Linux操作系统分析
Chapter 9 Linux中的进程地址空间

陈香兰（xlanchen@ustc.edu.cn）

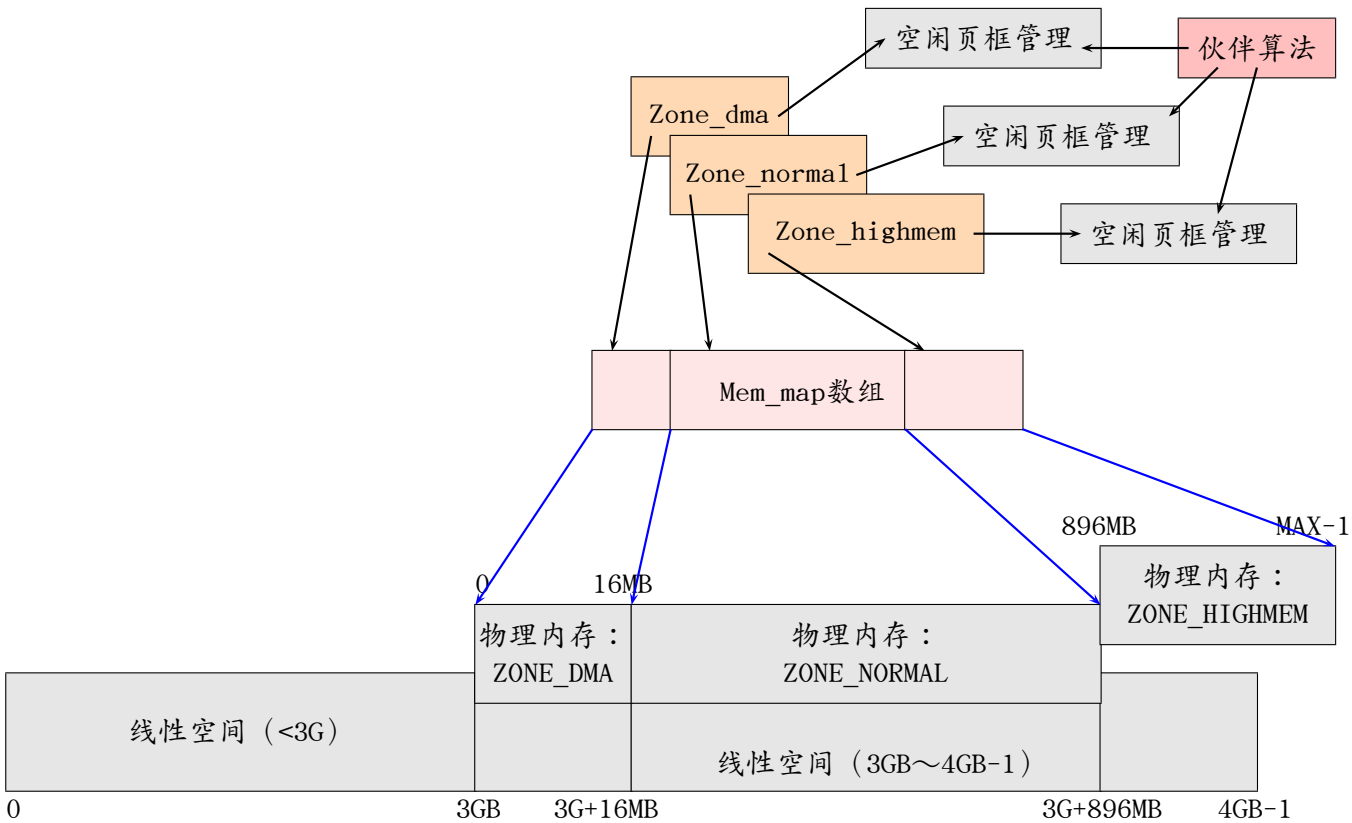
计算机应用教研室@计算机学院
嵌入式系统实验室@苏州研究院
中国科学技术大学
Fall 2014

November 4, 2014

Outline

内核态和用户态分配内存的不同

- 内核中的函数以直接了当的方式获得动态内存
  - 内核是操作系统中优先级最高的成分。
  - 内核信任自己
  - 采用我们上次课介绍的页面级内存分配和小内存分配



内核态和用户态分配内存的不同

- 给用户态进程分配内存时
  - 请求被认为是不紧迫的
  - 用户进程不可信任
- 因此，当用户态进程请求动态内存时，并没有立即获得实际的物理页框，而仅仅获得对一个新的线性地址区间的使用权
- 这个线性地址区间会成为进程地址空间的一部分，称作线性区(memory areas)

