Userfaultfd

Objective

Userfaults allow the implementation of on-demand paging from userland and more generally they allow userland to take control of various memory page faults, something otherwise only the kernel code could do.

For example userfaults allows a proper and more optimal implementation of the `PROT_NONE+SIGSEGV` trick.

Design

Userfaults are delivered and resolved through the `userfaultfd` syscall.

The `userfaultfd` (aside from registering and unregistering virtual memory ranges) provides two primary functionalities:

1. `read/POLLIN` protocol to notify a userland thread of the faults happening
2. various `UFFDIO_*` ioctls that can manage the virtual memory regions registered in the `userfaultfd` that allows userland to efficiently resolve the userfaults it receives via 1) or to manage the virtual memory in the background

The real advantage of userfaults if compared to regular virtual memory management of mremap/mprotect is that the userfaults in all their operations never involve heavyweight structures like vmas (in fact the `userfaultfd` runtime load never takes the mmap_lock for writing).

Vmas are not suitable for page- (or hugepage) granular fault tracking when dealing with virtual address spaces that could span Terabytes. Too many vmas would be needed for that.

The `userfaultfd` once opened by invoking the syscall, can also be passed using unix domain sockets to a manager process, so the same manager process could handle the userfaults of a multitude of different processes without them being aware about what is going on (well of course unless they later try to use the `userfaultfd` themselves on the same region the manager is already tracking, which is a corner case that would currently return `-EBUSY`).

API

When first opened the `userfaultfd` must be enabled invoking the `UFFDIO_API` ioctl specifying a `uffdio_api.api` value set to `UFFD_API` (or a later API version) which will specify the `read/POLLIN` protocol userland intends to speak on the `UFFD` and the `uffdio_api.features` userland requires. The `UFFDIO_API` ioctl if successful (i.e. if the requested `uffdio_api.api` is spoken also by the running kernel and the requested features are going to be enabled) will return into `uffdio_api.features` and `uffdio_api.ioctls` two 64bit bitmasks of respectively all the available features of the read(2) protocol and the generic ioctl available.

The `uffdio_api.features` bitmask returned by the `UFFDIO_API` ioctl defines what memory types are supported by the `userfaultfd` and what events, except page fault notifications, may be generated:

- The `UFFD_FEATURE_EVENT_*` flags indicate that various other events other than page faults are supported. These events are described in more detail below in the Non-cooperative userfaultfd section.
- `UFFD_FEATURE_MISSING_HUGETLBFS` and `UFFD_FEATURE_MISSING_SHMEM` indicate that the kernel supports `UFFDIO_REGISTER_MODE_MISSING` registrations for hugetlbfs and shared memory (covering all shmem APIs, i.e. tmpfs, `IPCSHM`, `/dev/zero`, `MAP_SHARED`, `memfd_create`, etc) virtual memory areas, respectively.
- `UFFD_FEATURE_MINOR_HUGETLBFS` indicates that the kernel supports `UFFDIO_REGISTER_MODE_MINOR` registration for hugetlbfs virtual memory areas. `UFFD_FEATURE_MINOR_SHMEM` is the analogous feature indicating support for shmem virtual memory areas.

The userland application should set the feature flags it intends to use when invoking the `UFFDIO_API` ioctl, to request that those features be enabled if supported.

Once the `userfaultfd` API has been enabled the `UFFDIO_REGISTER` ioctl should be invoked (if present in the returned `uffdio_api.ioctls` bitmask) to register a memory range in the `userfaultfd` by setting the uffdio_register structure accordingly. The `uffdio_register.mode` bitmask will specify to the kernel which kind of faults to track for the range. The `UFFDIO_REGISTER` ioctl will return the `uffdio_register.ioctls` bitmask of ioctls that are suitable to resolve userfaults on the range registered. Not all ioctls will necessarily be supported for all memory types (e.g. anonymous memory vs. shmem vs. hugetlbfs), or all types of intercepted faults.

Userland can use the `uffdio_register.ioctls` to manage the virtual address space in the background (to add or potentially also remove memory from the `userfaultfd` registered range). This means a userfault could be triggering just before userland maps in the background the user-faulted page.

Resolving Userfaults

There are three basic ways to resolve userfaults:

- `UFFDIO_COPY` atomically copies some existing page contents from userspace.
- `UFFDIO_ZEROPAGE` atomically zeros the new page.
- `UFFDIO_CONTINUE` maps an existing, previously-populated page.

These operations are atomic in the sense that they guarantee nothing can see a half-populated page, since readers will keep userfaulting until the operation has finished.

