内存管理

陈海波 / 夏虞斌

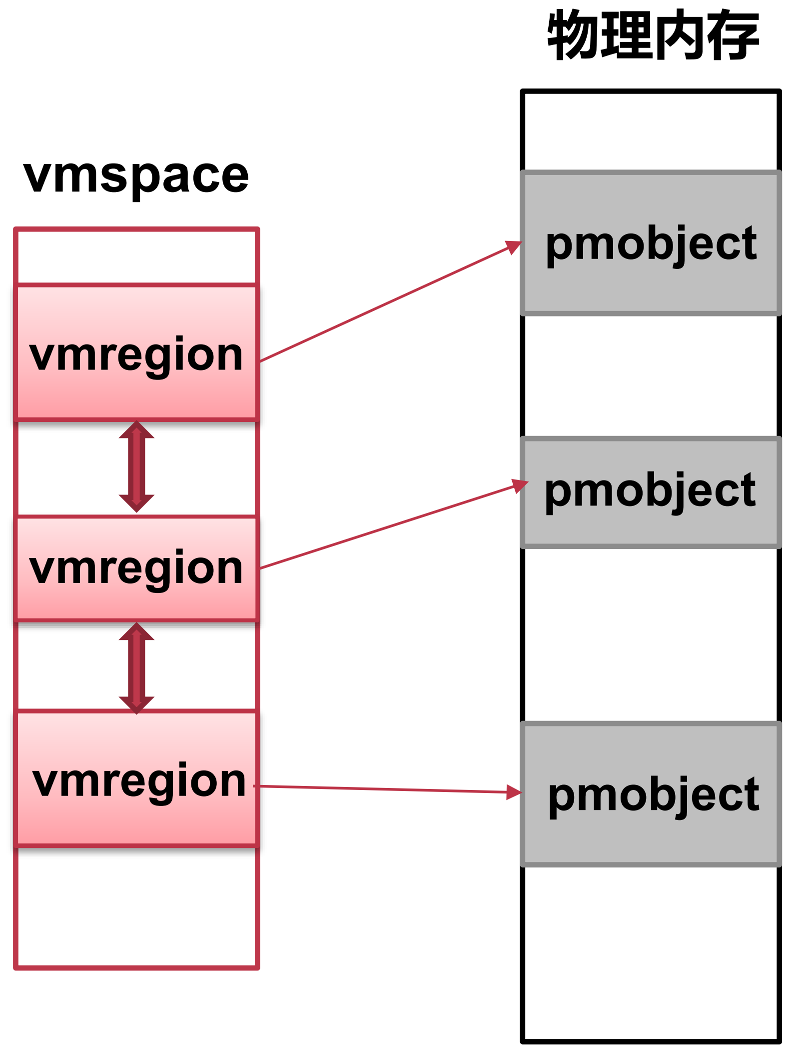
上海交通大学并行与分布式系统研究所

https://ipads.se.sjtu.edu.cn

版权声明

核心数据结构

- 虚拟地址空间vmspace
- 虚拟地址区域vmregion
- 物理内存对象pmobject



进程虚拟地址空间 vmspace （图）

- **vmr_list**
  - vmregion链表头
  - vmspace包含多个vmregion
- **pgtbl**
  - 页表基地址 （虚拟地址）
- **locks**
  - 多线程并发控制

```c
struct vmspace {
    /* List head of vmregion (vmr_list) */
    struct list_head vmr_list;
    /* Root page table */
    vaddr_t *pgtbl;

    /* The lock for manipulating vmregions */
    struct lock vmspace_lock;
    /* The lock for manipulating the page table */
    struct lock pgtbl_lock;

    /*
     * For TLB flushing:
     * Record the all the CPU that a vmspace ran on.
     */
    #ifdef CHCORE
    u8 history_cpus[PLAT_CPU_NUM];
    #endif

    /* Heap-related: only used for user processes */
    struct lock heap_lock;
    struct vmregion *heap_vmr;
    vaddr_t user_current_heap;

    /* For the virtual address of mmap */
    vaddr_t user_current_mmap_addr;
};
```

