



# 第三次实例分析第四部分

李颖彦 李昊宸 郭泓锐 任敏思

# pde\_t\*copyuvm(pde\_t \*pgdir, uint sz)

- Copyuvm的作用是拷贝父进程的页表
- 首先, Setupkvm()会分配一页内存来放置页目录, 然后调用 mappages 来建立内核需要的映射, 这些映射可以在 kmap数组中找到。这里的映射包括内核的指令和数据, PHYSTOP以下的物理内存, 以及 I/O 设备所占的内存。需要注意的是, setupkvm 不会建立任何用户内存的映射。
- 这一步完成之后, 子进程的内核映射已经完成。

```
316 copyuvm(pde_t *pgdir, uint sz)
317 {
318     pde_t *d;
319     pte_t *pte;
320     uint pa, i, flags;
321     char *mem;
322
323     if((d = setupkvm()) == 0)
324         return 0;
325     for(i = 0; i < sz; i += PGSIZE){
326         if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
327             panic("copyuvm: pte should exist");
328         if(!(*pte & PTE_P))
329             panic("copyuvm: page not present");
330         pa = PTE_ADDR(*pte);
331         flags = PTE_FLAGS(*pte);
332         if((mem = kalloc()) == 0)
333             goto bad;
334         memmove(mem, (char*)P2V(pa), PGSIZE);
335         if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0) {
336             kfree(mem);
337             goto bad;
338         }
339     }
340     return d;
341
342 bad:
343     freevm(d);
344     return 0;
345 }
```

# pde\_t\*copyuvm(pde\_t \*pgdir, uint sz)

- 随后，进入一个以页为单位，遍历父进程所使用的地址空间的循环。
- Walkpgdir()模仿 x86 的分页硬件为一个虚拟地址寻找 PTE 的过程。walkpgdir 通过虚拟地址的前 10 位来找到在页目录中的对应条目，如果该条目不存在，说明要找的页表页尚未分配；如果 alloc 参数被设置了，walkpgdir 会分配页表页并将其物理地址放到页目录中。最后用虚拟地址的接下来 10 位来找到其在页表中的 PTE 地址。

```
316 copyuvm(pde_t *pgdir, uint sz)
317 {
318     pde_t *d;
319     pte_t *pte;
320     uint pa, i, flags;
321     char *mem;
322
323     if((d = setupkvm()) == 0)
324         return 0;
325     for(i = 0; i < sz; i += PGSIZE){
326         if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
327             panic("copyuvm: pte should exist");
328         if(!(*pte & PTE_P))
329             panic("copyuvm: page not present");
330         pa = PTE_ADDR(*pte);
331         flags = PTE_FLAGS(*pte);
332         if((mem = kalloc()) == 0)
333             goto bad;
334         memmove(mem, (char*)P2V(pa), PGSIZE);
335         if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0) {
336             kfree(mem);
337             goto bad;
338         }
339     }
340     return d;
341
342 bad:
343     freevm(d);
344     return 0;
345 }
```

# pde\_t\*copyuvm(pde\_t \*pgdir, uint sz)

- 随后，进入一个以页为单位，遍历父进程所使用的地址空间的循环。
- 之后，memmove将映射好的内容复制到通过kalloc分配的地址上，再通过mappages建立映射。
- Mappages()做的工作是在页表中建立一段虚拟内存到一段物理内存的映射。它是在页的级别，即一页一页地建立映射的。对于每一个待映射虚拟地址，mappages调用walkpgdir来找到该地址对应的 PTE 地址。然后初始化该 PTE 以保存对应物理页号、许可级别（`PTE\_W` 和/或 `PTE\_U`）以及 `PTE\_P` 位来标记该 PTE 是否是有效的。

```
316 copyuvm(pde_t *pgdir, uint sz)
317 {
318     pde_t *d;
319     pte_t *pte;
320     uint pa, i, flags;
321     char *mem;
322
323     if((d = setupkvm()) == 0)
324         return 0;
325     for(i = 0; i < sz; i += PGSIZE){
326         if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
327             panic("copyuvm: pte should exist");
328         if(!(*pte & PTE_P))
329             panic("copyuvm: page not present");
330         pa = PTE_ADDR(*pte);
331         flags = PTE_FLAGS(*pte);
332         if((mem = kalloc()) == 0)
333             goto bad;
334         memmove(mem, (char*)P2V(pa), PGSIZE);
335         if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0) {
336             kfree(mem);
337             goto bad;
338         }
339     }
340     return d;
341
342 bad:
343     freevm(d);
344     return 0;
345 }
```

# pde\_t\*copyuvm(pde\_t \*pgdir, uint sz)

- 最后，copyuvm返回d，即由setupkvm分配的页表，记录了由mappages完成虚拟内存到物理内存的映射。

```
316 copyuvm(pde_t *pgdir, uint sz)
317 {
318     pde_t *d;
319     pte_t *pte;
320     uint pa, i, flags;
321     char *mem;
322
323     if((d = setupkvm()) == 0)
324         return 0;
325     for(i = 0; i < sz; i += PGSIZE){
326         if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
327             panic("copyuvm: pte should exist");
328         if(!(*pte & PTE_P))
329             panic("copyuvm: page not present");
330         pa = PTE_ADDR(*pte);
331         flags = PTE_FLAGS(*pte);
332         if((mem = kalloc()) == 0)
333             goto bad;
334         memmove(mem, (char*)P2V(pa), PGSIZE);
335         if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0) {
336             kfree(mem);
337             goto bad;
338         }
339     }
340     return d;
341
342 bad:
343     freevm(d);
344     return 0;
345 }
```



