User virtual maps (brk)

2016-12-15 (http://jake.dothome.co.kr/user-virtual-maps-brk/) Moon Young-il (http://jake.dothome.co.kr/author/admin/) Linux Kernel (http://jake.dothome.co.kr/category/linux/)

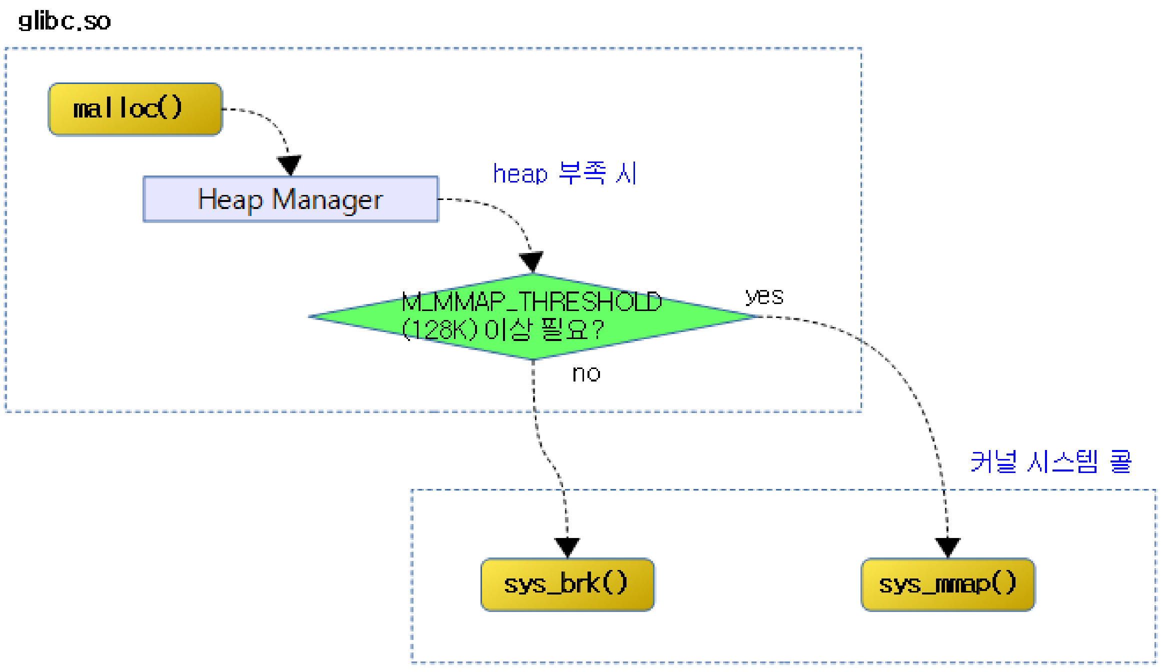
<kernel v5.0>

Requesting heap or stack memory increase/decrease in user process

Heap Manager

When using the malloc() function in user space, the heap memory manager included in libraries such as glibc manages the heap pool. If more heap memory is needed, the kernel will make a Posix system call to the user anon memory, and the function called by the kernel is brk() or mmap().

- M_MMAP_THRESHOLD (128K)
  - You can use the mallopt() function to change the threadold threshold.
  - When you want to expand the heap below the threadzold, the brk() function is called, which is a simplified version of mmap() that only treats memory of type anonymouse.
  - If you extend the heap beyond the threadzold, the mmap() function is called.



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

Custom Heap Manager

In most cases, it's easy and comfortable to use GNU's glibc library in C and the FASMLIB library in assembly for heap management. However, there are times when you don't want to use these libraries and need to use a custom heap manager specifically. Of course, developers can write custom heap memory managers that call brk() or map() system call functions to create special heap managers for complex threads.

sys_brk()

mm/mmap.c -1/2-

```
SYSCALL_DEFINE1(brk, unsigned long, brk)
{
        unsigned long retval;
        unsigned long newbrk, oldbrk, origbrk;
        struct mm_struct *mm = current->mm;
        struct vm_area_struct *next;
        unsigned long min_brk;
        bool populate;
        bool downgraded = false;
        LIST_HEAD(uf);

        if (down_write_killable(&mm->mmap_sem))
                return -EINTR;

        origbrk = mm->brk;

#ifdef CONFIG_COMPAT_BRK
        /*
         * CONFIG_COMPAT_BRK can still be overridden by setting
         * randomize_va_space to 2, which will still cause mm->start_brk
         * to be arbitrarily shifted
         */
        if (current->brk_randomized)
                min_brk = mm->start_brk;
        else
                min_brk = mm->end_data;
#else
        min_brk = mm->start_brk;
#endif
        if (brk < min_brk)
                goto out;

        /*
         * Check against rlimit here. If this check is done later after the test
         * of oldbrk with newbrk then it can escape the test and let the data
         * segment grow beyond its set limit the in case where the limit is
         * not page aligned -Ram Gupta
         */
        if (check_data_rlimit(rlimit(RLIMIT_DATA), brk, mm->start_brk,
                              mm->end_data, mm->start_data))
                goto out;

        newbrk = PAGE_ALIGN(brk);
        oldbrk = PAGE_ALIGN(mm->brk);
        if (oldbrk == newbrk) {
                mm->brk = brk;
                goto success;
        }
```

```
49
50              /*
51               * Always allow shrinking brk.
52               * __do_munmap() may downgrade mmap_sem to read.
53               */
54              if (brk <= mm->brk) {
55                      int ret;
56
57                      /*
58                       * mm->brk must to be protected by write mmap_sem so upd
ate it
59                       * before downgrading mmap_sem. When __do_munmap() fail
s,
60                       * mm->brk will be restored from origbrk.
61                       */
62                      mm->brk = brk;
63                      ret = __do_munmap(mm, newbrk, oldbrk-newbrk, &uf, true);
64                      if (ret < 0) {
65                              mm->brk = origbrk;
66                              goto out;
67                      } else if (ret == 1) {
68                              downgraded = true;
69                      }
70                      goto success;
71              }
```

Expand or shrink the heap (data) area. When expanding, it is assigned to the anon user page, and when it is reduced, the area is unmapped.

