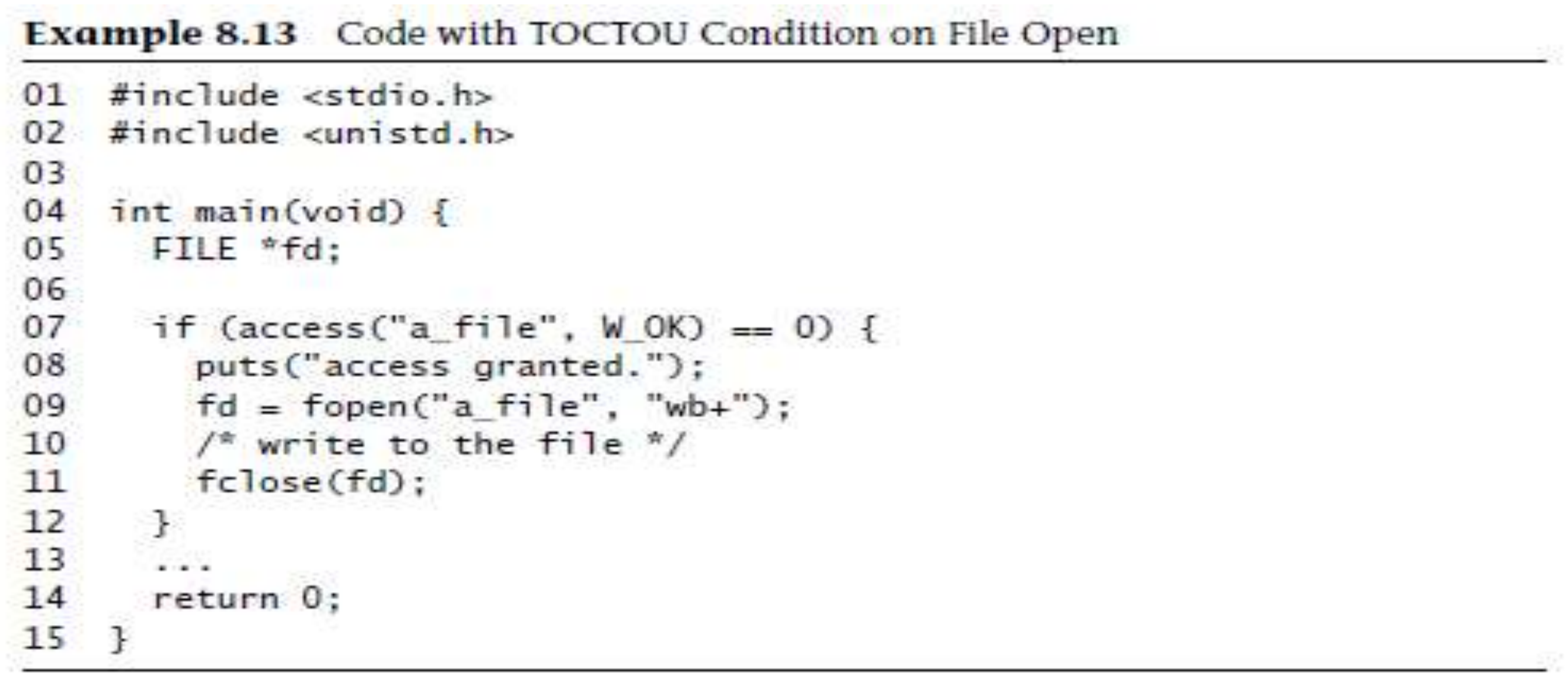
A Time of Check, Time of Use (TOCTOU) race condition occurs when a program checks the state of a resource (like a file or directory) at one point in time and then uses it later, but between the check and the use, the resource has changed. This can lead to unexpected behavior or security vulnerabilities, especially when the resource is modified by another process or thread during the delay.

For example, consider a scenario where a program checks if a file exists and has certain permissions (Time of Check) and then performs an action on it (Time of Use). If, in the time between these two steps, another process changes the file, such as altering its permissions or deleting it, the program might perform the action on the wrong file or fail unexpectedly, leading to potential security risks. Attackers can exploit TOCTOU conditions by manipulating the resource between the check and the use, gaining unauthorized access or causing other issues.

To prevent TOCTOU issues, it's important to minimize the gap between checking a resource's state and acting on it, and to use atomic operations when possible.

UNIT-5—File I/O—Race Conditions

Time of Check and Time of Use(TOCTOU)