# Int allocuvm(pde\_t \*pgdir, uint oldsz, uint newsz)

➤ Allocuvm的作用是为ELF文件的每个段分配内存。分配的范围是从oldsz所在的页面到newsz所在的页面。

首先，会检查请求分配的虚拟地址是否是在KERNBASE之下，不是的话就没有必要分配并返回0。

随后，以页为单位（PGSIZE）分配物理内存到，并用mappages建立物理内存到虚拟内存的映射。

最后，全部分配完之后，返回虚拟地址newsz。

```
221 int
222 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
223 {
224     char *mem;
225     uint a;
226
227     if(newsz >= KERNBASE)
228         return 0;
229     if(newsz < oldsz)
230         return oldsz;
231
232     a = PGROUNDUP(oldsz);
233     for(; a < newsz; a += PGSIZE){
234         mem = kalloc();
235         if(mem == 0){
236             cprintf("allocuvm out of memory\n");
237             deallocuvm(pgdir, newsz, oldsz);
238             return 0;
239         }
240         memset(mem, 0, PGSIZE);
241         if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
242             cprintf("allocuvm out of memory (2)\n");
243             deallocuvm(pgdir, newsz, oldsz);
244             kfree(mem);
245             return 0;
246         }
247     }
248     return newsz;
249 }
```

# Int loadvm(pde\_t \*pgdir, char \*addr, struct inode \*ip, uint offset, uint sz)

- Loadvm的作用是将ELF文件段的内容载入内存中。
- 首先，检查地址是否页对齐。如果没有对齐，则发出panic。
- 随后进入以页为单位的循环。

调用walkpgdir查找pte。由于一个进程创建完成之后，对应的虚拟地址的pte应该已经被创建好了。现在如果查询不到，说明该页面因意外丢失，发出panic。在每一个循环中，考虑到文件的大小未必是页对齐的，在做最后一次拷贝时需要注意规定读取字节的大小，不要把多余的部分读取进来。

- 最后，如果加载顺利完成，会返回0。

```
197 int
198 loadvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
199 {
200     uint i, pa, n;
201     pte_t *pte;
202
203     if((uint) addr % PGSIZE != 0)
204         panic("loadvm: addr must be page aligned");
205     for(i = 0; i < sz; i += PGSIZE){
206         if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
207             panic("loadvm: address should exist");
208         pa = PTE_ADDR(*pte);
209         if(sz - i < PGSIZE)
210             n = sz - i;
211         else
212             n = PGSIZE;
213         if(readi(ip, P2V(pa), offset+i, n) != n)
214             return -1;
215     }
216     return 0;
217 }
```

## Void clearpteu(pde\_t \*pgdir, char \*uva)

- clearpteu的作用是将目标页面的PTE\_U位设置为0。
- 用户代码只能使用带有`PTE\_U`设置的页。
- 对于一个进程，xv6 只把该进程所使用的内存对应的 PTE 的PTE\_U设为 1。

```
302 void
303 clearpteu(pde_t *pgdir, char *uva)
304 {
305     pte_t *pte;
306
307     pte = walkpgdir(pgdir, uva, 0);
308     if(pte == 0)
309         panic("clearpteu");
310     *pte &= ~PTE_U;
311 }
312
```



# Void switchvm(struct proc \*p)

- switchvm通知硬件开始使用目标进程的页表。
- 首先要做的是关中断，防止处理过程被打断。
- 接着会设置任务状态段 gdt[SEG\_TSS]。

```
struct taskstate ts; // used by kernel to find search for  
struct segdesc gdt[NSEGS]; // x86 global descriptor table  
volatile uint started; // Has the CPU started?
```

```
// Segment Descriptor  
struct segdesc {  
    uint lim_15_0 : 16; // Low bits of segment limit  
    uint base_15_0 : 16; // Low bits of segment base address  
    uint base_23_16 : 8; // Middle bits of segment base address  
    uint type : 4; // Segment type (see STS_ constants)  
    uint s : 1; // 0 = system, 1 = application  
    uint dpl : 2; // Descriptor Privilege Level  
    uint p : 1; // Present  
    uint lim_19_16 : 4; // High bits of segment limit  
    uint avl : 1; // Unused (available for software use)  
    uint rsv1 : 1; // Reserved  
    uint db : 1; // 0 = 16-bit segment, 1 = 32-bit segment  
    uint g : 1; // Granularity: limit scaled by 4K when set  
    uint base_31_24 : 8; // High bits of segment base address  
};
```

```
156 void  
157 switchvm(struct proc *p)  
158 {  
159     if(p == 0)  
160         panic("switchvm: no process");  
161     if(p->kstack == 0)  
162         panic("switchvm: no kstack");  
163     if(p->pgdir == 0)  
164         panic("switchvm: no pgdir");  
165  
166     pushcli();  
167     mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A, &mycpu()->ts,  
168                                     sizeof(mycpu()->ts)-1, 0);  
169     mycpu()->gdt[SEG_TSS].s = 0;  
170     mycpu()->ts.ss0 = SEG_KDATA << 3;  
171     mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;  
172     // setting IOPL=0 in eflags *and* iomb beyond the tss segment limit  
173     // forbids I/O instructions (e.g., inb and outb) from user space  
174     mycpu()->ts.iomb = (ushort) 0xFFFF;  
175     ltr(SEG_TSS << 3);  
176     lcr3(V2P(p->pgdir)); // switch to process's address space  
177     popcli();  
178 }
```

