

NAME

membarrier – issue memory barriers on a set of threads

SYNOPSIS

```
#include <linux/membarrier.h>
```

```
int membarrier(int cmd, int flags);
```

DESCRIPTION

The **membarrier()** system call helps reducing the overhead of the memory barrier instructions required to order memory accesses on multi-core systems. However, this system call is heavier than a memory barrier, so using it effectively is *not* as simple as replacing memory barriers with this system call, but requires understanding of the details below.

Use of memory barriers needs to be done taking into account that a memory barrier always needs to be either matched with its memory barrier counterparts, or that the architecture's memory model doesn't require the matching barriers.

There are cases where one side of the matching barriers (which we will refer to as "fast side") is executed much more often than the other (which we will refer to as "slow side"). This is a prime target for the use of **membarrier()**. The key idea is to replace, for these matching barriers, the fast-side memory barriers by simple compiler barriers, for example:

```
asm volatile (" : : : \"memory\"")
```

and replace the slow-side memory barriers by calls to **membarrier()**.

This will add overhead to the slow side, and remove overhead from the fast side, thus resulting in an overall performance increase as long as the slow side is infrequent enough that the overhead of the **membarrier()** calls does not outweigh the performance gain on the fast side.

The *cmd* argument is one of the following:

MEMBARRIER_CMD_QUERY (since Linux 4.3)

Query the set of supported commands. The return value of the call is a bit mask of supported commands. **MEMBARRIER_CMD_QUERY**, which has the value 0, is not itself included in this bit mask. This command is always supported (on kernels where **membarrier()** is provided).

MEMBARRIER_CMD_GLOBAL (since Linux 4.16)

Ensure that all threads from all processes on the system pass through a state where all memory accesses to user-space addresses match program order between entry to and return from the **membarrier()** system call. All threads on the system are targeted by this command.

MEMBARRIER_CMD_GLOBAL_EXPEDITED (since Linux 4.16)

Execute a memory barrier on all running threads of all processes that previously registered with **MEMBARRIER_CMD_REGISTER_GLOBAL_EXPEDITED**.

Upon return from the system call, the calling thread has a guarantee that all running threads have passed through a state where all memory accesses to user-space addresses match program order between entry to and return from the system call (non-running threads are de facto in such a state). This guarantee is provided only for the threads of processes that previously registered with **MEMBARRIER_CMD_REGISTER_GLOBAL_EXPEDITED**.

Given that registration is about the intent to receive the barriers, it is valid to invoke **MEMBARRIER_CMD_GLOBAL_EXPEDITED** from a process that has not employed **MEMBARRIER_CMD_REGISTER_GLOBAL_EXPEDITED**.

The "expedited" commands complete faster than the non-expedited ones; they never block, but have the downside of causing extra overhead.

MEMBARRIER_CMD_REGISTER_GLOBAL_EXPEDITED (since Linux 4.16)

Register the process's intent to receive **MEMBARRIER_CMD_GLOBAL_EXPEDITED** memory barriers.

MEMBARRIER_CMD_PRIVATE_EXPEDITED (since Linux 4.14)

Execute a memory barrier on each running thread belonging to the same process as the calling thread.

Upon return from the system call, the calling thread has a guarantee that all its running thread siblings have passed through a state where all memory accesses to user-space addresses match program order between entry to and return from the system call (non-running threads are de facto in such a state). This guarantee is provided only for threads in the same process as the calling thread.

The "expedited" commands complete faster than the non-expedited ones; they never block, but have the downside of causing extra overhead.

A process must register its intent to use the private expedited command prior to using it.

MEMBARRIER_CMD_REGISTER_PRIVATE_EXPEDITED (since Linux 4.14)

Register the process's intent to use **MEMBARRIER_CMD_PRIVATE_EXPEDITED**.

MEMBARRIER_CMD_PRIVATE_EXPEDITED_SYNC_CORE (since Linux 4.16)

In addition to providing the memory ordering guarantees described in **MEMBARRIER_CMD_PRIVATE_EXPEDITED**, upon return from system call the calling thread has a guarantee that all its running thread siblings have executed a core serializing instruction. This guarantee is provided only for threads in the same process as the calling thread.

The "expedited" commands complete faster than the non-expedited ones, they never block, but have the downside of causing extra overhead.

A process must register its intent to use the private expedited sync core command prior to using it.

MEMBARRIER_CMD_REGISTER_PRIVATE_EXPEDITED_SYNC_CORE (since Linux 4.16)

Register the process's intent to use **MEMBARRIER_CMD_PRIVATE_EXPEDITED_SYNC_CORE**.

MEMBARRIER_CMD_SHARED (since Linux 4.3)

This is an alias for **MEMBARRIER_CMD_GLOBAL** that exists for header backward compatibility.

The *flags* argument is currently unused and must be specified as 0.

All memory accesses performed in program order from each targeted thread are guaranteed to be ordered with respect to **membarrier()**.

