Large Files in Solaris: A White Paper

Solaris OS group

2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Part No.: 96115-001
Revision 1, March 1996

Please
Recycle

Contents

Introduction 1

‘The Large File Summit’ is an industry initiative to produce a common specification for support of files that are bigger than the current limit of 2GB on existing 32-bit systems. It details the modifications to X/Open’s Single UNIX Specification to support large files. SunSoft’s implementation is based on Draft 8, the latest specification, available at the time of writing this paper. We expect Draft 8 will be submitted to X/Open for standardization.

The intent of this document is two fold:

- It is an aid to Solaris programmers who would like to convert their applications to take advantage of 64-bit file offsets or to those who would like to make sure that their applications work properly in a large file environment. This information is also useful for developing new applications on Solaris.
- It highlights the implementation of the Large File Summit’s API in Solaris so that the users can understand the functionality offered.

This document does NOT describe an implementation of 64-bit Solaris.

It is assumed that readers have the basic knowledge of manipulating files and their offsets in an application so the question of method is not addressed. This document is meant to be used in conjunction with other supporting documentation such as the Solaris reference manual and the Large File Summit’s API specifications.

Chapter 2, ‘Terminology and Concepts’, introduces terminology and concepts which will be used in the following chapters.

Chapter 3, ‘Converting Applications’, suggests a number of ways an application can be made **aware** or **safe** in a large file environment.

Chapter 4, ‘Converting Libraries’, suggests how to convert a library to function in a large file environment and provide explicit **64-bit interfaces**.

Chapter 5, ‘Implementation in Solaris’, highlights the SunSoft’s implementation of the Large File Summit’s API, and includes a list of utilities and libraries which are **aware** or **safe**.

Terminology and Concepts 2

2.1 Small file

A small file is a regular file whose size is less than **2GB** i.e. <= (2^{31} - 1) bytes.

2.2 Large file

A large file is a regular file whose size is greater than or equal to the former limit of **2GB** i.e. >= 2^{31} bytes.

2.3 Large file safe

An application is called **large file safe** if it causes no data loss or corruption when it encounters a large file. In other words, it need not properly process a large file but it has the appropriate logic to handle errors detected by the file manipulating functions.

2.4 Large file aware

An application is called **large file aware** when it has been converted so that it can process large files with the same functionality it has when processing small files. It must be able to handle large files as input and generate large files as output. Note that this functionality is relevant to a class of files that the application operates upon and does not necessarily pertain to all file access

within that program. For example, an application may be large file aware with respect to its data files, but only large file safe with respect to its configuration files, as they are not likely to grow to large file size.

2.5 Large file summit (LFS) specification

The LFS has produced the **Large File Support Specification**. Draft 7, published on 12/15/95, is robust enough for the purposes of implementation. See the LFS URL- http://www.sas.com/standard/large.file/ for more details on the LFS proceedings.

2.5.1 Extension to the current API

The LFS has extended the behavior of the **current interfaces** to handle errors correctly when an action cannot be performed or an attribute correctly represented for a large file. For example `open()` will set `errno` to **EOVERFLOW** when it encounters a regular file whose size is greater that or equal to **2GB**.

2.5.2 Transitional API

The LFS has defined a transitional API, **64-bit functions** and types, whose interfaces provide correct access to both large and small files. This specification provides a new function named `xxx64()` for each function named `xxx()` that passes or returns file offsets and a new type named `xxx64_t` for each type named `xxx_t` that is related to file offsets. For example, the LFS specification defines `open64()` and `off64_t` to deal with large files.

2.6 Compilation environment

2.6.1 Regular compilation environment

This is the existing compilation environment where all `xxx()` source interfaces map to `xxx()` calls in the resulting binary. All the interfaces assume small files or 32-bit offsets and return appropriate errors when presented with large files. This environment is sufficient for making an application large file **safe**.

2.6.2 Transitional compilation environment

The transitional compilation environment exports all the explicit 64-bit functions and types in addition to all the regular 32-bit functions and types. For example, both `xxx()` and `xxx64()` functions are available to the program source. An application must use `xxx64()` in order to manipulate large files. In this environment, a program has a choice of using `xxx()`, `xxx64()`, or both interfaces.

Setting **_LARGEFILE64_SOURCE** to **1** before including any system headers enables 64-bit transitional interfaces.

