64 Bit File Access

Adam Sweeney

1.0 Introduction

In order to support the access of 64 bit files from 32 bit applications, new interfaces must be defined which take 64 bit parameters. These interfaces must be cleanly supported in the kernel, without the information of whether an application is 64 or 32 bit filtering down below the system call level.This document describes the new interfaces to be used to access 64 bit files from 32 bit applications, the policies involved in 64 bit file access, and the changes necessary to kernel internals to support 64 bit files.

It is important to understand that the need for these extensions is only to allow 32 bit applications to access 64 bit files. They are not needed by 64 bit applications or by 32 bit applications which have no need to deal with 64 bit files.

2.0 New Interfaces

2.1 New User Visible Types

To support the access of 64 bit files, the following new types will be exported for use by user programs.

2.1.1 off64_t

The off64_t type will be defined as “long long.” This will make it a 64 bit value useful for specifying 64 bit file offsets and sizes.

2.1.2 stat64

The new system calls will include extended stat(2) calls. They will all use the new stat64 structure.

TABLE 1. stat64 structure definition

field name	field type	comments
st_dev	dev_t	
st_pad1	long[3]	
st_ino	ino_t	
st_mode	mode_t	
st_nlink	nlink_t	
st_uid	uid_t	
st_gid	gid_t	

TABLE 1. stat64 structure definition

field name	field type	comments
st_rdev	dev_t	
st_pad	long[2]	
st_size	off64_t	now 64 bits
st_pad3	long	
st_atim	timestruc_t	
st_mtim	timestruc_t	
st_ctim	timestruct_t	
st_blksize	long	
st_blocks	long	
st_fstype	char[_ST_FSTYPSZ]	
st_pad4	long[8]	

2.1.3 flock64

In order to allow file record locking on large files, the fcntl(2) system call will be extended to accept a flock64 structure as defined below.

TABLE 2. flock64 structure definition

field name	field type	comments
l_type	short	
l_whence	short	
l_start	off64_t	64 bits
l_len	off64_t	64 bits
l_sysid	long	
l_pid	pid_t	
pad	long[4]	

2.1.4 rlim64_t and rlimit64

The addition of large files requires the resource limit subsystem to understand limits which are greater than 32 bits in length. For this we will add extended setrlimit(2) and getrlimit(2) system calls which work with 64 bit values.

The rlim64_t will be defined as “unsigned long long rlim64_t.” The rlimit64 structure will be:

TABLE 3. rlimit64 structure definition

field name	field type	comments
rlim_cur	rlim64_t	64 bits
rlim_max	rlim64_t	64 bits

2.1.5 fpos64_t

The standard I/O library routines fsetpos() and fgetpos() use the fpos_t type to describe file offsets. It would be nice if we could just extend this type to 64 bits, but then we would break old

applications which use it. Therefore we will define the new type fpos64_t which will be fpos_t's 64 bit equivalent. It will be defined as "long long fpos64_t."

2.2 New and Extended System Calls

This section describes each of the new and extended system calls to be added to support 64 bit file access.

2.2.1 stat64(), fstat64(), lstat64()

These routines are the 64 bit equivalents of the 32 bit stat(), fstat(), and lstat() system calls. They take a pointer to a stat64 structure in place of the stat structure accepted by the 32 bit versions.

TABLE 4. stat64 system call prototypes

```
int stat64(const char *path, struct stat64 *buf);
int fstat64(int fildes, struct stat64 *buf);
int lstat64(const char *path, struct stat64 *buf);
```

2.2.2 truncate64(), ftruncate64()

These are the 64 bit equivalents of the 32 bit truncate() and ftruncate() system calls. In place of an off_t they take a off64_t to specify the new length of the file.

TABLE 5. truncate64 system call prototypes

```
int truncate64(const char *path, off64_t length);
int ftruncate64(int fildes, off64_t length);
```

2.2.3 lseek64()

This is the 64 bit equivalent of the 32 bit lseek() system call. It takes a off64_t in place of an off_t to specify the offset amount, and it returns a off64_t in place of an off_t.

TABLE 6. lseek64 system call prototype

```
off64_t lseek64(int fildes, off64_t offset, int whence);
```

2.2.4 mmap64()

This is a version of the mmap() system call which allows the caller to specify a 64 bit file offset. It replaces mmap()'s off_t file offset parameter with a 64 bit off64_t value.

TABLE 7. mmap64 system call prototype

```
void *mmap64(void *addr, size_t len, int prot, int flags, int fildes, off64_t offset);
```

2.2.5 getrlimit64(), setrlimit64()

In order to access or set file size limits which are greater than 64 bits, 64 bit versions of the rlimit system calls will be added. These will use the rlimit64 structure in place of the rlimit structure.