If we use the semantic *barrier()* to represent a compiler barrier forcing memory accesses to be performed in program order across the barrier, and *smp_mb()* to represent explicit memory barriers forcing full memory ordering across the barrier, we have the following ordering table for each pairing of *barrier()*, **membarrier()** and *smp_mb()*. The pair ordering is detailed as (O: ordered, X: not ordered):

	<i>barrier()</i>	<i>smp_mb()</i>	membarrier()
<i>barrier()</i>	X	X	O
<i>smp_mb()</i>	X	O	O
membarrier()	O	O	O

RETURN VALUE

On success, the **MEMBARRIER_CMD_QUERY** operation returns a bit mask of supported commands, and the **MEMBARRIER_CMD_GLOBAL**, **MEMBARRIER_CMD_GLOBAL_EXPEDITED**, **MEMBARRIER_CMD_REGISTER_GLOBAL_EXPEDITED**, **MEMBARRIER_CMD_PRIVATE_EXPEDITED**, **MEMBARRIER_CMD_REGISTER_PRIVATE_EXPEDITED**, **MEMBARRIER_CMD_PRIVATE_EXPEDITED_SYNC_CORE**, and **MEMBARRIER_CMD_REGISTER_PRIVATE_EXPEDITED_SYNC_CORE** operations return zero. On error, -1 is returned, and *errno* is set appropriately.

For a given command, with *flags* set to 0, this system call is guaranteed to always return the same value until reboot. Further calls with the same arguments will lead to the same result. Therefore, with *flags* set to 0, error handling is required only for the first call to **membarrier()**.

ERRORS**EINVAL**

cmd is invalid, or *flags* is nonzero, or the **MEMBARRIER_CMD_GLOBAL** command is disabled because the *nohz_full* CPU parameter has been set, or the **MEMBARRIER_CMD_PRIVATE_EXPEDITED_SYNC_CORE** and **MEMBARRIER_CMD_REGISTER_PRIVATE_EXPEDITED_SYNC_CORE** commands are not implemented by the architecture.

ENOSYS

The **membarrier()** system call is not implemented by this kernel.

EPERM

The current process was not registered prior to using private expedited commands.

VERSIONS

The **membarrier()** system call was added in Linux 4.3.

CONFORMING TO

membarrier() is Linux-specific.

NOTES

A memory barrier instruction is part of the instruction set of architectures with weakly-ordered memory models. It orders memory accesses prior to the barrier and after the barrier with respect to matching barriers on other cores. For instance, a load fence can order loads prior to and following that fence with respect to stores ordered by store fences.

Program order is the order in which instructions are ordered in the program assembly code.

Examples where **membarrier()** can be useful include implementations of Read-Copy-Update libraries and garbage collectors.

EXAMPLE

Assuming a multithreaded application where "fast_path()" is executed very frequently, and where "slow_path()" is executed infrequently, the following code (x86) can be transformed using **membarrier()**:

```
#include <stdlib.h>

static volatile int a, b;

static void
fast_path(int *read_b)
{
    a = 1;
    asm volatile ("mfence" : : : "memory");
    *read_b = b;
}

static void
slow_path(int *read_a)
{
    b = 1;
    asm volatile ("mfence" : : : "memory");
    *read_a = a;
}

int
main(int argc, char **argv)
{
    int read_a, read_b;

    /*
```

```

    * Real applications would call fast_path() and slow_path()
    * from different threads. Call those from main() to keep
    * this example short.
    */

slow_path(&read_a);
fast_path(&read_b);

/*
 * read_b == 0 implies read_a == 1 and
 * read_a == 0 implies read_b == 1.
 */

if (read_b == 0 && read_a == 0)
    abort();

exit(EXIT_SUCCESS);
}

```

The code above transformed to use **membarrier()** becomes:

```

#define _GNU_SOURCE
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <linux/membarrier.h>

static volatile int a, b;

static int
membarrier(int cmd, int flags)
{
    return syscall(__NR_membarrier, cmd, flags);
}

static int
init_membarrier(void)
{
    int ret;

    /* Check that membarrier() is supported. */

    ret = membarrier(MEMBARRIER_CMD_QUERY, 0);
    if (ret < 0) {
        perror("membarrier");
        return -1;
    }

    if (!(ret & MEMBARRIER_CMD_GLOBAL)) {
        fprintf(stderr,
            "membarrier does not support MEMBARRIER_CMD_GLOBAL\n");
        return -1;
    }

    return 0;
}

```

```
    }

    static void
    fast_path(int *read_b)
    {
        a = 1;
        asm volatile ("": : : "memory");
        *read_b = b;
    }

    static void
    slow_path(int *read_a)
    {
        b = 1;
        membarrier(MEMBARRIER_CMD_GLOBAL, 0);
        *read_a = a;
    }

    int
    main(int argc, char **argv)
    {
        int read_a, read_b;

        if (init_membarrier())
            exit(EXIT_FAILURE);

        /*
         * Real applications would call fast_path() and slow_path()
         * from different threads. Call those from main() to keep
         * this example short.
         */

        slow_path(&read_a);
        fast_path(&read_b);

        /*
         * read_b == 0 implies read_a == 1 and
         * read_a == 0 implies read_b == 1.
         */

        if (read_b == 0 && read_a == 0)
            abort();

        exit(EXIT_SUCCESS);
    }
}
```

COLOPHON

This page is part of release 5.05 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at <https://www.kernel.org/doc/man-pages/>.