2.6.3 'Large file' compilation environment

In this environment, all `xxx()` source interfaces will map to `xxx64()` calls in the resulting binary. This facility also ensures that POSIX data types will be defined to be the correct size (i.e. `off_t` will be typedef'd to be a `long long` 64-bit entity). A program compiled in this environment will be able to use the `xxx()` source interfaces to access large files rather than having to explicitly utilize the transitional `xxx64()` interface calls. In this environment, a program can only use `xxx()` which manipulates both small and large files.

Setting **_FILE_OFFSET_BITS** to **64** before including any system headers enables 64-bit interfaces.

Not defining this macro or setting it as to **32** will result in the regular compilation environment as discussed above.

2.7 open() protection

To prevent accidental data loss, the `open()` system call will fail on a large file in the regular compilation environment. Before a file is opened, it's size will be verified to not be greater than the maximum offset representable by a 32 bit value. A successful `open()` will set the new **offset maximum** field in the open file descriptor to be the largest possible file size that can be successfully manipulated in the targeted environment. Other calls, such as `read()` and `write()`, respect the offset maximum value and refuse to venture beyond it. This facility is an integral component to making a program large file safe.

Converting Applications 3

Applications that manipulate files descriptors or offsets need to be examined for correct behavior in the large file environment. When porting an application to the large file environment, there are essentially three types of source code modification which may take place. These map directly to one of three desired operating modes:

- Large file safe
- Large file aware
- Mixed mode operability, where the program utilizes both **small** and **large** file descriptions.

In an effort to make the issues most easily identifiable, the three modes are considered individually below.

3.1 Making an application large file safe

Utilities, tools or libraries, which may encounter large files but should not process them need to be made safe. For example, *lp* should fail gracefully when given a large file to print.

3.1.1 Compilation environment

Ensuring that an application is large file safe does **not** require any special feature test macros. The source should be compiled as usual. No modifications to the makefiles are necessary.

3.1.2 Source changes

The process of making an application large file safe involves examining the source and ensuring that the existing interfaces behave appropriately (e.g. have the correct error handling facilities) should they encounter a large file.

Traditionally, `stat()` and `open()` have returned errors when a file did not exist or the specified path was mangled. The LFS specification requires **EOVERFLOW** or **EFBIG** to be returned in `errno` when the called function cannot correctly manipulate or represent some attribute of a large file, such as size or offset information that exceeds the 32-bit limitation.

Usually, an application first encounters a large file during `open()/fopen()` or `lstat()/stat()/fstat()` and fails at that point. But in the cases where a program inherits a large file descriptor via `fork()` or `exec()`, it can fail during functions like `lseek()/fseek()` or `mmap()`.

The general philosophy for ensuring that a program is safe is to:

- look at all places where the program obtains file descriptors
- handle **EOVERFLOW** or **EFBIG** errors appropriately
- ensure that the program recovers or exits gracefully

For example:

```
lseek(fd, offset, 0);
```

or

```
if (lseek(fd, offset, 0) < 0) {
   /* No error handling for large files */
}
```

may require changing to:

```
if (lseek(fd, offset, 0) < 0) {
   if (errno == EOVERFLOW) {
      /* Handling for large files */
}
```

3.2 Making an application large file aware

Utilities, tools or libraries that process large files need to be made large file aware. For example, *cp* should be able to copy a large file.

3.2.1 Compilation environment

The preferred method for achieving large file awareness at the source level is to utilize the large file compilation environment option (see section 2.6.3).

Pass -**D_FILE_OFFSET_BITS=64** to the compiler in your makefiles.

Setting **_FILE_OFFSET_BIT**S to **64** will enable a new `#pragma`, which causes the compiler to map the current `xxx()` source level interfaces to their corresponding `xxx64()` binary entry points. This will also typedef types in all relevant system headers to the correct size. As an example, consider `off_t`. It is currently typedef'd in <sys/types.h> to be a `long`, but when **_FILE_OFFSET_BIT**S is set to **64**, it will be typedef'd to be a `long long`.

Compilers which do not have the new `#pragma` feature will map `xxx()` to `xxx64()` using `#define`. See section 5.2 for an example.

If **_FILE_OFFSET_BIT**S is set to **32** or if it is undefined, the mapping will not occur.

3.2.2 Source changes

Porting existing applications to the large file environment mostly involves cleaning up code which contains undetected type clashes. Frequently a fundamental type of similar size has been used instead of the variable's defined type. Also, much code has never been updated to reflect standards such as POSIX.

