userfaultfd(2) — Linux manual page

Search online pages

USERFAULTFD(2) Linux Programmer's Manual USERFAULTFD(2)

NAME top

userfaultfd - create a file descriptor for handling page faults in user space

SYNOPSIS top

```
#include <fcntl.h>            /* Definition of O_* constants */
#include <sys/syscall.h>      /* Definition of SYS_* constants */
#include <unistd.h>

int syscall(SYS_userfaultfd, int flags);
```

Note: glibc provides no wrapper for **userfaultfd**(), necessitating the use of syscall(2).

DESCRIPTION top

userfaultfd() creates a new userfaultfd object that can be used for delegation of page-fault handling to a user-space application, and returns a file descriptor that refers to the new object. The new userfaultfd object is configured using ioctl(2).

Once the userfaultfd object is configured, the application can use read(2) to receive userfaultfd notifications. The reads from userfaultfd may be blocking or non-blocking, depending on the value of *flags* used for the creation of the userfaultfd or subsequent calls to fcntl(2).

The following values may be bitwise ORed in *flags* to change the behavior of **userfaultfd**():

O_CLOEXEC
    Enable the close-on-exec flag for the new userfaultfd file descriptor. See the description of the **O_CLOEXEC** flag in

open(2).

O_NONBLOCK
Enables non-blocking operation for the userfaultfd object. See the description of the **O_NONBLOCK** flag in open(2).

When the last file descriptor referring to a userfaultfd object is closed, all memory ranges that were registered with the object are unregistered and unread events are flushed.

Userfaultfd supports two modes of registration:

UFFDIO_REGISTER_MODE_MISSING (since 4.10)
When registered with **UFFDIO_REGISTER_MODE_MISSING** mode, user-space will receive a page-fault notification when a missing page is accessed. The faulted thread will be stopped from execution until the page fault is resolved from user-space by either an **UFFDIO_COPY** or an **UFFDIO_ZEROPAGE** ioctl.

UFFDIO_REGISTER_MODE_WP (since 5.7)
When registered with **UFFDIO_REGISTER_MODE_WP** mode, user-space will receive a page-fault notification when a write-protected page is written. The faulted thread will be stopped from execution until user-space write-unprotects the page using an **UFFDIO_WRITEPROTECT** ioctl.

Multiple modes can be enabled at the same time for the same memory range.

Since Linux 4.14, a userfaultfd page-fault notification can selectively embed faulting thread ID information into the notification. One needs to enable this feature explicitly using the **UFFD_FEATURE_THREAD_ID** feature bit when initializing the userfaultfd context. By default, thread ID reporting is disabled.

Usage

The userfaultfd mechanism is designed to allow a thread in a multithreaded program to perform user-space paging for the other threads in the process. When a page fault occurs for one of the regions registered to the userfaultfd object, the faulting thread is put to sleep and an event is generated that can be read via the userfaultfd file descriptor. The fault-handling thread reads events from this file descriptor and services them using the operations described in ioctl_userfaultfd(2). When servicing the page fault events, the fault-handling thread can trigger a wake-up for the sleeping thread.

It is possible for the faulting threads and the fault-handling threads to run in the context of different processes. In this case, these threads may belong to different programs, and the program that executes the faulting threads will not necessarily cooperate with the program that handles the page faults. In such non-cooperative mode, the process that monitors userfaultfd and handles page faults needs to be aware of the changes in the virtual memory layout of the faulting process to avoid memory corruption.

Since Linux 4.11, userfaultfd can also notify the fault-handling threads about changes in the virtual memory layout of the faulting process. In addition, if the faulting process invokes fork(2), the userfaultfd objects associated with the parent may be duplicated into the child process and the userfaultfd monitor will be notified (via the **UFFD_EVENT_FORK** described below) about the file descriptor associated with the userfault objects created for the child process, which allows the userfaultfd monitor to perform user-space paging for the child process. Unlike page faults which have to be synchronous and require an explicit or implicit wakeup, all other events are delivered asynchronously and the non-cooperative process resumes execution as soon as the userfaultfd manager executes read(2). The userfaultfd manager should carefully synchronize calls to **UFFDIO_COPY** with the processing of events.

