

System Call

Fallocate()

Contents

System call	2
What is fallocate?	2
Why fallocate?	2
How fallocate?	3
ERRORS	5
Flags and their purposes	6
Implementations	7
Sources	10

System call

A system call is a controlled entry point into the kernel, allowing a process to request that the kernel perform some action on the process's behalf. The kernel makes a range of services accessible to programs via the system call application programming interface (API). These services include, for example, creating a new process, performing I/O, and creating a pipe for interprocess communication.

Before going into the details of how my given (`fallocate`) system call works, we note some general points:

- A system call changes the processor state from user mode to kernel mode, so that the CPU can access protected kernel memory.
- The set of system calls is fixed. Each system call is identified by a unique number. (This numbering scheme is not normally visible to programs, which identify system calls by name.)
- Each system call may have a set of arguments that specify information to be transferred from user space (i.e., the process's virtual address space) to kernel space and vice versa.

What is `fallocate`?

This is a nonportable, Linux-specific system call. For the portable, POSIX.1-specified method of ensuring that space is allocated for a file.

`fallocate()` allows the caller to directly manipulate the allocated disk space for the file referred to by `fd` for the byte range starting at `offset` and continuing for `len` bytes. The mode argument determines the operation to be performed on the given range. Details of the supported operations are given in the subsections below.

SYNOPSIS

```
#define _GNU_SOURCE
#include <fcntl.h>

int fallocate(int fd, int mode, off_t offset, off_t len);
```

RETURN VALUE

On success, `fallocate()` returns zero. On error, `-1` is returned and `errno` is set to indicate the error.

Why `fallocate`?

`fallocate` is used to manipulate the allocated disk space for a file, either to deallocate or preallocate it. For filesystems which support the `fallocate` system call, preallocation is done quickly by allocating blocks and marking them as uninitialized, requiring no IO to the data blocks. This is much faster than creating a file by filling it with zeroes.

How fallocate?

Allocating disk space

The default operation (i.e., *mode* is zero) of *fallocate()* allocates the disk space within the range specified by *offset* and *len*. The file size will be changed if *offset+len* is greater than file size. Any subregion within the range specified by *offset* and *len* that did not contain data before the call will be initialized to zero. This default behavior closely resembles the behavior of the *posix_fallocate()* library function, and is intended as a method of optimally implementing that function.

After a successful call, subsequent writes into the range specified by *offset* and *len* are guaranteed not to fail because of lack of disk space.

If the *FALLOC_FL_KEEP_SIZE* flag is specified in *mode*, the behavior of the call is similar, but the file size will not be changed even if *offset+len* is greater than the file size. Preallocating zeroed blocks beyond the end of the file in this manner is useful for optimizing append workloads.

If the *FALLOC_FL_UNSHARE* flag is specified in *mode*, shared file data extents will be made private to the file to guarantee that a subsequent write will not fail due to lack of space. Typically, this will be done by performing a copy-on-write operation on all shared data in the file. This flag may not be supported by all filesystems.

Because allocation is done in block size chunks, *fallocate()* may allocate a larger range of disk space than was specified.

Deallocating file space

Specifying the *FALLOC_FL_PUNCH_HOLE* flag (available since Linux 2.6.38) in *mode* deallocates space (i.e., creates a hole) in the byte range starting at *offset* and continuing for *len* bytes. Within the specified range, partial filesystem blocks are zeroed, and whole filesystem blocks are removed from the file. After a successful call, subsequent reads from this range will return zeros.

The *FALLOC_FL_PUNCH_HOLE* flag must be ORed with *FALLOC_FL_KEEP_SIZE* in *mode*; in other words, even when punching off the end of the file, the file size does not change.

Not all filesystems support *FALLOC_FL_PUNCH_HOLE*; if a filesystem doesn't support the operation, an error is returned. The operation is supported on at least the following filesystems:

- XFS (since Linux 2.6.38)
- ext4 (since Linux 3.0)
- Btrfs (since Linux 3.7)
- tmpfs(5) (since Linux 3.5)
- gfs2(5) (since Linux 4.16)

Collapsing file space

Specifying the *FALLOC_FL_COLLAPSE_RANGE* flag (available since Linux 3.15) in *mode* removes a byte range from a file, without leaving a hole. The byte range to be collapsed starts at *offset* and continues for *len* bytes. At the completion of the operation, the contents of the

file starting at the location *offset+len* will be appended at the location *offset*, and the file will be *len* bytes smaller.

A filesystem may place limitations on the granularity of the operation, in order to ensure efficient implementation. Typically, *offset* and *len* must be a multiple of the filesystem logical block size, which varies according to the filesystem type and configuration. If a filesystem has such a requirement, *fallocate()* fails with the error *EINVAL* if this requirement is violated.