Data types that have been extended to include the 64-bit version are listed below:

`ino_t`	file serial number
`off_t`	relative file pointer offsets and file sizes
`fpos_t`	represent uniquely every position within a file
`rlim_t`	used for resource limit values
`blkcnt_t`	number of disk blocks
`fsblkcnt_t`	file system block counts
`fsfilcnt_t`	file system inode counts

3.2.2.1 Correct interface parameter and return value type definitions

The code should be examined for interfaces where return values and constants were not declared to be one of these types. These declarations must be manually edited and should be changed to the appropriate POSIX data type.

For example, an instance of `lseek()` to return the current value of the file offset pointer for an fd, might have been coded as follows:

```
long curpos;
curpos = lseek(fd, 0L, SEEK_CUR);
```

In large file environment this would truncate the returned 64-bit offset to 32-bit, which might lead to data corruption later on. The code should be changed to:

```
off_t curpos;
curpos = lseek(fd, (off_t)0, SEEK_CUR);
```

3.2.2.2 Variables not directly involved in interface calls

Any variables used as counters or for temporary storage of sizes or offset information must of the correct type to avoid truncation. There is no sure-fire method of automating this process, so diligent perusal of the code is required. Care must be taken to declare all the variables in an expression to be of the same type. For example,

```
int delta;
long curpos;
....
curpos = lseek(fd, 0L, SEEK_CUR);
curpos =+ delta;
```

Not only `curpos` should be changed to `off_t`, but delta should also be changed to `off_t`. Correctly modified code is shown below.

```
off_t delta;
off_t curpos;
....
curpos = lseek(fd, 0L, SEEK_CUR);
curpos =+ delta;
```

Use of the POSIX type definitions is preferable for declarations.

3.2.2.3 Output and in-memory formatting strings

Any output or in-memory formatting strings utilized in reference to the large file sizing entities must be converted. A formatting string for an offset may in the current environment look like "%ld". In the new environment, it must be converted to "%lld" to accommodate value of type long long. Additionally, if any byte count information accompanies the format, it must also be modified to accommodate larger potential values.

For example:

```
off_t offset;
printf(" %7ld", offset);
```

should be modified to:

```
off_t offset;
printf(" %7lld", offset);
```

If the compilation environment may be set for either small files or large files, it is safest to place #ifdef around the format string, such as:

```
off_t offset;
#if _FILE_OFFSET_BITS - 0 == 64
   printf(" %7lld", offset);
#else
   printf(" %7ld", offset);
#endif
```

3.2.2.4 Conversion routines

It may be the case that variables representing size or offset information are being converted from string values and vice-versa. It is likely that the code would use routines such as atol(), strtol(), to derive offset information. Similar routines handle long long values, and these values should be used to convert any size and/or offset information when programming for a large file environment. Again, if the compilation environment is unknown, it is safest to place #ifdef around the conversion routines as in the example below:

```
off_t offset;
#if _FILE_OFFSET_BITS - 0 == 64
   offset = atoll(argv[1]);
#else
   offset = atol(argv[1]);
#endif
```

3.3 Making a mixed mode application

Some users may wish to use the 64-bit version of functions and types explicitly so that an application may have both a small and a large files environment. An application may use `open64()` to manipulate data files, but use `open()` to handle a configuration file. In other words, this application can process the large data files but will detect an error on encountering a large configuration file.

3.3.1 Compilation environment

Use the transitional compilation environment as described in section 2.6.2.

Pass **-D_LARGEFILE64_SOURCE=1** to the compiler in your makefiles.

If **_LARGEFILE64_SOURCE** is set to **1**, 64-bit source level interfaces, of the form `xxx64()` and types such as `off64_t` will be visible along with their existing 32-bit counterparts.

3.3.2 Source changes

If mixed mode is chosen for an application, the source must be examined carefully and decisions made as to which of the functions and types need to be modified to handle large files. Essentially, this involves all the steps that are required to make an application large file **safe** as well as **aware**.

3.3.2.1 Explicit use of 64-bit version functions and types

The rules as discussed in the section 3.2.2 are applicable here also, except that 64-bit version functions and types are used. Below is a list of those steps:

- Correct interface parameter and return value type definitions

  For example, an instance of `lseek()` to return the current value of the file offset pointer for an `fd`, will be modified from:

```
long curpos;
curpos = lseek(fd, 0L, SEEK_CUR);
```

  to:

```
off64_t curpos;
curpos = lseek64(fd, (off64_t)0, SEEK_CUR);
```

- Variables not directly involved in interface calls

  Care must be taken to declare all the variables in an expression to the same type. For example following code should be changed from:

```
int delta;
long curpos;
....
curpos = lseek(fd, 0L, SEEK_CUR);
curpos =+ delta;
```

  to

```
off64_t delta;
off64_t curpos;
....
curpos = lseek64(fd, (off64_t)0, SEEK_CUR);
curpos =+ delta;
```