- In lines 23~28 of the code, the heap position placed in the user space is used randomly for security purposes, but for compatible purposes such as legacy libc5, this kernel option is used to disable the heap start position random feature.
  - If you do "echo 2 > /proc/sys/kernel/randomize_va_space", use the random heap start position feature.
  - If you use the "norandmaps" kernel parameter, you can also disable the heap start position randomization feature.
- If lines 30~31 of the code request a heap address lower than the heap start address set in the task, it will give up the allocation and move to the out label.
- In lines 39~41 of the code, if the size of the data used by the current task exceeds the data area limit set for the current task, it abandons the allocation and moves to the out label.
  - RLIMIT_DATA: Data (heap usage + data section) size limit
- If there is no page-level change in the heap address requested in lines 43~48 of the code, move to the success label.
- If the heap address requested in code lines 54~71 is less than the existing heap address set in the task, the heap is shrinked by unmapping the reduced difference. If the collapse is successful, it goes to the success label, otherwise it goes to the out label.
  - 참고: mm: brk: downgrade mmap_sem to read when shrinking (https://github.com/torvalds/linux/commit/9bc8039e715da3b53dbac89525323a9f2f69b7b5#diff-fdaaedaf96a6a572e21218cb5e8691d9) (2018, v4.20-rc1)

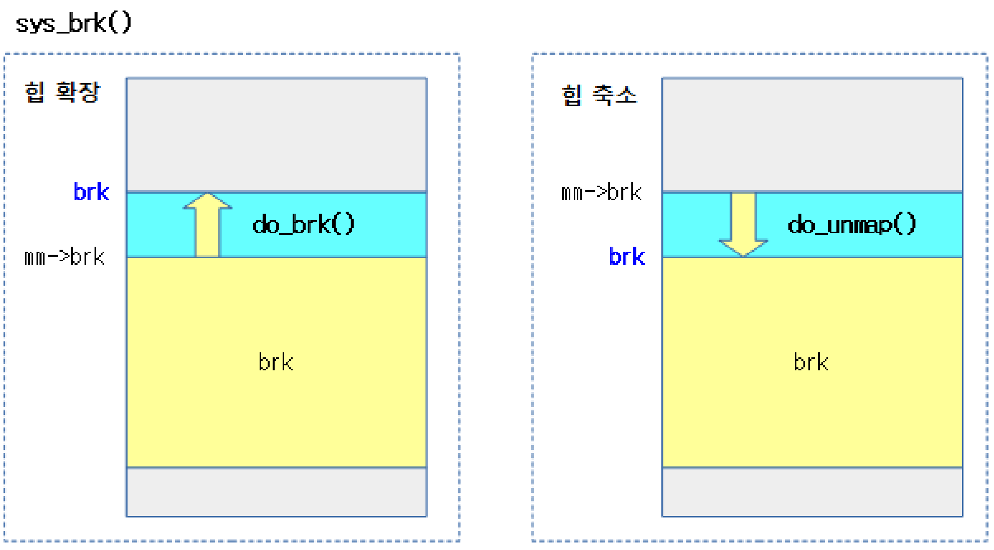
mm/mmap.c -2/2-

```
01              /* Check against existing mmap mappings. */
02              next = find_vma(mm, oldbrk);
```

```
03         if (next && newbrk + PAGE_SIZE > vm_start_gap(next))
04                 goto out;
05
06         /* Ok, looks good - let it rip. */
07         if (do_brk_flags(oldbrk, newbrk-oldbrk, 0, &uf) < 0)
08                 goto out;
09         mm->brk = brk;
10
11 success:
12         populate = newbrk > oldbrk && (mm->def_flags & VM_LOCKED) != 0;
13         if (downgraded)
14                 up_read(&mm->mmap_sem);
15         else
16                 up_write(&mm->mmap_sem);
17         userfaultfd_unmap_complete(mm, &uf);
18         if (populate)
19                 mm_populate(oldbrk, newbrk - oldbrk);
20         return brk;
21
22 out:
23         retval = origbrk;
24         up_write(&mm->mmap_sem);
25         return retval;
26 }
```

- If the heap space to be increased in line 2~4 of code is already mapped, give up the allocation and move it to the out label.
- In line 7~8 of the code, expand the existing VM and return it, or configure a new VM and import it. If it fails, move to the out label.
- In code lines 11~19, the success: label is. If there is an increased heap and it is a VMA in a mlocked area, all user pages for that area are assigned and mapped.
    - ARM64 Page Table-2- (mapping) (http://jake.dothome.co.kr/map64) | Qc
- Returns the end address of the new heap on line 20 of the code.
- In code lines 22~25, the out: label. Returns the address of the end of the existing heap.

The following illustration shows how the heap expands or shrinks when the do_brk() function is called.



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

check_data_rlimit()

include/linux/mm.h

```
static inline int check_data_rlimit(unsigned long rlim,
                                    unsigned long new,
                                    unsigned long start,
                                    unsigned long end_data,
                                    unsigned long start_data)
{
        if (rlim < RLIM_INFINITY) {
                if (((new - start) + (end_data - start_data)) > rlim)
                        return -ENOSPC;
        }

        return 0;
}
```

If the size of the data usage exceeds the rlim, it returns the -ENOSPC error value.

- data=heap usage + data section

do_brk_flags()

mm/mmap.c

```
/*
 *  this is really a simplified "do_mmap".  it only handles
 *  anonymous maps.  eventually we may be able to do some
 *  brk-specific accounting here.
 */
```

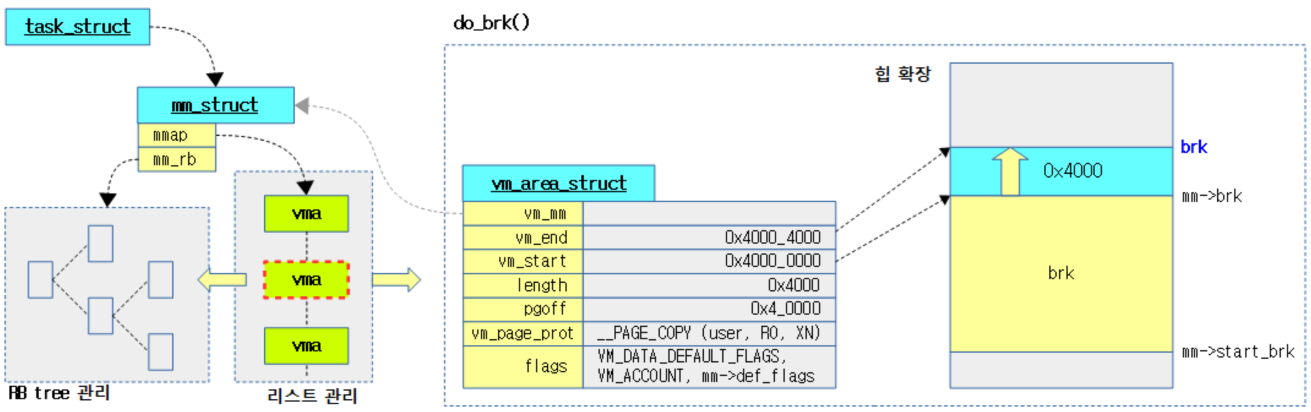
```
static int do_brk_flags(unsigned long addr, unsigned long len, unsigned long flags, struct list_headd
 *uf)
{
        struct mm_struct *mm = current->mm;
        struct vm_area_struct *vma, *prev;
        struct rb_node **rb_link, *rb_parent;
        pgoff_t pgoff = addr >> PAGE_SHIFT;
        int error;

        /* Until we need other flags, refuse anything except VM_EXEC. */
        if ((flags & (~VM_EXEC)) != 0)
                return -EINVAL;
        flags |= VM_DATA_DEFAULT_FLAGS | VM_ACCOUNT | mm->def_flags;

        error = get_unmapped_area(NULL, addr, len, 0, MAP_FIXED);
        if (offset_in_page(error))
                return error;

        error = mlock_future_check(mm, mm->def_flags, len);
        if (error)
                return error;

        /*
         * Clear old maps.  this also does some error checking for us
         */
        while (find_vma_links(mm, addr, addr + len, &prev, &rb_link,
                              &rb_parent)) {
                if (do_munmap(mm, addr, len, uf))
                        return -ENOMEM;
        }

        /* Check against address space limits *after* clearing old map
s... */
        if (!may_expand_vm(mm, flags, len >> PAGE_SHIFT))
34                 return -ENOMEM;
35 
36         if (mm->map_count > sysctl_max_map_count)
37                 return -ENOMEM;
38 
39         if (security_vm_enough_memory_mm(mm, len >> PAGE_SHIFT))
40                 return -ENOMEM;
41 
42         /* Can we just expand an old private anonymous mapping? */
43         vma = vma_merge(mm, prev, addr, addr + len, flags,
44                         NULL, NULL, pgoff, NULL, NULL_VM_UFFD_CTX);
45         if (vma)
46                 goto out;
47 
48         /*
49          * create a vma struct for an anonymous mapping
50          */
51         vma = vm_area_alloc(mm);
52         if (!vma) {
53                 vm_unacct_memory(len >> PAGE_SHIFT);
54                 return -ENOMEM;
55         }
56 
57         vma_set_anonymous(vma);
58         vma->vm_start = addr;
59         vma->vm_end = addr + len;
60         vma->vm_pgoff = pgoff;
61         vma->vm_flags = flags;
62         vma->vm_page_prot = vm_get_page_prot(flags);
63         vma_link(mm, vma, prev, rb_link, rb_parent);
64 out:
65         perf_event_mmap(vma);
66         mm->total_vm += len >> PAGE_SHIFT;
67         mm->data_vm += len >> PAGE_SHIFT;
68         if (flags & VM_LOCKED)
69                 mm->locked_vm += (len >> PAGE_SHIFT);
70         vma->vm_flags |= VM_SOFTDIRTY;
71         return 0;
72 }
```

