User virtual maps (mmap)

2016-12-20 (http://jake.dothome.co.kr/user-virtual-maps-mmap2/) Moon Young-il (http://jake.dothome.co.kr/author/admin/) Linux Kernel (http://jake.dothome.co.kr/category/linux/)

<kernel v5.0>

Memory Mapped Files & anon Mapping

Request to map or unmap a file (including devices) to a VM (virtual memory) in the current user address space. Use the following APIs:

- mmap()
  - Request a file or anon mapping to a VM (virtual memory) in the current user address space.
- munmap()
  - Request to unmap a file or anon to a VM (virtual memory) in the current user address space.
- mmap2()
  - It's the same as mmap(), except that the last argument uses a page-by-page offset.
- mmap_pgoff()
  - It's the same as mmap(), except that the last argument uses a page-by-page offset.

mmap() – for user application

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

Map files and devices to memory. You can also map anon memory by giving it an anon mapping property.

mmap2() – for user application

```
#include <sys/mman.h>

void *mmap2(void *addr, size_t length, int prot, int flags, int fd, off_t pgoffset);
```

mmap_pgoff() – for user application

```
#include <sys/mman.h>

void *mmap_pgoff(void *addr, size_t length, int prot, int flags, int fd, unsigned long pgoffset);
```

factor

- addr
    - This is the starting virtual address value that you want to map, and the kernel uses it as a hint to try to find the appropriate address. If null is entered, the kernel looks for empty space.
- length
    - Length to map (bytes)
- prot
    - Memory Protection Properties
        - PROT_EXEC: The page is executable.
        - PROT_READ: The page is readable.
        - PROT_WRITE: Pages are writable.
        - PROT_NONE: The page is inaccessible.
- flags
    - flag
- fd
    - File Descriptors
- offset
    - byte offset
- pgoff
    - Page offset
    - If you are mapping a file, use the number of pages to skip.

Request Flags

- MAP_FIXED
    - Maps to the specified starting virtual address. If the specified address is not available, mmap() fails.
    - The virtual start address addr must be a multiple of the page size.
    - If the request area overlaps with an existing mapping, the existing mapping of the overlapping area is removed.
- MAP_ANONYMOUS
    - An area that is not associated with any file and is initialized to zero.
    - fd is ignored, but in some cases -1 is required.
    - offset to 0.
    - It can also be used in conjunction with MAP_SHARED. (kernel v2.4)
- MAP_FILE
    - File Mapping
- MAP_SHARED
    - Shared mapping for different processes and areas.
- MAP_PRIVATE

  - Create a private Copy On Write (COW) mapping. This area is not shared with other processes.
- MAP_DENYWRITE
  - Write-protected mapping regions
- MAP_EXECUTABLE
  - Actionable mapping areas
- MAP_GROWSDOWN
  - Used for stacks that grow downward.
- MAP_HUGETLB (kernel v2.6.32)
  - huge pages.
- MAP_HUGE_2MB, MAP_HUGE_1GB (kernel v3.8)
  - MAP_HUGETLB and you can choose huge page.
  - You can see the huge page types available in the "/sys/kernel/mm/hugepages" directory.
- MAP_LOCKED
  - As with mlock(), the request area is mapped with pre-allocated physical memory.
  - Unlike mlock(), if it fails when pre-allocating and mapping physical memory, it does not immediately throw an -ENOMEM error.
- MAP_NONBLOCK
  - It is only meaningful when used in conjunction with MAP_POPULATE.
- MAP_NORESERVE
  - This area does not prepare the swap space.
- MAP_POPULATE
  - Prefault the page table for the mapping.
  - It is only used in private mappings. (kernel v2.6.23)
- MAP_STACK (kernel v2.6.27)
  - Allocate a process or thread stack.
- MAP_UNINITIALIZED (kernel v2.6.33)
  - anonymous pages are not cleared.

Memory Mapped Mapping Stories

private anon example)

- mmap((void *) 0x200000000000ULL, 4096 * 30, PROT_READ | PROT_WRITE, MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
- The first argument addr is usually null. Addresses have been specified to make it easier to check the results.
- The anon vma zone is created, but the physical memory is not mapped, as shown by the Rss value of 0. A fault occurs when the memory is accessed in the future, and the physical memory is allocated by the fault handler and then mapped. (lazy allocation)

```
200000000000-20000001e000 rw-p 00000000 00:00 0
Size:                120 kB
KernelPageSize:        4 kB
MMUPageSize:           4 kB
Rss:                   0 kB
Pss:                   0 kB
Shared_Clean:          0 kB
Shared_Dirty:          0 kB
Private_Clean:         0 kB
Private_Dirty:         0 kB
Referenced:            0 kB
Anonymous:             0 kB
LazyFree:              0 kB
AnonHugePages:         0 kB
ShmemPmdMapped:        0 kB
Shared_Hugetlb:        0 kB
Private_Hugetlb:       0 kB
Swap:                  0 kB
SwapPss:               0 kB
Locked:                0 kB
THPeligible:    1
VmFlags: rd wr mr mw me ac
```

private file 예)

- mmap(NULL, 15275, PROT_READ, MAP_PRIVATE, fd, 0);
- fd is the file descriptor that opened abc.txt.
- The file vma area is created, but the physical memory is not mapped, as shown by the Rss value of 0. A fault occurs when the memory is accessed later, the physical memory is allocated by the fault handler, and the contents of the file are read and mapped thereafter. (lazy allocation)

```
ffffb2d14000-ffffb2d15000 r--p 00000000 fe:00 416016                     /root/work
space/test/mmap/abc.txt
Size:                  4 kB
KernelPageSize:        4 kB
MMUPageSize:           4 kB
Rss:                   0 kB
Pss:                   0 kB
Shared_Clean:          0 kB
Shared_Dirty:          0 kB
Private_Clean:         0 kB
Private_Dirty:         0 kB
Referenced:            0 kB
Anonymous:             0 kB
LazyFree:              0 kB
AnonHugePages:         0 kB
ShmemPmdMapped:        0 kB
Shared_Hugetlb:        0 kB
Private_Hugetlb:       0 kB
Swap:                  0 kB
SwapPss:               0 kB
Locked:                0 kB
THPeligible:    0
VmFlags: rd mr mw me
```