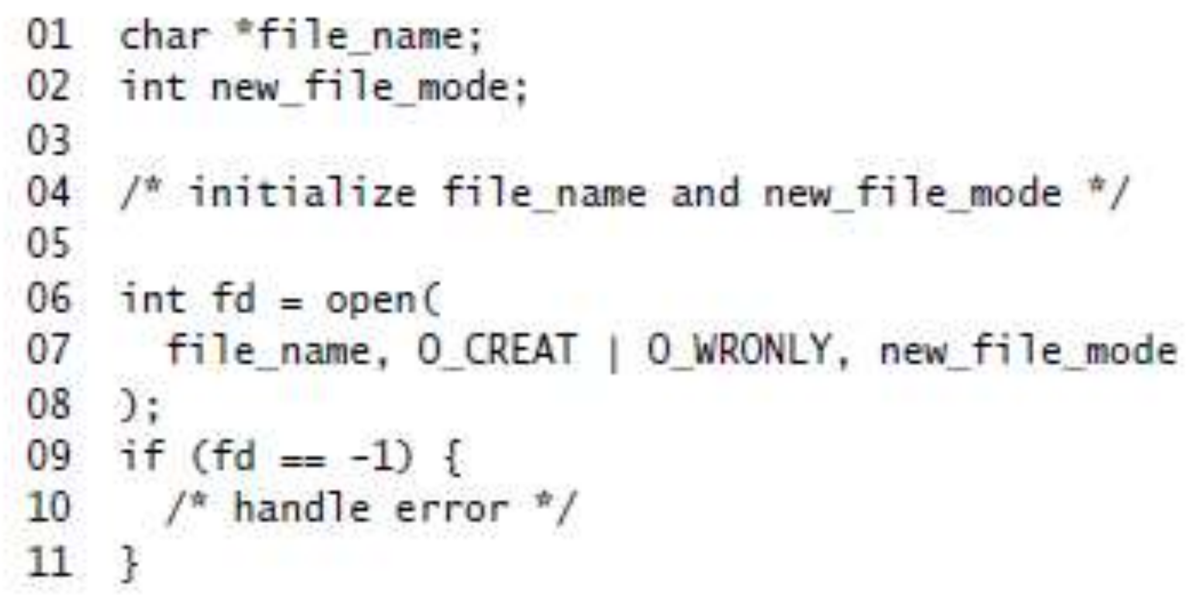


Example 8.13 Code with TOCTOU Condition on File Open

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

int main(void) {
  FILE *fd;

  if (access("a_file", W_OK) == 0) {
    puts("access granted.");
    fd = fopen("a_file", "wb+");
    /* write to the file */
    fclose(fd);
  }
  ...
  return 0;
}
```

UNIT-5—File I/O—Race Conditions

Create Without Replace



```
char *file_name;
int new_file_mode;

/* initialize file_name and new_file_mode */

int fd = open(
  file_name, O_CREAT | O_WRONLY, new_file_mode
);
if (fd == -1) {
