VE482 - Lab 8

Introduction to Operating Systems

Weili Shi 519370910011

Kexuan Huang 518370910126

November 27, 2021

2. Memory in Minix 3

2.1 Memory management at kernel level

What does vm stands for?

Virtual Memory

Find the page table definition and search what fields each entry contain?

Reference from `minix/minix/servers/vm/pt.h`

```
/* A pagetable. */
typedef struct {
    /* Directory entries in VM addr space - root of page table.
       */
    u32_t *pt_dir;       /* page aligned (ARCH_VM_DIR_ENTRIES) */
    u32_t pt_dir_phys;   /* physical address of pt_dir */

    /* Pointers to page tables in VM address space. */
    u32_t *pt_pt[ARCH_VM_DIR_ENTRIES];

    /* When looking for a hole in virtual address space, start
     * looking here. This is in linear addresses, i.e.,
     * not as the process sees it but the position in the page
     * page table. This is just a hint.
     */
    u32_t pt_virtop;
} pt_t;
```

How is memory allocated within the kernel?[1] Why are not `malloc` and `calloc` used?

Linux provides a variety of APIs for memory allocation. You can allocate small chunks using `kmalloc` or `kmem_cache_alloc` families, large virtually contiguous areas using `vmalloc` and its derivatives, or you can directly request pages from the page allocator with `alloc_pages`.

We can't use user libraries in the kernel, and Linux does not define a `malloc` or `calloc`, so we can't call it.

[1] The Linux Kernel documentation

What basic functions are used to handle virtual memory?

Reference from `minix/minix/servers/vm/pagetable.c`

```c
void pt_sanitycheck(pt_t *pt, const char *file, int line);
static u32_t findhole(int pages);
void vm_freepages(vir_bytes vir, int pages);
static void *vm_getsparepage(phys_bytes *phys);
static void *vm_getsparepagedir(phys_bytes *phys);
void *vm_allocpages(phys_bytes *phys, int reason, int pages);
void vm_pagelock(void *vir, int lockflag);
int vm_addrok(void *vir, int writeflag);
static int pt_ptalloc(pt_t *pt, int pde, u32_t flags);
int pt_ptalloc_in_range(pt_t *pt, vir_bytes start, vir_bytes end,
    u32_t flags, int verify);
int pt_map_in_range(struct vmproc *src_vmp, struct vmproc *dst_vmp
    , vir_bytes start, vir_bytes end)
int pt_ptmap(struct vmproc *src_vmp, struct vmproc *dst_vmp);
int pt_writemap(struct vmproc * vmp,
            pt_t *pt,
            vir_bytes v,
            phys_bytes physaddr,
            size_t bytes,
            u32_t flags,
            u32_t writemapflags);
int pt_checkrange(pt_t *pt, vir_bytes v, size_t bytes, int write);
int pt_new(pt_t *pt);
void pt_init(void);
int pt_bind(pt_t *pt, struct vmproc *who);
void pt_free(pt_t *pt);
int pt_mapkernel(pt_t *pt);
```

Find all the places where the vm used inside the kernel, Why does it appear in so many different places?

With command `find minix/ -name "*.c"| xargs grep -l "vm"> vm.log`, we can find it in 1625 files.

For detailed list, please refer to `vm.log`

Virtual memory is widely used, which is a feature of an operating system that enables a computer to be able to compensate shortages of physical memory by transferring pages of data from random access memory to disk storage.

While allocating memory, how does the functions in kernel space switch back and forth between user and kernel spaces?[2] How is that boundary crossed? How good or bad it is to put vm in userspace?

To allocate memory, a typical flow of control is:

1. Ask the memory management unit for a block of memory
2. Map amd write the block onto the page table
3. Return the pointer to the userspace for accessing the memory block.