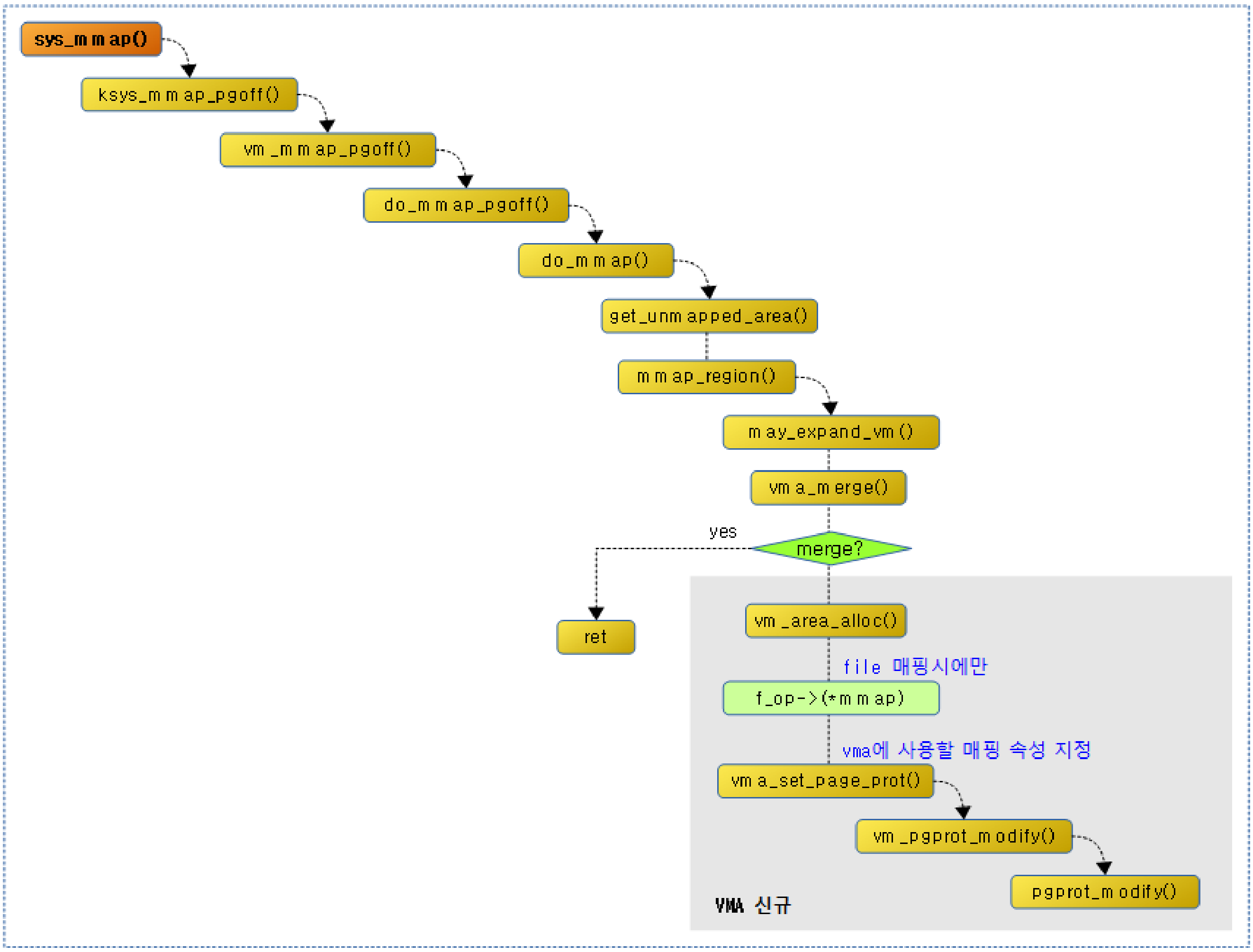
locked private file 예)

- mmap(NULL, 15275, PROT_READ, MAP_PRIVATE | MAP_LOCKED, fd, 0);
- MAP_LOCKED was used to pre-map all of the file's contents to physical memory. You can see that the Size entry and the RSS entry are the same.

```
ffffba677000-ffffba67b000 r--p 00000000 fe:00 416016                     /root/work
space/test/mmap/abc.txt
Size:                 16 kB
KernelPageSize:        4 kB
MMUPageSize:           4 kB
Rss:                  16 kB
Pss:                  16 kB
Shared_Clean:          0 kB
Shared_Dirty:          0 kB
Private_Clean:        16 kB
Private_Dirty:         0 kB
Referenced:           16 kB
Anonymous:             0 kB
LazyFree:              0 kB
AnonHugePages:         0 kB
ShmemPmdMapped:        0 kB
Shared_Hugetlb:        0 kB
Private_Hugetlb:       0 kB
Swap:                  0 kB
SwapPss:               0 kB
Locked:               16 kB
THPeligible:    0
VmFlags: rd mr mw me lo
```

Memory Mapped Mapping for ARM32

The following figure shows the relationship between the mmap syscall and the function call in the kernel.



(http://jake.dothome.co.kr/wp-content/uploads/2016/12/sys_mmap-1.png)

sys_mmap2() – ARM32

arch/arm/kernel/entry-common. S

```
/*
 * Note: off_4k (r5) is always units of 4K.  If we can't do the requested
 * offset, we return EINVAL.
 */
```

```
sys_mmap2:
                streq   r5, [sp, #4]
                beq     sys_mmap_pgoff
ENDPROC(sys_mmap2)
```

Map files and devices to memory. You can also map anon memory by giving it an anon mapping property.

sys_mmap_pgoff() – Generic (ARM32, …)

mm/mmap.c

```
SYSCALL_DEFINE6(mmap_pgoff, unsigned long, addr, unsigned long, len,
                unsigned long, prot, unsigned long, flags,
                unsigned long, fd, unsigned long, pgoff)
{
        return ksys_mmap_pgoff(addr, len, prot, flags, fd, pgoff);
}
```

Memory Mapped Mapping for ARM64

sys_mmap() – ARM64

arch/arm64/kernel/sys.c

```
SYSCALL_DEFINE6(mmap, unsigned long, addr, unsigned long, len,
                unsigned long, prot, unsigned long, flags,
                unsigned long, fd, off_t, off)
{
        if (offset_in_page(off) != 0)
                return -EINVAL;

        return ksys_mmap_pgoff(addr, len, prot, flags, fd, off >> PAGE_SHIFT);
}
```

sys_mmap_pgoff() – ARM64

arch/arm64/kernel/sys.c

```
SYSCALL_DEFINE6(mmap_pgoff, unsigned long, addr, unsigned long, len,
                unsigned long, prot, unsigned long, flags,
                unsigned long, fd, unsigned long, pgoff)
{
        return ksys_mmap_pgoff(addr, len, prot, flags, fd, pgoff);
}
```

Common Part – Memory Mapped Mapping

ksys_mmap_pgoff()

mm/mmap.c

```
unsigned long ksys_mmap_pgoff(unsigned long addr, unsigned long len,
                              unsigned long prot, unsigned long flags,
                              unsigned long fd, unsigned long pgoff)
{
        struct file *file = NULL;
        unsigned long retval;

08         if (!(flags & MAP_ANONYMOUS)) {
09                 audit_mmap_fd(fd, flags);
10                 file = fget(fd);
11                 if (!file)
12                         return -EBADF;
13                 if (is_file_hugepages(file))
14                         len = ALIGN(len, huge_page_size(hstate_file(file)));
15                 retval = -EINVAL;
16                 if (unlikely(flags & MAP_HUGETLB && !is_file_hugepages(file)))
17                         goto out_fput;
18         } else if (flags & MAP_HUGETLB) {
19                 struct user_struct *user = NULL;
20                 struct hstate *hs;
21 
22                 hs = hstate_sizelog((flags >> MAP_HUGE_SHIFT) & MAP_HUGE_MASK);
23                 if (!hs)
24                         return -EINVAL;
25 
26                 len = ALIGN(len, huge_page_size(hs));
27                 /*
28                  * VM_NORESERVE is used because the reservations will be
29                  * taken when vm_ops->mmap() is called
30                  * A dummy user value is used because we are not locking
31                  * memory so no accounting is necessary
32                  */
33                 file = hugetlb_file_setup(HUGETLB_ANON_FILE, len,
34                                 VM_NORESERVE,
35                                 &user, HUGETLB_ANONHUGE_INODE,
36                                 (flags >> MAP_HUGE_SHIFT) & MAP_HUGE_MASK);
37                 if (IS_ERR(file))
38                         return PTR_ERR(file);
39         }
40 
41         flags &= ~(MAP_EXECUTABLE | MAP_DENYWRITE);
42 
43         retval = vm_mmap_pgoff(file, addr, len, prot, flags, pgoff);
44 out_fput:
45         if (file)
46                 fput(file);
47         return retval;
48 }
```

In the file descriptor, map the requested virtual address to the requested virtual address for the length of the request from the pgoff page to the prot attribute.