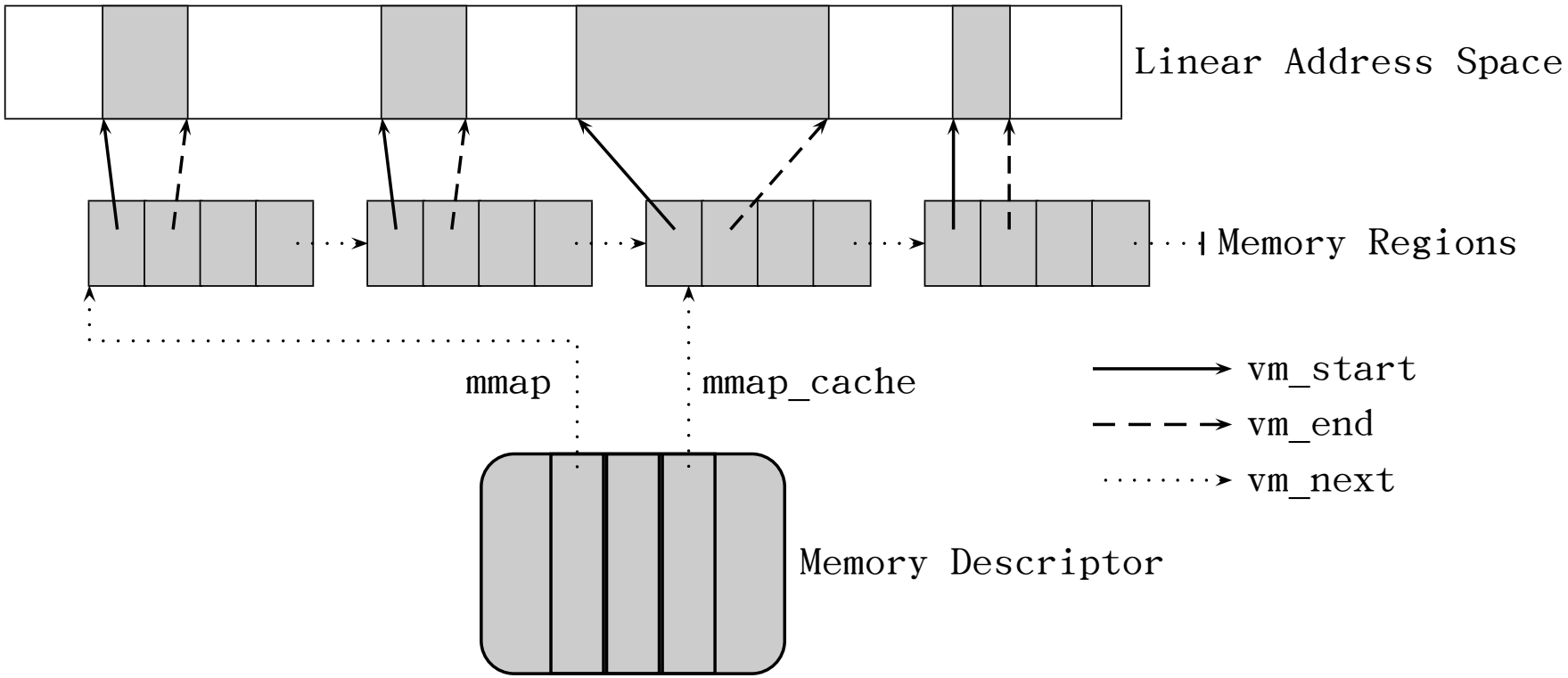
Outline

进程地址空间和线性区

- 进程最多能访问4GB的线性地址空间
- 但进程在访问某个线性空间之前，必须获得该线性空间的许可
- 因此，一个进程的地址空间是由允许该进程访问的全部线性地址组成
- 内核使用线性区资源来表示线性地址空间
- 每个线性区由起始线性地址、长度和一些存取权限描述

与进程地址空间相关的描述符

Figure 8-2. Descriptors related to the address space of a process



与进程地址空间相关的描述符

- 线性区的开始和结束都必须4KB对齐
- 进程获得新线性区的一些典型情况：
  - 刚刚创建的新进程
  - 使用exec系统调用装载一个新的程序运行
  - 将一个文件（或部分）映射到进程地址空间中
  - 当用户堆栈不够用的时候，扩展堆栈对应的线性区
  - ……