By default, these wake up userfaults blocked on the range in question. They support a `UFFDIO_*_MODE_DONTWAKE` `mode` flag, which indicates that waking will be done separately at some later time.

Which ioctl to choose depends on the kind of page fault, and what we'd like to do to resolve it:

- For `UFFDIO_REGISTER_MODE_MISSING` faults, the fault needs to be resolved by either providing a new page (`UFFDIO_COPY`), or mapping the zero page (`UFFDIO_ZEROPAGE`). By default, the kernel would map the zero page for a missing fault. With userfaultfd, userspace can decide what content to provide before the faulting thread continues.
- For `UFFDIO_REGISTER_MODE_MINOR` faults, there is an existing page (in the page cache). Userspace has the option of modifying the page's contents before resolving the fault. Once the contents are correct (modified or not), userspace asks the kernel to map the page and let the faulting thread continue with `UFFDIO_CONTINUE`.

Notes:

- You can tell which kind of fault occurred by examining `pagefault.flags` within the `uffd_msg`, checking for the `UFFD_PAGEFAULT_FLAG_*` flags.
- None of the page-delivering ioctls default to the range that you registered with. You must fill in all fields for the appropriate ioctl struct including the range.
- You get the address of the access that triggered the missing page event out of a struct uffd_msg that you read in the thread from the uffd. You can supply as many pages as you want with these IOCTLs. Keep in mind that unless you used DONTWAKE then the first of any of those IOCTLs wakes up the faulting thread.
- Be sure to test for all errors including (`pollfd[0].revents & POLLERR`). This can happen, e.g. when ranges supplied were incorrect.

Write Protect Notifications

This is equivalent to (but faster than) using mprotect and a SIGSEGV signal handler.

Firstly you need to register a range with `UFFDIO_REGISTER_MODE_WP`. Instead of using mprotect(2) you use `ioctl(uffd, UFFDIO_WRITEPROTECT, struct *uffdio_writeprotect)` while `mode = UFFDIO_WRITEPROTECT_MODE_WP` in the struct passed in. The range does not default to and does not have to be identical to the range you registered with. You can write protect as many ranges as you like (inside the registered range). Then, in the thread reading from uffd the struct will have `msg.arg.pagefault.flags & UFFD_PAGEFAULT_FLAG_WP` set. Now you send `ioctl(uffd, UFFDIO_WRITEPROTECT, struct *uffdio_writeprotect)` again while `pagefault.mode` does not have `UFFDIO_WRITEPROTECT_MODE_WP` set. This wakes up the thread which will continue to run with writes. This allows you to do the bookkeeping about the write in the uffd reading thread before the ioctl.

If you registered with both `UFFDIO_REGISTER_MODE_MISSING` and `UFFDIO_REGISTER_MODE_WP` then you need to think about the sequence in which you supply a page and undo write protect. Note that there is a difference between writes into a WP area and into a !WP area. The former will have `UFFD_PAGEFAULT_FLAG_WP` set, the latter `UFFD_PAGEFAULT_FLAG_WRITE`. The latter did not fail on protection but you still need to supply a page when `UFFDIO_REGISTER_MODE_MISSING` was used.

QEMU/KVM

QEMU/KVM is using the `userfaultfd` syscall to implement postcopy live migration. Postcopy live migration is one form of memory externalization consisting of a virtual machine running with part or all of its memory residing on a different node in the cloud. The `userfaultfd` abstraction is generic enough that not a single line of KVM kernel code had to be modified in order to add postcopy live migration to QEMU.

Guest async page faults, `FOLL_NOWAIT` and all other `GUP*` features work just fine in combination with userfaults. Userfaults trigger async page faults in the guest scheduler so those guest processes that aren't waiting for userfaults (i.e. network bound) can keep running in the guest vcpus.

It is generally beneficial to run one pass of precopy live migration just before starting postcopy live migration, in order to avoid generating userfaults for readonly guest regions.

The implementation of postcopy live migration currently uses one single bidirectional socket but in the future two different sockets will be used (to reduce the latency of the userfaults to the minimum possible without having to decrease `/proc/sys/net/ipv4/tcp_wmem`).

The QEMU in the source node writes all pages that it knows are missing in the destination node, into the socket, and the migration thread of the QEMU running in the destination node runs `UFFDIO_COPY|ZEROPAGE` ioctls on the `userfaultfd` in order to map the received pages into the guest (`UFFDIO_ZEROCOPY` is used if the source page was a zero page).