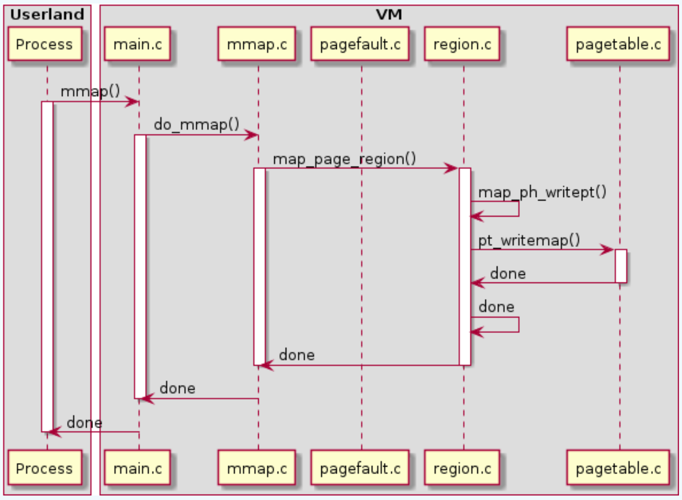

Figure 1: Flow of Allocating Memory

With `mmap()`, the boundary is crossed.

It depends. Putting `vm` in kernel space serves to provide memory protection and hardware protection from malicious or errant software behaviour, but crossing the boundary between usersapce and kernel space is time and resource consuming. Putting `vm` in userspace does speed up for some real-time or embedded system, but the protection is lost.

[2] Minix 3 Wiki

In **minix**, the boundary is crossed several times:

- from kernel to vm: `kernel/arch/earm/memory.c`

```
int vm_memset(...) {
    ...
    void vm_suspend();
    ...
    send_sig(VM_PROC_NR, SIGKMEM);
}
```

A signal "SIGKMEM" sent to VM (specified by the process number of VM)

- from vm to kernel:

1. `servers/vm/main.c`

```
... sef_cb_signal_handler(...) {
    ...
    case : SIGKMEM
        ...
        do_memory();
}
```

2. `servers/vm/pagefaults.c`, requests for memreq from kernel

```
... do_memory() {
    sys_vmctl_get_memreq(...);
}
...
// where the mem_req is handled
message m;
int r = _kernel_call(SYS_VMCTL &m);
```

- from kernel to vm: in `kernel/system/do_vmctl.c`

```
/*
 * m_type: SYS_VMCTL
 */
...
case: VMCTL_MEMREQ_GET
    ...
// then we return to do_memory(), which is in vm
```

- from vm to kernel: `servers/vm/pagefaults.c`

```
if(sys_vmctl(requestor, VMCTL_MEMREQ_REPLY, r) != OK)
            panic("do_memory: sys_vmctl failed: %d", r);
```

  - The VMCTL_MEMREQ_REPLY sent to kernel, telling him/her that things are OK.

How are pagefaults handled?[3]

A page fault occurs when a process accesses a virtual page for which there is no PTE in the page table or whose PTE in some way prohibits the access, e.g., because the page is not present or because the access is in conflict with the access rights of the page. Page faults are triggered by the CPU and handled in the `page_fault_handler`.

Because Linux uses demand paging and page-fault-based optimizations such as copy-on-write, page faults occur during the normal course of operation and do not necessarily indicate an error. Thus, when the page fault handler is invoked, it first needs to determine whether the page fault is caused by an access to a valid page. If not, the page fault handler simply sends a segmentation violation signal to the faulting process and returns. Otherwise, it takes one of several possible actions:

- If the page is being accessed for the first time, the handler allocates a new page frame and initializes it, e.g., by reading its content from disk. Page faults caused by first-time accesses are called demand_page_faults.
- If the page has been paged out to swap space, the handler reads it back from disk into a newly allocated page frame.
- If the page fault occurred because of a page-fault-based optimization (such as copy-on-write), the handler takes the appropriate recovery action (such as performing the delayed page copy).

[3] Virtual Memory in the IA-64 Linux Kernel

In **minix**, the implementation is as follows:

- Kernel sends the pagefault message in `/kernel/arch/earm/exception.c`

```
m_pagefault.m_type    = VM_PAGEFAULT;
mini_send(pr, VM_PROC_NR, &m_pagefault, FROM_KERNEL)
```

- VM receives the message in `/servers/vm/main.c`

```
else if (msg.m_type == VM_PAGEFAULT)
    do_pagefaults(&msg);
```

- inside `do_pagefaults`

```
void do_pagefaults(message *m) {
    /* See if address is valid at all. */
    /* If process was writing, see if it's writable. */
    /* Access is allowed; handle it. */
    /* Pagefault is handled, so now reactivate the process.
       */
    if((s=sys_vmctl(ep, VMCTL_CLEAR_PAGEFAULT, 0 /*unused*/))
        != OK)
      panic("do_pagefaults: sys_vmctl failed: %d", ep);
}
```

- `sys_vmctl` notifies the kernel that the page fault has been cleared

```
...
message m;
m.SVMCTL_PARAM = VMCTL_CLEAR_PAGEFAULT;
_kernel_call(SYS_VMCTL, &m);
...
```

2.2 MRU and LRU

What algorithm is used by default in Minix 3 to handle pagefault? Find its implementation and study it closely.