- Output and in-memory formatting strings

  Any output or in-memory formatting strings utilized in reference to the large file sizing entities must be converted. For example:

```
off_t offset;
printf(" %7ld", offset);
```

  should be modified to:

```
off64_t offset;
printf(" %7lld", offset);
```

- Conversion routines

  It is likely that the code is currently using routines such as `atol()`, `strtol()`, to derive offset information. Similar routines handle `long long` values, and those routines should be used to convert any size and/or offset information when programming for large file environment. The following code should be modified from:

```
off_t offset
offset = atol(argv[1]);
```

  to:

```
off64_t offset;
offset = atoll(argv[1]);
```

3.3.2.2 *Existing 32-bit version functions and types*

The remaining 32-bit versions of functions and types should be made large file safe. This involves examining the source and ensuring that the existing interfaces have correct error handling procedures should they encounter a large file. Refer to section 3.1.2 for details.

The general philosophy for ensuring that a program is safe is to:

- look at all places where the program obtains file descriptors
- handle **EOVERFLOW** or **EFBIG** errors appropriately
- ensure that the program recovers or exits gracefully

Converting Libraries 4

Libraries that manipulate files descriptors or offsets need to be examined for the correct behavior in the large file environment. When porting a library to the large file environment, there are essentially three types of source code modification which may take place:

- Making library functions large file safe
- Making library functions large file aware
- Creating a transitional 64-bit function to provide mixed mode operability, where the program utilizes both **small** and **large** file descriptions.

In an effort to make the issues most easily identifiable, the three modes are considered individually below.

4.1 Making a library function large file safe

Library functions which may encounter large files but are not intended to process them need to be made large file safe. For example, `fopen()` should fail gracefully when given a large file to open. If this is the case, the function code should be compiled as usual but the function code must be examined for large file error handling.

The steps involved to make a library function safe are the same as those described in the previous chapter in section 3.1- “Making an application large file safe”.

4.2 Making a library function large file aware

Library functions which may encounter large files and must process them need to be made large file aware. For example, `fopen64()` should be able to successfully return a FILE pointer when given a large file to open. If this is the case, then the function needs to be compiled in the large file compilation environment and the function code must be examined for correct return and parameter types.

The steps involved to make a function aware are the same as those described in the previous chapter in section 3.2- “Making an application large file aware”.

4.3 Creating a transitional 64-bit interface

Some library providers may wish to export the 64-bit version of functions and types explicitly so that a library may continue to provide both small and large file environments. A library may export `fopen64()` to manipulate large files, as well as the standard `fopen()` to handle a regular file.

Any function that takes or returns a 64-bit type must be large file safe and have a new transitional interface created. Transitional interfaces are created to allow applications to continue to use pre-existing code while converting to the large file environment.

4.3.1 An example

Lets take the example of `fopen()` in libc.so.1 in Solaris that needed to provide the 64-bit transitional `fopen64()` interface. There are two steps in creating such an interface.

1. Introduce a new function called `fopen64()` which can deal with large files successfully, and modify existing `fopen()` to return **EOVERFLOW** on encountering large files.

2. Modify the <stdio.h> header to include the new function prototype. The header file changes are done in such a way that in:
   - a regular compilation environment -

     Source symbol `fopen()` maps to binary symbol `fopen()`
   - a large file compilation environment -

     Source symbol `fopen()` maps to binary symbol `fopen64()`

- a mixed mode compilation environment. -

  Source symbol `fopen()` maps to binary symbol `fopen()`

  Source symbol `fopen64()` maps to binary symbol `fopen64()`

4.3.2 Header changes

A new compiler directive `#pragma redefine_extname` is used to change one symbol to another symbol during compilation. See section 5.2 “Compilation Environment” for details on this feature.

```
..
#if _FILE_OFFSET_BITS == 64    /* large file environment */
#ifdef _PRAGMA_REDEFINE_EXTNAME
#pragma redefine_extname fopen  fopen64
#else                          /* __PRAGMA_REDEFINE_EXTNAME */
#define fopen  fopen64
#endif                         /* __PRAGMA_REDEFINE_EXTNAME */
#endif                         /* _FILE_OFFSET_BITS == 64   */
..
#if defined(__STDC__)
extern FILE  *fopen(const char *, const char *);
#ifdef _LARGEFILE64_SOURCE     /* explicit 64-bit interface */
extern FILE *fopen64(const char *, const char *);
#endif                         /* _LARGEFILE64_SOURCE */
#else                          /* __STDC__ */
extern FILE  *fopen();
#ifdef _LARGEFILE64_SOURCE
extern FILE  *fopen64();
#endif                         /* _LARGEFILE64_SOURCE */
#endif                         /* __STDC__ */
..
```