Static map of the area of the virtual address and length requested in the current task user area. Returns 0 on success.

- In line 7 of code, put the page number for the virtual address in pgoff.
  - Cache aliasing is used to compare whether the start address is appropriate on a system that uses aliasing.
- In lines 11~12 of the code, all flags except VM_EXEC are not allowed.
- Line 13 through line 19 of code contains the def_flags of the VM_DATA_DEFAULT_FLAGS and the VM_ACCOUNT and the current task.
  - VM_DATA_DEFAULT_FLAGS:
    - VM_EXEC (optional) | VM_READ | VM_WRITE | VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC
- In lines 15~17 of the code, we find out whether there is an empty space equal to the length of len from the request virtual address in the current task user area to be fixed mapped. If there is no space to map, it returns an error.
- In line 19~21 of the code, if the number of pages using the VM_LOCKED flag in the current task exceeds the maximum locked page limit, it returns a -EAGAIN error.
- In line 26~30 of the code, if the VMAs overlap with the request virtual address area, perform an unmap.

- In line 33~34 of the code, if the total number of vm pages in the current task exceeds the address space limit (RLIMIT_AS), it returns a -ENOMEM error.
- In code lines 36~37, if the number of VMAs in the current task exceeds the maximum number of mapping counts, returns -ENOMEM.
    - sysctl_max_map_count
        - By default, it is 65530, which can be changed via "/proc/sys/vm/max_map_count".
- In line 39~40 of the code, check whether the allocation of the new virtual mapping is sufficient via the Linux Security Module (LSM).
    - LSM (Linux Security Module) (http://jake.dothome.co.kr/lsm) | 문c
- In line 43~46 of the code, if the existing VMA uses the Private Anonymous mapping, make sure it can be extended by merging, and if possible, go to the out label.
- In line 51~55 of the code, it allocates memory for the vma structure and reverts to the already increased number of vm commit pages in case of failure and returns a -ENOMEM error.
- Configure and add VMA information in code lines 57~63.
    - The mapping properties that go into the vm_page_prot are basically __PAGE_COPY, even if there are no properties specifically added to the memory descriptor (mm->def_flags).
- This is the starting position of the out label in code lines 64~67 and updates the total number of used VM pages in the memory descriptor.
- If you buy the VM_LOCKED flag in line 68~69 of code, update the locked_vm page counter in the memory descriptor.
- In line 70 of the code, add VM_SOFTDIRTY to the VMA's flag.
- Success returns 71 on line 0 of the code.

The following figure shows how the brk() request is extended using an existing VMA.



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

VM Flags - > Page Mapping Properties

vm_get_page_prot()

mm/mmap.c

```
pgprot_t vm_get_page_prot(unsigned long vm_flags)
{
        return __pgprot(pgprot_val(protection_map[vm_flags &
                                (VM_READ|VM_WRITE|VM_EXEC|VM_SHARED)]) |
                        pgprot_val(arch_vm_get_page_prot(vm_flags)));
}
EXPORT_SYMBOL(vm_get_page_prot);
```

Use the read, write, exec, and shared flag values among the VM flags to find out which page mapping attribute values are needed.

Global protection_map[] array

mm/mmap.c

```
/* description of effects of mapping type and prot in current implementation.
 * this is due to the limited x86 page protection hardware.  The expected
 * behavior is in parens:
 *
 * map_type     prot
 *              PROT_NONE       PROT_READ       PROT_WRITE      PROT_EXEC
 * MAP_SHARED   r: (no) no      r: (yes) yes    r: (no) yes     r: (no) yes
 *              w: (no) no      w: (no) no      w: (yes) yes    w: (no) no
 *              x: (no) no      x: (no) yes     x: (no) yes     x: (yes) yes
 *
 * MAP_PRIVATE  r: (no) no      r: (yes) yes    r: (no) yes     r: (no) yes
 *              w: (no) no      w: (no) no      w: (copy) copy  w: (no) no
 *              x: (no) no      x: (no) yes     x: (no) yes     x: (yes) yes
 *
 */
```

```
pgprot_t protection_map[16] __ro_after_init = {
        __P000, __P001, __P010, __P011, __P100, __P101, __P110, __P111,
        __S000, __S001, __S010, __S011, __S100, __S101, __S110, __S111
};
```

There are 4 arrays with 16 flag bits.

16 Page Mapping Properties – ARM32

arch/arm/include/asm/pgtable.h

```
/*
 * The table below defines the page protection levels that we insert into our
 * Linux page table version.  These get translated into the best that the
 * architecture can perform.  Note that on most ARM hardware:
 *  1) We cannot do execute protection
 *  2) If we could do execute protection, then read is implied
 *  3) write implies read permissions
 */
```

```
#define __P000  __PAGE_NONE
#define __P001  __PAGE_READONLY
#define __P010  __PAGE_COPY
```

```
#define __P011   __PAGE_COPY
#define __P100   __PAGE_READONLY_EXEC
#define __P101   __PAGE_READONLY_EXEC
#define __P110   __PAGE_COPY_EXEC
#define __P111   __PAGE_COPY_EXEC

#define __S000   __PAGE_NONE
#define __S001   __PAGE_READONLY
#define __S010   __PAGE_SHARED
#define __S011   __PAGE_SHARED
#define __S100   __PAGE_READONLY_EXEC
#define __S101   __PAGE_READONLY_EXEC
#define __S110   __PAGE_SHARED_EXEC
#define __S111   __PAGE_SHARED_EXEC
```