# Void switchvm(struct proc \*p)

➡ 接着会设置任务状态段 gdt[SEG\_TSS]。

```
// Segment Descriptor
struct segdesc {
    uint lim_15_0 : 16; // Low bits of segment limit
    uint base_15_0 : 16; // Low bits of segment base address
    uint base_23_16 : 8; // Middle bits of segment base address
    uint type : 4; // Segment type (see STS_constants)
    uint s : 1; // 0 = system, 1 = application
    uint dpl : 2; // Descriptor Privilege Level
    uint p : 1; // Present
    uint lim_19_16 : 4; // High bits of segment limit
    uint avl : 1; // Unused (available for software use)
    uint rsv1 : 1; // Reserved
    uint db : 1; // 0 = 16-bit segment, 1 = 32-bit segment
    uint g : 1; // Granularity: limit scaled by 4K when set
    uint base_31_24 : 8; // High bits of segment base address
};
```

Lim\_15\_0: (sizeof(mycpu()->ts) - 1) & 0xffff

Base\_15\_0: (uint)&mycpu()->ts & 0xffff

Base\_23\_16: ((uint)&mycpu()->ts)>>16 & 0xff

Type: SEG16 S:0 dpl:0 p:1

Lim\_19\_16: (sizeof(mycpu()->ts) - 1) >>16

Avl:0 rsv1:0 db:1 g:0 Base\_31\_24: ((uint)&mycpu()->ts)>>24

```
156 void
157 switchvm(struct proc *p)
158 {
159     if(p == 0)
160         panic("switchvm: no process");
161     if(p->kstack == 0)
162         panic("switchvm: no kstack");
163     if(p->pgdir == 0)
164         panic("switchvm: no pgdir");
165
166     pushcli();
167     mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A, &mycpu()->ts,
168                                 sizeof(mycpu()->ts)-1, 0);
169     mycpu()->gdt[SEG_TSS].s = 0;
170     mycpu()->ts.ss0 = SEG_KDATA << 3;
171     mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
172     // setting IOPL=0 in eflags *and* iomb beyond the tss segment limit
173     // forbids I/O instructions (e.g., inb and outb) from user space
174     mycpu()->ts.iomb = (ushort) 0xFFFF;
175     ltr(SEG_TSS << 3);
176     lcr3(V2P(p->pgdir)); // switch to process's address space
177     popcli();
178 }
```

# Void switchvm(struct proc \*p)

- switchvm通知硬件开始使用目标进程的页表。
- 首先要做的是关中断，防止处理过程被打断。
- 接着会设置任务状态段 gdt[SEG\_TSS]。
- 然后设置任务状态ts的部分(特权级、栈指针、I/O地址映射的基地址)。当特权级从用户模式向内核模式转换时，内核不能使用用户的栈，因为它可能不是有效的。用户进程可能是恶意的或者包含了一些错误，使得用户的 %esp 指向一个不是用户内存的地方。xv6 会使得在内陷发生的时候进行一个栈切换，栈切换的方法是让硬件从一个任务段描述符中读出新的栈选择符和一个新的 %esp 的值。在switchvm中，用户进程的内核栈顶地址被存入任务段描述符 (mtcpu()->ts.esp0) 中。

```
156 void
157 switchvm(struct proc *p)
158 {
159     if(p == 0)
160         panic("switchvm: no process");
161     if(p->kstack == 0)
162         panic("switchvm: no kstack");
163     if(p->pgdir == 0)
164         panic("switchvm: no pgdir");
165
166     pushcli();
167     mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A, &mycpu()->ts,
168                                   sizeof(mycpu()->ts)-1, 0);
169     mycpu()->gdt[SEG_TSS].s = 0;
170     mycpu()->ts.ss0 = SEG_KDATA << 3;
171     mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
172     // setting IOPL=0 in eflags *and* iomb beyond the tss segment limit
173     // forbids I/O instructions (e.g., inb and outb) from user space
174     mycpu()->ts.iomb = (ushort) 0xFFFF;
175     ltr(SEG_TSS << 3);
176     lcr3(V2P(p->pgdir)); // switch to process's address space
177     popcli();
178 }
```