If the region specified by *offset* plus *len* reaches or passes the end of file, an error is returned; instead, use *ftruncate()* to truncate a file.

No other flags may be specified in *mode* in conjunction with *FALLOC_FL_COLLAPSE_RANGE*.

As at Linux 3.15, *FALLOC_FL_COLLAPSE_RANGE* is supported by ext4 (only for extent based files) and XFS.

Zeroing file space

Specifying the *FALLOC_FL_ZERO_RANGE* flag (available since Linux 3.15) in *mode* zeros space in the byte range starting at *offset* and continuing for *len* bytes. Within the specified range, blocks are preallocated for the regions that span the holes in the file. After a successful call, subsequent reads from this range will return zeros.

Zeroing is done within the filesystem preferably by converting the range into unwritten extents. This approach means that the specified range will not be physically zeroed out on the device (except for partial blocks at the either end of the range), and I/O is (otherwise) required only to update metadata.

If the *FALLOC_FL_KEEP_SIZE* flag is additionally specified in *mode*, the behavior of the call is similar, but the file size will not be changed even if *offset+len* is greater than the file size. This behavior is the same as when preallocating space with *FALLOC_FL_KEEP_SIZE* specified.

Not all filesystems support *FALLOC_FL_ZERO_RANGE*; if a filesystem doesn't support the operation, an error is returned. The operation is supported on at least the following filesystems:

- XFS (since Linux 3.15)
- ext4, for extent-based files (since Linux 3.15)
- SMB3 (since Linux 3.17)
- Btrfs (since Linux 4.16)

Increasing file space

Specifying the *FALLOC_FL_INSERT_RANGE* flag (available since Linux 4.1) in *mode* increases the file space by inserting a hole within the file size without overwriting any existing data. The hole will start at *offset* and continue for *len* bytes. When inserting the hole inside file, the contents of the file starting at *offset* will be shifted upward (i.e., to a higher file offset) by *len* bytes. Inserting a hole inside a file increases the file size by *len* bytes.

This mode has the same limitations as *FALLOC_FL_COLLAPSE_RANGE* regarding the granularity of the operation. If the granularity requirements are not met, *fallocate()* fails with the error *EINVAL*. If the offset is equal to or greater than the end of file, an error is returned. For such operations (i.e., inserting a hole at the end of file), *ftruncate()* should be used.

No other flags may be specified in mode in conjunction with *FALLOC_FL_INSERT_RANGE*.

FALLOC_FL_INSERT_RANGE requires filesystem support. Filesystems that support this operation include XFS (since Linux 4.1) and ext4 (since Linux 4.2).

ERRORS

EBADF *fd* is not a valid file descriptor, or is not opened for writing.

EFBIG *offset+len* exceeds the maximum file size.

EFBIG mode is *FALLOC_FL_INSERT_RANGE*, and the current file *size+len* exceeds the maximum file size.

EINTR A signal was caught during execution;

EINVAL *offset* was less than 0, or *len* was less than or equal to 0.

EINVAL mode is *FALLOC_FL_COLLAPSE_RANGE* and the range specified by *offset* plus *len* reaches or passes the end of the file.

EINVAL mode is *FALLOC_FL_INSERT_RANGE* and the range specified by *offset* reaches or passes the end of the file.

EINVAL mode is *FALLOC_FL_COLLAPSE_RANGE* or *FALLOC_FL_INSERT_RANGE*, but either *offset* or *len* is not a multiple of the filesystem block size.

EINVAL mode contains one of *FALLOC_FL_COLLAPSE_RANGE* or *FALLOC_FL_INSERT_RANGE* and also other flags; no other flags are permitted with *FALLOC_FL_COLLAPSE_RANGE* or *FALLOC_FL_INSERT_RANGE*.

EINVAL mode is *FALLOC_FL_COLLAPSE_RANGE* or *FALLOC_FL_ZERO_RANGE* or *FALLOC_FL_INSERT_RANGE*, but the file referred to by *fd* is not a regular file.

EIO An I/O error occurred while reading from or writing to a filesystem.

ENODEV *fd* does not refer to a regular file or a directory. (If *fd* is a pipe or FIFO, a different error results.)

ENOSPC There is not enough space left on the device containing the file referred to by *fd*.

ENOSYS This kernel does not implement *fallocate()*.

EOPNOTSUPP

The filesystem containing the file referred to by *fd* does not support this operation; or the mode is not supported by the filesystem containing the file referred to by *fd*.

EPERM The file referred to by *fd* is marked immutable.