The following example of `off_t` in <sys/types.h> shows how to declare types which are to be made available in large file compilation environment as well as in mixed mode via explicit 64-bit type.

```
..
#if _FILE_OFFSET_BITS == 32
typedef long  off_t;       /* offsets within files */
#elif _FILE_OFFSET_BITS == 64
typedef longlong_t  off_t;  /* offsets within files */
#endif
#if defined(_LARGEFILE64_SOURCE)
typedef longlong_t  off64_t;/* offsets within files */
#endif
```

Implementation in Solaris 5

5.1 Overview

The main goals of this implementation are to:

- Provide large file support for all file systems that are capable of supporting large file semantics.I.e. UFS, NFS, swapfs, tmpfs, and CacheFS.
- Conform to the final API specified by the Large Files Summit. The LFS is going to submit its final specification to X/Open for standardization.
- Provide a compile time environment so that existing interfaces can be used to access large files.
- Convert utilities and libraries that are listed below.

5.1.1 Limitations

This implementation does NOT provide support for:

- large files other than regular unix files.
- mapping of character devices, because mapping these devices using `mmap()` will lead to ddi-dki interface breakage. However, we support reading and writing to these devices at offsets > 2GB and up to maximum value (1 terabyte).
- 64-bit block devices.

5.2 Compilation environment

A new compiler directive `#pragma redefine_extname` is used to change one symbol to another symbol during compilation.This changes the symbol table to use the new name wherever the original name is referenced. The effect is similar to `#define` except that it is not a source level replacement of all the occurrences of the string. For example, suppose `myfunc()` is an existing interface which is a 32-bit interface and has been made large file safe. Another interface, `myfunc64()`, is created to handle large files.

```
..
#if _FILE_OFFSET_BITS == 64    /* large file environment */
#ifdef _PRAGMA_REDEFINE_EXTNAME
#pragma redefine_extname myfunc  myfunc64
#else                          /* __PRAGMA_REDEFINE_EXTNAME */
#define myfunc  myfunc64
#endif                         /* __PRAGMA_REDEFINE_EXTNAME */
#endif                         /* _FILE_OFFSET_BITS == 64   */
..
extern off_t myfunc();
#ifdef _LARGEFILE64_SOURCE     /* explicit 64-bit interface*/
extern off64_t myfunc64();
#endif                         /* _LARGEFILE64_SOURCE */
..
```

When an application uses the declarations above and a program is compiled, there could be four possible scenarios:

Table 1: Compile environment

Compile environment	Flags defined during compilation	Source symbol exported	Maps to binary symbol
Existing environment	None	`myfunc`	`myfunc`
Large file environment	`_FILE_OFFSET_BITS = 64`	`myfunc`	`myfunc64`
Explicit 64-bit interface or Mixed mode environment	`_LARGEFILE64_SOURCE = 1`	`myfunc` `myfunc64`	`myfunc` `myfunc64`
Why_would_you_do_this? mode	`_FILE_OFFSET_BITS = 64` `_LARGEFILE64_SOURCE = 1`	`myfunc` `myfunc64`	`myfunc64` `myfunc64`

The new #pragma feature has been provided in SunSoft's latest compilers (ProCompilers 4.2) for Sparc, x86 and PowerPC. These compilers predefine the macro **__PRAGMA_REDEFINE_EXTNAME** to indicate that the new #pragma feature is supported. For applications built using other compilers (including older versions of Sun compilers) that do not support the new #pragma feature, the source level mapping takes place via #define.

Note that using #define may break applications that #undef a function name before passing the address of that function as a parameter to another function. The #pragma redefine_extname avoids this problem.

5.3 Extension of interfaces

5.3.1 Data types

Some of the data structures have been modified to correctly handle large files. The following is the list of structures and the changes.

