NAME

memfd_create − create an anonymous file

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#define _GNU_SOURCE /* See feature_test_macros(7) */
#include <sys/mman.h>

int memfd_create(const char *_name_, unsigned int _fla gs_);

DESCRIPTION

memfd_create() creates an anonymous file and returns a file descriptor that refers to it. The file behaves like a regular file, and so can be modified, truncated, memory-mapped, and so on. However, unlike a regular file, it lives in RAM and has a volatile backing storage. Once all references to the file are dropped, it is automatically released. Anonymous memory is used for all backing pages of the file. Therefore, files created by **memfd_create**() have the same semantics as other anonymous memory allocations such as those allocated using **mmap**(2) with the **MAP_ANONYMOUS** flag.

The initial size of the file is set to 0. Following the call, the file size should be set using **ftruncate**(2). (Alternatively, the file may be populated by calls to **write**(2) or similar.)

The name supplied in *name* is used as a filename and will be displayed as the target of the corresponding symbolic link in the directory */proc/self/fd/*. The displayed name is always prefixed with *memfd:* and serves only for debugging purposes. Names do not affect the behavior of the file descriptor, and as such multiple files can have the same name without any side effects.

The following values may be bitwise ORed in *fla gs* to change the behavior of **memfd_create**():

MFD_CLOEXEC
: Set the close-on-exec (**FD_CLOEXEC**) flag on the new file descriptor. See the description of the **O_CLOEXEC** flag in **open**(2) for reasons why this may be useful.

MFD_ALLOW_SEALING
: Allow sealing operations on this file. See the discussion of the **F_ADD_SEALS** and **F_GET_SEALS** operations in **fcntl**(2), and also NOTES, below. The initial set of seals is empty. If this flag is not set, the initial set of seals will be **F_SEAL_SEAL**, meaning that no other seals can be set on the file.

MFD_HUGETLB (since Linux 4.14)
: The anonymous file will be created in the hugetlbfs filesystem using huge pages. See the Linux kernel source file *Documentation/admin−guide/mm/hugetlbpage.rst* for more information about hugetlbfs. Specifying both**MFD_HUGETLB** and **MFD_ALLO W_SEALING** in *flags* is supported since Linux 4.16.

MFD_HUGE_2MB, **MFD_HUGE_1GB**, **...**
: Used in conjunction with **MFD_HUGETLB** to select alternative hugetlb page sizes (respectively, 2 MB, 1 GB, ...) on systems that support multiple hugetlb page sizes. Definitions for known huge page sizes are included in the header file<*linux/memfd.h*>.

  For details on encoding huge page sizes not included in the header file, see the discussion of the similarly named constants in **mmap**(2).

Unused bits in *fla gs* must be 0.

As its return value, **memfd_create**() returns a new file descriptor that can be used to refer to the file. This file descriptor is opened for both reading and writing (**O_RDWR**) and **O_LARGEFILE** is set for the file descriptor.

With respect to **fork**(2) and **execve**(2), the usual semantics apply for the file descriptor created by **memfd_create**(). A copy of the file descriptor is inherited by the child produced by **fork**(2) and refers to the same file. The file descriptor is preserved across **execve**(2), unless the close-on-exec flag has been set.

RETURN VALUE

On success, **memfd_create**() returns a new file descriptor. On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EFAULT
The address in *name* points to invalid memory.

EINVAL
flags included unknown bits.

EINVAL
name was too long. (The limit is 249 bytes, excluding the terminating null byte.)

EINVAL
Both **MFD_HUGETLB** and **MFD_ALLOW_SEALING** were specified in *flags*.

EMFILE
The per-process limit on the number of open file descriptors has been reached.

ENFILE
The system-wide limit on the total number of open files has been reached.

ENOMEM
There was insufficient memory to create a new anonymous file.

VERSIONS

The **memfd_create**() system call first appeared in Linux 3.17; glibc support was added in glibc 2.27.

EPERM
The **MFD_HUGETLB** flag was specified, but the caller was not privileged (did not have the **CAP_IPC_LOCK** capability) and is not a member of the *sysctl_hugetlb_shm_group* group; see the description of */proc/sys/vm/sysctl_hugetlb_shm_group* in **proc**(5).

STANDARDS

The **memfd_create**() system call is Linux-specific.