- In line 8~17 of the code, set the audit if it is a file mapping, and if it is a file that uses a huge page, arrange the length to match the huge page.
    - If the audit_context of the current task is set, set the request fd and flags in the audit_context and assign the AUDIT_MMAP as the type.
    - If it fails to retrieve the file from the file descriptor, it returns an error with the out label.
    - If the file is shared using hugetlbfs or huge page mappings, the length is sorted by huge pages.
    - If a MAP_HUGETLB flag request is made, but the file is not of type huge page request, it returns a -EINVAL error.
- If there is a request for MAP_HUGETLB in line 18~39 of the code, prepare a file in the form of a HUGETLB_ANON_FILE by sorting the length in huge pages. If no huge page information is found in the flag, it returns a -EINVAL error.

  - The size of the flag was converted to log units and stored in bit26~bit31.
    - x86 예) 21 -> 2M huge page, 30 -> 1G huge page
- In line 41~43 of the code, remove the MAP_EXECUTABLE and MAP_DENYWRITE from the flag and do the mapping.

is_file_hugepages()

include/linux/hugetlb.h

```
static inline int is_file_hugepages(struct file *file)
{
        if (file->f_op == &hugetlbfs_file_operations)
                return 1;
        if (is_file_shm_hugepages(file))
                return 1;

        return 0;
}
```

Returns 1 if file is a file used by hugetlbfs or a shared memory file that uses huge pages. Otherwise, it returns 0.

Audit-related functions

audit_mmap_fd()

include/linux/audit.h

```
static inline void audit_mmap_fd(int fd, int flags)
{
        if (unlikely(!audit_dummy_context()))
                __audit_mmap_fd(fd, flags);
}
```

If there is a small probability that the audit_context of the current task is set, set the request fd and flags in the audit_context and assign the AUDIT_MMAP as the type.

audit_dummy_context()

include/linux/audit.h

```
static inline int audit_dummy_context(void)
{
        void *p = current->audit_context;
        return !p || *(int *)p;
}
```

Returns 0 if the current task's audit_context is set, or non-0 if it is not set.

__audit_mmap_fd()

kernel/auditsc.c

```
void __audit_mmap_fd(int fd, int flags)
```

```
{
        struct audit_context *context = current->audit_context;
        context->mmap.fd = fd;
        context->mmap.flags = flags;
        context->type = AUDIT_MMAP;
}
```

Set the request fd and flags for the audit_context of the current task, and assign AUDIT_MMAP as the type.

vm_mmap_pgoff()

mm/util.c

```
01 unsigned long vm_mmap_pgoff(struct file *file, unsigned long addr,
02         unsigned long len, unsigned long prot,
03         unsigned long flag, unsigned long pgoff)
04 {
05         unsigned long ret;
06         struct mm_struct *mm = current->mm;
07         unsigned long populate;
08         LIST_HEAD(uf);
09 
10         ret = security_mmap_file(file, prot, flag);
11         if (!ret) {
12                 if (down_write_killable(&mm->mmap_sem))
13                         return -EINTR;
14                 ret = do_mmap_pgoff(file, addr, len, prot, flag, pgoff,
15                                     &populate, &uf);
16                 up_write(&mm->mmap_sem);
17                 userfaultfd_unmap_complete(mm, &uf);
18                 if (populate)
19                         mm_populate(ret, populate);
20         }
21         return ret;
22 }
```

Map the @file from the @pgoff page to the @prot attribute of the virtual address @addr for the length of the request.

- In line 10 of the code, we find out whether the file mapping is allowed via LSM and LIM. 0=Success
  - This is done by calling the hook api of LSM (Linux Security Module) and Linux Integrity Module (LIM).
- In lines 10~15 of the code, use mmap_sem write semaphore lock to request VM mapping.
- In line 16 of the code, userfaultfd instructs the unmap to complete.
  - userfaultfd
    - Linux using on-demand paging does not allocate the actual memory when requesting a memory allocation, but instead allocates physical memory for the kernel after a fault occurs when the user application accesses the allocated area. Userfaultfd is designed to handle this for the user application.
- In line 17~18 of the code, enable the mapping area (map physical RAM).

do_mmap_pgoff()

mm/mmap.c

```c
static inline unsigned long
do_mmap_pgoff(struct file *file, unsigned long addr,
        unsigned long len, unsigned long prot, unsigned long flags,
        unsigned long pgoff, unsigned long *populate,
        struct list_head *uf)
{
        return do_mmap(file, addr, len, prot, flags, 0, pgoff, populate, uf);
}
```

do_mmap()

mm/mmap.c -1/3-

```c
/*
 * The caller must hold down_write(&current->mm->mmap_sem).
 */
```

```c
unsigned long do_mmap(struct file *file, unsigned long addr,
                        unsigned long len, unsigned long prot,
                        unsigned long flags, vm_flags_t vm_flags,
                        unsigned long pgoff, unsigned long *populate,
                        struct list_head *uf)
{
        struct mm_struct *mm = current->mm;
        int pkey = 0;

        *populate = 0;

        if (!len)
                return -EINVAL;

        /*
         * Does the application expect PROT_READ to imply PROT_EXEC?
         *
         * (the exception is when the underlying filesystem is noexec
         *  mounted, in which case we dont add PROT_EXEC.)
         */
        if ((prot & PROT_READ) && (current->personality & READ_IMPLIES_EXEC))
                if (!(file && path_noexec(&file->f_path)))
                        prot |= PROT_EXEC;

        /* force arch specific MAP_FIXED handling in get_unmapped_area */
        if (flags & MAP_FIXED_NOREPLACE)
                flags |= MAP_FIXED;

        if (!(flags & MAP_FIXED))
                addr = round_hint_to_min(addr);

        /* Careful about overflows.. */
        len = PAGE_ALIGN(len);
        if (!len)
                return -ENOMEM;

        /* offset overflow? */
        if ((pgoff + (len >> PAGE_SHIFT)) < pgoff)
                return -EOVERFLOW;

        /* Too many mappings? */
```

```
42         if (mm->map_count > sysctl_max_map_count)
43                 return -ENOMEM;
44 
45         /* Obtain the address to map to. we verify (or select) it and ensure
46          * that it represents a valid section of the address space.
47          */
48         addr = get_unmapped_area(file, addr, len, pgoff, flags);
49         if (offset_in_page(addr))
50                 return addr;
51 
52         if (flags & MAP_FIXED_NOREPLACE) {
53                 struct vm_area_struct *vma = find_vma(mm, addr);
54 
55                 if (vma && vma->vm_start < addr + len)
56                         return -EEXIST;
57         }
58 
59         if (prot == PROT_EXEC) {
60                 pkey = execute_only_pkey(mm);
61                 if (pkey < 0)
62                         pkey = 0;
63         }
64 
65         /* Do simple checking here so the lower-level routines won't have
66          * to. we assume access permissions have been handled by the open
67          * of the memory object, so we don't do any here.
68          */
69         vm_flags |= calc_vm_prot_bits(prot, pkey) | calc_vm_flag_bits(flags) |
70                         mm->def_flags | VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC;
71 
72         if (flags & MAP_LOCKED)
73                 if (!can_do_mlock())
74                         return -EPERM;
75 
76         if (mlock_future_check(mm, vm_flags, len))
77                 return -EAGAIN;
```

Map file to the virtual address addr for the length of the request from the pgoff page to the prot attribute. If physical memory is mapped, assign the length to the output argument populate.