Minix uses LRU to handle pagefaults.

Use the `top` command to keep track of your used memory and cache, then run time `grep -r "mum" /usr/src`. Run the command again. What do you notice?

- first trial: 3.76 real 0.65 user 1.95 sys

```
load averages:  0.20, 0.00, 0.00
44 processes: 1 running, 43 sleeping
main memory: 2095548K total, 2022356K free, 1936540K contig free, 6256K cached
CPU states:   0.15% user,   0.50% system,   0.30% kernel,  99.05% idle
CPU time displayed ('t' to cycle): user ;  sort order ('o' to cycle): cpu

  PID USERNAME PRI NICE    SIZE STATE   TIME     CPU COMMAND
   -1 root       0        2613K         0:00   0.30% kernel
    7 root       5    0   1052K         0:00   0.21% vfs
   12 root       2    0   5104K         0:01   0.13% vm
   27 root       7    0    836K   RUN   0:00   0.08% procfs
   89 service    7    0    120K         0:00   0.05% random
   62 root       7    0    192K         0:00   0.04% devman
  546 root       7    0    580K         0:00   0.04% top
    5 root       4    0    248K         0:00   0.03% pm
   83 root       7    0    220K         0:00   0.03% devmand
   36 service    5    0   6900K         0:00   0.03% mfs
   10 root       1    0    228K         0:00   0.02% tty
    4 root       4    0   1372K         0:00   0.00% rs
    6 root       4    0     52K         0:00   0.00% sched
    8 root       3    0    104K         0:00   0.00% memory
    9 root       2    0    144K         0:00   0.00% log
    3 root       4    0    164K         0:00   0.00% ds
```

- second trial: 1.41 real 0.41 user 0.88 sys

```
load averages:  0.60, 0.01, 0.01
44 processes: 1 running, 43 sleeping
main memory: 2095548K total, 1847660K free, 1761836K contig free, 179212K cached
CPU states:   0.10% user,   0.31% system,   0.20% kernel,  99.39% idle
CPU time displayed ('t' to cycle): user ;  sort order ('o' to cycle): cpu

  PID USERNAME PRI NICE    SIZE STATE   TIME     CPU COMMAND
   -1 root       0        2613K         0:00   0.20% kernel
    7 root       5    0   1052K         0:01   0.13% vfs
   12 root       2    0   6844K         0:01   0.10% vm
   89 service    7    0    120K         0:00   0.04% random
   62 root       7    0    192K         0:00   0.03% devman
   27 root       7    0    836K   RUN   0:00   0.02% procfs
   36 service    5    0   6900K         0:00   0.02% mfs
   83 root       7    0    220K         0:00   0.02% devmand
   10 root       1    0    228K         0:00   0.02% tty
    5 root       4    0    248K         0:00   0.02% pm
  549 root       7    0    580K         0:00   0.01% top
    4 root       4    0   1372K         0:00   0.00% rs
   42 root       7    0   1060K         0:00   0.00% is
  104 root       7    0    112K         0:00   0.00% lance
    8 root       3    0    104K         0:00   0.00% memory
    9 root       2    0    144K         0:00   0.00% log
```

We observe that the free memory has decreased, the cached memory has increased.

Adjust the implementation of LRU into MRU and recompile the kernel.