16 macro constants combined with read, write, exec, and shared flag bits

arch/arm/include/asm/pgtable.h

```
#define __PAGE_NONE                __pgprot(_L_PTE_DEFAULT | L_PTE_RDONLY | L_PTE_XN | L_PTE_NONE)
#define __PAGE_SHARED              __pgprot(_L_PTE_DEFAULT | L_PTE_USER | L_PTE_XN)
#define __PAGE_SHARED_EXEC         __pgprot(_L_PTE_DEFAULT | L_PTE_USER)
#define __PAGE_COPY                __pgprot(_L_PTE_DEFAULT | L_PTE_USER | L_PTE_RDONLY | L_PTE_XN)
#define __PAGE_COPY_EXEC           __pgprot(_L_PTE_DEFAULT | L_PTE_USER | L_PTE_RDONLY)
#define __PAGE_READONLY            __pgprot(_L_PTE_DEFAULT | L_PTE_USER | L_PTE_RDONLY | L_PTE_XN)
#define __PAGE_READONLY_EXEC       __pgprot(_L_PTE_DEFAULT | L_PTE_USER | L_PTE_RDONLY)
```

- __PAGE_NONE
  - It is used for Automatic NUMA balancing, which considers the migration of tasks that use the page after it fails due to an accesss permission failure when reading the page from a NUMA system.
- __PAGE_COPY
  - L_PTE_PRESENT: Mapped
  - L_PTE_YOUNG: Accessible to the page
  - L_PTE_USER: Allow users
  - L_PTE_RDONLY: Read-only
  - L_PTE_XN: Excute Never

16 Page Mapping Properties – ARM64

arch/arm64/include/asm/pgtable-prot.h

```
#define __P000   PAGE_NONE
#define __P001   PAGE_READONLY
#define __P010   PAGE_READONLY
#define __P011   PAGE_READONLY
#define __P100   PAGE_EXECONLY
#define __P101   PAGE_READONLY_EXEC
#define __P110   PAGE_READONLY_EXEC
#define __P111   PAGE_READONLY_EXEC

#define __S000   PAGE_NONE
#define __S001   PAGE_READONLY
#define __S010   PAGE_SHARED
```

```
#define __S011  PAGE_SHARED
#define __S100  PAGE_EXECONLY
#define __S101  PAGE_READONLY_EXEC
#define __S110  PAGE_SHARED_EXEC
#define __S111  PAGE_SHARED_EXEC
```

arch/arm64/include/asm/pgtable-prot.h

```
#define PAGE_NONE               __pgprot(((_PAGE_DEFAULT) & ~PTE_VALID) | PTE_PROT_NONE | PTE_RDONLYY
 | PTE_NG | PTE_PXN | PTE_UXN)
#define PAGE_SHARED             __pgprot(_PAGE_DEFAULT | PTE_USER | PTE_NG | PTE_PXN | PTE_UXN | PTEE
_WRITE)
#define PAGE_SHARED_EXEC        __pgprot(_PAGE_DEFAULT | PTE_USER | PTE_NG | PTE_PXN | PTE_WRITE)
#define PAGE_READONLY           __pgprot(_PAGE_DEFAULT | PTE_USER | PTE_RDONLY | PTE_NG | PTE_PXN |
PTE_UXN)
#define PAGE_READONLY_EXEC      __pgprot(_PAGE_DEFAULT | PTE_USER | PTE_RDONLY | PTE_NG | PTE_PXN)
#define PAGE_EXECONLY           __pgprot(_PAGE_DEFAULT | PTE_RDONLY | PTE_NG | PTE_PXN)
```

- _PAGE_DEFAULT
  - PTE_ATTRINDX(MT_NORMAL) | PTE_TYPE_PAGE | PTE_AF | PTE_SHARED

Find unmapped address zones

get_unmapped_area()

mm/mmap.c

```
unsigned long
get_unmapped_area(struct file *file, unsigned long addr, unsigned long len,
                unsigned long pgoff, unsigned long flags)
{
        unsigned long (*get_area)(struct file *, unsigned long,
                                  unsigned long, unsigned long, unsigned long);

        unsigned long error = arch_mmap_check(addr, len, flags);
        if (error)
                return error;

        /* Careful about overflows.. */
        if (len > TASK_SIZE)
                return -ENOMEM;

        get_area = current->mm->get_unmapped_area;
        if (file) {
                if (file->f_op->get_unmapped_area)
                        get_area = file->f_op->get_unmapped_area;
        } else if (flags & MAP_SHARED) {
                /*
                 * mmap_region() will call shmem_zero_setup() to create a file,
                 * so use shmem's get_unmapped_area in case it can be huge.
24                  * do_mmap_pgoff() will clear pgoff, so match alignment.
25                  */
26                 pgoff = 0;
27                 get_area = shmem_get_unmapped_area;
28         }
29 
30         addr = get_area(file, addr, len, pgoff, flags);
31         if (IS_ERR_VALUE(addr))
32                 return addr;
33 
34         if (addr > TASK_SIZE - len)
35                 return -ENOMEM;
36         if (offset_in_page(addr))
37                 return -EINVAL;
38 
39         error = security_mmap_addr(addr);
40         return error ? error : addr;
41 }
42 EXPORT_SYMBOL(get_unmapped_area);
```

Search in the mmap mapping space of the current task user area to find the virtual start address of an empty unmapped area where the request length can fit.

- If the user space requested in line 8~10 of the code is a mapping space that is not supported by the architecture, it returns an error.
- In lines 13~14 of the code, if the request allocation length is greater than the size of the user space, it returns a -ENOMEM error.
    - TASK_SIZE: The size of the user space
        - e.g. rpi2: 2G – 16M (arm size excluding kernel space and module (16M) space)
- Identify the unmapped area in code lines 16~32.
    - If file is a mapping, use a handler function that knows the unmapped area from the file descriptor.
        - Note that do_brk() doesn't use the file mapping, but calls it by assigning null to the file argument for the anon mapping.
    - The handler function for the shared area uses the shmem_get_unmapped_area() function.
- If the virtual address learned from lines 34~35 of the code cannot be placed in user space, it returns a -ENOMEM error.
- If the virtual address learned from lines 36~37 is 0 pages, it returns a -EINVAL error.
- When SELinux, which is used for security in lines 39~30 of the code, returns the virtual address if it is authorized, and returns an error if it does not.

arch_mmap_check()

arch/arm/include/uapi/asm/mman.h

```
#define arch_mmap_check(addr, len, flags) \
        (((flags) & MAP_FIXED && (addr) < FIRST_USER_ADDRESS) ? -EINVAL : 0)
```

If a static mapping is required to the requested virtual address, it returns -EINVAL if the virtual address is less than or equal to the starting userspace address.