与创建、删除线性区相关的系统调用

brk( )	改变进程堆的大小
execve( )	Loads a new executable file, thus changing the process address space
_exit( )	Terminates the current process and destroys its address space
fork( )	Creates a new process, and thus a new address space
mmap( ) mmap2( )	Creates a memory mapping for a file, thus enlarging the process address space
mremap( )	扩大或缩小线性区
remap_file_pages( )	Creates a non-linear mapping for a file
munmap( )	Destroys a memory mapping for a file, thus contracting the process address space
shmat( )	Attaches a shared memory region
shmdt( )	Detaches a shared memory region

线性区（memory area）

- 比如0x08048000—0x0804C000这段线性地址空间被分配给了一个进程，进程就可以访问这段地址空间
- 进程只能访问某个有效的memory area。进一步讲，这个area可以被标志为只读或者不可执行(nonexecutable)
- 如果进程试图访问一个有效的area之外的地址或者用不正确的方式访问一个有效的area，内核将通过段异常(segmentation fault)杀死这个进程

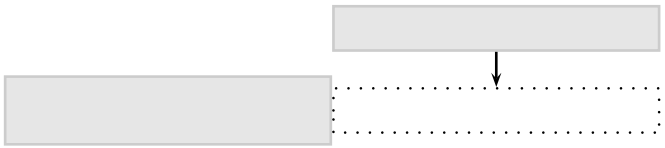
线性区（memory area）

- 线性区中可以包含各种内容
  - 可执行文件代码段的内存映射，就是.text section
  - 数据段的内存映射，.data section
  - zero page的内存映射用来包含未初始化的全局变量，.bss section
  - 为库函数和链接器附加的代码、数据、bss段
  - 文件的内存映射
  - 共享内存的映射
  - 匿名内存区域的映射，比如通过malloc()函数申请的内存区域

线性区（memory area）

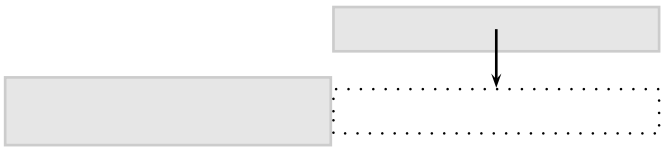
- 进程地址空间中所有有效的线性地址都确定的存在于一个area中
  - memory areas不重叠
- 进程中每个单独的area对应一个不同内存区：
  堆栈、二进制代码、全局变量、文件映射等等

增加或删除一个线性区



(a) Access rights of interval to be added are equal to those of contiguous region

(a') The exsiting region is enlarged



(b) Access rights of interval to be added are different from those of contiguous region

(b') A new memory region is created

(c) Interval to be removed is at the end of existing region

(c') The existing region is shortened

(d) Interval to be removed is inside existing region

(d') Two smaller regions are created

Adress space before operation

Address space after operation

内存描述符mm_struct

- task_struct中的内存描述符（include/linux/sched.h）

```
struct mm_struct *mm, *active_mm;
```

- 内存描述符数据结构mm_struct，参见include/linux/mm_types.h
  主要数据项的说明参见ULK3中文版354页

分配一个内存描述符

- copy_mm函数用来在fork()调用中从父进程拷贝内存描述符

$$\text{do\_fork} \rightarrow \text{copy\_process} \rightarrow \text{copy\_mm}$$

  - mm_struct数据结构本身的空间是从mm_cachep指向的slab缓存中通过allocate_mm()宏分配得到的

```
#define allocate_mm() (kmem_cache_alloc(mm_cachep, GFP_KERNEL))
#define free_mm(mm) (kmem_cache_free(mm_cachep, (mm)))
```

  - 如果父进程在fork()创建子进程时，通过一些标志指明要和子进程共享地址空间。那么，只需要

```
if (clone_flags & CLONE_VM) {
    atomic_inc(&oldmm->mm_users);
    mm = oldmm;
    goto good_mm;
}
```

释放一个内存描述符

- 在进程退出时，exit_mm() @kernel/exit.c函数被调用

$$\text{do\_exit} \rightarrow \text{exit\_mm}$$

  - 首先做一些清除工作，更新一些内核全局统计数据
  - 接着调用mmput()，这个函数减内存描述符的mm_users域
  - 如果mm_users域变成了0，就调用mmdrop()函数来减mm_count域
  - 如果mm_count域变成了0，就由free_mm()宏调用kmem_cache_free()函数把mm_struct返还给mm_cachp指向slab缓存