Table 2: Large file sensitive data structures

Existing Definition	Modified/New Definition	Header File
	typedef long **blkcnt_t**; typedef ulong_t **fsblkcnt_t**; typedef ulong_t **fsfilcnt_t**;	<sys/types.h>
struct stat long st_blocks;	struct stat **blkcnt_t** st_blocks;	<sys/stat.h>
struct statvfs u_long f_blocks; u_long f_bfree; u_long f_bavial; u_long f_files; u_long f_ffree; u_long f_favail;	struct statvfs **fsblkcnt_t** f_blocks; **fsblkcnt_t** f_bfree; **fsblkcnt_t** f_bavial; **fsfilcnt_t** f_files; **fsfilcnt_t** f_ffree; **fsfilcnt_t** f_favail;	<sys/statvfs.h>
	RLIM_SAVED_MAX RLIM_SAVED_CUR	<sys/resource.h>
	_PC_FILESIZEBITS	<sys/unistd.h>

5.3.2 System interfaces

Many of the existing 32-bit system interfaces have been modified to handle the large files. These functions are listed below with their expected behavior.

Table 3: Large file sensitive interfaces

Existing Interface	Error	Remarks
`execl(), execv(), execle(), execlp(), execvp()`		the process's resource limits are copied to its saved resource limits
`fclose(), fflush(), fprintf(), fputc(), fputs(), fputwc(), fputws(), fseek(), fwrite(), printf(), putc(), putchar(), puts(), putw(), putwchar(), vfprintf(), vprintf()`	EFBIG	functions fail if either the output stream or the stream's buffer needs to flushed and the starting point is greater than or equal to the offset maximum established during open
`fcntl()`	EOVERFLOW	one or more of the values won't fit
`fgetc(), fgets(), fgetwc(), fgetws(), fread(), fscanf(), getc(), gets(), getw(), getwc(), getwchar(), scanf()`	EOVERFLOW	functions fail if data needs to be read and the starting point is greater than or equal to the offset maximum established during open
`fgetops()`	EOVERFLOW	`fpos_t` can not hold the current file position
`fopen(), freopen(), tmpfile()`	EOVERFLOW	`off_t` can not hold the size of the file
`fpathconf(), pathconf()`		added support for `FILESIZEBITS`
`fseek()`	EOVERFLOW	The file offset can not be stored in a **`long`**
`fstat(), lstat(), stat()`	EOVERFLOW	**`stat`** structure can not represent file size (`off_t st_size`), inode (`ino_t st_ino`) or block count (**`blkcnt_t`** `st_blocks`)
`fstatvfs(), statvfs()`	EOVERFLOW	**`statvfs`** structure can not represent total blocks (`fsblkcnt_t f_blocks`), free blocks (`fsblkcnt_t f_bfree`), available blocks (`fsblkcnt_t f_bavail`), total inodes (`fsfilcnt_t f_files`), free inodes (`fsfilcnt f_ffree`) or available inodes (`fsfilcnt_t f_favail`)

Table 3: Large file sensitive interfaces

Existing Interface	Error	Remarks
`ftell()`	EOVERFLOW	the file offset can not be stored in **`long`**
`ftruncate()`	EFBIG	if the `length` is greater than the maximum offset established at open
`getrlimit(), setrlimit()`		use of saved resource limits described
`lockf()`	EOVERFLOW	a lock offset can not be stored in `off_t`
`lseek()`	EOVERFLOW	The file offset can not be stored in `off_t`
`open(), creat()`	EOVERFLOW	`off_t` can not hold the file size
`mmap()`	EOVERFLOW	if `off + len` exceeds the maximum offset established at open
`read(), readv(), pread()`	EOVERFLOW	if trying to read a regular file beyond the maximum offset established at open
`readdir(), readdir_r()`	EOVERFLOW	**`dirent`** structure can not represent either inode number (`ino_t d_ino`) or offset (`off_t d_off`)
`write(), writev(), pwrite()`	EFBIG	if trying to write a regular file beyond the maximum offset established at open

5.4 New transitional interfaces

The interfaces, macros and data types in this section are explicit 64-bit instances of the standard API. The function prototype and semantics of a transitional interface are equivalent to those of the standard version of the call.

Setting **_LARGEFILE64_SOURCE** to **1** enables these interfaces.

5.4.1 Data types

The following table shows the standard data or struct types and their corresponding 64-bit types. Please refer to headers for more details.