  /* handle error */
}
```

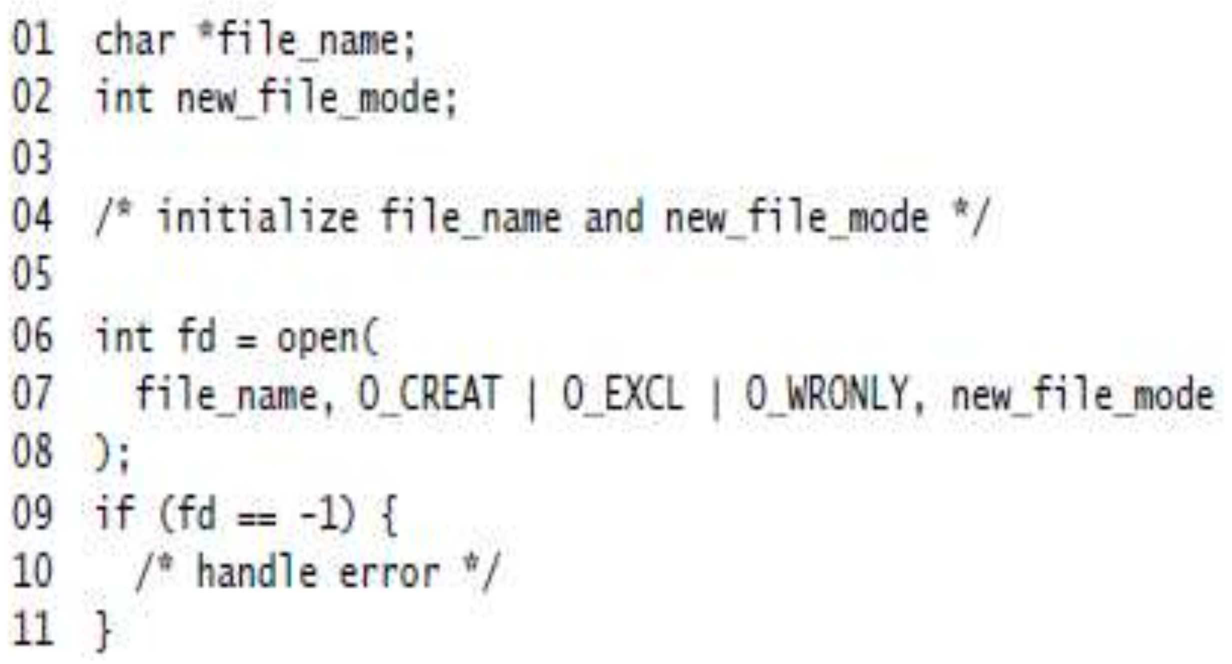
If file_name already exists at the time the call to open() executes, then that file is opened and truncated.

Furthermore, if file_name is a symbolic link, then the target file referenced by the link is truncated.

All an attacker needs to do is create a symbolic link at file_name before this call. Assuming the vulnerable process has appropriate permissions, the targeted file will be overwritten.

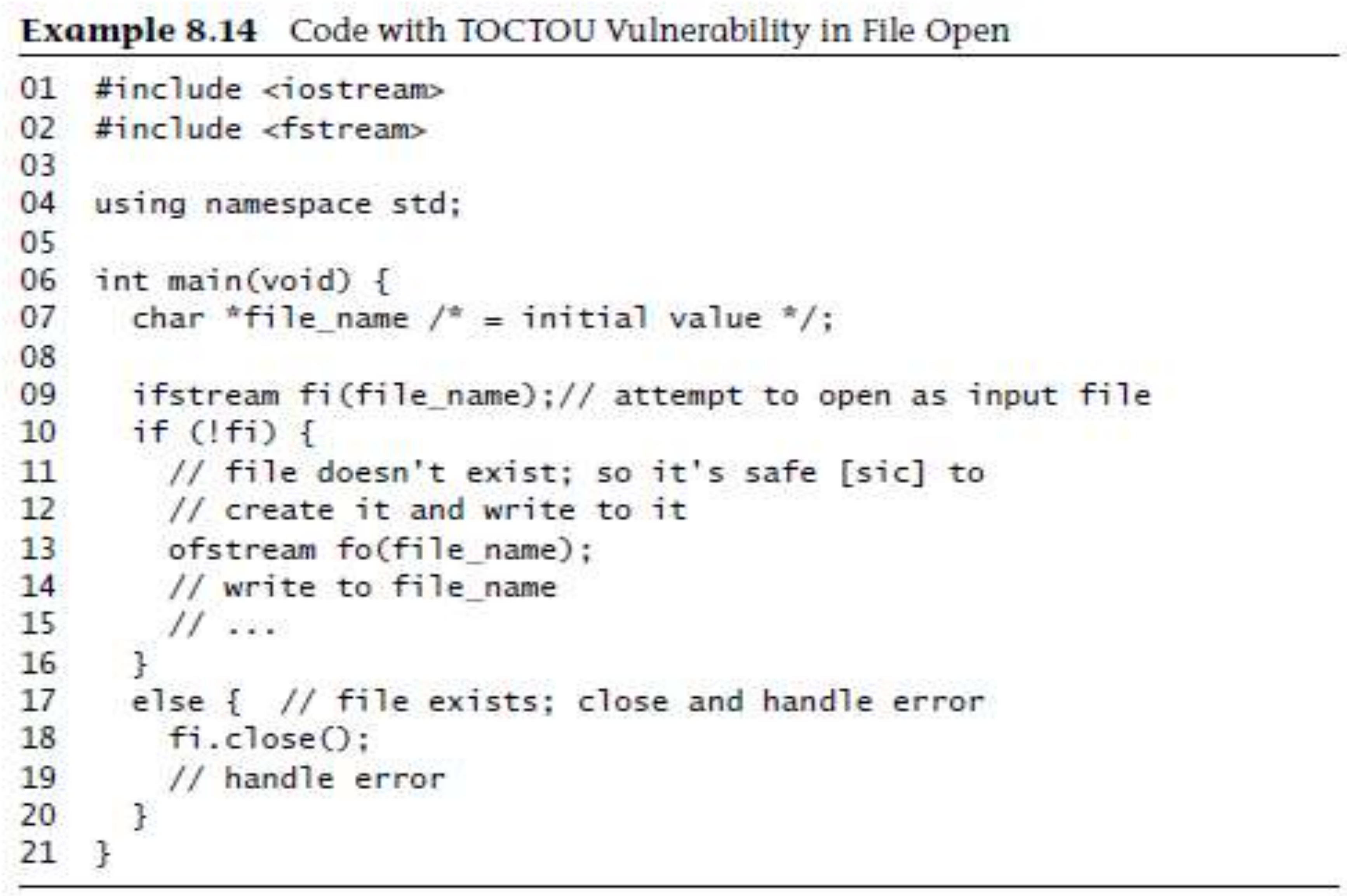
UNIT-5—File I/O—Race Conditions

Create Without Replace