TABLE 8. rlimit64 system call prototypes

```
int getrlimit64(int resource, struct rlimit64 *rlp);
int setrlimit64(int resource, const struct rlimit64 *rlp);
```

2.2.6 fcntl()

The fcntl() system call will be extended by adding new operations. These new operations are:

- F_GETLK64
- F_SETLK64
- F_SETLKW64
- F_FREESP64
- F_ALLOCSP64

All of these operations are equivalent to those without the ‘64,’ and they are different only in that they take a pointer to a flock64 structure in place of an flock structure.

2.3 New Library Interfaces

The standard I/O library will be extended so that 64 bit files can be accessed with the standard I/O file access model. The new interfaces are described below.

2.3.1 fseek64()

This is a version of the fseek() call which takes a 64 bit value for an offset and returns the new file offset as a 64 bit value.

TABLE 9. fseek64 function prototype

```
off64_t fseek64(FILE *file, off64_t offset, int whence);
```

2.3.2 ftell64()

This is a version of the ftell() call which returns a 64 bit value for the current file offset.

TABLE 10. ftell64 function prototype

```
off64_t ftell64(FILE *file);
```

2.3.3 fgetpos64(), fsetpos64()

These are versions of the fgetpos() and fsetpos() routines which take pointers to fpos64_t parameters rather than fpos_t.

TABLE 11. fgetpos64 and fsetpos64 function prototypes

```
int fgetpos64(FILE *file, fpos64_t *pos);
int fsetpos64(FILE *file, const fpos64_t *pos);
```

2.3.4 ftw64(), nftw64()

If it is considered worthwhile, the file tree walk library routines will be extended to support stat64 structures instead of stat structures.

3.0 Semantics and Error Returns

This section explains what happens when the old interfaces are used to access large files. It is important to make this behavior consistent, but there do not seem to be any clearly “correct” answers here.

3.1 The Model

The model described below is almost identical to one presented by Convex in the Winter 1992 USENIX conference proceedings. Their model seems to be consistent and easily implementable without any major drawbacks, so I am using most of it.

32 bit programs either recognize the existence of large files or they do not. Those that do will be using new interfaces to access these files. Those that do not will see all files as having a maximum size of 2GB-1 bytes. Data beyond offset 2GB-1 in a file will be inaccessible by applications which do not recognize the existence of large files, and the size of a file which is greater than 2GB-1 will not be visible to an application which does not know of such files. The implications of these statements are explained below.

3.1.1 open()

In order for the system to be able to determine that a program is aware of large files, the program must “tell” the system. This will be done with a new open flag O_LARGEFILE. Opening a file with this flag allows the program to see the file as larger then 2GB-1 if it is so.

This flag will be inherited across both fork(2) and exec(2). This behavior is pretty much defined by the implementation, because the file structure in the kernel is shared across these calls and that is where the open flags are kept.

3.1.2 fcntl()

In addition to specifying the O_LARGEFILE flag to open, a program may get the equivalent effect by setting the FLARGEFILE flag for the file with fcntl(F_SETFL).

3.1.3 fopen(), fdopen(), freopen()

These open routines will accept an added flag, the “l” flag. This will have the equivalent effect of specifying O_LARGEFILE to the open() system call.

3.1.4 write()

In order for a 32 bit program to write beyond offset 2GB-1 in a file, it must have set the FLARGE-FILE flag for the file. If the program has not done so and it attempts to write beyond offset 2GB-1in the file, write() will return -1 and errno will be set to EFBIG. Actually, a write which starts before 2GB-1 and extends beyond it will first return as a partial write with write() returning the number of bytes written before the 2GB-1 boundary was reached. The next write() call would then return -1 with errno set to EFBIG.

3.1.5 read()

In order for a 32 bit program to read beyond offset 2GB-1 in a file, it must have set the FLARGE-FILE flag for the file. If the program has not done so and it attempts to read beyond offset 2GB-1 in the file, read() will return -1 and errno will be set to EFBIG. As with write(), read() will actually do a partial read up to the 2GB-1 boundary and return EFBIG on the subsequent read() call.

3.1.6 lseek()

It is not possible for a 32 bit program to use seek to manipulate the file offset beyond 2GB-1, because the 32 bit value returned by lseek() cannot correctly describe the new offset. If a program attempts to manipulate the seek offset beyond offset 2GB-1 in the file, lseek() will return -1 and errno will be set to EFBIG

3.1.7 truncate(), ftruncate()

Since these system calls take only 32 bit values for the new file size, the worst they could be doing is to shrink a file from greater than 2GB-1 in size to less than 2GB-1. This will be allowed.

3.1.8 mmap()

If the sum of the offset and length parameters to mmap() is greater than 2GB-1 and the application has not set the FLARGEFILE flag in the file, then the call will fail returning -1 and setting errno to EFBIG.

