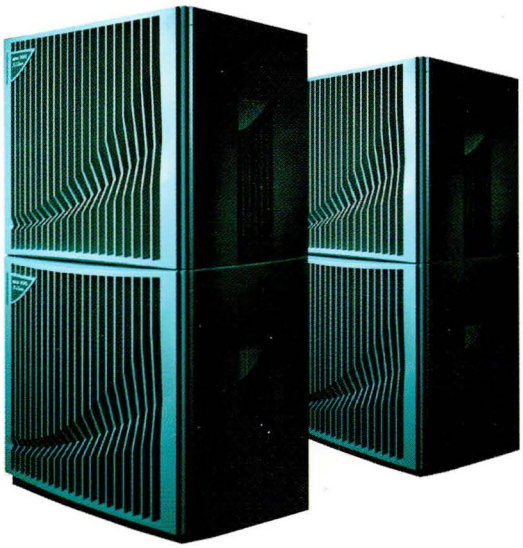
HEWLETT® PACKARD

Exemplar
S-Class and
X-Class Servers



SPP-UX Large Files User's Guide

Third Edition

SPP-UX Large Files User's Guide

Exemplar S-Class and X-Class Servers

B5655-90032

Third Edition

June 1997

Hewlett-Packard Company
Convex Division
Richardson, Texas
United States of America

SPP-UX Large Files User's Guide

Exemplar S-Class and X-Class Servers

B5655-90032

© Copyright Hewlett-Packard Company 1997. All Rights Reserved. Reproduction, adaptation, or translation without prior written permission is prohibited, except as allowed under the copyright laws.

Notice

The information contained in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

This entire book is recyclable.

Printed in the United States of America

Revision Information for

SPP-UX Large Files User's Guide

Exemplar S-Class and X-Class Servers

Edition	Document No.	Description
Third	B5655-90032	Released June 1997 with SPP-UX V5.2.
Second	B5655-90006	Released January 1997 with SPP-UX V5.1.
First	700-060530-000	Initial release February 1996.

Contents

Figure 2

Table 7

Preface

Notational conventions

This section discusses notational conventions used in this book.

`Bold monospace`

In command examples, text shown in **`bold monospace`** identifies user input that must be typed exactly as shown.

`Monospace`

In paragraph text, `monospace` identifies command names, system calls, and data structures and types.

In command examples, `monospace` identifies command output, including error messages.

In command syntax diagrams, text shown in `monospace` must be typed exactly as shown.

Italic

In paragraph text, *italic* identifies new and important terms and titles of documents.

In command syntax diagrams, *italic* identifies variables that must be supplied by the user.

Notes and cautions

This document presents notes and cautions in the following formats.

Note

A Note highlights supplemental information.

Caution

A Caution highlights information necessary to avoid damage to the system.

Associated documents

Using large files may require information not specific to the tasks described in this document.Useful related information is located in the following documents:

- For more information on programming Exemplar systems, refer to the *Exemplar C and Fortran 77 Programmer's Guide* (B5600-90002).
- For general information on using SPP-UX, refer to the *Exemplar User's Guide* (B5655-90003).
- For general information on administering SPP-UX, refer to the *SPP-UX System Administration Guide* (B5655-90023).
- For additional support documentation, refer to the *Guide to Exemplar Documentation* (B5655-90001).

Ordering documents

To order additional copies of this document, send requests to:

Hewlett-Packard Company
Convex Division
Customer Service
P.O. Box 833851
Richardson TX 75083-3851 USA

Please include the order number (xxxxx-9xxxx) or the exact title of the document.

Technical assistance

If you have questions that are not answered in this book, contact the Hewlett-Packard Convex Technical Assistance Center (TAC) at the following locations:

Within the continental U.S., call 1 (800) 952-0379.

From Canada, call 1 (800) 345-2384.

All other locations, contact the local Hewlett-Packard office.

You can also use the `contact` utility, if you would like to report any problems you may have with SPP-UX or its associated documentation. For more information refer, to the contact(1) man page.

Working with large files

1

This chapter covers large files concepts and instructions, including:

- What are large files?
- File systems and large files
  - Explains factors of your file system environment that are important when working with large files
  - Explains how to find out if your file system supports large files
- SPP-UX utilities and large files
  - Explains conditions that apply to utilities in a large files environment
  - Discusses limitations of certain functions; information about the limitations is located both in the section where the function is described and in the section "Limitations summary."