进程虚拟地址区域vmregion

- node：连接前后vmregion
- start：区域起始位置
- size：区域大小
- perm：区域读写执行权限
- pmo：对应的物理内存

```c
struct vmregion {
	struct list_head node; /* vmr_list */
	vaddr_t start;
	size_t size;
	vmr_prop_t perm;
	struct pmobject *pmo;
};
```

物理内存对象pmobject

- **物理内存资源以对象的形式进行分配**
  - 映射给vmregion

```
typedef u64 pmo_type_t;
#define PMO_ANONYM      0 /* lazy allocation */
#define PMO_DATA        1 /* immediate allocation */
#define PMO_FILE        2 /* file backed */
#define PMO_SHM         3 /* shared memory */
#define PMO_USER_PAGER 4 /* support user pager */
#define PMO_DEVICE      5 /* memory mapped device registers */

struct pmobject {
    struct radix *radix; /* record physical pages */
    paddr_t start;
    size_t size;
    pmo_type_t type;
    atomic_cnt refcnt; /* free this pmo when refcnt is 0 */
};
```

物理内存对象pmobject

- **每个类型的作用不同**
  - PMO_ANONYM：按需分配，一般用于堆的分配
  - PMO_DATA：立即分配，比如用于加载代码和数据
  - PMO_FILE：访问文件，支持POSIX标准中的mmap
  - PMO_SHM：共享内存
  - PMO_USER_PAGER：支持用户态虚拟内存管理
  - PMO_DEVICE：访问设备，如MMIO和DMA

进程创建流程

- **首先创建虚拟地址空间vmspace**
- **然后创建主线程**
  - 分配栈空间 （栈的vmregion和相应的pmoject）
  - 加载代码和数据…

示例：为用户进程创建地址空间

- 创建并且初始化vmspace对象

```
    vmspace = obj_alloc(TYPE_VMSPACE, sizeof(*vmspace));
    if (!vmspace) {
        r = -ENOMEM;
        goto out_free_obj_vmspace;
    }
    vmspace_init(vmspace);
```

初始化vmspace

- 初始化vmregion的链表
- 分配顶层页表页
- 体系结构相关的初始化
  - 如aarch64上可设置ASID
- 其它域的初始化

```c
int vmspace_init(struct vmspace *vmspace)
{
        init_list_head(&vmspace->vmr_list);
        /* Allocate the root page table page */
        vmspace->pgtbl = get_pages(0);
        BUG_ON(vmspace->pgtbl == NULL);
        memset((void*)vmspace->pgtbl, 0, PAGE_SIZE);

        /* Architecture-dependent initilization */
        arch_vmspace_init(vmspace);

        /*
         * Note: acquire vmspace_lock before pgtbl_lock
         * when locking them together.
         */
        lock_init(&vmspace->vmspace_lock);
        lock_init(&vmspace->pgtbl_lock);

        vmspace->user_current_heap = HEAP_START;
        lock_init(&vmspace->heap_lock);

        /* The vmspace does not run on any CPU for now */
        reset_history_cpus(vmspace);

        /* Set the mmap area: this variable is protected
        vmspace->user_current_mmap_addr = MMAP_START;

        return 0;
}
```

示例：为第一个用户线程创建栈

- 分配并初始化物理内存对象，然后映射到地址空间

```
    init_vmspace = obj_get(cap_group, VMSPACE_OBJ_ID, TYPE_VMSPACE);
    obj_put(init_vmspace);

    /* Allocate and setup a user stack for the init thread */
    stack_pmo = obj_alloc(TYPE_PMO, sizeof(*stack_pmo));
    if (!stack_pmo) {
        ret = -ENOMEM;
        goto out_fail;
    }
    pmo_init(stack_pmo, PMO_DATA, stack_size, 0);
    stack_pmo_cap = cap_alloc(cap_group, stack_pmo, 0);
    if (stack_pmo_cap < 0) {
        ret = stack_pmo_cap;
        goto out_free_obj_pmo;
    }
    ret = vmspace_map_range(init_vmspace, stack_base, stack_size,
                            VMR_READ | VMR_WRITE, stack_pmo);
    BUG_ON(ret != 0);
```