```c
char *file_name;
int new_file_mode;

/* initialize file_name and new_file_mode */

int fd = open(
  file_name, O_CREAT | O_EXCL | O_WRONLY, new_file_mode
);
if (fd == -1) {
  /* handle error */
}
```

UNIT-5—File I/O—Race Conditions



Example 8.14 Code with TOCTOU Vulnerability in File Open

```
#include <iostream>
#include <fstream>

using namespace std;

int main(void) {
  char *file_name /* = initial value */;

  ifstream fi(file_name);// attempt to open as input file
  if (!fi) {
    // file doesn't exist; so it's safe [sic] to
    // create it and write to it
    ofstream fo(file_name);
    // write to file_name
    // ...
  }
  else {  // file exists; close and handle error
    fi.close();
    // handle error
  }
}
```

UNIT-5—File I/O—Race Conditions

Exclusive Access

- Exclusive access and file locking are mechanisms used to prevent multiple processes or threads from simultaneously accessing or modifying the same file, ensuring data consistency and preventing conflicts.

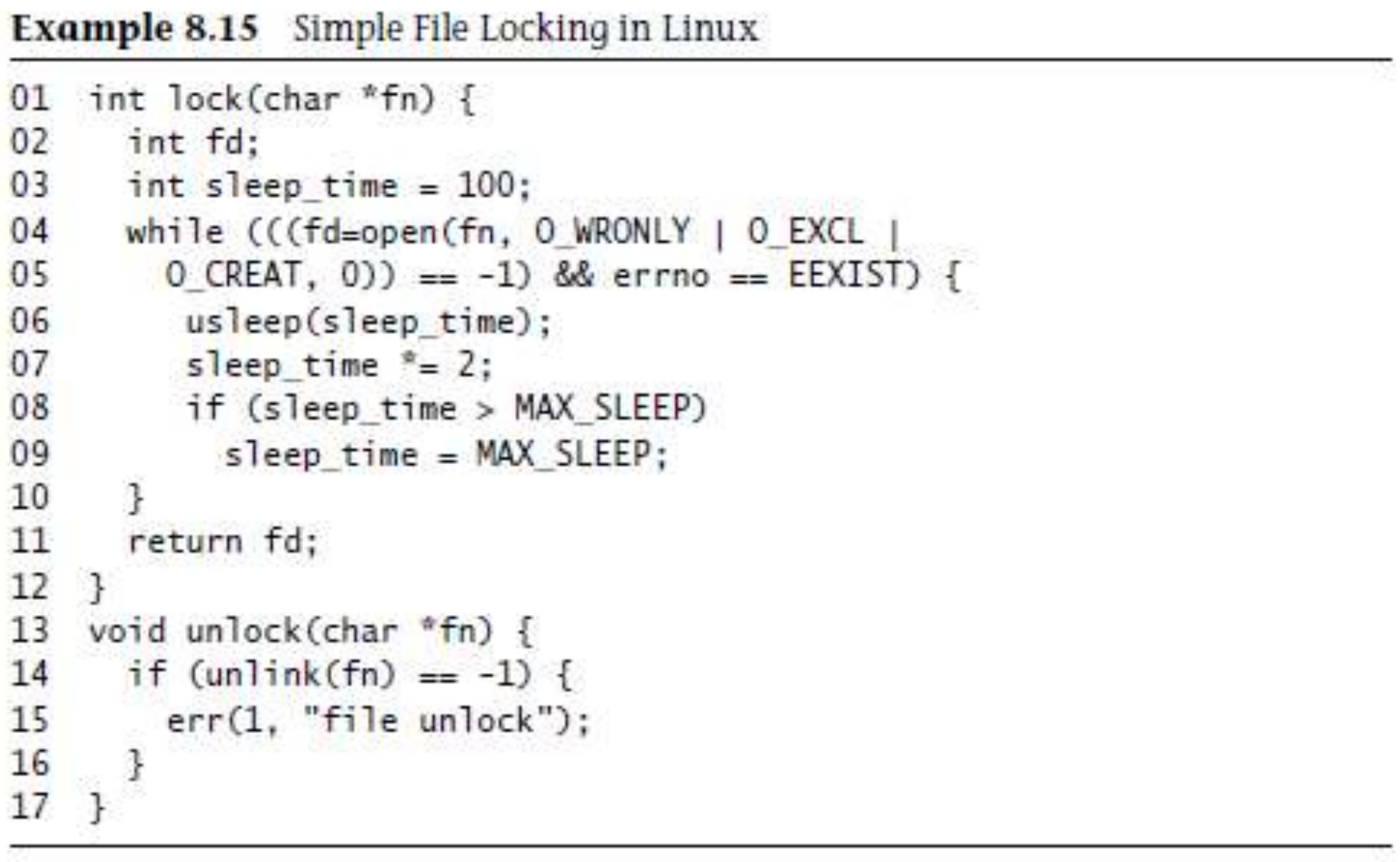
- File locking is a technique used in operating systems to control access to a file by multiple processes. It allows a process to "lock" a file while it is using it, preventing other processes from reading or modifying the file until the lock is released. There are two common types of locks:

- Exclusive lock: This lock prevents other processes from accessing the file at all. Only the process holding the exclusive lock can read or write to the file, ensuring that no other process can interfere.

- Shared lock: This allows multiple processes to read a file simultaneously, but prevents any process from writing to it while the lock is in place.
- The purpose of file locking is to avoid race conditions or data corruption that can occur when multiple processes try to modify a file at the same time. It is commonly used in scenarios where data integrity is critical, such as databases or configuration files. However, improper use of file locking (e.g., not releasing a lock or using it incorrectly) can lead to performance issues, deadlocks, or even system crashes.

UNIT-5—File I/O—Race Conditions

Exclusive Access



Example 8.15 Simple File Locking in Linux