What are large files?

In SPP-UX, the term *large files* refers to files that are greater than 2^{31} - 1 bytes in size (approximately 2 gigabytes). The current upper limit on the size of a large file is one terabyte minus 512 bytes. (One terabyte is 2^{40} bytes.) SPP-UX V3.0 (and later) allows you to create and manipulate large files.

Use the `sysinfo` command to determine what version of SPP-UX you are using.

Some utilities and applications cannot properly process large files. Refer to the section "Utilities you can use with large files." for a list of utilities that have been modified to function properly on large files.

File systems and large files

File systems are structures that computers use to organize and store information.

Each file is connected or related to another file, starting with the root (/) directory file and continuing down through an unlimited number of files and directories.

The term *file system* can refer to the entire file hierarchy or to a subsection of the file tree. In this document, the term refers to a subsection of the file tree (a collection of files, directories, and file management structures that is assigned by your system manager to a certain portion of the disk system).

In SPP-UX, the file system is hierarchical, as illustrated in Figure 1.

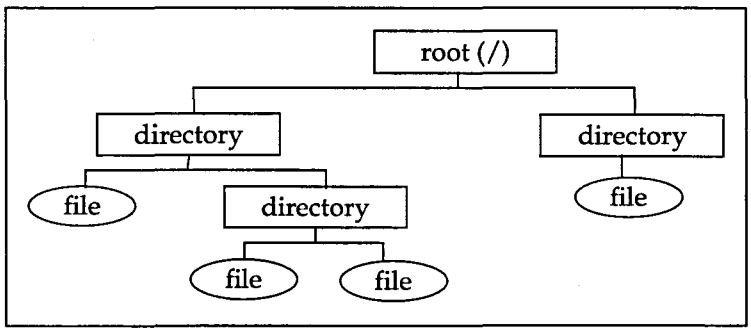


Figure 1 File system hierarchy

Determining if your file system supports large files

Before you create or manipulate a large file, make sure your file system supports large files by checking the following:

1. You are *not* in an NFS-mounted file system.
2. The file system has the magic number/featurebit pair indicating support for large files.

The steps for determining large file support are shown below.

Step 1 Check that you are *not* working in an NFS-mounted file system.

Large files are not supported on NFS file systems. The currently supported NFS protocol limits the size of files you can access via NFS to two (2) gigabytes or less.

The Network File System (NFS) allows you to mount file systems on various machines enabling you to access files across all the mounted file systems. You access standard, non-large, files in NFS-mounted file systems similarly to how you access files mounted directly on a specified machine.

To find out if you are working in an NFS-mounted file system, use the `bdf` command to report file system information:

a. Enter `bdf .` (sample output from `bdf` is shown below).

```
% bdf .
File system      kbytes      used      avail      capacity      Mounted on
/dev/st3        4140152   2980928     745232         80%        /test
```

b. If the "File system" column of the `bdf` output starts with `/dev/xxx` (where *xxx* represents a three-character string), you are in a system that can support large files. Thus, in the example above, the file system does support large files, assuming the file system has the correct magic number/featurebit pair.

```
% bdf .
File system      kbytes      used      avail      capacity      Mounted on
fantasia:/home    249138    102512    121712         46%        /rmt/fantasia:/home
```

If, however, the "File system" column contains *machinename*, you are working in a file system that is NFS-mounted from remote host *machinename*. If this is the case, you cannot create or manipulate large files in this file system. In the example above, the machine `fantasia` is listed in the "File System" column indicating that large files are not supported on `fantasia`.

Step 2 Determine if the file system has the appropriate magic number/featurebit pair which indicates support for large files. This is accomplished by using the `cnx_dumpfs` command as shown below.

a. Enter

`% cnx_dumpfs -l` *file_system_mount_point*

file_system_mount_point is the directory given in the "Mounted on" column in the `bdf` output.

Sample output follows:

```
magic    FD_MAGIC_2
featurebits    FSF_LARGEFILES
```

Use the `statfs()` system call to check for large file support from within a C or Fortran program. Additional information on statfs may be obtained in the section herein, titled "Checking large file support with statfs()."

b. Verify that the magic number is `FD_MAGIC_2` and that the featurebits field contains `FSF_LARGEFILES`.

If the magic number and feature bits fields do not indicate that large files are supported on your file system, contact the system administrator for the computer in question.