- If the length of code lines 12~13 is zero, it returns a -EINVAL error.
- If the read attribute is requested in line 21~23 of the code, add the exec attribute if the READ_IMPLIES_EXEC is set to the personality of the current task. However, this is not the case with sysfs and proc mounts that have a MNT_NOEXEC mount flag as shown in the file above.
- In code lines 26~27
- Unless a fixed mapping is requested in line 29~30 of the code, use the page-aligned address, but change it to the mmap_min_addr address if it is smaller than mmap_min_addr.
- Sort the length in lines 33~35 by page. However, if it is 0, it returns a -ENOMEM error.
- In line 38~39 of the code, if the pgoff page + len page exceeds the system address range, it returns a -EOVERFLOW error.
- If lines 42~43 of the current task are mapped to the memory descriptors and exceed the maximum allowable number, it returns a -ENOMEM error.
    - Default mapping maximum: 65530 (set in "/proc/sys/vm/max_map_count")

- Find the unmapped area in lines 48~50 of the code to get the starting virtual address. If it is an error, it returns an error code.
- If there is a MAP_FIXED_NOREPLACE flag request in code lines 52~57, it can only be mapped if there is no zone in the VMA for that address. Therefore, if it already exists, it returns an error with -EEXIST.
- In line 59~63 of the code, check the Protection Key if the Run attribute exists.
  - It is only supported on x86 and powerpc architectures and is not yet used in ARM and ARM64.
- In line 69~70 of the code, add mayread, maywrite, and mayexec to the prot flag converted to vm flags, flags converted to vm flags, and the default flag of the memory descriptor in the vm_flags.
- If there is a MAP_LOCKED flag request in code lines 72~74, it returns a -EPERM error if it cannot be performed due to the mlock maximum limit, etc.
- If there is a VM_LOCKED request from lines 76~77 of the code and the request length is added to the existing mlock page, the mlock page maximum limit is reached, it returns a -EAGAIN error.

mm/mmap.c -2/3-

```
01 if (file) {
02         struct inode *inode = file_inode(file);
03         unsigned long flags_mask;
04 
05         if (!file_mmap_ok(file, inode, pgoff, len))
06                 return -EOVERFLOW;
07 
08         flags_mask = LEGACY_MAP_MASK | file->f_op->mmap_supported_flags;
09 
10         switch (flags & MAP_TYPE) {
11         case MAP_SHARED:
12                 /*
13                  * Force use of MAP_SHARED_VALIDATE with non-legacy
14                  * flags. E.g. MAP_SYNC is dangerous to use with
15                  * MAP_SHARED as you don't know which consistency model
16                  * you will get. We silently ignore unsupported flags
17                  * with MAP_SHARED to preserve backward compatibility.
18                  */
19                 flags &= LEGACY_MAP_MASK;
20                 /* fall through */
21         case MAP_SHARED_VALIDATE:
22                 if (flags & ~flags_mask)
23                         return -EOPNOTSUPP;
24                 if ((prot&PROT_WRITE) && !(file->f_mode&FMODE_WRITE))
25                         return -EACCES;
26 
27                 /*
28                  * Make sure we don't allow writing to an append-only
29                  * file..
30                  */
31                 if (IS_APPEND(inode) && (file->f_mode & FMODE_WRITE))
32                         return -EACCES;
33 
34                 /*
35                  * Make sure there are no mandatory locks on the file.
36                  */
37                 if (locks_verify_locked(file))
38                         return -EAGAIN;
39 
40                 vm_flags |= VM_SHARED | VM_MAYSHARE;
41                 if (!(file->f_mode & FMODE_WRITE))
42                         vm_flags &= ~(VM_MAYWRITE | VM_SHARED);
43
44                 /* fall through */
45         case MAP_PRIVATE:
46                 if (!(file->f_mode & FMODE_READ))
47                         return -EACCES;
48                 if (path_noexec(&file->f_path)) {
49                         if (vm_flags & VM_EXEC)
50                                 return -EPERM;
51                         vm_flags &= ~VM_MAYEXEC;
52                 }
53
54                 if (!file->f_op->mmap)
55                         return -ENODEV;
56                 if (vm_flags & (VM_GROWSDOWN|VM_GROWSUP))
57                         return -EINVAL;
58                 break;
59
60         default:
61                 return -EINVAL;
62         }
```

- In code lines 1~8, get the inode value if file is a mapping.
- In line 10~42 of the code, if it is a shared file mapping (MAP_SHARED) flag request, it is handled as follows:
    - If a write request is made to a file that does not have a write attribute, it returns a -EACCESS error.
    - If the file is append-only, it returns a -EACCESS error if a write request is made.
    - If the file is locked mandatory, it returns a -EAGAIN error.
    - Add the share and mayshare attributes to the vm_flags.
    - If the file doesn't have a write attribute, remove maywrite and shared. Continue with your private request below.
- In line 45~58 of the code, if it is a private file mapping (MAP_PRIVATE) flag request, it is handled as follows:
    - If the file does not have a read attribute, it returns a -EACCESS error.
    - If the mount flag where the file is mounted, remove mayexec. However, if there is a VM_EXEC request, it returns a -EPERM error.
        - Where "/proc and /sys" are mounted, there cannot be executable files.
    - Files with no mapping return the -ENODEV error.
    - growsdown and growsup returns a -EINVAL error if requested.

mm/mmap.c -3/3-

```
01         } else {
02                 switch (flags & MAP_TYPE) {
03                 case MAP_SHARED:
04                         if (vm_flags & (VM_GROWSDOWN|VM_GROWSUP))
05                                 return -EINVAL;
06                         /*
07                          * Ignore pgoff.
08                          */
09                         pgoff = 0;
10                         vm_flags |= VM_SHARED | VM_MAYSHARE;
11                         break;
12                 case MAP_PRIVATE:
13                         /*
14                          * Set pgoff according to addr for anon_vma.
15                          */
16                         pgoff = addr >> PAGE_SHIFT;
17                         break;
18                 default:
19                         return -EINVAL;
20                 }
21         }
22 
23         /*
24          * Set 'VM_NORESERVE' if we should not account for the
25          * memory use of this mapping.
26          */
27         if (flags & MAP_NORESERVE) {
28                 /* We honor MAP_NORESERVE if allowed to overcommit */
29                 if (sysctl_overcommit_memory != OVERCOMMIT_NEVER)
30                         vm_flags |= VM_NORESERVE;
31 
32                 /* hugetlb applies strict overcommit unless MAP_NORESERV
   E */
33                 if (file && is_file_hugepages(file))
34                         vm_flags |= VM_NORESERVE;
35         }
36 
37         addr = mmap_region(file, addr, len, vm_flags, pgoff, uf);
38         if (!IS_ERR_VALUE(addr) &&
39             ((vm_flags & VM_LOCKED) ||
40              (flags & (MAP_POPULATE | MAP_NONBLOCK)) == MAP_POPULATE))
41                 *populate = len;
42         return addr;
43 }
```

- In line 1~21 of the code, if it is an anon mapping, make the following two requests:
  - If it's a shared anon mapping, add pgoff=0, shared, and maysahre flags. However, growsdown and growsup return an error if -EINVAL is requested.
  - In the case of a private ANON mapping, assign the virtual address page number to pgoff.
- In lines 27~35 of the code, add a vm_noreserve flag if it is a no reserve request, if the over commit mode is not OVERCOMMIT_NEVER, or if it is also a huge page file.
- In line 37 of the code, file the pgoff page from the virtual address addr to configure and register the VMA with the vm_flags attribute as long as the len.
- If physical memory is mapped in code lines 38~41, assign the length to the output argument populate.

round_hint_to_min()

mm/mmap.c

```
/*
 * If a hint addr is less than mmap_min_addr change hint to be as
 * low as possible but still greater than mmap_min_addr
 */
```

```
static inline unsigned long round_hint_to_min(unsigned long hint)
{
        hint &= PAGE_MASK;
        if (((void *)hint != NULL) &&
            (hint < mmap_min_addr))
                return PAGE_ALIGN(mmap_min_addr);
        return hint;
}
```

Cut and return a virtual address hint page by page. However, if the address is lower than mmap_min_addr, it returns the mmap_min_addr address sorted by page.

calc_vm_prot_bits()

include/linux/mman.h

```
/*
 * Combine the mmap "prot" argument into "vm_flags" used internally.
 */
```

```
static inline unsigned long
calc_vm_prot_bits(unsigned long prot)
{
        return _calc_vm_trans(prot, PROT_READ,  VM_READ ) |
               _calc_vm_trans(prot, PROT_WRITE, VM_WRITE) |
               _calc_vm_trans(prot, PROT_EXEC,  VM_EXEC) |
               arch_calc_vm_prot_bits(prot);
}
```

If the prot value contains PROT_READ, PROT_WRITE, and PROT_EXEC bits, it is converted to the VM_READ, VM_WRITE, and VM_EXEC flag attributes, respectively, and returned.