- FIRST_USER_ADDRESS: (PAGE_SIZE * 2)

  - If you're using low vectors in ARM, the bottom two pages are spaces that aren't available to the user.
- For ARM64 and x86, it always returns 0 success.

arch_get_unmapped_area() – ARM32

arch/arm/mm/mmap.c

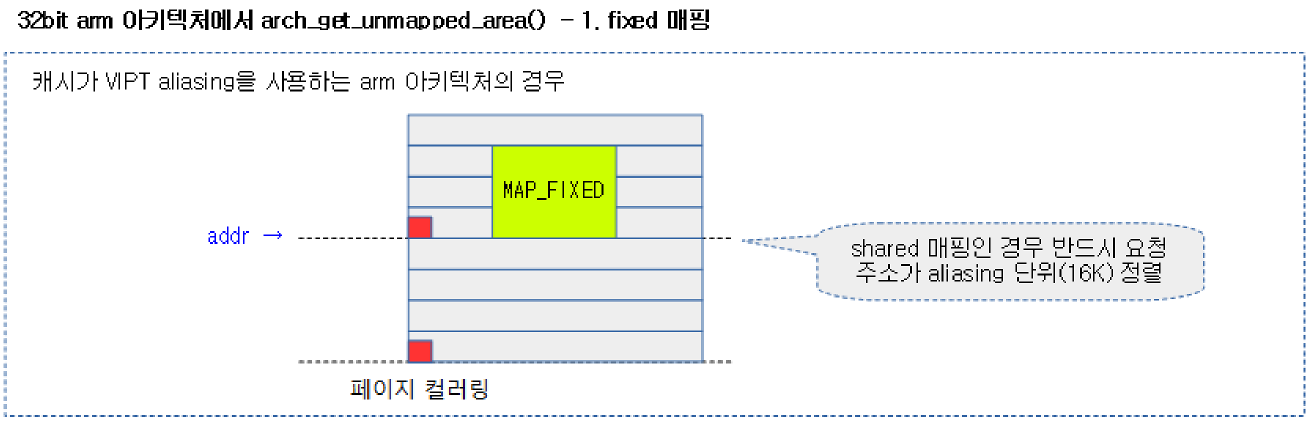
```
/*
 * We need to ensure that shared mappings are correctly aligned to
 * avoid aliasing issues with VIPT caches.  We need to ensure that
 * a specific page of an object is always mapped at a multiple of
 * SHMLBA bytes.
 *
 * We unconditionally provide this function for all cases, however
 * in the VIVT case, we optimise out the alignment rules.
 */
```

```
unsigned long
arch_get_unmapped_area(struct file *filp, unsigned long addr,
                unsigned long len, unsigned long pgoff, unsigned long flags)
{
        struct mm_struct *mm = current->mm;
        struct vm_area_struct *vma;
        int do_align = 0;
        int aliasing = cache_is_vipt_aliasing();
        struct vm_unmapped_area_info info;

        /*
         * We only need to do colour alignment if either the I or D
         * caches alias.
         */
        if (aliasing)
                do_align = filp || (flags & MAP_SHARED);

        /*
         * We enforce the MAP_FIXED case.
         */
        if (flags & MAP_FIXED) {
                if (aliasing && flags & MAP_SHARED &&
                    (addr - (pgoff << PAGE_SHIFT)) & (SHMLBA - 1))
                        return -EINVAL;
                return addr;
        }

        if (len > TASK_SIZE)
                return -ENOMEM;

        if (addr) {
                if (do_align)
                        addr = COLOUR_ALIGN(addr, pgoff);
                else
                        addr = PAGE_ALIGN(addr);

                vma = find_vma(mm, addr);
                if (TASK_SIZE - len >= addr &&
                    (!vma || addr + len <= vma->vm_start))
                        return addr;
        }

        info.flags = 0;
        info.length = len;
        info.low_limit = mm->mmap_base;
46          info.high_limit = TASK_SIZE;
47          info.align_mask = do_align ? (PAGE_MASK & (SHMLBA - 1)) : 0;
48          info.align_offset = pgoff << PAGE_SHIFT;
49          return vm_unmapped_area(&info);
50  }
```

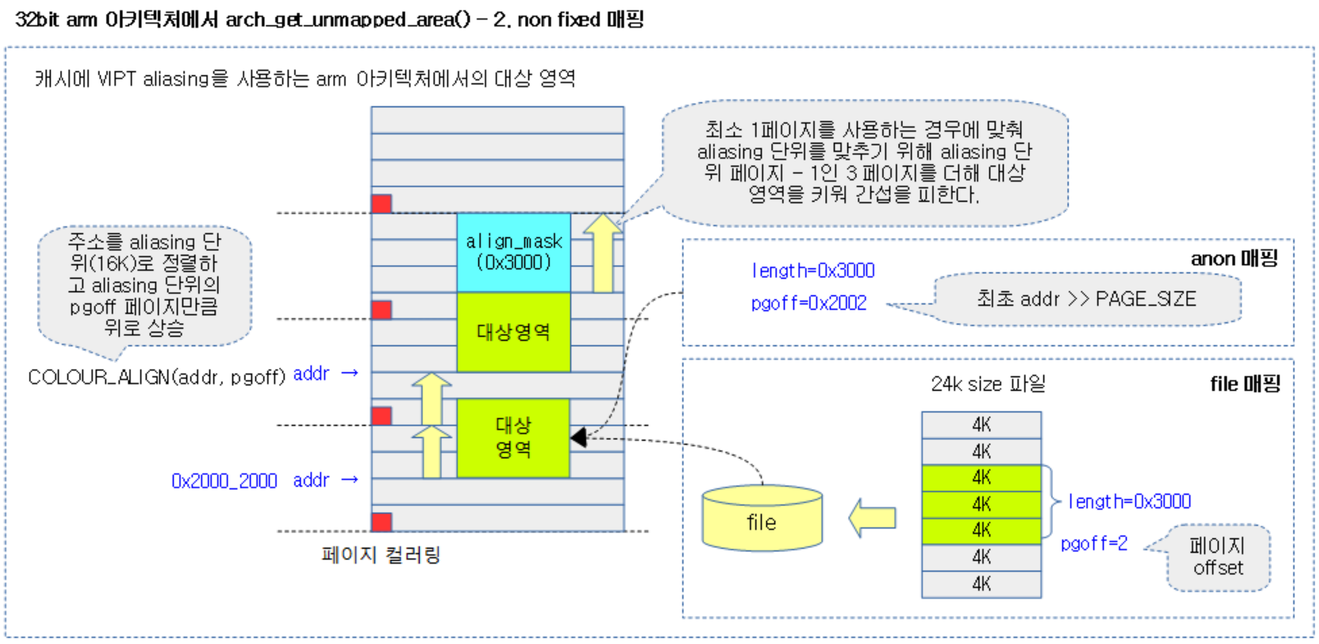
A function implemented for the ARM architecture that searches from bottom to top in the mmap mapping space of the current task user area to find the virtual start address of an empty unmapped area where the request length can fit.