```
int lock(char *fn) {
  int fd;
  int sleep_time = 100;
  while (((fd=open(fn, O_WRONLY | O_EXCL |
    O_CREAT, 0)) == -1) && errno == EEXIST) {
     usleep(sleep_time);
     sleep_time *= 2;
     if (sleep_time > MAX_SLEEP)
       sleep_time = MAX_SLEEP;
  }
  return fd;
}
void unlock(char *fn) {
  if (unlink(fn) == -1) {
    err(1, "file unlock");
  }
}
```

UNIT-5—File I/O—Race Conditions

Shared Directories

Shared directories are directly related to race conditions because multiple processes or users accessing the same files in a shared directory can lead to conflicts if proper synchronization is not in place. In environments with shared directories, race conditions can occur when two or more processes try to read from or write to the same file at the same time, resulting in unpredictable behavior or data corruption.

For example, if two processes simultaneously try to write to the same file in a shared directory, the first process might check the file's state, perform its operation, and then finish. Meanwhile, the second process could perform its operation without knowing that the first one has already modified the file, causing inconsistent or incorrect data. Similarly, if one process checks the file's state and another process changes it in between the check and the actual operation (the Time of Check, Time of Use or TOCTOU race condition), this can also lead to errors.

To prevent race conditions in shared directories, mechanisms like file locking, where processes ensure exclusive or coordinated access to a file, are used. This ensures that one process can safely operate on a file without interference from others, maintaining data integrity and preventing conflicts.

UNIT-5—File I/O—Mitigation Strategies

Closing the Race Window

a. Mutual Exclusion Mitigation
b. Thread Safe Functions
c. Use of Atomic Operations
d. Reopening Files
e. Checking for Symbolic Links

UNIT-5—File I/O—Mitigation Strategies

Closing the Race Window

a. Mutual Exclusion Mitigation

Synchronization Primitives:
UNIX and Windows provide various synchronization primitives like mutexes, semaphores, critical sections, condition variables, and lock variables. These are used to protect critical sections in multithreaded applications and prevent race conditions by ensuring mutual exclusion.