In the file `/servers/vm/region.c`, we change the `lru_oldest` to `lru_youngest`, then recompile the kernel by:

```
su
cd /usr/src
make build
reboot
```

The modified code is shown below:

```
/*===========================================================================*
 *                              free_yielded                                 *
 *===========================================================================*/
vir_bytes free_yielded(vir_bytes max_bytes)
{

/* PRIVATE yielded_t *lru_youngest = NULL, *lru_oldest = NULL; */
        vir_bytes freed = 0;
        int blocks = 0;

        while(freed < max_bytes && lru_youngest) {
                SLABSANE(lru_youngest);
                freed += freeyieldednode(lru_youngest, 1);
                blocks++;
        }

        return freed;
}

/*===========================================================================*
 *                              map_free_proc                                *
 *===========================================================================*/
int map_free_proc(vmp)
```

Use the `top` command to keep track of your used memory and cache, then run time `grep -r "mum" /usr/src`. Run the command again. What do you notice?

- first trial: 5.28 real 1.62 user 3.66 sys

```
load averages:  0.30, 0.00, 0.00
42 processes: 1 running, 41 sleeping
main memory: 2095548K total, 2061284K free, 2045660K contig free, 0K cached
CPU states:   0.12% user,   0.43% system,   0.25% kernel,  99.20% idle
CPU time displayed ('t' to cycle): user ;  sort order ('o' to cycle): cpu

  PID USERNAME PRI NICE    SIZE STATE   TIME     CPU COMMAND
   -1 root       0        2613K         0:00   0.25% kernel
    7 root       5    0   1048K         0:00   0.18% vfs
   12 root       2    0   3700K         0:00   0.10% vm
   27 root       7    0    836K   RUN   0:00   0.08% procfs
  145 root       7    0    564K         0:00   0.04% top
   89 service    7    0    120K         0:00   0.03% random
   10 root       1    0    228K         0:00   0.03% tty
   62 root       7    0    192K         0:00   0.02% devman
    5 root       4    0    248K         0:00   0.02% pm
   36 service    5    0   5288K         0:00   0.02% mfs
   83 root       7    0    208K         0:00   0.02% devmand
    4 root       4    0   1356K         0:00   0.00% rs
  130 root       7    0    340K         0:00   0.00% nonamed
  108 service    7    0   1052K         0:00   0.00% inet
    6 root       4    0     52K         0:00   0.00% sched
    8 root       3    0    104K         0:00   0.00% memory
_
```

- second trial: 2.15 real 0.71 user 1.40 sys

```
load averages:  0.70, 0.02, 0.00
42 processes: 1 running, 41 sleeping
main memory: 2095548K total, 1736484K free, 1735180K contig free, 292248K cached
CPU states:   0.15% user,   0.94% system,   0.33% kernel,  98.58% idle
CPU time displayed ('t' to cycle): user ;  sort order ('o' to cycle): cpu

  PID USERNAME PRI NICE    SIZE STATE   TIME     CPU COMMAND
   12 root       2    0   7392K         0:00   0.41% vm
   -1 root       0        2613K         0:00   0.33% kernel
    7 root       5    0   1048K         0:00   0.26% vfs
   27 root       7    0    836K   RUN   0:00   0.11% procfs
   10 root       1    0    228K         0:00   0.10% tty
  148 root       7    0    564K         0:00   0.07% top
   36 service    5    0   5288K         0:00   0.04% mfs
   89 service    7    0    120K         0:00   0.04% random
    5 root       4    0    248K         0:00   0.03% pm
   62 root       7    0    192K         0:00   0.03% devman
   83 root       7    0    208K         0:00   0.02% devmand
    4 root       4    0   1356K         0:00   0.00% rs
  128 root       7    0    352K         0:00   0.00% dhcpd
  104 root       7    0    112K         0:00   0.00% lance
   42 root       7    0   1060K         0:00   0.00% is
    8 root       3    0    104K         0:00   0.00% memory
```

We observe that the free memory has decreased and the cached memory has increased. Also, for `MRU`, the cached memory seems to be bigger. Compared to the `LRU` implementation, the overall runtime has been slower.

Discuss the different behaviours of LRU and MRU as well as the consequences for the users. Can you think of any situation where MRU would be better than LRU?

In terms of page replacement, `LRU` replaces the least recently used pages, while `MRU` replaces the most recently used pages. Due to time and spatial locality, the user may use some pages frequently but rarely others. The `MRU` will make access to those frequently or recently used pages slower. It is likely that `MRU` performs better, if we keep asking for the least frequently used pages.