The current asynchronous model of the event delivery is optimal for single threaded non-cooperative userfaultfd manager implementations.

Since Linux 5.7, userfaultfd is able to do synchronous page dirty tracking using the new write-protect register mode. One should check against the feature bit **UFFD_FEATURE_PAGEFAULT_FLAG_WP** before using this feature. Similar to the original userfaultfd missing mode, the write-protect mode will generate a userfaultfd notification when the protected page is written. The user needs to resolve the page fault by unprotecting the faulted page and kicking the faulted thread to continue. For more information, please refer to the "Userfaultfd write-protect mode" section.

Userfaultfd operation

After the userfaultfd object is created with **userfaultfd**(), the application must enable it using the **UFFDIO_API** **ioctl**(2) operation. This operation allows a handshake between the kernel and user space to determine the API version and supported features. This operation must be performed before any of the other ioctl(2) operations described below (or those operations fail with the **EINVAL** error).

After a successful **UFFDIO_API** operation, the application then registers memory address ranges using the **UFFDIO_REGISTER** ioctl(2) operation. After successful completion of a **UFFDIO_REGISTER** operation, a page fault occurring in the requested memory range, and satisfying the mode defined at the registration time, will be forwarded by the kernel to the user-space application. The application can then use the **UFFDIO_COPY** or **UFFDIO_ZEROPAGE** **ioctl**(2) operations to resolve the page fault.

Since Linux 4.14, if the application sets the **UFFD_FEATURE_SIGBUS** feature bit using the **UFFDIO_API** **ioctl**(2), no page-fault notification will be forwarded to user space. Instead a **SIGBUS** signal is delivered to the faulting process. With this feature, userfaultfd can be used for robustness purposes to simply catch any access to areas within the registered address range that do not have pages allocated, without having to listen to userfaultfd events. No userfaultfd monitor will be required for dealing with such memory accesses. For example, this feature can be useful for applications that want to prevent the kernel from automatically allocating pages and filling holes in sparse files when the hole is accessed through a memory mapping.

The **UFFD_FEATURE_SIGBUS** feature is implicitly inherited through fork(2) if used in combination with **UFFD_FEATURE_FORK**.

Details of the various ioctl(2) operations can be found in ioctl_userfaultfd(2).

Since Linux 4.11, events other than page-fault may enabled during **UFFDIO_API** operation.

Up to Linux 4.11, userfaultfd can be used only with anonymous private memory mappings. Since Linux 4.11, userfaultfd can be also used with hugetlbfs and shared memory mappings.

Userfaultfd write-protect mode (since 5.7)

Since Linux 5.7, userfaultfd supports write-protect mode. The user needs to first check availability of this feature using **UFFDIO_API** ioctl against the feature bit **UFFD_FEATURE_PAGEFAULT_FLAG_WP** before using this feature.

To register with userfaultfd write-protect mode, the user needs to initiate the **UFFDIO_REGISTER** ioctl with mode **UFFDIO_REGISTER_MODE_WP** set. Note that it is legal to monitor the same memory range with multiple modes. For example, the user can do **UFFDIO_REGISTER** with the mode set to **UFFDIO_REGISTER_MODE_MISSING | UFFDIO_REGISTER_MODE_WP**. When there is only **UFFDIO_REGISTER_MODE_WP** registered, user-space will *not* receive any notification when a missing page is written. Instead, user-space will receive a write-protect page-fault notification only when an existing but write-protected page got written.

After the **UFFDIO_REGISTER** ioctl completed with **UFFDIO_REGISTER_MODE_WP** mode set, the user can write-protect any existing memory within the range using the ioctl **UFFDIO_WRITEPROTECT** where *uffdio_writeprotect.mode* should be set to **UFFDIO_WRITEPROTECT_MODE_WP**.