- In line 8 of the code, we find out whether the data cache in the current architecture uses vipt aliasing.
    - rpi2: Don't use aliasing.
- If aliasing is used in lines 15~16 of the code, assign the flip(file struct pointer) value + MAP_SHARED(1) flag to the do_align value to determine whether alignment is required.
- If you request a VM static mapping in lines 21~26 of the code, you need to use the requested virtual address as it is, so you can exit the function without changing the virtual address. However, if aliasing and sharing maps are requested, the virtual address with the pgoff page reduced by the amount of the pgoff page is not sorted by the SHMLBA size required for aliasing, it returns -EINVAL.
    - SHMLBA: In pre-armv7 architectures, compensation for aliasing prevents the use of virtual address mappings of 4 pages (16K) between tasks if mapping pages need to be shared.
        - See: Cache – VIPT Cache Coloring (http://jake.dothome.co.kr/cache6) | Qc
- In lines 28~29 of the code, if the length exceeds the size of the user space, it returns a -ENOMEM error.
- If the virtual address is specified in lines 31~35 of the code, it should be sorted by page, but if the do_align is set, the virtual address should be sorted according to aliasing.
- In line 37 of the code, the VMA is retrieved from the VMA information registered with the current task.
- If the controlled virtual address in line 38~40 is placeable in user space and is below the VMA area, it exits the function.
- In lines 43~49 of the code, configure the VM unmapped zone information as follows, and then find and return the unmapped zone.
    - In the low_limit, substitute the address of the lower limit of the mapping area.
    - In the high_limit, the upper limit of the user space is assigned.
    - align_mask assigns the required size-1 page (3 pages = 0x3000) for ARM architectures that require aliasing, so that the back of the area to be mapped is not used.
    - align_offset assign the offset page number to be read from the page.

The two figures below show that the start address and size of the mapping space can be changed by cache aliasing, depending on whether it is fixed mapping or not.


32bit arm 아키텍처에서 arch_get_unmapped_area() – 1. fixed 매핑
캐시가 VIPT aliasing을 사용하는 arm 아키텍처의 경우
MAP_FIXED
addr →
shared 매핑인 경우 반드시 요청 주소가 aliasing 단위(16K) 정렬
페이지 컬러링

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


32bit arm 아키텍처에서 arch_get_unmapped_area() – 2. non fixed 매핑
캐시에 VIPT aliasing을 사용하는 arm 아키텍처에서의 대상 영역
최소 1페이지를 사용하는 경우에 맞춰 aliasing 단위를 맞추기 위해 aliasing 단위 페이지 – 1인 3 페이지를 더해 대상 영역을 키워 간섭을 피한다.
주소를 aliasing 단위(16K)로 정렬하고 aliasing 단위의 pgoff 페이지만큼 위로 상승
align_mask (0x3000)
대상영역
COLOUR_ALIGN(addr, pgoff) addr →
대상 영역
0x2000_2000 addr →
페이지 컬러링
anon 매핑
length=0x3000
pgoff=0x2002
최초 addr >> PAGE_SIZE
24k size 파일
file 매핑
4K
length=0x3000
pgoff=2
페이지 offset
file

(http://jake.dothome.co.kr/wp-content/uploads/2016/12/arch_get_unmapped_area-2.png)

COLOUR_ALIGN()

mm/mmap.c

```
#define COLOUR_ALIGN(addr,pgoff)                \
        ((((addr)+SHMLBA-1)&~(SHMLBA-1)) +      \
         (((pgoff)<<PAGE_SHIFT) & (SHMLBA-1)))
```

If the data cache uses VIP aliasing, it will align the request virtual address with the page coloring.

- 예) SHMLBA=16K , addr=0x1234_6000, pgoff=7
  - Sort the virtual addresses in 16K units + divide the pgoff page by 16K pages.
    - 0x1234_8000 + 0x3000 = 0x1234_b000

arch_get_unmapped_area() – Generic (x86, ARM64, …)

mm/mmap.c

```
/* Get an address range which is currently unmapped.
 * For shmat() with addr=0.
 *
 * Ugly calling convention alert:
 * Return value with the low bits set means error value,
 * ie
 *      if (ret & ~PAGE_MASK)
 *              error = ret;
 *
 * This function "knows" that -ENOMEM has the bits set.
 */
```

```
01 #ifndef HAVE_ARCH_UNMAPPED_AREA
02 unsigned long
03 arch_get_unmapped_area(struct file *filp, unsigned long addr,
04                 unsigned long len, unsigned long pgoff, unsigned long fl
   ags)
05 {
06         struct mm_struct *mm = current->mm;
07         struct vm_area_struct *vma, *prev;
08         struct vm_unmapped_area_info info;
09         const unsigned long mmap_end = arch_get_mmap_end(addr);
10 
11         if (len > mmap_end - mmap_min_addr)
12                 return -ENOMEM;
13 
14         if (flags & MAP_FIXED)
15                 return addr;
16 
17         if (addr) {
18                 addr = PAGE_ALIGN(addr);
19                 vma = find_vma_prev(mm, addr, &prev);
20                 if (mmap_end - len >= addr && addr >= mmap_min_addr &&
21                     (!vma || addr + len <= vm_start_gap(vma)) &&
22                     (!prev || addr >= vm_end_gap(prev)))
23                         return addr;
24         }
25 
26         info.flags = 0;
27         info.length = len;
28         info.low_limit = mm->mmap_base;
29         info.high_limit = mmap_end;
30         info.align_mask = 0;
31         return vm_unmapped_area(&info);
32 }
33 #endif
```

A function implemented in generic (including x86, ARM64, etc.) that searches from bottom to top in the mmap mapping space of the current task user area to find the virtual start address of an empty unmapped area where the request length can fit.

- If lines 11~12 of the code exceed the maximum size supported by the architecture, it returns a -ENOMEM error.
- If you use the MAP_FIXED flag in lines 14~15 of the code, it will return the @addr as it is without address conversion.
- If the virtual address @addr is specified in lines 17~24 of the code, sort it page by page and retrieve the VMA from the VMA information registered in the current task. If the virtual address is placeable in user space and below the VMA area, exit the function.
- In lines 26~31 of the code, find the unmapped area with the following information and return its virtual address.
  - In the low_limit, substitute the address of the lower limit of the mapping area.
  - In the high_limit, the upper limit of the user space is assigned.

arch_get_mmap_base() & arch_get_mmap_end() – ARM64

arch/arm64/include/asm/processor.h

```
#ifndef CONFIG_ARM64_FORCE_52BIT
#define arch_get_mmap_end(addr) ((addr > DEFAULT_MAP_WINDOW) ? TASK_SIZE :\
                                DEFAULT_MAP_WINDOW)

#define arch_get_mmap_base(addr, base) ((addr > DEFAULT_MAP_WINDOW) ? \
                                        base + TASK_SIZE - DEFAULT_MAP_WINDOW :\
                                        base)
#endif /* CONFIG_ARM64_FORCE_52BIT */
```

vm_unmapped_area()

include/linux/mm.h

```
/*
 * Search for an unmapped address range.
 *
 * We are looking for a range that:
 * - does not intersect with any VMA;
 * - is contained within the [low_limit, high_limit) interval;
 * - is at least the desired size.
 * - satisfies (begin_addr & align_mask) == (align_offset & align_mask)
 */
```

```
static inline unsigned long
vm_unmapped_area(struct vm_unmapped_area_info *info)
{
        if (!(info->flags & VM_UNMAPPED_AREA_TOPDOWN))
                return unmapped_area(info);
        else
                return unmapped_area_topdown(info);
}
```

Determine the virtual address of the unmapped area.

- Whether or not VM_UNMAPPED_AREA_TOPDOWN flag is used determines the direction of the search.