Minimizing Critical Section Size:
To avoid performance issues, the size of critical sections should be minimized, ensuring synchronization primitives are only used where absolutely necessary.

Object-Oriented Approach:
An alternative approach uses decorator modules to isolate access to shared resources. Wrapper functions test for mutual exclusion, ensuring safe access.

Synchronization Between Processes:
For race conditions involving multiple processes, synchronization primitives must be in shared memory and support multi-process awareness. Examples include Windows named mutexes or POSIX named semaphores, which can be used across processes.

UNIX File-Based Locks:
A less efficient solution in UNIX involves using a file as a lock, though it's not as robust as other synchronization mechanisms.

Deadlock Avoidance Strategy:
Resources are numbered (e.g., r1, r2, …, rn), and a process must capture resources in a specific order (r1, r2, …, rk) to avoid deadlock.

UNIT-5—File I/O—Mitigation Strategies

Closing the Race Window

a. Thread Safe Functions

- Race conditions can occur not just within an application’s code but also in functions that the application calls, even if the function is from an external library or API.
- A function is considered thread-safe if it can be safely called by multiple threads simultaneously without causing race conditions.
- If a function is not thread-safe, it should be treated as a critical section, meaning access to it should be synchronized to prevent race conditions when called by multiple threads.

UNIT-5—File I/O—Mitigation Strategies

Closing the Race Window

C. Use of Atomic Operations:

- ✓ EnterCriticalRegion() or pthread_mutex_lock()
- ✓ LeaveCriticalRegion()

D. Reopening of Files

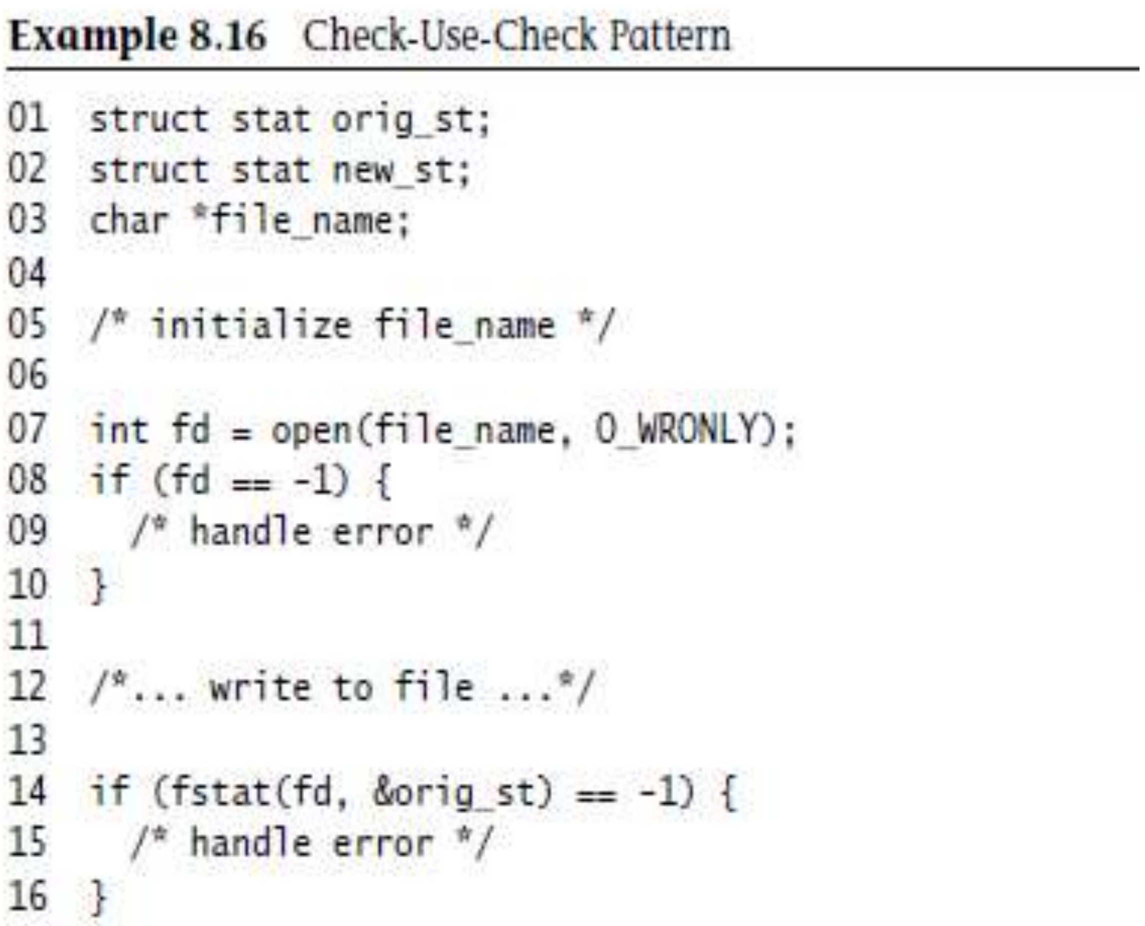
Reopening of file stream should generally be avoided but may be necessary in long running applications to avoid depleting file descriptors.

Because the file name is reassociated with the file each time it is opened, there are no guarantees that reopened file is the same as the original file.

UNIT-5—File I/O—Mitigation Strategies

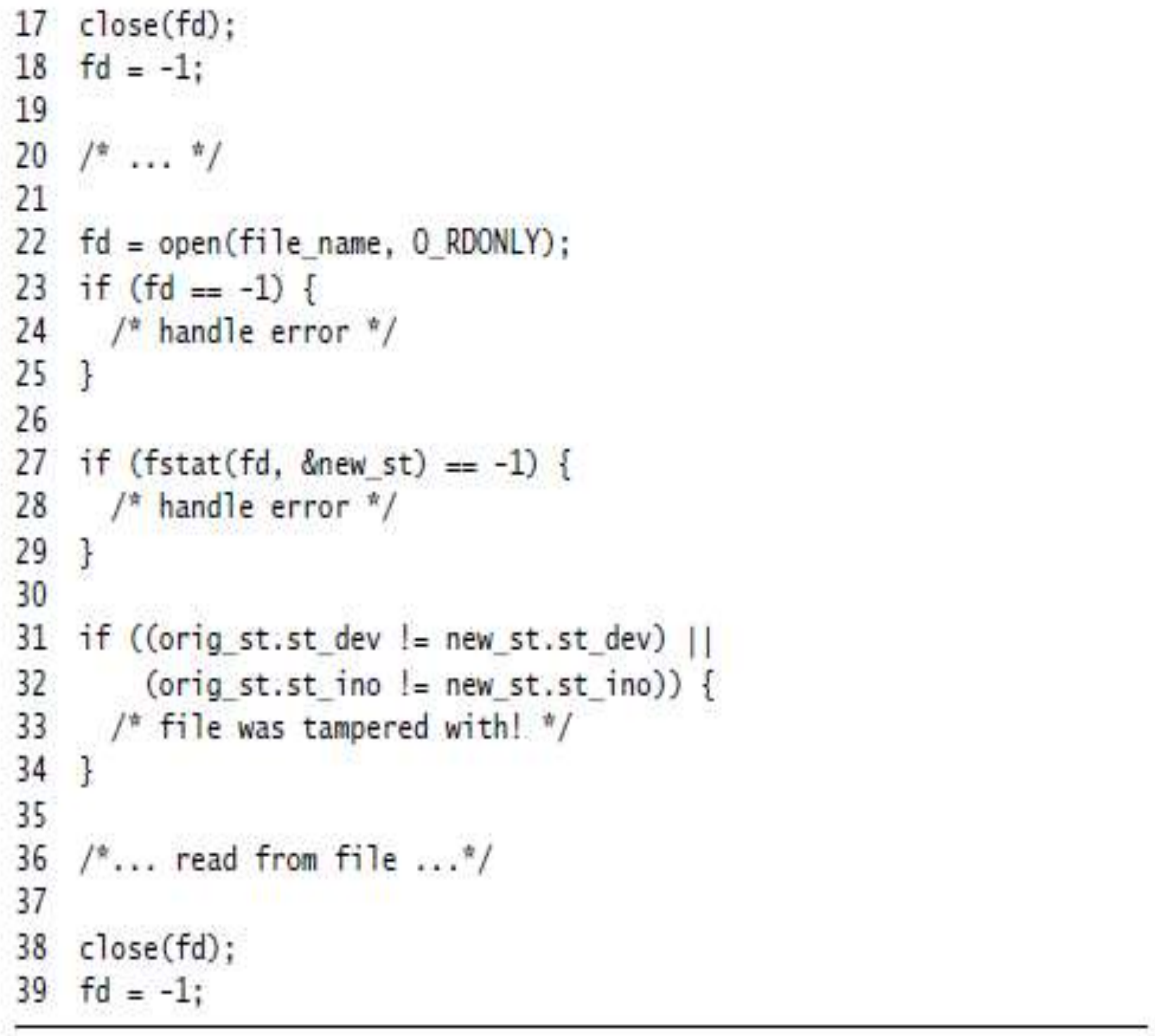
Closing the Race Window

D. Reopening of Files



Example 8.16 Check-Use-Check Pattern