- You can add properties to suit your specific architecture. (no additional arm)

_calc_vm_trans()

include/linux/mman.h

```
/*
 * Optimisation macro.  It is equivalent to:
 *      (x & bit1) ? bit2 : 0
 * but this version is faster.
 * ("bit1" and "bit2" must be single bits)
 */
```

```
#define _calc_vm_trans(x, bit1, bit2) \
  ((!(bit1) || !(bit2)) ? 0 : \
  ((bit1) <= (bit2) ? ((x) & (bit1)) * ((bit2) / (bit1)) \
   : ((x) & (bit1)) / ((bit1) / (bit2))))
```

If the x flag has a bi1 attribute, convert it to a bit attribute and return it. If it doesn't exist, it returns 0.

calc_vm_flag_bits()

include/linux/mman.h

```
/*
 * Combine the mmap "flags" argument into "vm_flags" used internally.
 */
```

```
static inline unsigned long
calc_vm_flag_bits(unsigned long flags)
{
        return _calc_vm_trans(flags, MAP_GROWSDOWN,  VM_GROWSDOWN ) |
               _calc_vm_trans(flags, MAP_DENYWRITE,  VM_DENYWRITE ) |
               _calc_vm_trans(flags, MAP_LOCKED,     VM_LOCKED    ) |
               _calc_vm_trans(flags, MAP_SYNC,       VM_SYNC      );
}
```

If the flags value contains MAP_GROWSDOWN, MAP_DENYWRITE, MAP_LOCKED, and MAP_SYNC bits, it is converted to VM_GROWSDOWN, VM_DENYWRITE, VM_LOCKED, and VM_SYNC flag attributes, respectively.

VMA Zone Configuration (Extended/Merged/New)

mmap_region()

mm/mmap.c -1/3-

```
unsigned long mmap_region(struct file *file, unsigned long addr,
                unsigned long len, vm_flags_t vm_flags, unsigned long pgoff,
                struct list_head *uf)
{
        struct mm_struct *mm = current->mm;
        struct vm_area_struct *vma, *prev;
        int error;
        struct rb_node **rb_link, *rb_parent;
        unsigned long charged = 0;

        /* Check against address space limit. */
        if (!may_expand_vm(mm, vm_flags, len >> PAGE_SHIFT)) {
                unsigned long nr_pages;

                /*
                 * MAP_FIXED may remove pages of mappings that intersects with
                 * requested mapping. Account for the pages it would unmap.
                 */
                nr_pages = count_vma_pages_range(mm, addr, addr + len);

                if (!may_expand_vm(mm, vm_flags,
                                        (len >> PAGE_SHIFT) - nr_pages))
                        return -ENOMEM;
        }

        /* Clear old maps */
        while (find_vma_links(mm, addr, addr + len, &prev, &rb_link,
                              &rb_parent)) {
                if (do_munmap(mm, addr, len, uf))
                        return -ENOMEM;
        }

        /*
         * Private writable mapping: check memory availability
         */
        if (accountable_mapping(file, vm_flags)) {
                charged = len >> PAGE_SHIFT;
                if (security_vm_enough_memory_mm(mm, charged))
                        return -ENOMEM;
                vm_flags |= VM_ACCOUNT;
        }

        /*
         * Can we just expand an old mapping?
         */
        vma = vma_merge(mm, prev, addr, addr + len, vm_flags,
                        NULL, file, pgoff, NULL, NULL_VM_UFFD_CTX);
        if (vma)
                goto out;
```

```
50
51         /*
52          * Determine the object being mapped and call the appropriate
53          * specific mapper. the address has already been validated, but
54          * not unmapped, but the maps are removed from the list.
55          */
56         vma = vm_area_alloc(mm);
57         if (!vma) {
58                 error = -ENOMEM;
59                 goto unacct_error;
60         }
61
62         vma->vm_start = addr;
63         vma->vm_end = addr + len;
64         vma->vm_flags = vm_flags;
65         vma->vm_page_prot = vm_get_page_prot(vm_flags);
66         vma->vm_pgoff = pgoff;
```

file from the pgoff page to the virtual address addr for the length of the request, configure and register the VMA with the prot attribute. Existing VMA areas can be expanded or merged, and if they don't exist, new ones can be created.

- In line 12~24 of the code, if the space cannot be expanded by the amount of len pages, it will be handled as follows, depending on whether it is fixed mapping or not.
  - If it is not a fix mapping, it returns a -ENOMEM error.
  - In the case of fix mapping, we plan to subtract and map the overlapping areas. If the page does not overlap with the existing area, it returns a -ENOMEM error.
- In lines 27~31 of the code, unmap the existing areas that overlap with the request area.
- In line 36~41 of the code, if the mapping requires VM memory metering, commit it as a VM memory allocation as much as the charged page via the LSM and add the VM_ACCOUNT flag if allowed.
  - Checking accountable mappings (private writable mappings)
    - It's not a huge file mapping, there's no noreserve and shared, and there's a write request.
  - Depending on whether the LSM module uses the SELinux module, the selinux_vm_enough_memory() function is first called to obtain the additional area with admin privileges, and then the __vm_enough_memory() function is called to return the allocation of the amount of commit through the VM allowance limit according to the commit option.
  - If you are only using the default Posix Capability module in the LSM module, you can call the cap_vm_enough_memory() function first to get as much additional area as you have admin privileges, and then call the __vm_enough_memory() function to return the allocation of the amount of commit through the VM allowance limit according to the commit option.
- Merge with the existing area on code lines 46~49, and move to the out label if successful.
- In code lines 56~66, allocate and configure a new VMA (vm_area_struct structure).

mm/mmap.c -2/3-

```
.       if (file) {
                if (vm_flags & VM_DENYWRITE) {
                        error = deny_write_access(file);
04                      if (error)
05                              goto free_vma;
06              }
07              if (vm_flags & VM_SHARED) {
08                      error = mapping_map_writable(file->f_mapping);
09                      if (error)
10                              goto allow_write_and_free_vma;
11              }
12
13              /* ->mmap() can change vma->vm_file, but must guarantee that
14               * vma_link() below can deny write-access if VM_DENYWRITE is set
15               * and map writably if VM_SHARED is set. This usually means the
16               * new file must not have been exposed to user-space, yet.
17               */
18              vma->vm_file = get_file(file);
19              error = call_mmap(file, vma);
20              if (error)
21                      goto unmap_and_free_vma;
22
23              /* Can addr have changed??
24               *
25               * Answer: Yes, several device drivers can do it in their
26               *         f_op->mmap method. -DaveM
27               * Bug: If addr is changed, prev, rb_link, rb_parent should
28               *      be updated for vma_link()
29               */
30              WARN_ON_ONCE(addr != vma->vm_start);
31
32              addr = vma->vm_start;
33              vm_flags = vma->vm_flags;
34      } else if (vm_flags & VM_SHARED) {
35              error = shmem_zero_setup(vma);
36              if (error)
37                      goto free_vma;
38      } else {
39              vma_set_anonymous(vma);
40      }
41
42      vma_link(mm, vma, prev, rb_link, rb_parent);
43      /* Once vma denies write, undo our temporary denial count */
44      if (file) {
45              if (vm_flags & VM_SHARED)
46                      mapping_unmap_writable(file->f_mapping);
47              if (vm_flags & VM_DENYWRITE)
48                      allow_write_access(file);
49      }
50      file = vma->vm_file;
```

- In line 1~33 of the code, if it is a file mapping, it is handled as follows:
  - If there is a denywrite request, the file is put into a denywrite state. If it fails, move to the free_vma label.
  - If there is a shared request, set the next writable file mapping request to be rejected for the mapping area. If it fails, move on to the allow_write_and_free_vma label.
  - Increment the f_count of the file's usage counter and assign the file to the vma_vm_file.
  - Perform file mapping with VMA information. If it fails, move to the unmap_and_free_vma label.
- In lines 34~37 of code, prepare the shared anon mapping. If it fails, move to the free_vma label.