When a write-protect event happens, user-space will receive a page-fault notification whose *uffd_msg.pagefault.flags* will be with **UFFD_PAGEFAULT_FLAG_WP** flag set. Note: since only writes can trigger this kind of fault, write-protect notifications will always have the **UFFD_PAGEFAULT_FLAG_WRITE** bit set along with the **UFFD_PAGEFAULT_FLAG_WP** bit.

To resolve a write-protection page fault, the user should initiate another **UFFDIO_WRITEPROTECT** ioctl, whose *uffd_msg.pagefault.flags* should have the flag **UFFDIO_WRITEPROTECT_MODE_WP** cleared upon the faulted page or range.

Write-protect mode supports only private anonymous memory.

Reading from the userfaultfd structure

Each read(2) from the userfaultfd file descriptor returns one or more *uffd_msg* structures, each of which describes a page-fault event or an event required for the non-cooperative userfaultfd usage:

```
struct uffd_msg {
    __u8  event;            /* Type of event */
    ...
    union {
        struct {
            __u64 flags;    /* Flags describing fault */
            __u64 address;  /* Faulting address */
            union {
                __u32 ptid; /* Thread ID of the fault */
            } feat;
        } pagefault;

        struct {            /* Since Linux 4.11 */
            __u32 ufd;      /* Userfault file descriptor
                               of the child process */
        } fork;

        struct {            /* Since Linux 4.11 */
            __u64 from;     /* Old address of remapped area */
            __u64 to;       /* New address of remapped area */
            __u64 len;      /* Original mapping length */
        } remap;

        struct {            /* Since Linux 4.11 */
            __u64 start;    /* Start address of removed area */
            __u64 end;      /* End address of removed area */
        } remove;
        ...
    } arg;
```

```
        /* Padding fields omitted */
    } __packed;
```

If multiple events are available and the supplied buffer is large enough, read(2) returns as many events as will fit in the supplied buffer. If the buffer supplied to read(2) is smaller than the size of the *uffd_msg* structure, the read(2) fails with the error **EINVAL**.

The fields set in the *uffd_msg* structure are as follows:

event The type of event. Depending of the event type, different fields of the *arg* union represent details required for the event processing. The non-page-fault events are generated only when appropriate feature is enabled during API handshake with **UFFDIO_API ioctl**(2).

The following values can appear in the *event* field:

UFFD_EVENT_PAGEFAULT (since Linux 4.3)
A page-fault event. The page-fault details are available in the *pagefault* field.

UFFD_EVENT_FORK (since Linux 4.11)
Generated when the faulting process invokes fork(2) (or clone(2) without the **CLONE_VM** flag). The event details are available in the *fork* field.

UFFD_EVENT_REMAP (since Linux 4.11)
Generated when the faulting process invokes mremap(2). The event details are available in the *remap* field.

UFFD_EVENT_REMOVE (since Linux 4.11)
Generated when the faulting process invokes madvise(2) with **MADV_DONTNEED** or **MADV_REMOVE** advice. The event details are available in the *remove* field.

UFFD_EVENT_UNMAP (since Linux 4.11)
Generated when the faulting process unmaps a memory range, either explicitly using munmap(2) or implicitly during mmap(2) or mremap(2). The event details are available in the *remove* field.

pagefault.address
The address that triggered the page fault.

pagefault.flags
A bit mask of flags that describe the event. For **UFFD_EVENT_PAGEFAULT**, the following flag may appear:

UFFD_PAGEFAULT_FLAG_WRITE
If the address is in a range that was registered with the **UFFDIO_REGISTER_MODE_MISSING** flag (see ioctl_userfaultfd(2)) and this flag is set, this a write fault; otherwise it is a read fault.