```
struct stat orig_st;
struct stat new_st;
char *file_name;

/* initialize file_name */

int fd = open(file_name, O_WRONLY);
if (fd == -1) {
  /* handle error */
}

/*... write to file ...*/

if (fstat(fd, &orig_st) == -1) {
  /* handle error */
}
```

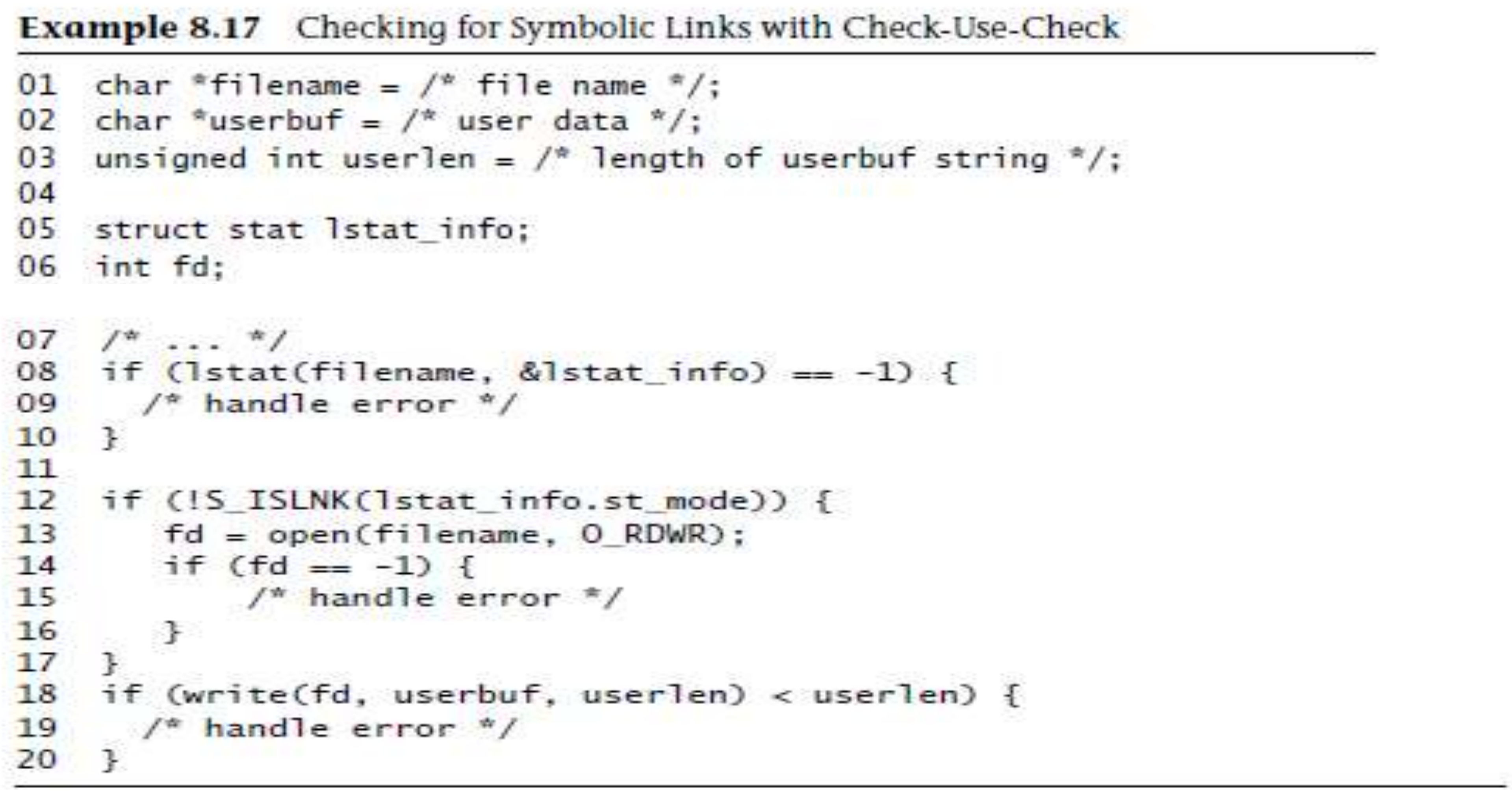


```
close(fd);
fd = -1;

/* ... */

fd = open(file_name, O_RDONLY);
if (fd == -1) {
  /* handle error */
}

if (fstat(fd, &new_st) == -1) {
  /* handle error */
}

if ((orig_st.st_dev != new_st.st_dev) ||
    (orig_st.st_ino != new_st.st_ino)) {
  /* file was tampered with! */
}

/*... read from file ...*/

close(fd);
fd = -1;
```

UNIT-5—File I/O—Mitigation Strategies

Closing the Race Window

E**. Checking for Symbolic links**



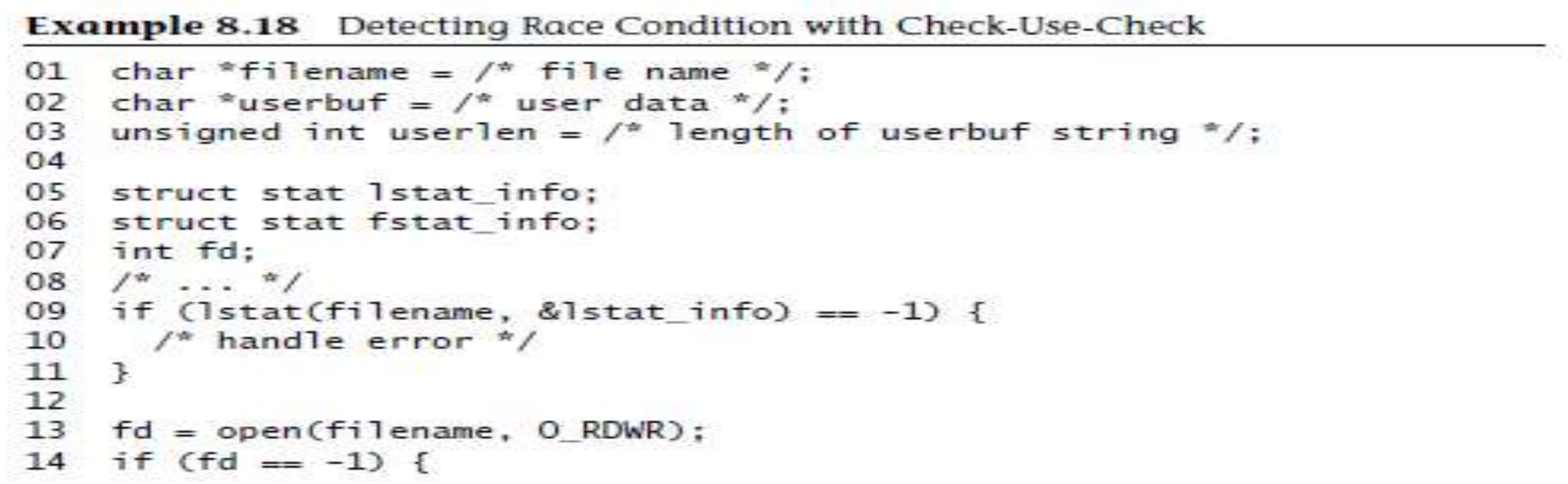
Example 8.17 Checking for Symbolic Links with Check-Use-Check