File system considerations

Although the stated limit for large files is currently one terabyte minus 512 bytes, some combinations of the items

- Block size
- Number of free blocks

can make it impossible to write a file that large. On file systems with block sizes less than 32 kilobytes, SPP-UX uses triple-indirect block pointers to write large files at the maximum size. In such a file system (with block size less than 32 kilobytes), the availability of block pointers determines the maximum file size you may write.

Use the `dumpfs` command to determine the block size on your system. See the dumpfs(1M) man page for more information.

The optimal block size is 64 kilobytes; this block size produces the greatest number of block pointers.

Consult your system administrator if you are interested in having a system's block size changed.

SPP-UX utilities and large files

Some SPP-UX utilities work on large files in the same manner as they do on regular files. Other utilities place restrictions on large file operations. The following sections explain how some of the SPP-UX utilities interact with large files.

Utilities you can use with large files

Not all utilities are practical for use with large files. For example, you are not likely to try to examine a large file using `more` because the size would make viewing the file unmanageable and make finding anything in it difficult. The SPP-UX utilities most useful with large files have been modified to work with large files. These utilities are:

- `cat`
- `chmod`
- `cp`
- `dd`
- `dump`
- `dumpfs, cnx_dumpfs`
- `find`
- `fbackup`
- `frecover`

- `fsck`
- `fsclean`
- `fsirand`
- `ftp`
- `ls`
- `mkfs, cnx_mkfs`
- `mount`
- `mv`
- `ncheck`
- `newfs, cnx_newfs`
- `restore`
- `rcp`
- `rm`
- `tail`

Use of all utilities is subject to the file system restrictions described in the section "File systems and large files."

Note

If you attempt to use any utility to place a large file in a file system where it is not allowed, your file may be truncated. Only the first two gigabytes of a large file will be captured, and the remaining data will be lost.

Using pipes with large files to work around shell limitations

The `sh` and `csh` shells and their variants cannot directly manipulate large files. This means that functions such as shell redirection do not work for large files.

You can use pipes with large files. Using pipes in combination with various utilities, you can replicate many shell functions as shown in Table 1.

Table 1 Using pipes to work around shell limitations

Limitation	Use	In place of
Using the shell to write large files	`% cat fname1 | dd of=fname2`	`% cat fname1 > fname2`
Using utilities that must be fed large files through pipes	`% cat fname | grep` *pattern*	`% grep` *pattern* `fname`

The second row of the table demonstrates how you can use pipes to feed large files to utilities that otherwise could not handle them. This method works with most utilities.

For more information on shells, please refer to Hewlett-Packard's *Using HP-UX* and *A Beginner's Guide to HP-UX*, each of which contains several chapters on how to use shells.

Backing up and restoring files

You can use

- `dump`
- `restore`
- `fbackup`
- `frecover`

to backup and restore file systems. Use these utilities with large files in the same manner as you use them with regular files.

The file system utilities noted below have limited usefulness with large files of eight gigabytes or greater:

- `tar`
- `pax`
- `cpio`

These utilities have a limit of 11 octal digits built into their format; you can use them to back up files up to eight gigabytes in size.

If any of these utilities (`tar`, `pax`, `cpio`) encounters a file larger than eight gigabytes, it issues a warning and skips that file.

Detecting and filling holes in large files

A *hole* is a region of a file that has not been written to but has been bypassed with the `lseek()` or `lseek64()` system call. Knowing about the holes in your files (and how to maintain or fill these holes) allows you to use disk space more effectively.

A hole is created, for example, when a new file is opened. The first 512 kilobytes (0 through 511) are skipped, then a string is written starting at kilobyte 512. The presence of the hole is recorded, but its content is not stored on disk, and it does not occupy any disk space.

A large file that contains a hole may fit on a particular file system, whereas a file of the same size without the hole may not fit. The utilities `cp` and `mv` fill in holes under some circumstances. The following sections explain how you can detect and fill holes.

Detecting holes in files

You can detect holes by comparing the output from `ls` and from `du`.

Suppose, for example, that the file myfile has a size of 10 megabytes as reported by `ls` but is only taking up 32 kilobytes of actual disk space, according to `du`:

```
% ls -l myfile
-rw------- 1 joe 10485760 Sep 10 15:26 myfile
% du myfile
32        myfile
```

This discrepancy occurs because myfile contains one or more holes.