# Void switchvm(struct proc \*p)

- switchvm通知硬件开始使用目标进程的页表。
- 首先要做的是关中断，防止处理过程被打断。
- 接着会设置任务状态段 gdt[SEG\_TSS]。
- 然后设置任务状态ts的部分(特权级、栈指针、I/O地址映射的基地址)。
- 然后设置tss寄存器，转移到用户地址空间。
- 最后开中断。

```
156 void
157 switchvm(struct proc *p)
158 {
159     if(p == 0)
160         panic("switchvm: no process");
161     if(p->kstack == 0)
162         panic("switchvm: no kstack");
163     if(p->pgdir == 0)
164         panic("switchvm: no pgdir");
165
166     pushcli();
167     mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A, &mycpu()->ts,
168                                   sizeof(mycpu()->ts)-1, 0);
169     mycpu()->gdt[SEG_TSS].s = 0;
170     mycpu()->ts.ss0 = SEG_KDATA << 3;
171     mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
172     // setting IOPL=0 in eflags *and* iomb beyond the tss segment limit
173     // forbids I/O instructions (e.g., inb and outb) from user space
174     mycpu()->ts.iomb = (ushort) 0xFFFF;
175     ltr(SEG_TSS << 3);
176     lcr3(V2P(p->pgdir)); // switch to process's address space
177     popcli();
178 }
```

# Void switchvm(struct proc \*p)

- Pushcli与popcli之间的关系由mycpu()->ncli来约束。有多少次pushcli，就需要多少次popcli，并且在pushcli的次数等于popcli的时候才会开中断。

```
104 void
105 pushcli(void)
106 {
107     int eflags;
108
109     eflags = readeflags();
110     cli();
111     if(mycpu()->ncli == 0)
112         mycpu()->intena = eflags & FL_IF;
113     mycpu()->ncli += 1;
114 }
115
116 void
117 popcli(void)
118 {
119     if(readeflags() & FL_IF)
120         panic("popcli - interruptible");
121     if(--mycpu()->ncli < 0)
122         panic("popcli");
123     if(mycpu()->ncli == 0 && mycpu()->intena)
124         sti();
125 }
126
127
```



# Exec操作

■ 在做完相关的准备工作 (begin\_op等) 后, 先检查ELF文件头, 检查魔术码是否对应。

随后setupkvm在内核分配一个页面存储页目录, 并映射内核所需要的页表项。


之后进入以段为单位的循环, 读取section header, 根据段表头的信息分配内存并建立映射 (allocuvm), 将段内容加载到内存中 (loaduvm)。

```
22 begin_op();
23
24 if((ip = namei(path)) == 0){
25     end_op();
26     cprintf("exec: fail\n");
27     return -1;
28 }
29 ilock(ip);
30 pgdir = 0;
31
32 // Check ELF header
33 if(readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
34     goto bad;
35 if(elf.magic != ELF_MAGIC)
36     goto bad;
37
38 if((pgdir = setupkvm()) == 0)
39     goto bad;
40
41 // Load program into memory.
42 sz = 0;
43 for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
44     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
45         goto bad;
46     if(ph.type != ELF_PROG_LOAD)
47         continue;
48     if(ph.memsz < ph.filesz)
49         goto bad;
50     if(ph.vaddr + ph.memsz < ph.vaddr)
51         goto bad;
52     if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
53         goto bad;
54     if(ph.vaddr % PGSIZE != 0)
55         goto bad;
56     if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
57         goto bad;
58 }
59 iunlockput(ip);
60 end_op();
```

## 在exec中执行的页表相关操作

```
if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
    goto bad;
if(ph.vaddr % PGSIZE != 0)
    goto bad;
if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
    goto bad;
```

首先，为程序的程序段分配页表（allocuvm），并将程序段拷贝到页表对应内存中（loaduvm）




之后调用allocuvm在原有基础上为分配两个新的页框，用于数据段

接下来的clearpteu将新分配的两个页框中的第一个页框的PTE\_U位置为无效，用作内核栈。而第二个页框则是用户栈

```
sz = PGROUNDUP(sz);  
if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)  
    goto bad;  
clearpteu(pgdir, (char*)(sz - 2*PGSIZE));  
sp = sz;
```

接下来处理参数。将exec时需要传入的参数复制到用户栈中。

```
71 // Push argument strings, prepare rest of stack in ustack
72 for(argc = 0; argv[argc]; argc++) {
73     if(argc >= MAXARG)
74         goto bad;
75     sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
76     if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
77         goto bad;
78     ustack[3+argc] = sp;
79 }
80 ustack[3+argc] = 0;
81
82 ustack[0] = 0xffffffff; // fake return PC
83 ustack[1] = argc;
84 ustack[2] = sp - (argc+1)*4; // argv pointer
85
86 sp -= (3+argc+1) * 4;
87 if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
88     goto bad;
89
90 // Save program name for debugging.
91 for(last=s=path; *s; s++)
92     if(*s == '/')
93         last = s+1;
94 safestrcpy(curproc->name, last, sizeof(curproc->name));
95
```