Find an address zone from bottom to top

unmapped_area()

mm/mmap.c -1/2-

```
unsigned long unmapped_area(struct vm_unmapped_area_info *info)
{
        /*
         * We implement the search by looking for an rbtree node that
         * immediately follows a suitable gap. That is,
         * - gap_start = vma->vm_prev->vm_end <= info->high_limit - length;
         * - gap_end   = vma->vm_start        >= info->low_limit  + length;
         * - gap_end - gap_start >= length
         */

```

```
11          struct mm_struct *mm = current->mm;
12          struct vm_area_struct *vma;
13          unsigned long length, low_limit, high_limit, gap_start, gap_end;
14
15          /* Adjust search length to account for worst case alignment over
    head */
16          length = info->length + info->align_mask;
17          if (length < info->length)
18                  return -ENOMEM;
19
20          /* Adjust search limits by the desired length */
21          if (info->high_limit < length)
22                  return -ENOMEM;
23          high_limit = info->high_limit - length;
24
25          if (info->low_limit > high_limit)
26                  return -ENOMEM;
27          low_limit = info->low_limit + length;
28
29          /* Check if rbtree root looks promising */
30          if (RB_EMPTY_ROOT(&mm->mm_rb))
31                  goto check_highest;
32          vma = rb_entry(mm->mm_rb.rb_node, struct vm_area_struct, vm_rb);
33          if (vma->rb_subtree_gap < length)
34                  goto check_highest;
35
36          while (true) {
37                  /* Visit left subtree if it looks promising */
38                  gap_end = vm_start_gap(vma);
39                  if (gap_end >= low_limit && vma->vm_rb.rb_left) {
40                          struct vm_area_struct *left =
41                                  rb_entry(vma->vm_rb.rb_left,
42                                           struct vm_area_struct, vm_rb);
43                          if (left->rb_subtree_gap >= length) {
44                                  vma = left;
45                                  continue;
46                          }
47                  }
```

A function implemented using augmented rbtrees searches the request range in user space from bottom to top to find the virtual start address of the unmapped gap between VMA regions where the request length can fit.

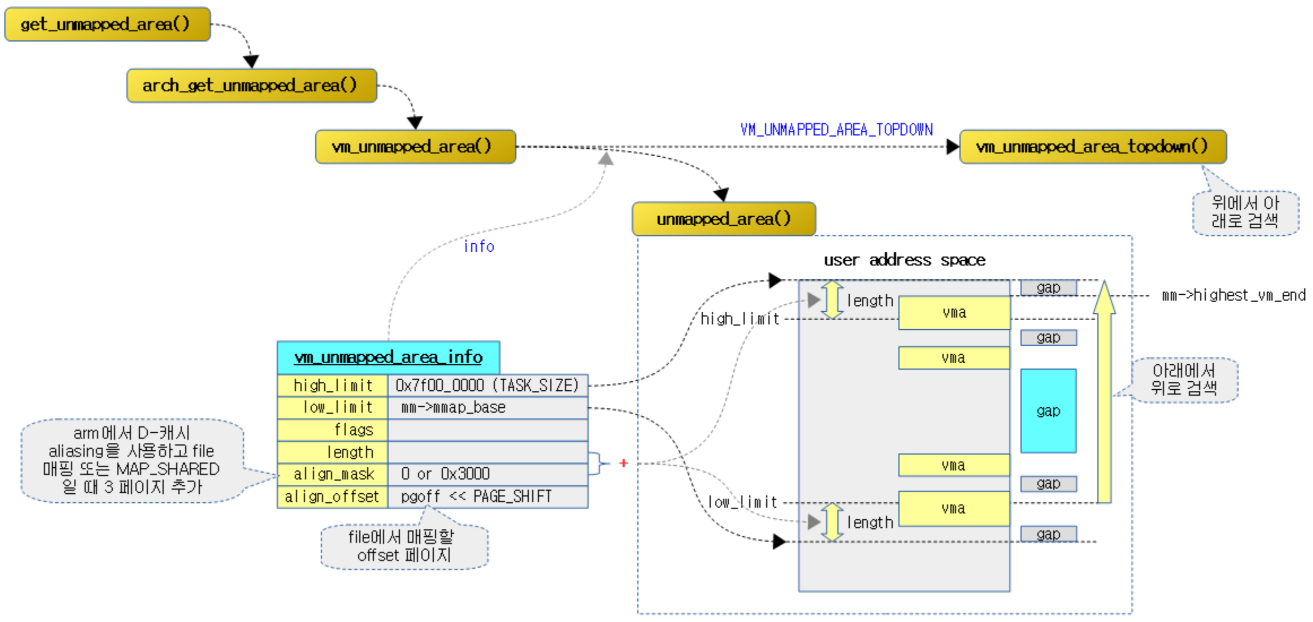
- Retrieve the length of the code lines 16~18 with the addition of align_mask, and return a -ENOMEM error if the length exceeds the system address area.
- Calculate the high_limit from code lines 21~23. If it goes out of range, it returns a -ENOMEM error.
  - Existing high_limit minus by length
- Calculate the low_limit from code lines 25~27. If it goes out of range, it returns a -ENOMEM error.
  - Values added by length from existing low_limit
- In line 30~31 of the code, if the rb tree of the memory descriptor is empty, go to the check_highest label.
- In lines 32~34 of code, get the VMA from the root node of the memory descriptor. If the rb_subtree_gap of the root node is less than its length, the nodes below the root node have no empty space, so go to the check_highest label to check the top-level space.
  - If the rb_subtree_gap of the root node is less than length, that is, the gap between each VMA is smaller than length and cannot be added in between, so the top-level area is examined.

- In lines 36~47 of the code, loop through the VMAs in the virtual memory area of the current task and find the node where a length of gap space is found in the direction of the left node in the RB tree.

mm/mmap.c -2/2-

```
01                 gap_start = vma->vm_prev ? vm_end_gap(vma->vm_prev) : 0;
02 check_current:
03                 /* Check if current node has a suitable gap */
04                 if (gap_start > high_limit)
05                         return -ENOMEM;
06                 if (gap_end >= low_limit &&
07                     gap_end > gap_start && gap_end - gap_start >= length)
08                         goto found;
09 
10                 /* Visit right subtree if it looks promising */
11                 if (vma->vm_rb.rb_right) {
12                         struct vm_area_struct *right =
13                                 rb_entry(vma->vm_rb.rb_right,
14                                          struct vm_area_struct, vm_rb);
15                         if (right->rb_subtree_gap >= length) {
16                                 vma = right;
17                                 continue;
18                         }
19                 }
20 
21                 /* Go back up the rbtree to find next candidate node */
22                 while (true) {
23                         struct rb_node *prev = &vma->vm_rb;
24                         if (!rb_parent(prev))
25                                 goto check_highest;
26                         vma = rb_entry(rb_parent(prev),
27                                        struct vm_area_struct, vm_rb);
28                         if (prev == vma->vm_rb.rb_left) {
29                                 gap_start = vm_end_gap(vma->vm_prev);
30                                 gap_end = vm_start_gap(vma);
31                                 goto check_current;
32                         }
33                 }
34         }
35 
36 check_highest:
37         /* Check highest gap, which does not precede any rbtree node */
38         gap_start = mm->highest_vm_end;
39         gap_end = ULONG_MAX;  /* Only for VM_BUG_ON below */
40         if (gap_start > high_limit)
41                 return -ENOMEM;
42 
43 found:
44         /* We found a suitable gap. Clip it with the original low_limit. */
45         if (gap_start < info->low_limit)
46                 gap_start = info->low_limit;
47 
48         /* Adjust gap address to the desired alignment */
49         gap_start += (info->align_offset - gap_start) & info->align_mask;
50 
51         VM_BUG_ON(gap_start + info->length > info->high_limit);
52         VM_BUG_ON(gap_start + info->length > gap_end);
53         return gap_start;
54 }
```