Table 4: 64-bit extended definitions

Standard Definition	64-bit Definition	Header

struct **dirent**
  ino_t d_ino;
  off_t d_off;
	struct **dirent64**
	  ino64_t d_ino;
	  off64_t d_off;
		<sys/dirent.h>

struct **flock**
  off_t l_start;
  off_t l_len;
F_SETLK, F_SETLKW, F_GETLK, F_FREESP
	struct **flock64**
	  off64_t l_start;
	  off64_t l_len;
	F_SETLK64, F_SETLKW64, F_GETLK64, F_FREESP64, O_LARGEFILE
		<sys/fcntl.h>

fpos_t	**fpos64_t**	<sys/stdio.h>

rlim_t
struct **rlimit**
  rlim_t rlim_cur;
  rlim_t rlim_max;
RLIM_INFINITY,RLIM_SAVED_MAX, RLIM_SAVED_CUR
	rlim64_t
	struct **rlimit64**
	  rlim64_t rlim_cur;
	  rlim64_t rlim_max;
	RLIM64_INFINITY ,RLIM64_SAVED_MAX, RLIM64_SAVED_CUR
		<sys/resource.h>

struct **stat**
  ino_t st_ino;
  off_t st_size;
  blkcnt_t st_blocks;
	struct **stat64**
	  ino64_t st_ino;
	  off64_t st_size;
	  blkcnt64_t st_blocks;
		<sys/stat.h>

struct **statvfs**
  fsblkcnt_t f_blocks;
  fsblkcnt_t f_bfree;
  fsblkcnt_t f_bavial;
  fsfilcnt_t f_files;
  fsfilcnt_t f_ffree;
  fsfilcnt_t f_favail;
	struct **statvfs64**
	  fsblkcnt64_t f_blocks;
	  fsblkcnt64_t f_bfree;
	  fsblkcnt64_t f_bavial;
	  fsfilcnt64_t f_files;
	  fsfilcnt64_t f_ffree;
	  fsfilcnt64_t f_favail;
		<sys/statvfs.h>

off_t; **ino_t;** **blkcnt_t;** **fsblkcnt_t;** **fsfilcnt_t;**	**off64_t;** **ino64_t;** **blkcnt64_t;** **fsblkcnt64_t;** **fsfilcnt64_t;**	<sys/types.h>
	_LFS_LARGEFILE64, _LFS64_STDIO	<unistd.h>
	_CS_LFS64_CFLAGS, _CS_LFS64_LDFLAGS _CS_LFS64_LIBS, _CS_LFS64_LINTFLAGS	<sys/unistd.h>

5.4.2 System interfaces

The following table shows the standard API and the corresponding 64-bit interfaces. The interface name and the affected data types are shown in bold faces.

Table 5: 64-bit extended interfaces

Existing Interface	64-bit Definition	Header File
`struct` **`dirent`** `*`**`readdir`**`()`	`struct dirent64 *readdir64()`	<dirent.h>
`int` **`creat`**`()` `int` **`open`**`()`	`int creat64()` `int open64()`	<fcntl.h>
`int` **`ftw`**`(.., const struct` **`stat`** `*, ..)` `int` **`nftw`**`(.., const struct` **`stat`** `*,..)`	`int ftw64(.., const struct stat64 *, ..)` `int nftw64(.., const struct stat64 *, ..)`	<ftw.h>

`int` **`fgetpos`**`()`
`FILE *`**`fopen`**`()`
`FILE *`**`freopen`**`()`
`int` **`fseeko`**`(..,`**`off_t`**`,..) NEW`
`int` **`fsetpos`**`(.., const` **`fpos_t`** `*)`
`off_t` **`ftello`**`() NEW`
`FILE *`**`tmpfile`**`()`
	`int` **`fgetpos64`**`()`
	`FILE *`**`fopen64`**`()`
	`FILE *`**`freopen64`**`()`
	`int` **`fseeko64`**`(..,` **`off64_t`**`, ..)`
	`int` **`fsetpos64`**`(.., const` **`fpos64_t`** `*)`
	`off64_t` **`ftello64`**`()`
	`FILE *`**`tmpfilr64`**`()`
		<stdio.h>

`void` **`mmap`**`(..,` **`off_t`**`)`	`void` **`mmap64`**`(..,` **`off64_t`**`)`	<sys/mman.h >
`int` **`getrlimit`**`(.., struct` **`rlimt`** `*)` `int` **`setrlimit`**`(.., const struct` **`rlimit`** `*)`	`int` **`getrlimit64`**`(.., struct` **`rlimt64`** `*)` `int` **`setrlimit64`**`(.., const struct` **`rlimit64`** `*)`	<sys/resource.h>