最后，修改curproc结构体中的各个参数，比如进程大小，进程的页表，进程的返回地址，栈指针等，通过switchvm，通知硬件开始使用目标进程的页表。

```
// Commit to the user image.  
oldpgdir = curproc->pgdir;  
curproc->pgdir = pgdir;  
curproc->sz = sz;  
curproc->tf->eip = elf.entry; // main  
curproc->tf->esp = sp;  
switchvm(curproc);  
freevm(oldpgdir);  
return 0;
```




## 在fork中执行的页表相关操作

```
// Allocate process.  
if((np = allocproc()) == 0){  
    return -1;  
}
```

Allocproc创建进程。

```
if((np->pgdir = copyvm(curproc->pgdir, curproc->sz)) == 0  
    kfree(np->kstack);  
    np->kstack = 0;  
    np->state = UNUSED;  
    return -1;  
}
```

Copyvm先为子进程分配页表，之后将父进程的页表拷贝进来，通过这种方法产生的子进程与父进程一模一样。



随后修改保存的eax值  
记录父进程打开的文件  
拷贝父进程名字  
将状态修改为可运行  
最后返回进程号

```
203 // Clear %eax so that fork returns 0 in the child.
204 np->tf->eax = 0;
205
206 for(i = 0; i < NOFILE; i++)
207     if(curproc->ofile[i])
208         np->ofile[i] = filedup(curproc->ofile[i]);
209 np->cwd = idup(curproc->cwd);
210
211 safestrcpy(np->name, curproc->name, sizeof(curproc->name));
212
213 pid = np->pid;
214
215 acquire(&ptable.lock);
216
217 np->state = RUNNABLE;
218
219 release(&ptable.lock);
220
221 return pid;
222 }
223
```

# entryother.S->start

- 从实模式开始，设置ds,es,ss段寄存器为0
- ds 为数据段寄存器，一般用于存放数据；
- ds地址对应的数据 相当于c语言中的全局变量
- ss 为栈段寄存器，一般作为栈使用 和sp搭档；
- ss地址对应的数据 相当于c语言中的局部变量
- ss相当于堆栈段的首地址 sp相当于堆栈段的偏移地址
- es 为扩展段寄存器；

```
7 // Control Register flags
8 #define CR0_PE          0x00000001 // Protection Enable

22 .code16
23 .globl start
24 start:
25     cli
26
27     # Zero data segment registers DS, ES, and SS.
28     xorw    %ax,%ax
29     movw    %ax,%ds
30     movw    %ax,%es
31     movw    %ax,%ss
32
33     # Switch from real to protected mode. Use a bootstrap GDT that makes
34     # virtual addresses map directly to physical addresses so that the
35     # effective memory map doesn't change during the transition.
36     lgdt    gdt_desc
37     movl    %cr0, %eax
38     orl     $CR0_PE, %eax
39     movl    %eax, %cr0
40
41     # Complete the transition to 32-bit protected mode by using a long jmp
42     # to reload %cs and %eip. The segment descriptors are set up with no
43     # translation, so that the mapping is still the identity mapping.
44     ljmpl   $(SEG_KCODE<<3), $(start32)
45
46 //PAGEBREAK!
47 .code32 # Tell assembler to generate 32-bit code now.
48 start32:
```

# entryother.S->start

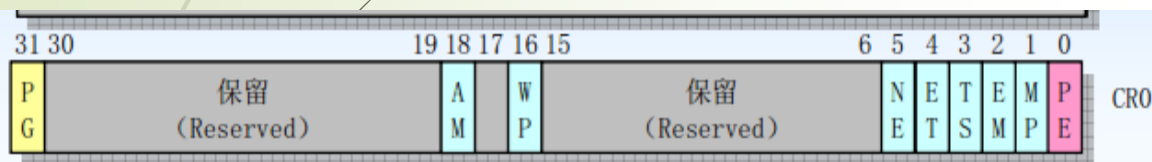
- 从实模式开始，设置ds,es,ss段寄存器为0
- 加载GDT表的地址到GDTR寄存器
- GDTR寄存器中用于存放全局描述符表GDT的32位的线性基地址和16位的表限长值。基地址指定GDT表中字节0在线性地址空间中的地址，表长度指明GDT表的字节长度值。

```
7 // Control Register flags
8 #define CR0_PE          0x00000001 // Protection Enable

22 .code16
23 .globl start
24 start:
25     cli
26
27     # Zero data segment registers DS, ES, and SS.
28     xorw    %ax,%ax
29     movw    %ax,%ds
30     movw    %ax,%es
31     movw    %ax,%ss
32
33     # Switch from real to protected mode. Use a bootstrap GDT that makes
34     # virtual addresses map directly to physical addresses so that the
35     # effective memory map doesn't change during the transition.
36     lgdt    gdt_desc
37     movl    %cr0, %eax
38     orl     $CR0_PE, %eax
39     movl    %eax, %cr0
40
41     # Complete the transition to 32-bit protected mode by using a long jmp
42     # to reload %cs and %eip. The segment descriptors are set up with no
43     # translation, so that the mapping is still the identity mapping.
44     ljmpl   $(SEG_KCODE<<3), $(start32)
45
46 //PAGEBREAK!
47 .code32 # Tell assembler to generate 32-bit code now.
48 start32:
```