UFFD_PAGEFAULT_FLAG_WP
If the address is in a range that was registered with the **UFFDIO_REGISTER_MODE_WP** flag, when this bit is set, it means it is a write-protect fault. Otherwise it is a page-missing fault.

pagefault.feat.pid
The thread ID that triggered the page fault.

fork.ufd
The file descriptor associated with the userfault object created for the child created by fork(2).

remap.from
The original address of the memory range that was remapped using mremap(2).

remap.to
The new address of the memory range that was remapped using mremap(2).

remap.len
The original length of the memory range that was remapped using mremap(2).

remove.start
The start address of the memory range that was freed using madvise(2) or unmapped

remove.end
The end address of the memory range that was freed using madvise(2) or unmapped

A read(2) on a userfaultfd file descriptor can fail with the following errors:

EINVAL The userfaultfd object has not yet been enabled using the **UFFDIO_API** **ioctl**(2) operation

If the **O_NONBLOCK** flag is enabled in the associated open file description, the userfaultfd file descriptor can be monitored with poll(2), select(2), and epoll(7). When events are available, the file descriptor indicates as readable. If the **O_NONBLOCK** flag is not enabled, then poll(2) (always) indicates the file as having a **POLLERR** condition, and select(2) indicates the file descriptor as both readable and writable.

RETURN VALUE top

On success, **userfaultfd**() returns a new file descriptor that refers to the userfaultfd object. On error, -1 is returned, and *errno* is set to indicate the error.

ERRORS top

EINVAL An unsupported value was 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 Insufficient kernel memory was available.

EPERM (since Linux 5.2)
The caller is not privileged (does not have the **CAP_SYS_PTRACE** capability in the initial user namespace), and */proc/sys/vm/unprivileged_userfaultfd* has the value 0.

VERSIONS top

The **userfaultfd**() system call first appeared in Linux 4.3.

The support for hugetlbfs and shared memory areas and non-page-fault events was added in Linux 4.11

CONFORMING TO top

userfaultfd() is Linux-specific and should not be used in programs intended to be portable.

NOTES top

The userfaultfd mechanism can be used as an alternative to traditional user-space paging techniques based on the use of the **SIGSEGV** signal and mmap(2). It can also be used to implement lazy restore for checkpoint/restore mechanisms, as well as post-copy migration to allow (nearly) uninterrupted execution when transferring virtual machines and Linux containers from one host to another.

BUGS top

If the **UFFD_FEATURE_EVENT_FORK** is enabled and a system call from the fork(2) family is interrupted by a signal or failed, a stale userfaultfd descriptor might be created. In this case, a spurious **UFFD_EVENT_FORK** will be delivered to the userfaultfd monitor.

EXAMPLES top

The program below demonstrates the use of the userfaultfd mechanism. The program creates two threads, one of which acts as the page-fault handler for the process, for the pages in a demand-page zero region created using mmap(2).

The program takes one command-line argument, which is the number of pages that will be created in a mapping whose page faults will be handled via userfaultfd. After creating a userfaultfd object, the program then creates an anonymous private mapping of the specified size and registers the address range of that mapping using the **UFFDIO_REGISTER ioctl**(2) operation. The program then creates a second thread that will perform the task of handling page faults.

The main thread then walks through the pages of the mapping fetching bytes from successive pages. Because the pages have not yet been accessed, the first access of a byte in each page will trigger a page-fault event on the userfaultfd file descriptor.

Each of the page-fault events is handled by the second thread, which sits in a loop processing input from the userfaultfd file descriptor. In each loop iteration, the second thread first calls poll(2) to check the state of the file descriptor, and then reads an event from the file descriptor. All such events should be **UFFD_EVENT_PAGEFAULT** events, which the thread handles by copying a page of data into the faulting region using the **UFFDIO_COPY ioctl**(2) operation.