EPERM mode specifies *FALLOC_FL_PUNCH_HOLE* or *FALLOC_FL_COLLAPSE_RANGE* or *FALLOC_FL_INSERT_RANGE* and the file referred to by *fd* is marked append-only.

EPERM The operation was prevented by a file seal.

ESPIPE *fd* refers to a pipe or FIFO.

ETXTBSY

mode specifies *FALLOC_FL_COLLAPSE_RANGE* or *FALLOC_FL_INSERT_RANGE*, but the file referred to by *fd* is currently being executed.

Flags and their purposes

FALLOC_FL_KEEP_SIZE

This flag allocates and initializes to zero the disk space within the range specified by *offset* and *len*. After a successful call, subsequent writes into this range are guaranteed not to fail because of lack of disk space. Preallocating zeroed blocks beyond the end of the file is useful for optimizing append workloads. Preallocating blocks does not change the file size even if it is less than *offset+len*.

FALLOC_FL_UNSHARE

flag is specified in mode, shared file data extents will be made private to the file to guarantee that a subsequent write will not fail due to lack of space.

FALLOC_FL_PUNCH_HOLE

flag (available since Linux 2.6.38) in mode deallocates space (i.e., creates a hole) in the byte range starting at *offset* and continuing for *len* bytes.

FALLOC_FL_COLLAPSE_RANGE

flag (available since Linux 3.15) in mode removes a byte range from a file, without leaving a hole.

FALLOC_FL_ZERO_RANGE

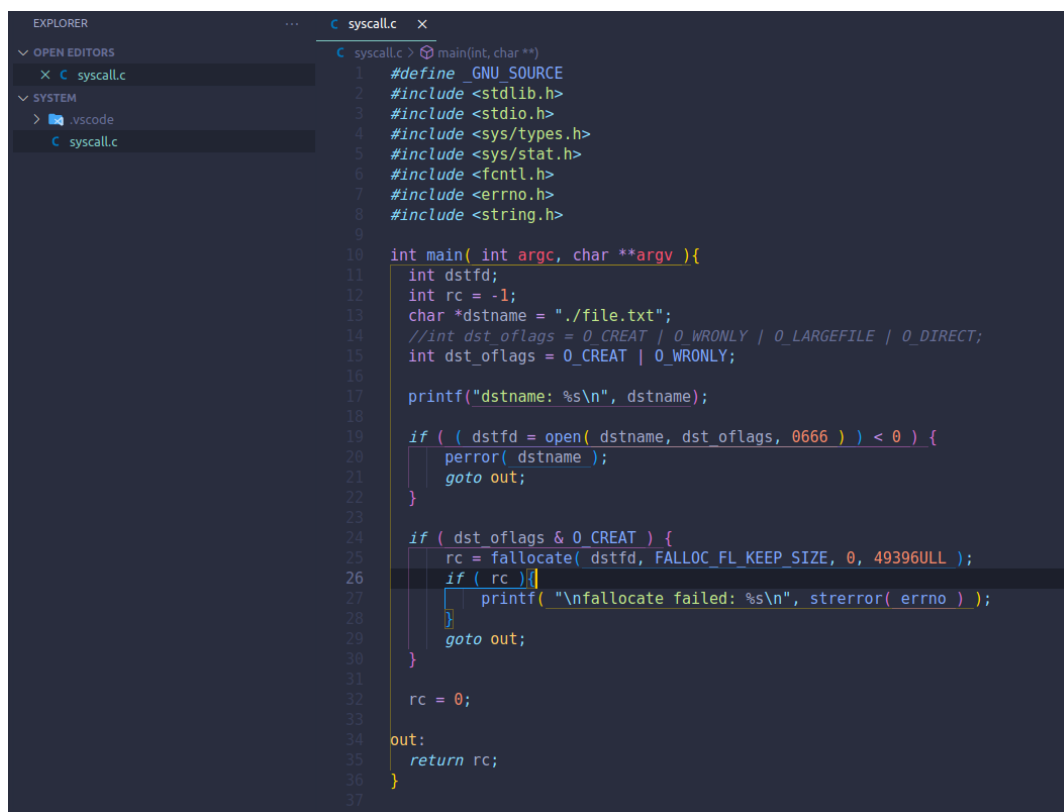
flag (available since Linux 3.15) in mode zeros space in the byte range starting at *offset* and continuing for *len* bytes.

FALLOC_FL_INSERT_RANGE

flag (available since Linux 4.1) in mode increases the file space by inserting a hole within the file size without overwriting any existing data.

Implementations

The implementation of *fallocate()* with flag *FALLOC_FL_KEEP_SIZE* is shown in the picture 1.1 and picture 1.2 below. First I create file named *file.txt* with command *touch* and run the command *du* to see the size of the file as you can see, it is zero KB and after I run the script the file size change into 52 KB as I try to show in the picture. The *FALLOC_FL_KEEP_SIZE* as I mention it in the flags section behave like mode 0 and allocate memory size.