  - Specify the "/dev/zero" file in vma->vm_file and assign the global shmem_vm_ops to vma->vm_ops.
- On lines 38~40 of code, prepare the private anon mapping.
- Add the VMA information in line 42 of code.
- In line 44~49 of the code, if it is a file mapping, it is handled as follows:
  - If it is a shared file mapping, change it to enable writable mapping for the mapping area.
  - And if it's a file mapping with a denywrite request, set it to enable writable mapping for the file.

mm/mmap.c -3/3-

```
01 out:
02         perf_event_mmap(vma);
03 
04         vm_stat_account(mm, vm_flags, len >> PAGE_SHIFT);
05         if (vm_flags & VM_LOCKED) {
06                 if ((vm_flags & VM_SPECIAL) || vma_is_dax(vma) ||
07                                         is_vm_hugetlb_page(vma) ||
08                                         vma == get_gate_vma(current->mm))
09                         vma->vm_flags &= VM_LOCKED_CLEAR_MASK;
10                 else
11                         mm->locked_vm += (len >> PAGE_SHIFT);
12         }
13 
14         if (file)
15                 uprobe_mmap(vma);
16 
17         /*
18          * New (or expanded) vma always get soft dirty status.
19          * Otherwise user-space soft-dirty page tracker won't
20          * be able to distinguish situation when vma area unmapped,
21          * then new mapped in-place (which must be aimed as
22          * a completely new data area).
23          */
24         vma->vm_flags |= VM_SOFTDIRTY;
25 
26         vma_set_page_prot(vma);
27 
28         return addr;
29 
30 unmap_and_free_vma:
31         vma->vm_file = NULL;
32         fput(file);
33 
34         /* Undo any partial mapping done by a device driver. */
35         unmap_region(mm, vma, prev, vma->vm_start, vma->vm_end);
36         charged = 0;
37         if (vm_flags & VM_SHARED)
38                 mapping_unmap_writable(file->f_mapping);
39 allow_write_and_free_vma:
40         if (vm_flags & VM_DENYWRITE)
41                 allow_write_access(file);
42 free_vma:
43         vm_area_free(vma);
44 unacct_error:
45         if (charged)
46                 vm_unacct_memory(charged);
47         return error;
48 }
```

- In code lines 1~2, the out: label. Outputs performance events information about mmap.
- In line 4 of code, add a few VM stat counters for virtual memory to the memory descriptor, as many as the LEN page.
- In lines 5~12 of the code, remove the VM_LOCKED flag for VM_SPECIAL, hugetlb page, or gate vma requests for mlock(VM_LOCKED) requests. If not, add a len page to mm->locked_vm.
- In line 14~15 of the code, if the file mapping is set up, and if the uprobe filter chain is attached, update the request address with the break point command code.
- Add the softdirty flag in lines 24~28 and log it to the VMA and return the virtual address normally.
- In code lines 30~38, unmap_and_free_vma: Label. Here, a routine of unmapping is performed.
- allow_write_and_free_vma in code lines 39~41: The label increments the i_writecount of the inode when there is a denywrite request.
- In code lines 42~43 free_vma: The label deallocates the VMA object.
- unacct_error in lines 44~47 of code: The label decrements the amount charged back to revert the amount of commit if it was metered (private writable mapping).

count_vma_pages_range()

mm/mmap.c

```
01 static unsigned long count_vma_pages_range(struct mm_struct *mm,
02                 unsigned long addr, unsigned long end)
03 {
04         unsigned long nr_pages = 0;
05         struct vm_area_struct *vma;
06 
07         /* Find first overlaping mapping */
08         vma = find_vma_intersection(mm, addr, end);
09         if (!vma)
10                 return 0;
11 
12         nr_pages = (min(end, vma->vm_end) -
13                 max(addr, vma->vm_start)) >> PAGE_SHIFT;
14 
15         /* Iterate over the rest of the overlaps */
16         for (vma = vma->vm_next; vma; vma = vma->vm_next) {
17                 unsigned long overlap_len;
18 
19                 if (vma->vm_start > end)
20                         break;
21 
22                 overlap_len = min(end, vma->vm_end) - vma->vm_start;
23                 nr_pages += overlap_len >> PAGE_SHIFT;
24         }
25 
26         return nr_pages;
27 }
```

Returns the number of pages that overlap between the request area and the existing VMA area. If there are no overlapping areas, it returns 0.

- In line 8~10 of the code, if the existing VMA area and the request area overlap, the relevant VMA is known. Returns 0 if there are no overlapping areas.
- In line 12~13 of the code, calculate the number of overlapping pages in the current VMA.

- In line 16~26 of the code, the number of overlapping pages is calculated and returned by comparing the following VMA areas.

accountable_mapping()

mm/mmap.c

```
/*
 * We account for memory if it's a private writeable mapping,
 * not hugepages and VM_NORESERVE wasn't set.
 */
```

```
static inline int accountable_mapping(struct file *file, vm_flags_t vm_flags)
{
        /*
         * hugetlb has its own accounting separate from the core VM
         * VM_HUGETLB may not be set yet so we cannot check for that flag.
         */
        if (file && is_file_hugepages(file))
                return 0;

        return (vm_flags & (VM_NORESERVE | VM_SHARED | VM_WRITE)) == VM_WRITE;
}
```

Verify that VM memory metering is the mapping that is required.

- Checking accountable mappings (private writable mappings)
  - It's not a huge file mapping, there's no noreserve and shared, and there's a write request.

Writable File Mapping(1)

inode - >i_writecount value status

- 0(writable)
  - A new writable permission request is possible without the write permission allowed
- 1(write)
  - Write permission is allowed, and new write permission requests are prohibited
- Negative (denywrite)
  - All write permission requests are prevented by denywrite requests

get_write_access()

include/linux/fs.h

```
/*
 * get_write_access() gets write permission for a file.
 * put_write_access() releases this write permission.
 * This is used for regular files.
 * We cannot support write (and maybe mmap read-write shared) accesses and
 * MAP_DENYWRITE mmappings simultaneously. The i_writecount field of an inode
 * can have the following values:
 * 0: no writers, no VM_DENYWRITE mappings
 * < 0: (-i_writecount) vm_area_structs with VM_DENYWRITE set exist
 * > 0: (i_writecount) users are writing to the file.
 *
 * Normally we operate on that counter with atomic_{inc,dec} and it's safe
 * except for the cases where we don't hold i_writecount yet. Then we need to
 * use {get,deny}_write_access() - these functions check the sign and refuse
 * to do the change if sign is wrong.
 */
```

```
static inline int get_write_access(struct inode *inode)
{
        return atomic_inc_unless_negative(&inode->i_writecount) ? 0 : -ETXTBSY;
}
```

Request write permission for the inode. Returns 0 if it succeeds, and -ETXTBSY error if it fails.