Filling holes in files

If you copy or move the file myfile to another directory, the holes are filled with zeros, and disk usage corresponds to actual file size:

```
% cp myfile bigfile.with.holes.filled
% ls -l bigfile.with.holes.filled
-rw------- 1 joe 10485760 Sep 10 15:30 bigfile.with.holes.filled
% du -a bigfile.with.holes.filled
10240     bigfile.with.holes.filled
```

The next section "Preventing hole-filling." explains how to maintain holes when copying files.

Preventing hole-filling

Use `cp` with the `-z` option to preserve the holes in large files (and consequently, reduce disk usage) when copying:

```
% cp -z mtfile bigfile.with.holes.preserved
```

Limitations summary

The following list summarizes the conditions you should keep in mind when working with large files:

- Not all utilities and applications can properly process large files.
- The file system's magic number must be `FD_MAGIC_2`, and the system's featurebits field must contain `FSF_LARGEFILES` on *all* file systems where you are using large files.
- The currently supported NFS protocol limits the size of files you can access via NFS to at most two gigabytes.
- The `sh` and `csh` shells cannot directly manipulate large files; however, there are workarounds.
- You may unknowingly fill holes in large files when copying or moving files. (Hole-filling can result in inefficient use of disk space.)
- If `tar`, `pax`, or `cpio` encounters a file larger than eight gigabytes, it issues a warning and skips that file.
- If you attempt to use any utility to place a large file in a file system where it is not allowed, your file may be truncated.

Programming with large files 2

This chapter covers:

- Fortran support for large files
- C support for large files
  - Include file values
  - System calls
  - Unsupported system calls
- Large file support from within a program
- File system considerations
- Large file use with existing programs

Programming with large files in Fortran

Exemplar Fortran supports large files—large formatted and unformatted files can be read and written. The Fortran I/O interface extends transparently to large files.

Direct access files are limited to 2^{31}-1 RECORDs. The actual size of such a file is then limited by the size of individual records. This restriction is due to the use of the INTEGER*4 data type for the following specifiers:

- REC= specifier in READ and WRITE statements for direct access files
- NEXTREC= specifier in INQUIRE statements

You must relink Fortran programs with Exemplar Fortran in order to read and write large files. All Fortran programs linked with the Fortran libraries have large file access. If you do not have access to Exemplar Fortran V9.3 (and later), contact your system administrator.

Programming with large files in C

Exemplar C supports large files. The C I/O interface requires you to specify certain include file values and system calls to properly handle large files. The following sections:

- List and describe the needed include files
- Describe how to prepare large files for reading and writing
- List and describe the necessary system calls

Include file values

The following include files contain definitions and prototypes that are useful in programming with large files:

<sys/cnx_fcntl.h>

Defines the `O_LARGEFILE` flag, `struct flock64`, `F_GETLK64`, `F_SETLK64`, and `F_SETLKW64`; includes prototypes of `creat64()` and `open64()`.

<sys/cnx_fs.h>

Defines new magic number of a file system that allows large files (`FD_MAGIC_2`); defines the large file feature bit (`FSF_LARGEFILES`).

<sys/cnx_ino.h>

Redefines `di_size` to be correct for an inode from a large-file-aware file system.

Note

<sys/ino.h> must be included before <sys/cnx_ino.h>

<sys/cnx_inode.h>

Defines the inode structure that is used on a large-file-aware file system.

Note

<sys/inode.h> should not be included with <sys/cnx_inode.h>; (the only difference between the two files is in the way `ic_size` is defined)

<sys/cnx_stat.h>

Defines `struct stat64`; includes prototypes for `stat64()`, `fstat64()`, and `lstat64()`.

<sys/cnx_types.h>

Defines `off64_t`, `fblkkcnt64_t`, and `fsblkcnt64_t`; `off64_t`, which is 64 bits long, defines possible offsets (and maximum size) of a file.

<sys/cnx_unistd.h>

Includes prototypes for `lseek64()`, `ftruncate64()`, and `truncate64()`.

Preparing large files for reading and writing in C

When programming with large files, you have to set the `O_LARGEFILE` (or large file) flag to allow a program to read or write beyond the two-gigabyte boundary in a file. Use one the options below to set the `O_LARGEFILE` flag.