线性区(memory areas)

- 每个线性区由一个vm_area_struct结构来表示，参见include/linux/mm_types.h
  - 这个结构描述了一段给定的内存区间
  - 区间中的地址都有同样的属性，比如同样的存取权限和相关的操作函数
  - 用这个结构可以表示各种线性区，比如映射可执行的二进制代码的线性区、用作用户态堆栈的线性区等等

线性区的存取权限

- vm_flags域描述有关这个线性区全部页的信息。例如，进程访问每个页的权限是什么。还有一些标志描述线性区自身，例如它应该如何增长

include/linux/mm.h

```
#define VM_READ 0x00000001 /* currently active flags */
#define VM_WRITE 0x00000002
#define VM_EXEC 0x00000004
#define VM_SHARED 0x00000008
...
#define VM_GROWSDOWN 0x00000100 /* general info on the segment */
#define VM_GROWSUP 0x00000200
...
```

线性区的链表和红黑树

```
struct mm_struct {
    struct vm_area_struct * mmap; /* list of VMAs */
    struct rb_root mm_rb;
    struct vm_area_struct * mmap_cache; /* last find_vma result */
    ...
};
```

- 通过内存描述符中的两个域mmap和mm_rb都可以访问线性区。事实上，它们都指向了同一组vm_area_struct结构，只是链接的方式不同
  1. mmap指向的线性区链表用来遍历整个进程的地址空间
  2. 红黑树mm_rb用来定位一个给定的线性地址落在进程地址空间中的哪一个线性区中
- mmap_cache用来缓存最近用过的线性区

处理线性区

- 内核进程需要对一个线性区进行处理，比如确定一个给定线性地址是否存在于一个线性地址空间中
- mm/mmap.c::find_vma()，
  查找一个线性地址
  - 两个参数：进程内存描述符的地址mm和线性地址addr
- include/linux/mm.h::find_vma_intersection()，
  查找一个与给定地址区间重叠的线性区
- mm/mmap.c::get_unmapped_area()，
  查找一个空闲的地址区间
  - arch_get_unmapped_area
  - shm_get_unmapped_area
- mm/mmap.c::insert_vm_struct()，
  向内存描述符链表中插入一个线性区

处理线性区

```
/* Look up the first VMA which satisfies addr < vm_end, NULL if none. */
struct vm_area_struct * find_vma(struct mm_struct * mm, unsigned long addr) {
    struct vm_area_struct *vma = NULL;
    if (mm) {
        /* Check the cache first. */
        /* (Cache hit rate is typically around 35%.) */
        vma = mm->mmap_cache;
        if (!(vma && vma->vm_end > addr && vma->vm_start <= addr)) {
            struct rb_node * rb_node;
            rb_node = mm->mm_rb.rb_node;
            vma = NULL;
            while (rb_node) {
                struct vm_area_struct * vma_tmp;
                vma_tmp = rb_entry(rb_node, struct vm_area_struct, vm_rb);
                if (vma_tmp->vm_end > addr) {
                    vma = vma_tmp;
                    if (vma_tmp->vm_start <= addr)     break;
                    rb_node = rb_node->rb_left;
                } else     rb_node = rb_node->rb_right;
            }
            if (vma) mm->mmap_cache = vma;
        }
    }
    return vma;
}
```

创建/删除一个线性区间

mmap, munmap - map or unmap files or devices into memory

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

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

1. 创建一个线性区

   do_mmap映射file中偏移量为offset，长度为len的一段内容

   - addr参数指明从何处开始查找一段可用的空闲线性地址区间
   - Prot参数指定这个区间所包含的页的存取权限
   - flags参数指定这个创建的线性区本身的一些标志

include/linux/mm.h

```
static inline unsigned long do_mmap(struct file *file, unsigned long addr,
                                    unsigned long len, unsigned long prot,
                                    unsigned long flag, unsigned long offset)
{...}
```

创建/删除一个线性区间

2. 删除一个线性区间

   do_munmap()函数从进程地址空间中删除一段线性空间

   - mm参数指向了当前进程的内存描述符
   - addr参数为线性区的起始地址
   - len参数指明要删除的区间大小

mm/mmap.c

```
int do_munmap(struct mm_struct *mm, unsigned long start, size_t len)
{...}
```