NOTES

The **memfd_create**() system call provides a simple alternative to manually mounting a **tmpfs**(5) filesystem and creating and opening a file in that filesystem. The primary purpose of **memfd_create**() is to create files and associated file descriptors that are used with the file-sealing APIs provided by **fcntl**(2).

The **memfd_create**() system call also has uses without file sealing (which is why file-sealing is disabled, unless explicitly requested with the **MFD_ALLOW_SEALING** flag). In particular, it can be used as an alternative to creating files in *tmp* or as an alternative to using the **open**(2) **O_TMPFILE** in cases where there is no intention to actually link the resulting file into the filesystem.

File sealing

In the absence of file sealing, processes that communicate via shared memory must either trust each other, or take measures to deal with the possibility that an untrusted peer may manipulate the shared memory region in problematic ways. For example, an untrusted peer might modify the contents of the shared memory at any time, or shrink the shared memory region. The former possibility leaves the local process vulnerable to time-of-check-to-time-of-use race conditions (typically dealt with by copying data from the shared memory region before checking and using it). The latter possibility leaves the local process vulnerable to **SIGBUS** signals when an attempt is made to access a now-nonexistent location in the shared memory region. (Dealing with this possibility necessitates the use of a handler for the **SIGBUS** signal.)

Dealing with untrusted peers imposes extra complexity on code that employs shared memory. Memory sealing enables that extra complexity to be eliminated, by allowing a process to operate secure in the knowledge that its peer can't modify the shared memory in an undesired fashion.

An example of the usage of the sealing mechanism is as follows:

(1) The first process creates a **tmpfs**(5) file using **memfd_create**(). The call yields a file descriptor used in subsequent steps.

(2) The first process sizes the file created in the previous step using **ftruncate**(2), maps it using **mmap**(2), and populates the shared memory with the desired data.

(3) The first process uses the **fcntl**(2) **F_ADD_SEALS** operation to place one or more seals on the file, in order to restrict further modifications on the file. (If placing the seal **F_SEAL_WRITE**, then it will be necessary to first unmap the shared writable mapping created in the previous step. Otherwise, behavior similar to **F_SEAL_WRITE** can be achieved by using **F_SEAL_FUTURE_WRITE**, which will prevent future writes via **mmap**(2) and **write**(2) from succeeding while keeping existing shared writable mappings).

(4) A second process obtains a file descriptor for the **tmpfs**(5) file and maps it. Among the possible ways in which this could happen are the following:

- The process that called **memfd_create**() could transfer the resulting file descriptor to the second process via a UNIX domain socket (see **unix**(7) and **cmsg**(3)). The second process then maps the file using **mmap**(2).
- The second process is created via **fork**(2) and thus automatically inherits the file descriptor and mapping. (Note that in this case and the next, there is a natural trust relationship between the two processes, since they are running under the same user ID. Therefore, file sealing would not normally be necessary.)
- The second process opens the file */proc/<pid>/fd/<fd>*, where<*pid*> is the PID of the first process (the one that called **memfd_create**()), and<*fd*> is the number of the file descriptor returned by the call to **memfd_create**() in that process. The second process then maps the file using **mmap**(2).

(5) The second process uses the **fcntl**(2) **F_GET_SEALS** operation to retrieve the bit mask of seals that has been applied to the file. This bit mask can be inspected in order to determine what kinds of restrictions have been placed on file modifications. If desired, the second process can apply further seals to impose additional restrictions (so long as the **F_SEAL_SEAL** seal has not yet been applied).

EXAMPLES

Below are shown two example programs that demonstrate the use of **memfd_create**() and the file sealing API.

The first program, *t_memfd_create.c*, creates a **tmpfs**(5) file using **memfd_create**(), sets a size for the file, maps it into memory, and optionally places some seals on the file. The program accepts up to three command-line arguments, of which the first two are required. The first argument is the name to associate with the file, the second argument is the size to be set for the file, and the optional third argument is a string of characters that specify seals to be set on the file.

The second program, *t_get_seals.c*, can be used to open an existing file that was created via **memfd_create**() and inspect the set of seals that have been applied to that file.