3.1.9 stat(), fstat(), lstat()

Each of these calls returns the attributes of the specified file in a stat structure. The stat structure has only a signed 32 bit field in which to store the size of the file. If any of these calls is used on a file whose size is greater than 2GB-1, then the system call will return -1 and errno will be set to EFBIG.

It has been suggested that the stat calls return successfully instead of failing and put 2GB-1 or -1 in the st_size field of the stat structure. This behavior, however friendly, will not notify the user that he or she is doing something wrong, and it can lead to incorrect and possibly damaging behavior by a correctly written program. Returning the EFBIG error should make it clear to the user what the problem is, and it should also cause most applications to exit somewhat cleanly.

3.2 More Implications

The FLARGFILE flag will be set explicitly by the open() and fcntl() calls. It will be set implicitly by calls to lseek64(), fstat64(), and ftruncate64(). The reason for this is that these calls indicate that the caller understands 64 bit files even if he or she did not say so explicitly. For the lseek64() call it is necessary to turn on the flag or fail the call. This is because if we allow the call to succeed and set the file offset beyond 2GB-1 but do not set the flag, subsequent reads and writes will fail. The other two will set the flag just to be consistent.

Another issue involves NFS access to large files. There is no way to communicate O_LARGE-FILE or a 64 bit file size or offset across the network using the NFS protocol. NFS accesses requiring information on the size of the file, which is all of them, will just have to fail.

4.0 Required Library Changes

Some parts of the libraries will need to be changed to use the new and extended interfaces defined above. The standard I/O code will be changed to use 64 bit file offsets and the new system calls internally. Other routines within libc such as opendir() and remove() will be changed to use the new system calls as well. It is not yet clear what changes if any will be necessary in other libraries.

5.0 Required Utility Changes

This section attempts to list the utility programs which must be changed to use the new 64 bit file access primitives. It does not include those which are file system specific and must be changed if the file supports 64 bit files anyway (e.g. dump, fsck).

ls, cp, mv, rcp, ftp, find, tail, cat, sh, csh, ksh, tcsh, dd, tar, cpio, cmp, od, compress, quotas, tail, bru, diff, wc, du, odiff

6.0 Required Kernel Changes

The goal of the changes to the kernel will be to support the following:

- 32 bit kernels without 64 bit files (K32FS32)
- 32 bit kernels with 64 bit files (K32FS64)
- 64 bit kernels with 64 bit files (K64FS64)

This support will be based on the compile time definitions indicated in parantheses in the list above. Whether or not we want to ship both the K32FS32 and the K32FS64 versions, however, is questionable. The arguments for each are:

For small systems we do not want to pay the increased memory and computational overhead associated with supporting 64 bit files.

Not supporting 64 bit files on all systems implies that we will have hardware independent functionality available on some platforms and not on others. While the 64 bit access interfaces will remain, there will be no way to access beyond 2GB-1 bytes in a file.

This is more a policy decision than anything else. The code will support all three types of kernels.

6.1 Kernel Type Changes

All of the types associated with file offsets will need to be expanded to 64 bits in K32FS64 and K64FS64 kernels. These changes will all be confined to the kernel and will not be user visible. The types to be changed are listed below.

- off_t -- This will become a 64 bit value.
- daddr_t -- This will become a 64 bit value.
- vattr_t -- The vnode attributes structure will be changed to contain a 64 bit value for the file size.
- uio_t -- The uio structure’s uio_offset, uio_limit, and uio_blkno fields will be expanded to 64 bits.
- pfdat_t -- The pfdata structure’s pf_pageno field will be expanded to 52 bits, increasing the size of the pfdat structure by 32 bits.
- pgno_t -- This will become a 64 bit value for describing pages associated with very large file offsets.
- file_t -- The f_offset field will be expanded to 64 bits.
- user_t -- The file offset fields in the user structure, u_r.r_off and u_offset, will be expanded to 64 bits.
- rlim_t -- This will become a 64 bit value for describing file size limits in a 64 bit file system.
- rlimit -- The rlim_cur and rlim_max fields of this structure are both rlim_ts, so this structure will be expanding.
- flock_t -- The l_start and l_len fields are of type off_t, so they will be growing to 64 bits.

- reg_t -- The r_fileoff field of the region structure is an off_t that will expand to 64 bits. The r_maxsize field may not need to be expanded and if not its type will be changed. The pregion structure will not need to be expanded, although the types of its p_offset adn p_pglen fields will be changed to they are 32 bit values in a K32FS64 kernel.
- swapres_t, xswapres_t, swapent_t, swapinfo_t -- These structures will be modified so that in K32FS64 kernels the off_t and pgno_t type fields will remain 32 bit values.
- buf_t -- The b_offset and b_blkno fields of the buf_t will be growing to 64 bits.

6.2 Kernel Routine Changes

All of the code in the kernel manipulating off_ts will need to be checked to make sure it does not depend on an off_t being the same size as an int or a long.