NO-MMU MEMORY MAPPING SUPPORT

The kernel has limited support for memory mapping under no-MMU conditions, such as are used in uClinux environments. From the userspace point of view, memory mapping is made use of in conjunction with the mmap() system call, the shmat() call and the execve() system call. From the kernel's point of view, execve() mapping is actually performed by the binfmt drivers, which call back into the mmap() routines to do the actual work.

Memory mapping behaviour also involves the way fork(), vfork(), clone() and ptrace() work. Under uClinux there is no fork(), and clone() must be supplied the CLONE_VM flag.

The behaviour is similar between the MMU and no-MMU cases, but not identical; and it's also much more restricted in the latter case:

(*) Anonymous mapping, MAP_PRIVATE

    In the MMU case: VM regions backed by arbitrary pages; copy-on-write across fork.

    In the no-MMU case: VM regions backed by arbitrary contiguous runs of pages.

(*) Anonymous mapping, MAP_SHARED

    These behave very much like private mappings, except that they're shared across fork() or clone() without CLONE_VM in the MMU case. Since the no-MMU case doesn't support these, behaviour is identical to MAP_PRIVATE there.

(*) File, MAP_PRIVATE, PROT_READ / PROT_EXEC, !PROT_WRITE

    In the MMU case: VM regions backed by pages read from file; changes to the underlying file are reflected in the mapping; copied across fork.

    In the no-MMU case:

    - If one exists, the kernel will re-use an existing mapping to the same segment of the same file if that has compatible permissions, even if this was created by another process.

    - If possible, the file mapping will be directly on the backing device if the backing device has the BDI_CAP_MAP_DIRECT capability and appropriate mapping protection capabilities. Ramfs, romfs, cramfs and mtd might all permit this.

    - If the backing device device can't or won't permit direct sharing, but does have the BDI_CAP_MAP_COPY capability, then a copy of the appropriate bit of the file will be read into a contiguous bit of memory and any extraneous space beyond the EOF will be cleared

    - Writes to the file do not affect the mapping; writes to the mapping are visible in other processes (no MMU protection), but should not

happen.

(*) File, MAP_PRIVATE, PROT_READ / PROT_EXEC, PROT_WRITE

In the MMU case: like the non-PROT_WRITE case, except that the pages in question get copied before the write actually happens. From that point on writes to the file underneath that page no longer get reflected into the mapping's backing pages. The page is then backed by swap instead.

In the no-MMU case: works much like the non-PROT_WRITE case, except that a copy is always taken and never shared.

(*) Regular file / blockdev, MAP_SHARED, PROT_READ / PROT_EXEC / PROT_WRITE

In the MMU case: VM regions backed by pages read from file; changes to pages written back to file; writes to file reflected into pages backing mapping; shared across fork.

In the no-MMU case: not supported.

(*) Memory backed regular file, MAP_SHARED, PROT_READ / PROT_EXEC / PROT_WRITE

In the MMU case: As for ordinary regular files.

In the no-MMU case: The filesystem providing the memory-backed file (such as ramfs or tmpfs) may choose to honour an open, truncate, mmap sequence by providing a contiguous sequence of pages to map. In that case, a shared-writable memory mapping will be possible. It will work as for the MMU case. If the filesystem does not provide any such support, then the mapping request will be denied.

(*) Memory backed blockdev, MAP_SHARED, PROT_READ / PROT_EXEC / PROT_WRITE

In the MMU case: As for ordinary regular files.

In the no-MMU case: As for memory backed regular files, but the blockdev must be able to provide a contiguous run of pages without truncate being called. The ramdisk driver could do this if it allocated all its memory as a contiguous array upfront.

(*) Memory backed chardev, MAP_SHARED, PROT_READ / PROT_EXEC / PROT_WRITE

In the MMU case: As for ordinary regular files.

In the no-MMU case: The character device driver may choose to honour the mmap() by providing direct access to the underlying device if it provides memory or quasi-memory that can be accessed directly. Examples of such are frame buffers and flash devices. If the driver does not provide any such support, then the mapping request will be denied.

==========
FURTHER NOTES ON NO-MMU MMAP
==========

(*) A request for a private mapping of a file may return a buffer that is not

page-aligned. This is because XIP may take place, and the data may not be paged aligned in the backing store.

(*) A request for an anonymous mapping will always be page aligned. If possible the size of the request should be a power of two otherwise some of the space may be wasted as the kernel must allocate a power-of-2 granule but will only discard the excess if appropriately configured as this has an effect on fragmentation.

(*) The memory allocated by a request for an anonymous mapping will normally be cleared by the kernel before being returned in accordance with the Linux man pages (ver 2.22 or later).

In the MMU case this can be achieved with reasonable performance as regions are backed by virtual pages, with the contents only being mapped to cleared physical pages when a write happens on that specific page (prior to which, the pages are effectively mapped to the global zero page from which reads can take place). This spreads out the time it takes to initialize the contents of a page - depending on the write-usage of the mapping.

In the no-MMU case, however, anonymous mappings are backed by physical pages, and the entire map is cleared at allocation time. This can cause significant delays during a userspace malloc() as the C library does an anonymous mapping and the kernel then does a memset for the entire map.