- Increments inode->i_writecount unless they are negative (-). Returns 0 if successful, and -ETXTBSY error if it is already negative (-) and cannot be increased.

put_write_access()

include/linux/fs.h

```
static inline void put_write_access(struct inode * inode)
{
        atomic_dec(&inode->i_writecount);
}
```

Remove the write permission for the inode.

- reduces inode->i_writecount.

deny_write_access()

include/linux/fs.h

```
static inline int deny_write_access(struct file *file)
{
        struct inode *inode = file_inode(file);
        return atomic_dec_unless_positive(&inode->i_writecount) ? 0 : -ETXTBSY;
}
```

It prohibits the creation of writable mappings by making the request file denywrite. Returns 0 if successful, and -ETXTBSY error if not.

- Decrements the inode->i_writecount of the request file unless it is positive (+). Returns 0 if successful, and -ETXTBSY error if it is already positive (+) and cannot be reduced.

allow_write_access()

include/linux/fs.h

```
static inline void allow_write_access(struct file *file)
{
        if (file)
                atomic_inc(&file_inode(file)->i_writecount);
}
```

Remove denywrite for the request file to allow for new writable mappings.

- Increments the inode->i_writecount of the request file.

Writable File Mapping(2)

mapping - >i_mapwritable value status (VM_SHARED counter)

- 0(writable)
  - A shared write is not mapped, and a new writable mapping request is possible
- Positive (write)
  - Shared write mapped state and new write mapping requests are prohibited
- Negative (denywrite)
  - All write mappings are forbidden by shared denywrite requests.

mapping_map_writable()

include/linux/fs.h

```
static inline int mapping_map_writable(struct address_space *mapping)
{
        return atomic_inc_unless_negative(&mapping->i_mmap_writable) ?
                0 : -EPERM;
}
```

Request a mapping area in a writable shared mapping state. Returns 0 if it succeeds, and -EPERM if it fails.

- Increase mapping->i_mmapwritable unless it is negative (-). Returns 0 if successful, and -EPERM error if it is already negative (-) and cannot be increased.

mapping_unmap_writable()

include/linux/fs.h

```
static inline void mapping_unmap_writable(struct address_space *mapping)
{
        atomic_dec(&mapping->i_mmap_writable);
}
```

Unmap the writable shared mapping area to change it to the writable state. You can get a new writable mapping request again.

- mapping-reduces >i_mmap_writable;

mapping_deny_writable()

include/linux/fs.h

```
static inline int mapping_deny_writable(struct address_space *mapping)
{
        return atomic_dec_unless_positive(&mapping->i_mmap_writable) ?
                0 : -EBUSY;
}
```

Prohibit the creation of writable mappings by changing the request mapping area to the denywritable state. Returns 0 if it succeeds, and -EBUSY if it fails.

- Reduces mapping->i_writecount unless it is positive (+). Returns 0 if successful, and -EPERM error if it is already positive (+) and cannot be reduced.

mapping_allow_writable()

include/linux/fs.h

```
static inline void mapping_allow_writable(struct address_space *mapping)
{
        atomic_inc(&mapping->i_mmap_writable);
}
```

Remove denywrite for the request mapping area to allow it to receive new writable mappings.

vm_stat_account()

mm/mmap.c

```
01 void vm_stat_account(struct mm_struct *mm, vm_flags_t flags, long npages)
02 {
03         mm->total_vm += npages;
04 
05         if (is_exec_mapping(flags))
06                 mm->exec_vm += npages;
07         else if (is_stack_mapping(flags))
08                 mm->stack_vm += npages;
09         else if (is_data_mapping(flags))
10                 mm->data_vm += npages;
11 }
```

Add as many pages for each VM stat as you want to map.

- In line 3 of code, add the number of pages to the mm->total_vm counter.
- If the code is executed on line 5~6, add the number of pages to the mm->exec_vm counter.
- In line 7~8 of the code, add the number of pages to the mm->stack_vm counter if it is a stack mapping.
- In line 9~10 of the code, add the number of pages to the mm->data_vm counter if the data is mapping.

Specifying Page Table Mapping Properties for VMAs

vma_set_page_prot()

mm/mmap.c

```
/* Update vma->vm_page_prot to reflect vma->vm_flags. */
void vma_set_page_prot(struct vm_area_struct *vma)
{
        unsigned long vm_flags = vma->vm_flags;

        vma->vm_page_prot = vm_pgprot_modify(vma->vm_page_prot, vm_flags);
        if (vma_wants_writenotify(vma)) {
                vm_flags &= ~VM_SHARED;
                vm_page_prot = vm_pgprot_modify(vma->vm_page_prot, vm_flags);
        }
        /* remove_protection_ptes reads vma->vm_page_prot without mmap_sem */
        WRITE_ONCE(vma->vm_page_prot, vm_page_prot);
}
```

Update the VMA->vm_page_prot attribute value with the memory attribute value that matches the VMA->vm_flags.

- If the architecture requires a customized cache transformation for the vm_page_prot properties of the request VMA zone, store the transformed vm_flags value. If it doesn't match, it just saves the vm_flags attributes.
- If VMA wants to use Write Notify, it removes the shared flag.
  - If the VMA area is a shared mapping and the pages are set to read only, it will return true when you want to track the write event.

vm_pgprot_modify()

mm/mmap.c

```
static pgprot_t vm_pgprot_modify(pgprot_t oldprot, unsigned long vm_flags)
{
        return pgprot_modify(oldprot, vm_get_page_prot(vm_flags));
}
```

If there is a predefined protocol translation for each architecture corresponding to oldprot, it will convert it and return a property.

- ARM converts the requesting OldProt attribute to noncache, writecombine, and device, respectively, to noncache, buffer, and noncache. If the property doesn't match, the newprot property will be returned as is, without conversion.

pgprot_modify()

include/asm-generic/pgtable.h

```
static inline pgprot_t pgprot_modify(pgprot_t oldprot, pgprot_t newprot)
{
        if (pgprot_val(oldprot) == pgprot_val(pgprot_noncached(oldprot)))
                newprot = pgprot_noncached(newprot);
        if (pgprot_val(oldprot) == pgprot_val(pgprot_writecombine(oldprot)))
```

```
                newprot = pgprot_writecombine(newprot);
        if (pgprot_val(oldprot) == pgprot_val(pgprot_device(oldprot)))
                newprot = pgprot_device(newprot);
        return newprot;
}
```

In the architecture, ODL cache properties that use old masks when mapping are converted to new cache properties.

- arm: no cache 또는 buffer

ARM32 Mapping Properties

arch/arm/include/asm/pgtable.h

```
#define __pgprot_modify(prot,mask,bits)         \
        __pgprot((pgprot_val(prot) & ~(mask)) | (bits))

#define pgprot_noncached(prot) \
        __pgprot_modify(prot, L_PTE_MT_MASK, L_PTE_MT_UNCACHED)

#define pgprot_writecombine(prot) \
        __pgprot_modify(prot, L_PTE_MT_MASK, L_PTE_MT_BUFFERABLE)

#define pgprot_stronglyordered(prot) \
        __pgprot_modify(prot, L_PTE_MT_MASK, L_PTE_MT_UNCACHED)

#ifdef CONFIG_ARM_DMA_MEM_BUFFERABLE
#define pgprot_dmacoherent(prot) \
        __pgprot_modify(prot, L_PTE_MT_MASK, L_PTE_MT_BUFFERABLE | L_PTE_XN)
#else
#define pgprot_dmacoherent(prot) \
        __pgprot_modify(prot, L_PTE_MT_MASK, L_PTE_MT_UNCACHED | L_PTE_XN)
#endif
```

Select the cache attribute that the ARM architecture uses for mapping. (uncached or buffrable)

- noncache - Use the > no cache attribute
- writecombine - use the > buffer attribute
- stronglyordered -> no cache attribute enabled
- dmacoherennt - > CONFIG_ARM_DMA_MEM_BUFFERABLE use buffer or no cache depending on kernel options
  - DMA buffer is available in armv6 or armv7.