`int` **`fstat`**`(.., struct` **`stat`** `*)`
`int` **`lstat`**`(.., struct` **`stat`** `*)`
`int` **`stat`**`(.., struct` **`stat`** `*)`
	`int` **`fstat64`**`(.., struct` **`stat64`** `*)`
	`int` **`lstat64`**`(.., struct` **`stat64`** `*)`
	`int` **`stat64`**`(.., struct` **`stat64`** `*)`
		<sys/stat.h>

`int` **`statvfs`**`(.., struct` **`statvfs`** `*)` `int` **`fstatvfs`**`(.., struct` **`statvfs`** `*)`	`int` **`statvfs64`**`(.., struct` **`statvfs64`** `*)` `int` **`fstatvfs64`**`(.., struct` **`statvfs64`** `*)`	<sys/statvfs.h>

`int` **`lockf`**`(..,` **`off_t`**`)`
`off_t` **`lseek`**`(..,` **`off_t`**`, ..)`
`int` **`ftruncate`**`(..,` **`off_t`**`)`
`int` **`truncate`**`(..,` **`off_t`**`)`
	`int` **`lockf64`**`(..,` **`off64_t`**`)`
	`off64_t` **`lseek64`**`(..,` **`off64_t`**`, ..)`
	`int` **`ftruncate64`**`(..,` **`off64_t`**`)`
	`int` **`truncate64`**`(..,` **`off64_t`**`)`
		<unistd.h>

5.5 Solaris utilities support

5.5.1 Large file aware utilities

adb	audioconvert	audioplay	audiorecord	awk
bdiff	cachefslog	cachefsstat	cachefswssize	cat
cfsadmin	cfsfstype	cfstagchk	chgrp	chmod
chown	cksum	clri	cmp	compress
cp	crash	csh	csplit	cut
dcopy	dd	df	du	edquota
egrep	ff	fgrep	file	find
fsck	fsdb	fsirand	fstyp	ftp
getconf	grep	head	in.ftpd	install
join	jsh	ksh	labelit	ln
lockfs	ls	makedbm	mkdir	mkfifo
mkfile	mkfs	mknod	more	mount
mv	mvdir	nawk	ncheck	newfs
page	paste	pathchk	pax	pg
quot	quota	quotacheck	quotaoff	quotaon
rcp	remsh	repquota	rksh	rm
rmdir	rsh	sed	sh	split
sum	swap	swapadd	tail	tee
test	touch	tr	truss	tunefs
umount	uncompress	volcopy	wc	zcat
/usr/ucb/chown	/usr/ucb/ln	/usr/ucb/ls	/usr/ucb/touch	

5.5.2 Large file safe utilities

accept	admind	cancel	comm	cpio
diff	diff3	diff3prog	diffh	diffmk
dircmp	disable	ed	enable	from
lp	lpadmin	lpfilter	lpforms	lpmove
lpr	lpsched	lpshut	lpstat	lpsystem
lpusers	mail	mailcompat	mailq	mailstats
mailx	pack	pcat	red	rmail
sdiff	sendmail	tar	unpack	uudecode
uuencode	vi	view		

5.6 Solaris library support

libTL	libadm	libaio	libauth	libbc
libbsm	libc	libcmd	libcrypt	libcurses
libdevinfo	libelf	libeti	libgen	libgenIO
libintl	libkrb	libkstat	libkvm	libmapmalloc
libnisdb	libnsl	libpkg	libplot	libposix4
libpthread	librac	libresolv	librpcsvc	libsocket
libthread	libthread_db	libtnf	libtnfprobe	libvolmgt
nametoaddr	nsswitch	scheme	ucblib/libcurses	ucblib/librpcsoc
ucblib/libucb	ucblib/libtermcap			

Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043
415 960-1300
FAX 415 969-9131

For U.S. Sales Office locations, call:
800 821-4643
In California:
800 821-4642

Australia: (02) 413 2666
Belgium: 32-2-759 5925
Canada: 416 477-6745
Finland: 358-0-502 27 00
France: (1) 30 67 50 00
Germany: (0) 89-46 00 8-0
Hong Kong: 852 802 4188
Italy: 039 60551
Japan: (03) 221-7021
Korea: 822-563-8700
Latin America: 415 688-9464
The Netherlands: 033 501234
New Zealand: (04) 499 2344
Nordic Countries: +46 (0) 8 623 90 00
PRC: 861-831-5568
Singapore: 224 3388
Spain: (91) 5551648
Switzerland: (1) 825 71 11
Taiwan: 2-514-0567
UK: 0276 20444

Elsewhere in the world,
call Corporate Headquarters:
415 960-1300
Intercontinental Sales: 415 688-9000