However, for memory that isn't required to be precleared - such as that returned by malloc() - mmap() can take a MAP_UNINITIALIZED flag to indicate to the kernel that it shouldn't bother clearing the memory before returning it. Note that CONFIG_MMAP_ALLOW_UNINITIALIZED must be enabled to permit this, otherwise the flag will be ignored.

uClibc uses this to speed up malloc(), and the ELF-FDPIC binfmt uses this to allocate the brk and stack region.

(*) A list of all the private copy and anonymous mappings on the system is visible through /proc/maps in no-MMU mode.

(*) A list of all the mappings in use by a process is visible through /proc/<pid>/maps in no-MMU mode.

(*) Supplying MAP_FIXED or a requesting a particular mapping address will result in an error.

(*) Files mapped privately usually have to have a read method provided by the driver or filesystem so that the contents can be read into the memory allocated if mmap() chooses not to map the backing device directly. An error will result if they don't. This is most likely to be encountered with character device files, pipes, fifos and sockets.

==========
INTERPROCESS SHARED MEMORY
==========

Both SYSV IPC SHM shared memory and POSIX shared memory is supported in NOMMU

mode. The former through the usual mechanism, the latter through files created on ramfs or tmpfs mounts.

=======
FUTEXES
=======

Futexes are supported in NOMMU mode if the arch supports them. An error will be given if an address passed to the futex system call lies outside the mappings made by a process or if the mapping in which the address lies does not support futexes (such as an I/O chardev mapping).

=============
NO-MMU MREMAP
=============

The mremap() function is partially supported. It may change the size of a mapping, and may move it[*] if MREMAP_MAYMOVE is specified and if the new size of the mapping exceeds the size of the slab object currently occupied by the memory to which the mapping refers, or if a smaller slab object could be used.

MREMAP_FIXED is not supported, though it is ignored if there's no change of address and the object does not need to be moved.

Shared mappings may not be moved. Shareable mappings may not be moved either, even if they are not currently shared.

The mremap() function must be given an exact match for base address and size of a previously mapped object. It may not be used to create holes in existing mappings, move parts of existing mappings or resize parts of mappings. It must act on a complete mapping.

[*] Not currently supported.

==
PROVIDING SHAREABLE CHARACTER DEVICE SUPPORT
==

To provide shareable character device support, a driver must provide a file->f_op->get_unmapped_area() operation. The mmap() routines will call this to get a proposed address for the mapping. This may return an error if it doesn't wish to honour the mapping because it's too long, at a weird offset, under some unsupported combination of flags or whatever.

The driver should also provide backing device information with capabilities set to indicate the permitted types of mapping on such devices. The default is assumed to be readable and writable, not executable, and only shareable directly (can't be copied).

The file->f_op->mmap() operation will be called to actually inaugurate the mapping. It can be rejected at that point. Returning the ENOSYS error will cause the mapping to be copied instead if BDI_CAP_MAP_COPY is specified.

The vm_ops->close() routine will be invoked when the last mapping on a chardev is removed. An existing mapping will be shared, partially or not, if possible without notifying the driver.

It is permitted also for the file->f_op->get_unmapped_area() operation to return -ENOSYS. This will be taken to mean that this operation just doesn't want to handle it, despite the fact it's got an operation. For instance, it might try directing the call to a secondary driver which turns out not to implement it. Such is the case for the framebuffer driver which attempts to direct the call to the device-specific driver. Under such circumstances, the mapping request will be rejected if BDI_CAP_MAP_COPY is not specified, and a copy mapped otherwise.

IMPORTANT NOTE:

> Some types of device may present a different appearance to anyone looking at them in certain modes. Flash chips can be like this; for instance if they're in programming or erase mode, you might see the status reflected in the mapping, instead of the data.
>
> In such a case, care must be taken lest userspace see a shared or a private mapping showing such information when the driver is busy controlling the device. Remember especially: private executable mappings may still be mapped directly off the device under some circumstances!

PROVIDING SHAREABLE MEMORY-BACKED FILE SUPPORT

Provision of shared mappings on memory backed files is similar to the provision of support for shared mapped character devices. The main difference is that the filesystem providing the service will probably allocate a contiguous collection of pages and permit mappings to be made on that.

It is recommended that a truncate operation applied to such a file that increases the file size, if that file is empty, be taken as a request to gather enough pages to honour a mapping. This is required to support POSIX shared memory.

Memory backed devices are indicated by the mapping's backing device info having the memory_backed flag set.

PROVIDING SHAREABLE BLOCK DEVICE SUPPORT

Provision of shared mappings on block device files is exactly the same as for character devices. If there isn't a real device underneath, then the driver should allocate sufficient contiguous memory to honour any supported mapping.

ADJUSTING PAGE TRIMMING BEHAVIOUR

==========

NOMMU mmap automatically rounds up to the nearest power-of-2 number of pages when performing an allocation. This can have adverse effects on memory fragmentation, and as such, is left configurable. The default behaviour is to aggressively trim allocations and discard any excess pages back in to the page allocator. In order to retain finer-grained control over fragmentation, this behaviour can either be disabled completely, or bumped up to a higher page watermark where trimming begins.

Page trimming behaviour is configurable via the sysctl `vm.nr_trim_pages'.