初始化pmobject

- **设置类型和大小**
- **根据类型执行具体初始化：**
  - 对于PMO_DATA类型，直接分配物理内存
  - 对于PMO_ANONYM和PMO_SHM类型，不会立即分配物理内存，而只是创建radix_tree。将来按需分配的物理内存将被记录在这个树中。

```
/*
 * Initialize an allocated pmobject.
 * @paddr is only used when @type == PMO_DEVICE.
 */
void pmo_init(struct pmobject *pmo, pmo_type_t type,
              size_t len, paddr_t paddr)
{
    memset((void*)pmo, 0, sizeof(*pmo));

    len = ROUND_UP(len, PAGE_SIZE);
    pmo->size = len;
    pmo->type = type;

    switch (type) {
    case PMO_DATA: {
        /*
         * For PMO_DATA, the user will use it soon (we expect).
         * So, we directly allocate the physical memory.
         * Note that kmalloc(>2048) returns continous physical pages.
         */
        pmo->start = (paddr_t)virt_to_phys(kmalloc(len));
        break;
    }
    case PMO_ANONYM:
    case PMO_SHM: {
        /*
         * For PMO_ANONYM (e.g., stack and heap) or PMO_SHM,
         * we do not allocate the physical memory at once.
         */
        pmo->radix = new_radix();
        init_radix(pmo->radix);
        break;
    }
```

将pmobject映射到地址空间中

- **vmspace_map_range**
  1. 分配vmregion
  2. 初始化vmregion
  3. 将vmregion加入vmspace

```c
int vmspace_map_range(struct vmspace *vmspace, vaddr_t va, size_t len,
                      vmr_prop_t flags, struct pmobject *pmo)
{
    struct vmregion *vmr;
    int ret;

    /* skip other code */

    vmr = alloc_vmregion();
    if (!vmr) {
        ret = -ENOMEM;
        goto out_fail;
    }
    vmr->start = va;
    vmr->size = len;
    vmr->perm = flags;
    if (unlikely(pmo->type == PMO_DEVICE))
        vmr->perm |= VMR_DEVICE;

    /* Currently, one vmr has exactly one pmo */
    vmr->pmo = pmo;

    /*
     * Note that each operation on the vmspace should be protected by
     * the per_vmspace lock, i.e., vmspace_lock
     */
    lock(&vmspace->vmspace_lock);
    ret = add_vmr_to_vmspace(vmspace, vmr);
    unlock(&vmspace->vmspace_lock);
```

将vmregion加入到vmspace中

- 检查vmregion是否合法，然后加入链表

```c
/*
 * This function should be surrounded with a lock.
 *
 * Modify the list with lock protection.
 * Otherwise, concurrent operations may lead to erros.
 */
static int add_vmr_to_vmspace(struct vmspace *vmspace, struct vmregion *vmr)
{
	if (check_vmr_intersect(vmspace, vmr) != 0) {
		kwarn("Detecting: vmr overlap\n");
		BUG_ON(1);
		return -EINVAL;
	}

	list_add(&(vmr->node), &(vmspace->vmr_list));
	return 0;
}
```

Cont. 将pmobject映射到地址空间中

- **vmspace_map_range**
  1. 分配vmregion
  2. 初始化vmregion
  3. 将vmregion加入vmspace
  4. 当pmobject已经分配好物理内存了，那么就直接填写页表

```c
int vmspace_map_range(struct vmspace *vmspace, vaddr_t va, size_t len,
                      vmr_prop_t flags, struct pmobject *pmo)
{
    ... /* Omit some code */

    /*
     * Case-1:
     * If the pmo type is PMO_DATA or PMO_DEVICE, we directly add mappings
     * in the page table because the corresponding physical pages are
     * prepared. In this case, early mapping avoids page faults and brings
     * better performance.
     *
     * Case-2:
     * Otherwise (for PMO_ANONYM and PMO_SHM), we use on-demand mapping.
     * In this case, lazy mapping reduces the usage of physical memory resource.
     */
    if ((pmo->type == PMO_DATA) || (pmo->type == PMO_DEVICE))
        fill_page_table(vmspace, vmr);
```