The following is an example of what we see when running the program:

```
$ ./userfaultfd_demo 3
Address returned by mmap() = 0x7fd30106c000

fault_handler_thread():
    poll() returns: nready = 1; POLLIN = 1; POLLERR = 0
    UFFD_EVENT_PAGEFAULT event: flags = 0; address = 7fd30106c00f
        (uffdio_copy.copy returned 4096)
Read address 0x7fd30106c00f in main(): A
Read address 0x7fd30106c40f in main(): A
Read address 0x7fd30106c80f in main(): A
Read address 0x7fd30106cc0f in main(): A
```

```
    fault_handler_thread():
        poll() returns: nready = 1; POLLIN = 1; POLLERR = 0
        UFFD_EVENT_PAGEFAULT event: flags = 0; address = 7fd30106d00f
            (uffdio_copy.copy returned 4096)
    Read address 0x7fd30106d00f in main(): B
    Read address 0x7fd30106d40f in main(): B
    Read address 0x7fd30106d80f in main(): B
    Read address 0x7fd30106dc0f in main(): B

    fault_handler_thread():
        poll() returns: nready = 1; POLLIN = 1; POLLERR = 0
        UFFD_EVENT_PAGEFAULT event: flags = 0; address = 7fd30106e00f
            (uffdio_copy.copy returned 4096)
    Read address 0x7fd30106e00f in main(): C
    Read address 0x7fd30106e40f in main(): C
    Read address 0x7fd30106e80f in main(): C
    Read address 0x7fd30106ec0f in main(): C
```