- Setting the `O_LARGEFILE` flag when opening a file, as shown in the following examples:

  ```
  fd=open("largefilename", O_RDWR | O_LARGEFILE);
  ```

  or

  ```
  fd=fopen("largefilename", "rwl");
  ```

  or

  ```
  fd=open64("largefilename", O_RDWR);
  ```

- Setting the `O_LARGEFILE` flag by using `fcntl()`:

  ```
  fd=open("largefilename", O_RDWR);
  flag=fcntl(fd, F_GETFL, 0);
  fcntl(fd, F_SETFL, flag|FLARGEFILE);
  ```

  You must use `fcntl` properly when working with large files; shortcut methods may lose the `O_LARGEFILE` flag. For example, the following call structure loses the `O_LARGEFILE` flag:

  ```
  fcntl(fd, F_SETFL, FASIO);
  /* Do not use the call structure above */
  ```

  However, the call structure below maintains the large file flag setting:

  ```
  flag=fcntl(fd, F_GETFL, 0);
  flag|=FASIO;
  fcntl(fd, F_SETFL, flag);
  ```

System calls for use with large files

The following system calls are available for use with large files:

- `creat64()`
- `fcntl()`
- `fgetpos64()`
- `fopen()`
- `fseek64()`
- `fsetpos64()`
- `fstat64()`
- `ftell64()`
- `ftruncate()`
- `ftruncate64()`
- `lseek()`
- `lseek64()`
- `lstat64()`
- `open()`
- `open64()`
- `read()`
- `stat64()`
- `truncate()`
- `truncate64()`
- `write()`

Each function with a name ending in 64 (indicating the function is a 64-bit function) has a 32-bit counterpart whose name is the same except for the ending 64 (`lseek64()`, `lseek()`; `fseek64()`, `fseek()`; etc.). Except as noted below, the 64-bit function has the same functionality as its 32-bit counterpart, but takes a 64-bit offset.

The list below describes the behaviors of various system calls with respect to large files. These system calls and libraries are located in the static library /usr/lib/libc.a and in the shared library /usr/lib/libc.sl.

`creat()` (Unsupported for large files)

Use `creat64()` in place of `creat()`.

The `creat()` system call is equivalent to

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

This equivalence causes `creat()` to fail if the file referenced by *path* exists and is a large file.

`creat64()`

The `creat64()` library call is equivalent to

`open(`*path*`, O_CREAT|O_TRUNC|O_WRONLY|O_LARGEFILE,` *mode*`);`

Unlike the `creat()` system call, `creat64()` successfully truncates and opens a large file.

`fcntl()`

Use the `fcntl()` system call to set or clear the `O_LARGEFILE` flag using the `F_SETFL` command. Use the `F_GETFL` command to get the current state of the `O_LARGEFILE` flag and of other flags you can set or clear.

`fcntl()` allows the `O_LARGEFILE` flag to be set and cleared regardless of the size of the open file.

`fgetpos64()`

Get stream position as a 64-bit offset.

`ftruncate(), ftruncate64()`

To use `ftruncate()` successfully on large files, make sure the `O_LARGEFILE` flag is set. If the flag is clear, the operation fails and returns -1 with errno set to EOVERFLOW.

`ftruncate64()` is the same as `ftruncate()`, but uses a 64-bit offset. Like `ftruncate()`, `ftruncate64()` requires the `O_LARGEFILE` flag to be set in order to truncate a large file.

fseek64()

Seek on a stream using 64-bit offsets. Implemented with `lseek64()`.

fsetpos64()

Set stream position as a 64-bit offset.

ftell64()

Returns stream position as a 64-bit offset.

lseek(), lseek64()

The `lseek64()` system call behaves like the `lseek()` system call except that the offset argument and returned value are defined as type `off64_t` instead of `off_t`. This allows `lseek64()` to take file offsets larger than `lseek()` and return any resulting large values.

If the resulting offset of an `lseek()` operation is legal (that is, less than the maximum file size), but too large to be represented in the return value (for example, a signed 32-bit integer for `lseek()`), the kernel returns -1 and sets errno to EOVERFLOW.

open()

The `open()` system call fails to open a large file unless the `O_LARGEFILE` flag is specified in the flags field. In addition to the normal errors, the `open()` system call returns -1 and sets errno to EOVERFLOW if the file specified by the *path* argument exists, is a large file, and `O_LARGEFILE` was not specified.