```
char *filename = /* file name */;
char *userbuf = /* user data */;
unsigned int userlen = /* length of userbuf string */;

struct stat lstat_info;
int fd;

/* ... */
if (lstat(filename, &lstat_info) == -1) {
  /* handle error */
}

if (!S_ISLNK(lstat_info.st_mode)) {
   fd = open(filename, O_RDWR);
   if (fd == -1) {
       /* handle error */
   }
}
if (write(fd, userbuf, userlen) < userlen) {
  /* handle error */
}
```

UNIT-5—File I/O—Mitigation Strategies

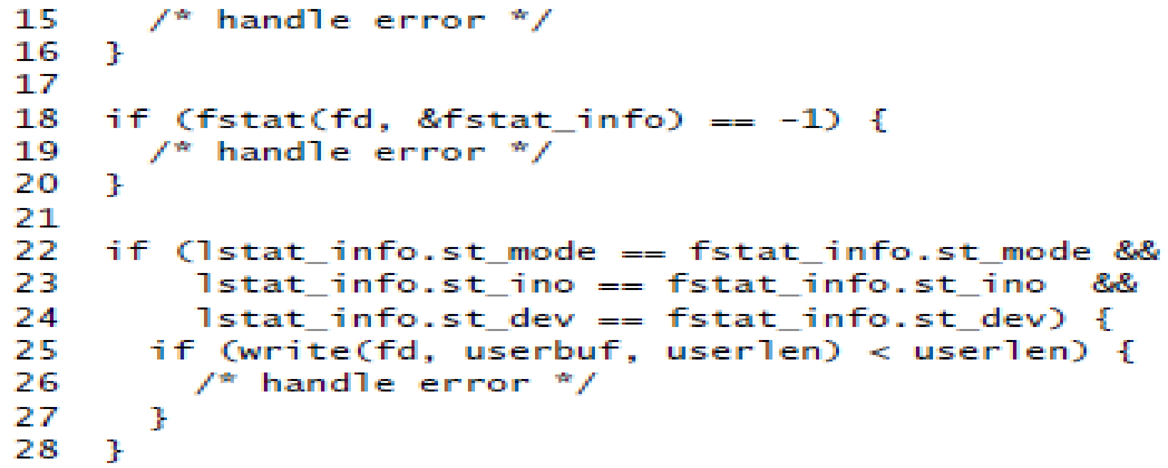
Closing the Race Window

E. Checking for Symbolic links



Example 8.18 Detecting Race Condition with Check-Use-Check

```
char *filename = /* file name */;
char *userbuf = /* user data */;
unsigned int userlen = /* length of userbuf string */;

struct stat lstat_info;
struct stat fstat_info;
int fd;
/* ... */
if (lstat(filename, &lstat_info) == -1) {
  /* handle error */
}

fd = open(filename, O_RDWR);
if (fd == -1) {
```



```
  /* handle error */
}

if (fstat(fd, &fstat_info) == -1) {
  /* handle error */
}

if (lstat_info.st_mode == fstat_info.st_mode &&
    lstat_info.st_ino == fstat_info.st_ino   &&
    lstat_info.st_dev == fstat_info.st_dev) {
  if (write(fd, userbuf, userlen) < userlen) {
    /* handle error */
  }
}
```

UNIT-5—File I/O—Mitigation Strategies

Eliminating the Race Object

If the shared object(s) can be eliminated or its shared access removed, there cannot be a race vulnerability.

✓ **Know What is Shared**

- In concurrent programming, sharing resources between threads or processes can lead to race conditions, where multiple execution flows interfere with each other. Key shared resources include system devices, the file system, and memory, and they must be carefully managed to avoid unpredictable behavior.

- Developers should minimize the use of global variables, system resources, and functions that rely on system settings, like ShellExecute(), to reduce vulnerabilities. Proper handling of file handles, environment variables, and memory, along with ensuring minimal access permissions, is essential to prevent race conditions and security issues.

UNIT-5—File I/O—Mitigation Strategies

Eliminating the Race Object

If the shared object(s) can be eliminated or its shared access removed, there cannot be a race vulnerability.

Use File Descriptors not File names

Once the file is opened , the file is not vulnerable to a symlink attack as long as it is accessed through its file descriptor.

Many file related race conditions can be eliminated by using

- ✓ rchown() rather than chown()
- ✓ fstat() rather than stat()
- ✓ fchmod() rather than chmod()

UNIT-5—File I/O—Mitigation Strategies

Eliminating the Race Object

If the shared object(s) can be eliminated or its shared access removed, there cannot be a race vulnerability.

Use File Descriptors not File names

Once the file is opened , the file is not vulnerable to a symlink attack as long as it is accessed through its file descriptor.

Many file related race conditions can be eliminated by using

- ✓ rchown() rather than chown()
- ✓ fstat() rather than stat()
- ✓ fchmod() rather than chmod()