ARM64 Mapping Properties

arch/arm64/include/asm/pgtable.h

```
/*
 * Mark the prot value as uncacheable and unbufferable.
 */
```

```
#define pgprot_noncached(prot) \
        __pgprot_modify(prot, PTE_ATTRINDX_MASK, PTE_ATTRINDX(MT_DEVICE_nGnRnE) | PTE_PXN | PTE_UXN)
#define pgprot_writecombine(prot) \
        __pgprot_modify(prot, PTE_ATTRINDX_MASK, PTE_ATTRINDX(MT_NORMAL_NC) | PTE_PXN | PTE_UXN)
#define pgprot_device(prot) \
        __pgprot_modify(prot, PTE_ATTRINDX_MASK, PTE_ATTRINDX(MT_DEVICE_nGnRE) | PTE_PXN | PTE_UXN)
#define __HAVE_PHYS_MEM_ACCESS_PROT
```

Select the cache attributes used by the arm64 architecture for mapping. (uncached or buffrable)

- noncache -> use nGnRnE attribute
- writecombine - use the > no cache attribute
- device -> Use nGnRE attribute

vma_wants_writenotify()

mm/mmap.c

```
/*
 * Some shared mappigns will want the pages marked read-only
 * to track write events. If so, we'll downgrade vm_page_prot
 * to the private version (using protection_map[] without the
 * VM_SHARED bit).
 */
```

```
01 int vma_wants_writenotify(struct vm_area_struct *vma)
02 {
03         vm_flags_t vm_flags = vma->vm_flags;
04 
05         /* If it was private or non-writable, the write bit is already clear */
06         if ((vm_flags & (VM_WRITE|VM_SHARED)) != ((VM_WRITE|VM_SHARED)))
07                 return 0;
08 
09         /* The backer wishes to know when pages are first written to? */
10         if (vma->vm_ops && vma->vm_ops->page_mkwrite || vm_ops->pfn_mkwrite))
11                 return 1;
12 
13         /* The open routine did something to the protections that pgprot_modify
14          * won't preserve? */
15         if (pgprot_val(vma->vm_page_prot) !=
16             pgprot_val(vm_pgprot_modify(vma->vm_page_prot, vm_flags)))
17                 return 0;
18 
19         /* Do we need to track softdirty? */
20         if (IS_ENABLED(CONFIG_MEM_SOFT_DIRTY) && !(vm_flags & VM_SOFTDIRTY))
21                 return 1;
22 
23         /* Specialty mapping? */
24         if (vm_flags & VM_PFNMAP)
25                 return 0;
26 
27         /* Can the mapping track the dirty pages? */
28         return vma->vm_file && vma->vm_file->f_mapping &&
29                 mapping_cap_account_dirty(vma->vm_file->f_mapping);
30 }
```

If the VMA area is a shared mapping and the pages are set to read only, it will return true when you want to track the write event.

- In line 6~7 of the code, it returns false(0) for VMA without write and shared requests.

- If the vm_ops->mkwrite callback function is specified in lines 10~11 of the code, it returns true(1).
- Returns false(15) on lines 17~0 if you don't need to change any of the vm_page_prot properties.
- Returns true(20) if lines 21~1 of the code use the CONFIG_MEM_SOFT_DIRTY kernel option and did not request the soft dirty function.
- If pfnmap mapping is requested in line 24~25 of the code, it returns false(0).
- In line 28~29 of the code, if the file mapping is set to Dirty Capable in BDI, it returns true(1). Otherwise, it returns flase(0).

consultation

- User virtual maps (brk) (http://jake.dothome.co.kr/user-virtual-maps-brk/) | 문c
- Understanding glibc malloc (https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/) | sploitfun

6 thoughts to “User virtual maps (mmap)”

GUNGMI
2021-06-10 13:51 (http://jake.dothome.co.kr/user-virtual-maps-mmap2/#comment-305507)

While looking at your well-organized article, I have a question and I am inquiring.

Is there any difference between memory allocated by mmap and memory allocated by malloc?

RESPONSE (/USER-VIRTUAL-MAPS-MMAP2/?REPLYTOCOM=305507#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)
2021-06-11 16:38 (http://jake.dothome.co.kr/user-virtual-maps-mmap2/#comment-305513)

Hello? Mr. Gungmi.

The mmap() function primarily performs the following processing:
Note: 1) When mapping a file to the user virtual space to access it directly
2) When mapping a device to the user virtual space and wanting
to control the hw Note: (https://slidesplayer.org/slide/15106509/)
https://slidesplayer.org/slide/15106509/ 3) When
you want to allocate memory in the user virtual space – this method is mainly used by heap managers in libraries such as glibc. When the heap is insufficient, use the MAP_ANONYMOUS flag to request free page memory from the kernel via mmap syscall. And in a normal user application, you don't use this method directly, but instead use malloc(), which allocates memory from the heap via glibc.

I appreciate it.

RESPONSE (/USER-VIRTUAL-MAPS-MMAP2/?REPLYTOCOM=305513#RESPOND)

TERRAIN TAK

2022-01-11 16:20 (http://jake.dothome.co.kr/user-virtual-maps-mmap2/#comment-306231)

I appreciate it.
When vm_mmap_pgoff calls do_mmap_pgoff, the last argument is populate, but the do_mmap_pgoff's declaration has uf.
Is there a mix of multiple kernel versions?

RESPONSE (/USER-VIRTUAL-MAPS-MMAP2/?REPLYTOCOM=306231#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)

2022-01-12 15:31 (http://jake.dothome.co.kr/user-virtual-maps-mmap2/#comment-306233)

Hello? Mr. Ji Ji-tak.

There was some leftover v4.9 code for the vm_mmap_pgoff() function. We've modified the version of this code to v5.0.

I appreciate it.

RESPONSE (/USER-VIRTUAL-MAPS-MMAP2/?REPLYTOCOM=306233#RESPOND)

GANGHWA

2022-02-25 17:34 (http://jake.dothome.co.kr/user-virtual-maps-mmap2/#comment-306346)

Thank you very much.
I have one question. If I
set the MAP_POPULATE flag when calling the mmap() function in the user area, does that mean that lazy allocation doesn't happen?

RESPONSE (/USER-VIRTUAL-MAPS-MMAP2/?REPLYTOCOM=306346#RESPOND)

MOON YOUNG-IL (HTTP://JAKE.DOTHOME.CO.KR)

2022-02-26 23:13 (http://jake.dothome.co.kr/user-virtual-maps-mmap2/#comment-306350)

Hello?

네. 맞습니다. page fault가 발생하지 않도록 미리 페이지 테이블을 구성합니다. 즉 메모리를 할당하고 미리 매핑합니다.

감사합니다.

응답 (/USER-VIRTUAL-MAPS-MMAP2/?REPLYTOCOM=306350#RESPOND)

댓글 남기기

이메일은 공개되지 않습니다. 필수 입력창은 * 로 표시되어 있습니다

댓글

이름 *

이메일 *

웹사이트

댓글 작성

< LIM(Linux Integrity Module) -1- (http://jake.dothome.co.kr/lim/)

ABI(Application Binary Interface) > (http://jake.dothome.co.kr/abi/)

문c 블로그 (2015 ~ 2024)