A screenshot of a code editor window titled 'syscall.c'. The editor shows a C program that demonstrates the use of `fallocate()` with the `FALLOC_FL_KEEP_SIZE` flag. The program opens a file named `./file.txt` and then calls `fallocate(dstfd, FALLOC_FL_KEEP_SIZE, 0, 49396ULL)`. It includes error handling with `perror` and `strerror`. The file explorer on the left shows the project structure with `syscall.c` selected.

```
1  #define _GNU_SOURCE
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <sys/types.h>
5  #include <sys/stat.h>
6  #include <fcntl.h>
7  #include <errno.h>
8  #include <string.h>
9
10 int main( int argc, char **argv ){
11     int dstfd;
12     int rc = -1;
13     char *dstname = "./file.txt";
14     //int dst_oflags = O_CREAT | O_WRONLY | O_LARGEFILE | O_DIRECT;
15     int dst_oflags = O_CREAT | O_WRONLY;
16
17     printf("dstname: %s\n", dstname);
18
19     if ( ( dstfd = open( dstname, dst_oflags, 0666 ) ) < 0 ) {
20         perror( dstname );
21         goto out;
22     }
23
24     if ( dst_oflags & O_CREAT ) {
25         rc = fallocate( dstfd, FALLOC_FL_KEEP_SIZE, 0, 49396ULL );
26         if ( rc ) {
27             printf( "\nfallocate failed: %s\n", strerror( errno ) );
28         }
29         goto out;
30     }
31
32     rc = 0;
33
34 out:
35     return rc;
36 }
37
```

Picture 1.1


```
kaleab@ubuntu: ~/Desktop/system
kaleab@ubuntu:~/Desktop/system$ touch file.txt
kaleab@ubuntu:~/Desktop/system$ ls
file.txt  syscall.c
kaleab@ubuntu:~/Desktop/system$ du file.txt
0          file.txt
kaleab@ubuntu:~/Desktop/system$ gcc syscall.c -o sys
kaleab@ubuntu:~/Desktop/system$ ./sys
dstname: ./file.txt
kaleab@ubuntu:~/Desktop/system$ du file.txt
52          file.txt
kaleab@ubuntu:~/Desktop/system$
```

Picture 1.2

Another implementation of *fallocate()* with flag *FALLOC_FL_PUNCH_HOLE* is the pictures down below, as you can see in the script part I only change the mode parameter in to *FALLOC_FL_PUNCH_HOLE* to deallocate the size, as I mention it above in the how to *fallocate?* Section, *FALLOC_FL_PUNCH_HOLE* flag only support some file systems. my storage device uses *FAT32* file system which is unsupported file system so it will say *operation not supported* like expected picture 1.4.

```
C syscall.c X
C syscall.c > main(int, char **)
1  #define _GNU_SOURCE
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <sys/types.h>
5  #include <sys/stat.h>
6  #include <fcntl.h>
7  #include <errno.h>
8  #include <string.h>
9
10 int main( int argc, char **argv ){
11     int dstfd;
12     int rc = -1;
13     char *dstname = "./file.txt";
14     //int dst_oflags = O_CREAT | O_WRONLY | O_LARGEFILE | O_DIRECT;
15     int dst_oflags = O_CREAT | O_WRONLY;
16
17     printf("dstname: %s\n", dstname);
18
19     if ( ( dstfd = open( dstname, dst_oflags, 0666 ) ) < 0 ) {
20         perror( dstname );
21         goto out;
22     }
23
24     if ( dst_oflags & O_CREAT ) {
25         rc = fallocate( dstfd, FALLOC_FL_PUNCH_HOLE, 0, 49396ULL );
26         if ( rc ){
27             printf( "\nfallocate failed: %s\n", strerror( errno ) );
28         }
29         goto out;
30     }
31
32     rc = 0;
33
34 out:
35     return rc;
36 }
37
```

Picture 1.3

```
kaleab@ubuntu:~/Desktop/system$ ls
file.txt  syscall.c
kaleab@ubuntu:~/Desktop/system$ du file.txt
52      file.txt
kaleab@ubuntu:~/Desktop/system$ gcc syscall.c -o sys
kaleab@ubuntu:~/Desktop/system$ ./sys
dstname: ./file.txt
fallocate failed: Operation not supported
kaleab@ubuntu:~/Desktop/system$ du file.txt
52      file.txt
kaleab@ubuntu:~/Desktop/system$
```

Picture 1.4

Sources

Linux manual page 2