# entryother.S->start

- 从实模式开始，设置ds,es,ss段寄存器为0
- 加载GDT表的地址到GDTR寄存器
- 设置CR0寄存器第0位为1，表示可以开启保护模式



- PE: CR0的0位是启用保护位 (protection enable)。当设置该位的时候即开启了保护模式，系统上电复位的时候该位默认为0,于是是实模式 real model. PE置1的保护，实质上是开启段级保护，就是只进行了分段，没有开启分页机制。如果要开启分页机制需要同时置位PE和PG

```
7 // Control Register flags
8 #define CR0_PE 0x00000001 // Protection Enable
```

```
22 .code16
23 .globl start
24 start:
25     cli
26
27     # Zero data segment registers DS, ES, and SS.
28     xorw    %ax,%ax
29     movw    %ax,%ds
30     movw    %ax,%es
31     movw    %ax,%ss
32
33     # Switch from real to protected mode. Use a bootstrap GDT that makes
34     # virtual addresses map directly to physical addresses so that the
35     # effective memory map doesn't change during the transition.
36     lgdt    gdt_desc
37     movl    %cr0, %eax
38     orl     $CR0_PE, %eax
39     movl    %eax, %cr0
40
41     # Complete the transition to 32-bit protected mode by using a long jmp
42     # to reload %cs and %eip. The segment descriptors are set up with no
43     # translation, so that the mapping is still the identity mapping.
44     ljmpl   $(SEG_KCODE<<3), $(start32)
45
46 //PAGEBREAK!
47 .code32 # Tell assembler to generate 32-bit code now.
48 start32:
```



# entryother.S->start

- 从实模式开始，设置ds,es,ss段寄存器为0
- 加载GDT表的地址到GDTR寄存器
- 设置CR0寄存器第0位为1，表示可以开启保护模式
- ljmp指令修改CS段寄存器，这时真正开启保护模式
- cs 为代码段寄存器，一般用于存放代码；通常和IP 使用用于处理下一条执行的代码

```
7 // Control Register flags
8 #define CR0_PE          0x00000001 // Protection Enable

22 .code16
23 .globl start
24 start:
25     cli
26
27     # Zero data segment registers DS, ES, and SS.
28     xorw    %ax,%ax
29     movw    %ax,%ds
30     movw    %ax,%es
31     movw    %ax,%ss
32
33     # Switch from real to protected mode. Use a bootstrap GDT that makes
34     # virtual addresses map directly to physical addresses so that the
35     # effective memory map doesn't change during the transition.
36     lgdt    gdt_desc
37     movl    %cr0, %eax
38     orl     $CR0_PE, %eax
39     movl    %eax, %cr0
40
41     # Complete the transition to 32-bit protected mode by using a long jmp
42     # to reload %cs and %eip. The segment descriptors are set up with no
43     # translation, so that the mapping is still the identity mapping.
44     ljmpl   $(SEG_KCODE<<3), $(start32)
45
46 //PAGEBREAK!
47 .code32 # Tell assembler to generate 32-bit code now.
48 start32:
```

# entryother.S->start

- Makefile 的103到106行将entryother编译为二进制代码，设置其入口地址为start，拷贝到0x7000处，这里是其他cpu最初运行的内核代码。
- 总体来看，这段代码初始化了ds、es、ss寄存器，修改了cs，开启段级保护，接下来进入保护模式。

```
7 // Control Register flags
8 #define CR0_PE          0x00000001 // Protection Enable

22 .code16
23 .globl start
24 start:
25     cli
26
27     # Zero data segment registers DS, ES, and SS.
28     xorw    %ax,%ax
29     movw    %ax,%ds
30     movw    %ax,%es
31     movw    %ax,%ss
32
33     # Switch from real to protected mode. Use a bootstrap GDT that makes
34     # virtual addresses map directly to physical addresses so that the
35     # effective memory map doesn't change during the transition.
36     lgdt    gdt_desc
37     movl    %cr0, %eax
38     orl     $CR0_PE, %eax
39     movl    %eax, %cr0
40
41     # Complete the transition to 32-bit protected mode by using a long jmp
42     # to reload %cs and %eip. The segment descriptors are set up with no
43     # translation, so that the mapping is still the identity mapping.
44     ljmpl   $(SEG_KCODE<<3), $(start32)
45
46 //PAGEBREAK!
47 .code32 # Tell assembler to generate 32-bit code now.
48 start32:
```