Program source

```
/* userfaultfd_demo.c

   Licensed under the GNU General Public License version 2 or later.
*/
#define _GNU_SOURCE
#include <inttypes.h>
#include <sys/types.h>
#include <stdio.h>
#include <linux/userfaultfd.h>
#include <pthread.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <signal.h>
#include <poll.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/syscall.h>
#include <sys/ioctl.h>
#include <poll.h>

#define errExit(msg)    do { perror(msg); exit(EXIT_FAILURE); \
                        } while (0)

static int page_size;

static void *
fault_handler_thread(void *arg)
{
    static struct uffd_msg msg;   /* Data read from userfaultfd */
    static int fault_cnt = 0;     /* Number of faults so far handled */
```

```
        long uffd;                    /* userfaultfd file descriptor */
        static char *page = NULL;
        struct uffdio_copy uffdio_copy;
        ssize_t nread;

        uffd = (long) arg;

        /* Create a page that will be copied into the faulting region. */

        if (page == NULL) {
            page = mmap(NULL, page_size, PROT_READ | PROT_WRITE,
                        MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
            if (page == MAP_FAILED)
                errExit("mmap");
        }

        /* Loop, handling incoming events on the userfaultfd
           file descriptor. */

        for (;;) {

            /* See what poll() tells us about the userfaultfd. */

            struct pollfd pollfd;
            int nready;
            pollfd.fd = uffd;
            pollfd.events = POLLIN;
            nready = poll(&pollfd, 1, -1);
            if (nready == -1)
                errExit("poll");

            printf("\nfault_handler_thread():\n");
            printf("    poll() returns: nready = %d; "
                    "POLLIN = %d; POLLERR = %d\n", nready,
                    (pollfd.revents & POLLIN) != 0,
                    (pollfd.revents & POLLERR) != 0);

            /* Read an event from the userfaultfd. */

            nread = read(uffd, &msg, sizeof(msg));
            if (nread == 0) {
                printf("EOF on userfaultfd!\n");
                exit(EXIT_FAILURE);
            }

            if (nread == -1)
                errExit("read");

            /* We expect only one kind of event; verify that assumption. */

            if (msg.event != UFFD_EVENT_PAGEFAULT) {
                fprintf(stderr, "Unexpected event on userfaultfd\n");
```

```
            exit(EXIT_FAILURE);
        }

        /* Display info about the page-fault event. */

        printf("    UFFD_EVENT_PAGEFAULT event: ");
        printf("flags = %"PRIx64"; ", msg.arg.pagefault.flags);
        printf("address = %"PRIx64"\n", msg.arg.pagefault.address);

        /* Copy the page pointed to by 'page' into the faulting
           region. Vary the contents that are copied in, so that it
           is more obvious that each fault is handled separately. */

        memset(page, 'A' + fault_cnt % 20, page_size);
        fault_cnt++;

        uffdio_copy.src = (unsigned long) page;

        /* We need to handle page faults in units of pages(!).
           So, round faulting address down to page boundary. */

        uffdio_copy.dst = (unsigned long) msg.arg.pagefault.address &
                                           ~(page_size - 1);
        uffdio_copy.len = page_size;
        uffdio_copy.mode = 0;
        uffdio_copy.copy = 0;
        if (ioctl(uffd, UFFDIO_COPY, &uffdio_copy) == -1)
            errExit("ioctl-UFFDIO_COPY");

        printf("        (uffdio_copy.copy returned %"PRId64")\n",
                uffdio_copy.copy);
    }
}

int
main(int argc, char *argv[])
{
    long uffd;          /* userfaultfd file descriptor */
    char *addr;         /* Start of region handled by userfaultfd */
    uint64_t len;       /* Length of region handled by userfaultfd */
    pthread_t thr;      /* ID of thread that handles page faults */
    struct uffdio_api uffdio_api;
    struct uffdio_register uffdio_register;
    int s;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s num-pages\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    page_size = sysconf(_SC_PAGE_SIZE);
    len = strtoull(argv[1], NULL, 0) * page_size;
```

```
    /* Create and enable userfaultfd object. */

    uffd = syscall(__NR_userfaultfd, O_CLOEXEC | O_NONBLOCK);
    if (uffd == -1)
        errExit("userfaultfd");

    uffdio_api.api = UFFD_API;
    uffdio_api.features = 0;
    if (ioctl(uffd, UFFDIO_API, &uffdio_api) == -1)
        errExit("ioctl-UFFDIO_API");

    /* Create a private anonymous mapping. The memory will be
       demand-zero paged--that is, not yet allocated. When we
       actually touch the memory, it will be allocated via
       the userfaultfd. */

    addr = mmap(NULL, len, PROT_READ | PROT_WRITE,
                MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    if (addr == MAP_FAILED)
        errExit("mmap");

    printf("Address returned by mmap() = %p\n", addr);

    /* Register the memory range of the mapping we just created for
       handling by the userfaultfd object. In mode, we request to track
       missing pages (i.e., pages that have not yet been faulted in). */

    uffdio_register.range.start = (unsigned long) addr;
    uffdio_register.range.len = len;
    uffdio_register.mode = UFFDIO_REGISTER_MODE_MISSING;
    if (ioctl(uffd, UFFDIO_REGISTER, &uffdio_register) == -1)
        errExit("ioctl-UFFDIO_REGISTER");

    /* Create a thread that will process the userfaultfd events. */

    s = pthread_create(&thr, NULL, fault_handler_thread, (void *) uffd);
    if (s != 0) {
        errno = s;
        errExit("pthread_create");
    }

    /* Main thread now touches memory in the mapping, touching
       locations 1024 bytes apart. This will trigger userfaultfd
       events for all pages in the region. */

    int l;
    l = 0xf;    /* Ensure that faulting address is not on a page
                   boundary, in order to test that we correctly
                   handle that case in fault_handling_thread(). */
    while (l < len) {
        char c = addr[l];
```

```
            printf("Read address %p in main(): ", addr + l);
            printf("%c\n", c);
            l += 1024;
            usleep(100000);         /* Slow things down a little */
        }

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO top

fcntl(2), ioctl(2), ioctl_userfaultfd(2), madvise(2), mmap(2)

Documentation/admin-guide/mm/userfaultfd.rst in the Linux kernel source tree

COLOPHON top

This page is part of release 5.13 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/.

Linux **2021-03-22** **USERFAULTFD(2)**

Pages that refer to this page: ioctl_userfaultfd(2), mmap(2), mremap(2), syscalls(2), proc(5)