复制线性区

- copy_mm间接调用dup_mmap()复制线性区

$$\text{copy\_mm} \rightarrow \text{dup\_mm} \rightarrow \text{dup\_mmap}$$

- 对于要复制的线性区
  1. 分配并复制线性区数据结构
  2. 复制相关页表映射

分配一个页目录

- copy_mm间接调用mm_alloc_pgd来分配一个新的页表

$$\text{copy\_mm} \rightarrow \text{dup\_mm} \rightarrow \text{mm\_init} \rightarrow \text{mm\_alloc\_pgd}$$

```
static inline int mm_alloc_pgd(struct mm_struct * mm) {
    mm->pgd = pgd_alloc(mm);
    if (unlikely(!mm->pgd)) return -ENOMEM;
    return 0;
}
```

```
pgd_t *pgd_alloc(struct mm_struct *mm) {
    pgd_t *pgd = (pgd_t *)__get_free_page(GFP_KERNEL | __GFP_ZERO);

    /* so that alloc_pmd can use it */
    mm->pgd = pgd;
    if (pgd) pgd_ctor(pgd);
    if (pgd && !pgd_prepopulate_pmd(mm, pgd)) {
        pgd_dtor(pgd);
        free_page((unsigned long)pgd);
        pgd = NULL;
    }
    return pgd;
}
```

Outline

缺页异常（i386中14号异常）

- 如前所述，内核只是通过mmap()等调用分配了一些线性地址空间给进程，并没有眞正的把实际的物理页框分配给进程
- 当进程试图访问这些分配给它的地址空间时，比如一段线性地址空间映射的是二进制代码，则进程被调度执行的时候会跳转到这个地址上去执行。
- 此时，并没有物理页框对应于这些线性地址，从而会引发一个缺页异常

缺页异常处理程序do_page_fault

- 缺页异常处理程序处理缺页异常。
  参见arch/x86/mm/fault.c
  - 它可以判断出这是不是一个合法的缺页异常，如果是，则负责给这段线性地址分配一些物理页框并把磁盘中对应的文件写入这些物理页框
  - 这样进程得以正常运行。

线性区缺页异常处理

- do_page_fault调用find_vma()找到缺页地址对应的线性区，然后根据线性区分别处理

1. 若用户栈区缺页

```
...
vma = find_vma(mm, address);
if (!vma) goto bad_area;
if (vma->vm_start <= address) goto good_area;
if (!(vma->vm_flags & VM_GROWSDOWN)) goto bad_area;
if (error_code & PF_USER) {
    /*
     * Accessing the stack below %sp is always a bug.
     * The large cushion allows instructions like enter
     * and pusha to work. ("enter $65535,$31" pushes
     * 32 pointers and then decrements %sp by 65535.)
     */
    if (address + 65536 + 32 * sizeof(unsigned long) < regs->sp) goto bad_area;
}
if (expand_stack(vma, address)) goto bad_area;
...
```

线性区缺页异常处理

2. 对于其他线性区，先排除非法错误，然后调用handle_mm_fault()处理

```
    /* * Ok, we have a good vm_area for this memory access, so
     * we can handle it..  */
good_area:
    si_code = SEGV_ACCERR;
    write = 0;
    switch (error_code & (PF_PROT|PF_WRITE)) {
    default: /* 3: write, present */
        /* fall through */
    case PF_WRITE: /* write, not present */
        if (!(vma->vm_flags & VM_WRITE)) goto bad_area;
        write++;
        break;
    case PF_PROT: /* read, present */
        goto bad_area;
    case 0: /* read, not present */
        if (!(vma->vm_flags & (VM_READ | VM_EXEC | VM_WRITE))) goto bad_area;
    }
    ...
    fault = handle_mm_fault(mm, vma, address, write);
```

线性区缺页异常处理

3. 若最后确定是用户地址空间的访问错误，则发出信号SIGSEGV

```
...
tsk->thread.cr2 = address;
/* Kernel addresses are always protection faults */
tsk->thread.error_code = error_code | (address >= TASK_SIZE);
tsk->thread.trap_no = 14;
force_sig_info_fault(SIGSEGV, si_code, address, tsk);
return;
...
```

4. ...

Outline

小结

作业

- 用户态和内核态分配内存有什么不同？
- 什么是线性区？列举4种最常见的线性区。
- Linux如何描述进程的地址空间？
- mm_struct中的mmap_cache有什么作用？

Thanks !

The end.