- Yield the gap start address on line 1 of the code.
- In code lines 2~5, check_current: Labels. If the gap start address is outside the high_limit, it returns a -ENOMEM error.
- In code lines 6~8, if the gap_end is more than low_limit and the gap area is larger than the length, move to the found label.
- In code lines 11~19, if there is a lower right node of the node and the gap area of that node is larger than the length, the right node is substituted into the VMA area and the loop continues.
- If there is no suitable gap area below the line 22~33, it will go back to the parent node, but if it is from the left node to the upper node, it will update the gap start and end and move to the check_current label.
- In code lines 36~41, check_highest: Label. In this case, there is no space needed for an empty area in the rb tree, so it looks for an empty space above it, and if there is no suitable space here, it returns a -ENOMEM error.
- In lines 43~46 of code, it is the found: label. Adjust the starting position of the found gap to at least info->low_limit.
- In code lines 49~53, the starting position of the gap is adjusted by adding the adjusted gap offset, and then cutting it with info->align_mask and returning.
    - File and cache aliasing can adjust the starting position of the found unmapped gap.



vm_start_gap()

include/linux/mm.h

```
static inline unsigned long vm_start_gap(struct vm_area_struct *vma)
{
        unsigned long vm_start = vma->vm_start;

        if (vma->vm_flags & VM_GROWSDOWN) {
                vm_start -= stack_guard_gap;
                if (vm_start > vma->vm_start)
                        vm_start = 0;
        }
        return vm_start;
}
```

Returns the VMA start address with the stack guard gap applied.

vm_end_gap()

include/linux/mm.h

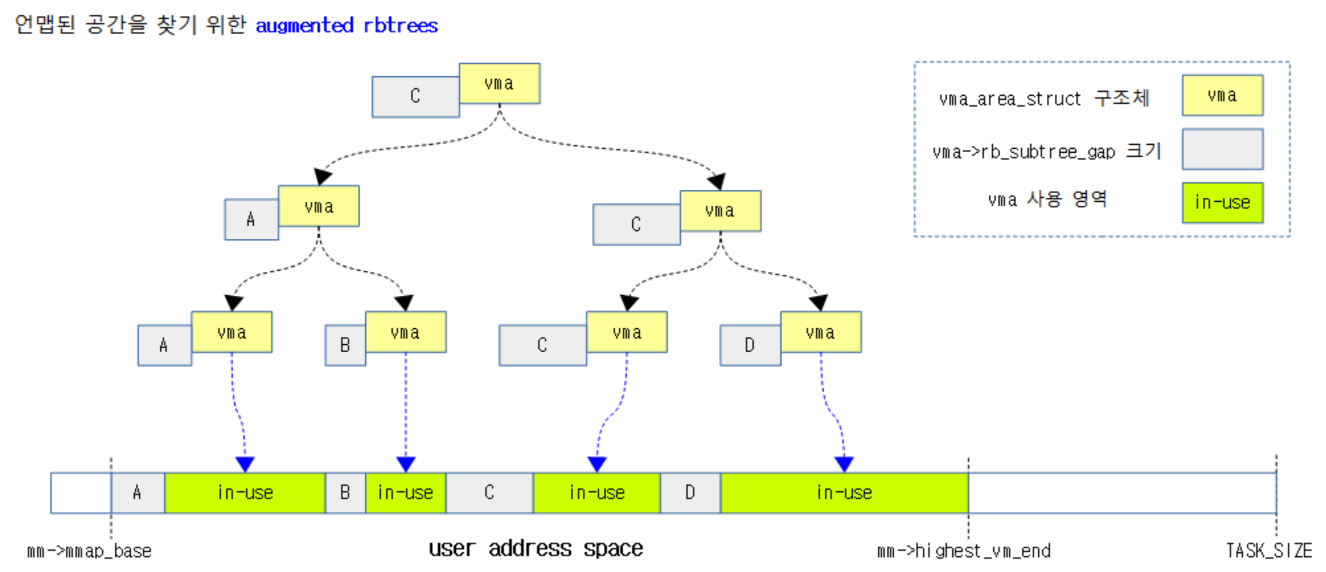
```
static inline unsigned long vm_end_gap(struct vm_area_struct *vma)
{
        unsigned long vm_end = vma->vm_end;

        if (vma->vm_flags & VM_GROWSUP) {
                vm_end += stack_guard_gap;
                if (vm_end < vma->vm_end)
                        vm_end = -PAGE_SIZE;
        }
        return vm_end;
}
```

Returns the address of the end of the VMA with the stack guard gap applied.

Augmented rbtrees

You can extend the red black tree to use the augmented rbtree, which can store additional information about the nodes. If the callback functions for propagation, copy, and rotation are written and registered by the user, they are called every time a change occurs on the node, enabling additional information management. In the kernel, VMA areas are also managed by RBRs, and the Augmented RBRons method is introduced to manage additional information about the gap areas managed by subtrees to be used to find empty areas that are not mapped.

The figure below shows how each node selects and manages the largest gap information from its subordinate nodes.



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

Limit the maximum number of VM_LOCKED pages

mlock_future_check()

mm/mmap.c

```
static inline int mlock_future_check(struct mm_struct *mm,
                                     unsigned long flags,
                                     unsigned long len)
{
        unsigned long locked, lock_limit;

        /*  mlock MCL_FUTURE? */
        if (flags & VM_LOCKED) {
                locked = len >> PAGE_SHIFT;
                locked += mm->locked_vm;
                lock_limit = rlimit(RLIMIT_MEMLOCK);
                lock_limit >>= PAGE_SHIFT;
                if (locked > lock_limit && !capable(CAP_IPC_LOCK))
                        return -EAGAIN;
        }
        return 0;
}
```

If the number of pages that use the VM_LOCKED flag in the current task exceeds the maximum locked page limit, it returns a -EAGAIN error. Returns 0 for all others.

- VM_LOCKED mapping areas are skipped by the Compaction and Reclaim system.

consultation

- User virtual maps (mmap) (http://jake.dothome.co.kr/user-virtual-maps-mmap2) | 문c
- LSM (Linux Security Module) -1- (http://jake.dothome.co.kr/lsm) | 문c
- LIM (Linux Integrity Module) -1- (http://jake.dothome.co.kr/lim) | 문c
- Understanding glibc malloc (https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/) | sploitfun
- Get User Page | 문c
- Mlock | Qc

LEAVE A COMMENT

Your email will not be published. Required fields are marked with *

Comments

name *

email *

Website

WRITE A COMMENT

Early ioremap (http://jake.dothome.co.kr/early-ioremap/)

LSM(Linux Security Module) -1- (http://jake.dothome.co.kr/lsm/)