When `O_TRUNC` is specified without `O_LARGEFILE`, open does not truncate a large file, but returns -1 and sets errno to EOVERFLOW.

open64()

Open a file, with `O_LARGEFILE` bit implied.

read()

Regardless of the maximum file size—which is determined by the setting of the O_LARGEFILE flag (see fcntl())—the read() system call does not read data at a file offset greater than the size of the file. In this case, -1 is returned and errno is set to EINVAL.

To illustrate this idea, consider the following three cases:

- If the current file offset is equal to the size of the file, a read requesting at least 1 byte of data returns a value of 0 (no bytes read) and leaves errno unchanged.
- If the current file offset is greater than the size of the file, a read requesting at least 1 byte of data returns -1 and sets errno to EINVAL.
- If the size of a file is greater than or equal to 2 gigabytes and the O_LARGEFILE flag is clear, then the read() system call behaves as if the file was 2 gigabytes - 1 byte in size, and therefore returns EINVAL.

stat(), lstat(), fstat() (Unsupported for large files)

Use stat64(), lstat64(), and fstat64() for large files in place of these calls.

The stat(), lstat(), and fstat() system calls fail if the specified file is a large file (larger than 2 gigabytes - 1 byte in size), because the size of the file cannot be represented in a signed 32-bit integer. In this case, stat(), lstat(), and fstat() return -1 and set errno to EOVERFLOW.

stat64(), lstat64(), fstat64()

The stat64(), lstat64() and fstat64() system calls use the stat64 structure instead of the stat structure to return information about a file. The main difference between these two structures is that the st_size field in the stat64 structure is defined to be of type off64_t instead of off_t.

This allows the stat64(), lstat64() and fstat64() system calls to return information for files larger than 2 gigabytes - 1 byte in size.

Other than this difference, stat64(), lstat64() and fstat64() behave exactly as their 32-bit counterparts.

truncate(), truncate64()

The truncate() system call truncates large files. The truncate64() system call redefines the length argument to be an off64_t.

`write()`

If a write requests that more bytes be written than there is room (for example, the maximum file size is exceeded or the storage medium is full), only as many bytes as there is room are written. For example, if there is only room for 20 bytes and a write requests that 512 bytes be written, only 20 bytes are written.

If the current file offset is equal to or greater than the maximum file size as determined by the setting of the `O_LARGEFILE` flag (see fcntl()), a write requesting at least 1 byte of data returns -1 and sets errno to EOVERFLOW.

Unsupported system calls

Because they require 32-bit file offsets, the following system calls do not work with large files:

- `mmap()`
- `getrlimit()`
- `setrlimit()`

These system calls do not adversely affect large files, but their usefulness with respect to large files is limited:

`mmap()` maps only the first two gigabytes of a file.

`setrlimit()` can limit file size with byte granularity only up to two gigabytes (which is less than the size of a large file). It may set the file size limit to `RLIM_INFINITY` to allow access to any size file.

Do not use `setrlimit` or `getrlimit`; `setrlimit()` cannot set, and `getrlimit()` cannot process, limits above two gigabytes.

Checking large file support with `statfs()`

Use `statfs()` to check large-file support from within a program.

Verify that in the output from `statfs()`, the magic number (f_magic) is `FD_MAGIC_2` and that the `FSF_LARGEFILES` bit (f_featurebits) is set. See the statfs(2) man page for more information.

If the magic number and feature bits fields do not indicate that large files are supported on your file system, contact the system administrator for the computer in question.

Using existing programs with large files

The behavior of existing programs with large files is determined by the behavior of the system calls those programs use. Recompiling an old program in a large file environment has no effect because it still uses the same system calls.

The danger in using an existing program with a large file is to the file, not the program. A program that does not understand large files cannot open a large file.

The following list explains how several system calls behave if they encounter a large file, assuming the programs that contain the system calls have not been modified in any manner that may facilitate working with large files. See the section "System calls for use with large files" for information on system calls that work properly with large files.

`lseek()`

Fails on a large file if the file offset is already greater than two gigabytes or if a successful seek would cause the offset to be greater than two gigabytes. It ignores the `O_LARGEFILE` bit.

`open()`

Fails when attempting to open a large file.

`stat()`

Fails when performed on a large file.

`truncate(), ftruncate()`

Truncates a large file at the two-gigabyte boundary.

These limitations also apply if a program that understands large files system calls another program that does not.

Index

B5655-90032