The following shell session demonstrates the use of these programs. First we create a **tmpfs**(5) file and set some seals on it:

```
$ ./t_memfd_create my_memfd_file 4096 sw &
[1] 11775
PID: 11775; fd: 3; /proc/11775/fd/3
```

At this point, the *t_memfd_create* program continues to run in the background. From another program, we can obtain a file descriptor for the file created by **memfd_create**() by opening the */proc/*pid*/fd* file that corresponds to the file descriptor opened by **memfd_create**(). Using that pathname, we inspect the content of the */proc/*pid*/fd* symbolic link, and use our *t_get_seals* program to view the seals that have been placed on the file:

```
$ readlink /proc/11775/fd/3
/memfd:my_memfd_file (deleted)
```

```
$ ./t_get_seals /proc/11775/fd/3
Existing seals: WRITE SHRINK
```

Program source: t_memfd_create.c

```
#define _GNU_SOURCE
#include <err.h>
#include <fcntl.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <unistd.h>

int
main(int argc, char *argv[])
{
    int           fd;
    char          *name, *seals_arg;
    ssize_t       len;
    unsigned int  seals;

    if (argc < 3) {
        fprintf(stderr, "%s name size [seals]\n", argv[0]);
        fprintf(stderr, "\t'seals' can contain any of the "
                "following characters:\n");
        fprintf(stderr, "\t\tg - F_SEAL_GROW\n");
        fprintf(stderr, "\t\ts - F_SEAL_SHRINK\n");
        fprintf(stderr, "\t\tw - F_SEAL_WRITE\n");
        fprintf(stderr, "\t\tW - F_SEAL_FUTURE_WRITE\n");
        fprintf(stderr, "\t\tS - F_SEAL_SEAL\n");
        exit(EXIT_FAILURE);
    }

    name = argv[1];
    len = atoi(argv[2]);
    seals_arg = argv[3];

    /* Create an anonymous file in tmpfs; allow seals to be
       placed on the file. */

    fd = memfd_create(name, MFD_ALLOW_SEALING);
    if (fd == -1)
        err(EXIT_FAILURE, "memfd_create");

    /* Size the file as specified on the command line. */

    if (ftruncate(fd, len) == -1)
        err(EXIT_FAILURE, "truncate");

    printf("PID: %jd; fd: %d; /proc/%jd/fd/%d\n",
           (intmax_t) getpid(), fd, (intmax_t) getpid(), fd);

    /* Code to map the file and populate the mapping with data
```

```
            omitted. */

        /* If a 'seals' command-line argument was supplied, set some
           seals on the file. */

        if (seals_arg != NULL) {
            seals = 0;

            if (strchr(seals_arg, 'g') != NULL)
                seals |= F_SEAL_GROW;
            if (strchr(seals_arg, 's') != NULL)
                seals |= F_SEAL_SHRINK;
            if (strchr(seals_arg, 'w') != NULL)
                seals |= F_SEAL_WRITE;
            if (strchr(seals_arg, 'W') != NULL)
                seals |= F_SEAL_FUTURE_WRITE;
            if (strchr(seals_arg, 'S') != NULL)
                seals |= F_SEAL_SEAL;

            if (fcntl(fd, F_ADD_SEALS, seals) == -1)
                err(EXIT_FAILURE, "fcntl");
        }

        /* Keep running, so that the file created by memfd_create()
           continues to exist. */

        pause();

        exit(EXIT_SUCCESS);
    }
```

Program source: t_get_seals.c

```
    #define _GNU_SOURCE
    #include <err.h>
    #include <fcntl.h>
    #include <stdio.h>
    #include <stdlib.h>

    int
    main(int argc, char *argv[])
    {
        int           fd;
        unsigned int  seals;

        if (argc != 2) {
            fprintf(stderr, "%s /proc/PID/fd/FD\n", argv[0]);
            exit(EXIT_FAILURE);
        }

        fd = open(argv[1], O_RDWR);
        if (fd == -1)
            err(EXIT_FAILURE, "open");

        seals = fcntl(fd, F_GET_SEALS);
```

```
        if (seals == -1)
            err(EXIT_FAILURE, "fcntl");

        printf("Existing seals:");
        if (seals & F_SEAL_SEAL)
            printf(" SEAL");
        if (seals & F_SEAL_GROW)
            printf(" GROW");
        if (seals & F_SEAL_WRITE)
            printf(" WRITE");
        if (seals & F_SEAL_FUTURE_WRITE)
            printf(" FUTURE_WRITE");
        if (seals & F_SEAL_SHRINK)
            printf(" SHRINK");
        printf("\n");

        /* Code to map the file and access the contents of the
           resulting mapping omitted. */

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

fcntl(2), **ftruncate**(2), **memfd_secret**(2), **mmap**(2), **shmget**(2), **shm_open**(3)