# void trap(struct trapframe \*tf)

## default选项

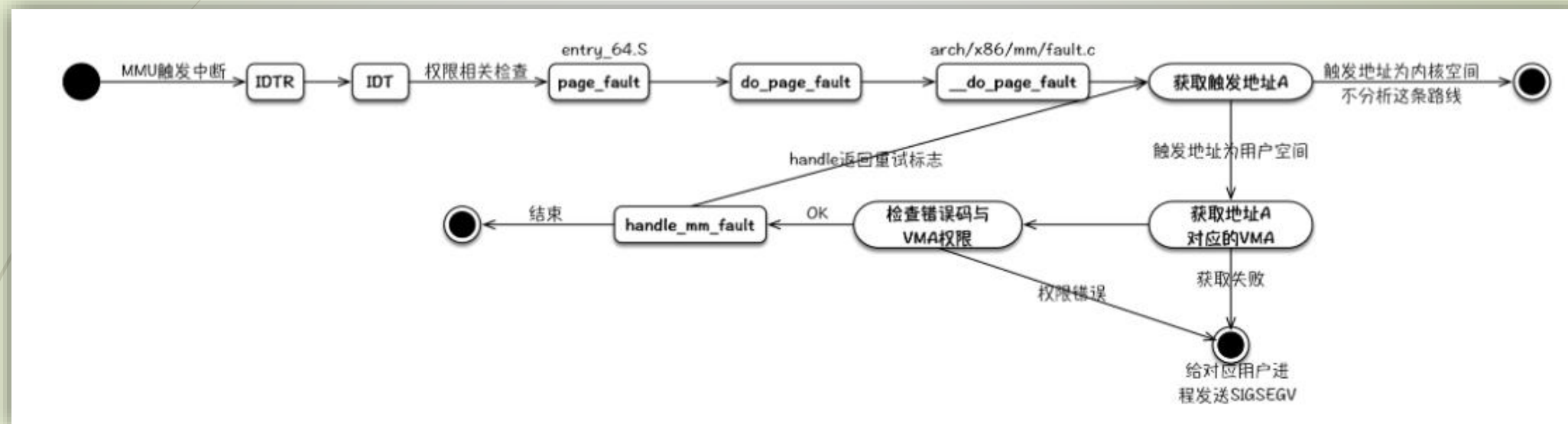
- T\_PGFLT表示page fault，首先进入all traps经过一系列通用中断处理流程最终进入trap中经过switch进入default中：
- myproc调用mycpu返回当前的proc(pcb)，它等于0说明当前没有记录进程
- tf->cs&3==0说明CPL寄存器的值是0
- 这2种情况下属于当前处于内核态，说明内核出错，进入panic函数，打印信息最终进入死循环
- 否则属于用户态，说明程序出错打印相关信息并结束进程

```
//PAGEBREAK: 13
default:
    if(myproc() == 0 || (tf->cs&3) == 0){
        // In kernel, it must be our mistake.
        cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
            tf->trapno, cpuid(), tf->eip, rcr2());
        panic("trap");
    }
    // In user space, assume process misbehaved.
    cprintf("pid %d %s: trap %d err %d on cpu %d "
        "eip 0x%x addr 0x%x--kill proc\n",
        myproc()->pid, myproc()->name, tf->trapno,
        tf->err, cpuid(), tf->eip, rcr2());
    myproc()->killed = 1;
}
```

```
panicked = 1; // freeze other CPU
for(;;)
    ;
```

# linux 系统中 14 号中断的处理流程

14号中断是Page Fault中断，进入Page Fault相关的处理：



流程与xv6相似，由中断表(IDT)跳转至`page_fault`，期间调用`do_page_fault`，从`cr2`寄存器中读出虚拟地址，并调用`_do_page_fault`检查地址属于`kernel/user`，若是`kernel`执行如下代码：

```
if (unlikely(fault_in_kernel_space(address))) {
    if (!(error_code & (PF_RSVD | PF_USER | PF_PROT))) {
        if (vmalloc_fault(address) >= 0)
            return;

        if (kmemcheck_fault(regs, address, error_code))
            return;
    }

    /* Can handle a stale RO->RW TLB: */
    if (spurious_fault(error_code, address))
        return;

    /* kprobes don't want to hook the spurious faults: */
    if (kprobes_fault(regs))
        return;
    /*
     * Don't take the mm semaphore here. If we fixup a prefetch
     * fault we could otherwise deadlock:
     */
    bad_area_nosemaphore(regs, error_code, address, NULL);

    return;
}
```

进行一些条件判断，并执行响应的处理函数，否则通过信号量死锁？  
(具体代码量过于庞大，我们只阅读主流流程)



```

vma = find_vma(mm, address);
if (unlikely(!vma)) {
    bad_area(regs, error_code, address);
    return;
}
if (likely(vma->vm_start <= address))
    goto good_area;
if (unlikely(!(vma->vm_flags & VM_GROWSDOWN))) {
    bad_area(regs, error_code, address);
    return;
}

```

- 若是user，通过find\_vma在内存中寻找内存段，如果没有找到说明是非法地址，执行bad\_area，如果找到执行good\_area，在good\_area中执handle\_mm\_fault函数

```

if (!arch_vma_access_permitted(vma, flags & FAULT_FLAG_WRITE,
                                flags & FAULT_FLAG_INSTRUCTION,
                                flags & FAULT_FLAG_REMOTE))
    return VM_FAULT_SIGSEGV;

if (unlikely(is_vm_hugetlb_page(vma)))
    ret = hugetlb_fault(vma->vm_mm, vma, address, flags);
else
    ret = __handle_mm_fault(vma, address, flags);