A different postcopy thread in the destination node listens with poll() to the `userfaultfd` in parallel. When a `POLLIN` event is generated after a userfault triggers, the postcopy thread read() from the `userfaultfd` and receives the fault address (or `-EAGAIN` in case the userfault was already resolved and waken by a UFFDIO_COPY|ZEROPAGE run by the parallel QEMU migration thread).

After the QEMU postcopy thread (running in the destination node) gets the userfault address it writes the information about the missing page into the socket. The QEMU source node receives the information and roughly "seeks" to that page address and continues sending all remaining missing pages from that new page offset. Soon after that (just the time to flush the tcp_wmem queue through the network) the migration thread in the QEMU running in the destination node will receive the page that triggered the userfault and it'll map it as usual with the UFFDIO_COPY|ZEROPAGE (without actually knowing if it was spontaneously sent by the source or if it was an urgent page requested through a userfault).

By the time the userfaults start, the QEMU in the destination node doesn't need to keep any per-page state bitmap relative to the live migration around and a single per-page bitmap has to be maintained in the QEMU running in the source node to know which pages are still missing in the destination node. The bitmap in the source node is checked to find which missing pages to send in round robin and we seek over it when receiving incoming userfaults. After sending each page of course the bitmap is updated accordingly. It's also useful to avoid sending the same page twice (in case the userfault is read by the postcopy thread just before UFFDIO_COPY|ZEROPAGE runs in the migration thread).

Non-cooperative userfaultfd

When the userfaultfd is monitored by an external manager, the manager must be able to track changes in the process virtual memory layout. Userfaultfd can notify the manager about such changes using the same read(2) protocol as for the page fault notifications. The manager has to explicitly enable these events by setting appropriate bits in uffdio_api.features passed to UFFDIO_API ioctl:

UFFD_FEATURE_EVENT_FORK
: enable userfaultfd hooks for fork(). When this feature is enabled, the userfaultfd context of the parent process is duplicated into the newly created process. The manager receives UFFD_EVENT_FORK with file descriptor of the new userfaultfd context in the uffd_msg.fork.

UFFD_FEATURE_EVENT_REMAP
: enable notifications about mremap() calls. When the non-cooperative process moves a virtual memory area to a different location, the manager will receive UFFD_EVENT_REMAP. The uffd_msg.remap will contain the old and new addresses of the area and its original length.

UFFD_FEATURE_EVENT_REMOVE
: enable notifications about madvise(MADV_REMOVE) and madvise(MADV_DONTNEED) calls. The event UFFD_EVENT_REMOVE will be generated upon these calls to madvise(). The uffd_msg.remove will contain start and end addresses of the removed area.

UFFD_FEATURE_EVENT_UNMAP
: enable notifications about memory unmapping. The manager will get UFFD_EVENT_UNMAP with uffd_msg.remove containing start and end addresses of the unmapped area.

Although the UFFD_FEATURE_EVENT_REMOVE and UFFD_FEATURE_EVENT_UNMAP are pretty similar, they quite differ in the action expected from the userfaultfd manager. In the former case, the virtual memory is removed, but the area is not, the area remains monitored by the userfaultfd, and if a page fault occurs in that area it will be delivered to the manager. The proper resolution for such page fault is to zeromap the faulting address. However, in the latter case, when an area is unmapped, either explicitly (with munmap() system call), or implicitly (e.g. during mremap()), the area is removed and in turn the userfaultfd context for such area disappears too and the manager will not get further userland page faults from the removed area. Still, the notification is required in order to prevent manager from using UFFDIO_COPY on the unmapped area.

Unlike userland page faults which have to be synchronous and require explicit or implicit wakeup, all the events are delivered asynchronously and the non-cooperative process resumes execution as soon as manager executes read(). The userfaultfd manager should carefully synchronize calls to UFFDIO_COPY with the events processing. To aid the synchronization, the UFFDIO_COPY ioctl will return -ENOSPC when the monitored process exits at the time of UFFDIO_COPY, and -ENOENT, when the non-cooperative process has changed its virtual memory layout simultaneously with outstanding UFFDIO_COPY operation.

The current asynchronous model of the event delivery is optimal for single threaded non-cooperative userfaultfd manager implementations. A synchronous event delivery model can be added later as a new userfaultfd feature to facilitate multithreading enhancements of the non cooperative manager, for example to allow UFFDIO_COPY ioctls to run in parallel to the event reception. Single threaded implementations should continue to use the current async event delivery model instead.