缺页异常

- **基于缺页异常，实现按需分配**
  - 例如，访问PMO_ANONYM映射的区域会触发缺页
- **（64位）用户态程序发生缺页时下陷到内核**
  - 在x86_64上，触发 #PF （异常号13）
  - 在aarch64上，触发64位EL0同步异常 （异常号8）
    - 同时错误状态寄存器（ESR）中的错误代码（EC）是 ESR_EL1_EC_DABT_LEL 或 ESR_EL1_EC_IABT_LEL

```
#define ESR_EL1_EC_IABT_LEL         0b100000    /* Instruction Abort from a lower Exception level) */
#define ESR_EL1_EC_DABT_LEL         0b100100    /* Data Abort from a lower Exception level */
```

缺页异常处理函数

```c
static inline vaddr_t get_fault_addr()
{
    vaddr_t addr;
    asm volatile ("mrs %0, far_el1\n\t" :"=r" (addr));
    return addr;
}
```

- **获取触发缺页的地址**
  - aarch64: far_el1
  - X86_64: CR2
- **根据ESR中错误状态码具体处理**
  - 翻译错误 （无映射）
  - 权限错误

```c
void do_page_fault(u64 esr, u64 fault_ins_addr)
{
    vaddr_t fault_addr;
    int fsc; // fault status code

    fault_addr = get_fault_addr();
    fsc = GET_ESR_EL1_FSC(esr);
    switch (fsc) {
    case DFSC_TRANS_FAULT_L0:
    case DFSC_TRANS_FAULT_L1:
    case DFSC_TRANS_FAULT_L2:
    case DFSC_TRANS_FAULT_L3: {
        int ret;

        ret = handle_trans_fault(current_thread->vmspace, fault_addr);
        break;
    }
    case DFSC_PERM_FAULT_L1:
    case DFSC_PERM_FAULT_L2:
    case DFSC_PERM_FAULT_L3:
        /* Support COW later */

        kinfo("do_page_fault: faulting ip is 0x%lx,"
              "faulting address is 0x%lx,"
              "fsc is perm_fault (0b%b)\n",
              fault_ins_addr, fault_addr, fsc);
        BUG_ON(1);
        break;
```

处理按需分配导致的翻译错误

- **找到出错地址属于的vmregion**
- **拿到vmregion关联的pmo**

```c
int handle_trans_fault(struct vmspace *vmspace, vaddr_t fault_addr)
{
	struct vmregion *vmr;
	struct pmobject *pmo;
	paddr_t pa;
	u64 offset;
	u64 index;

	/*
	 * Grab lock here.
	 * Because two threads (in same process) on different cores
	 * may fault on the same page, so we need to prevent them
	 * from adding the same mapping twice.
	 */

	lock(&vmspace->vmspace_lock);
	vmr = find_vmr_for_va(vmspace, fault_addr);
	if (vmr == NULL) {
		kinfo("handle_trans_fault: no vmr find for va 0x%lx!\n",
		      fault_addr);
		kprint_vmr(vmspace);
		// TODO: kill the process
		kwarn("TODO: kill such faulting process.\n");
		return -ENOMAPPING;
	}

	pmo = vmr->pmo;
```

Cont. 处理按需分配导致的翻译错误

- 分配物理页
- 在radix_tree中记录物理页信息
- 在页表中添加映射

```
    pmo = vmr->pmo;
    switch (pmo->type) {
    case PMO_ANONYM:
    case PMO_SHM: {
        /* Simplified code */

        pa = virt_to_phys(get_pages(0));
        BUG_ON(pa == 0);
        /*
         * Record the physical page in the radix tree:
         * the offset is used as index in the radix tree
         */
        kdebug("commit: index: %ld, 0x%lx\n", index, pa);
        commit_page_to_pmo(pmo, index, pa);

        /* Add mapping in the page table */
        lock(&vmspace->pgtbl_lock);
        fault_addr = ROUND_DOWN(fault_addr, PAGE_SIZE);
        map_range_in_pgtbl(vmspace, fault_addr, pa,
                           PAGE_SIZE, vmr->perm);
        unlock(&vmspace->pgtbl_lock);
    }
```