```

- 首先若权限判断错误，则返回信号量杀死进程？
- 比较重要的是调用\_\_handle\_mm\_fault



```
pgd = pgd_offset(mm, address);
p4d = p4d_alloc(mm, pgd, address);
...
vmf.pud = pud_alloc(mm, p4d, address);
...
vmf.pmd = pmd_alloc(mm, vmf.pud, address);
...
return handle_pte_fault(&vmf);
```

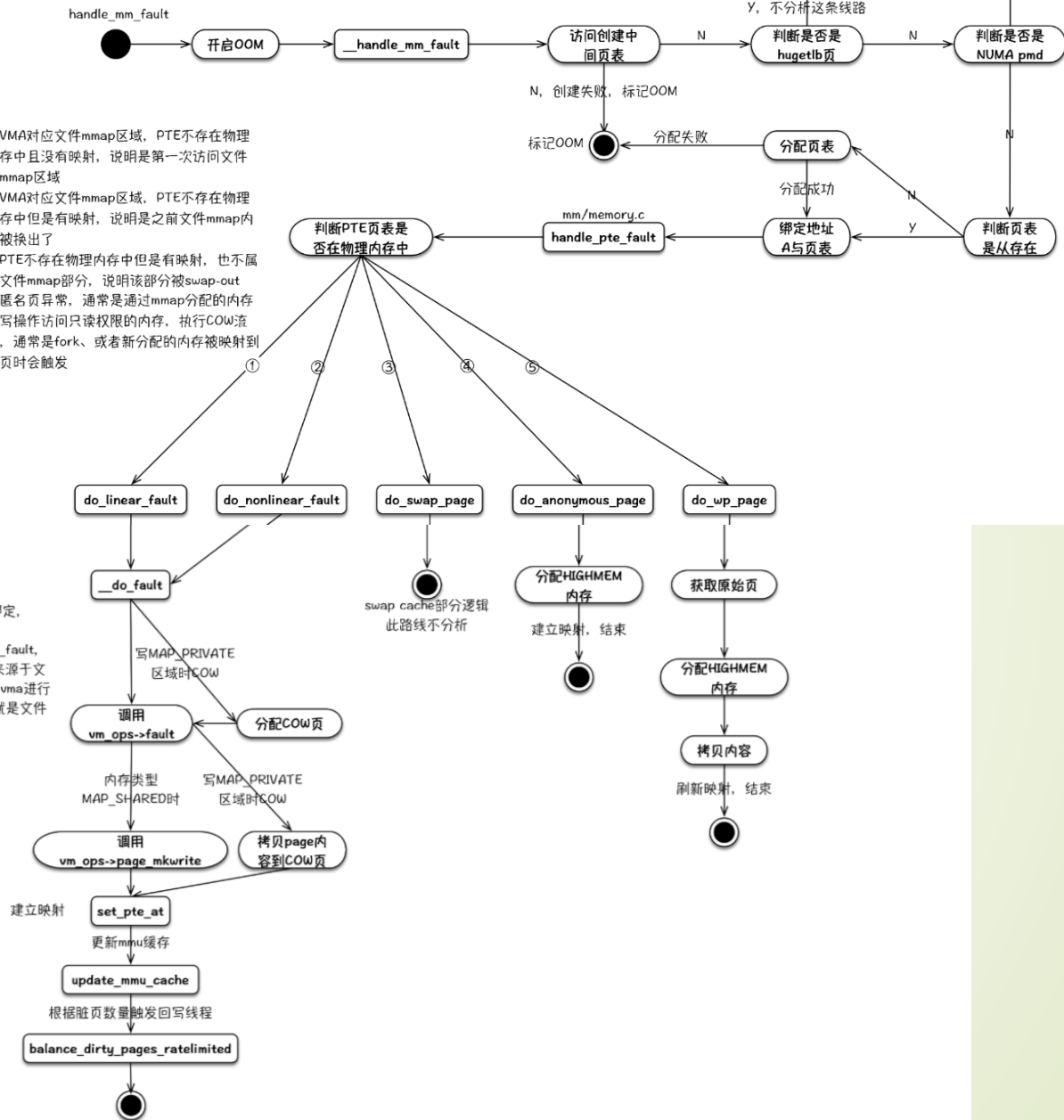
- 为pud、pmd等目录分配空间，最后调用handle\_pte\_fault,完成对物理页的分配。
- 一份讲的比较简单的参考资料：
- [https://cloud.tencent.com/developer/article/1459526?ADUIN=842793857&ADSESSION=1575213079&ADTAG=CLIENT.QQ.5603\\_.0&ADPUBNO=26933#userconsent#](https://cloud.tencent.com/developer/article/1459526?ADUIN=842793857&ADSESSION=1575213079&ADTAG=CLIENT.QQ.5603_.0&ADPUBNO=26933#userconsent#)

## 一份大致的处理流程

- ①VMA对应文件mmap区域，PTE不存在物理内存中且没有映射，说明是第一次访问文件的mmap区域
- ②VMA对应文件mmap区域，PTE不存在物理内存中但是有映射，说明是之前文件mmap内存被换出了
- ③PTE不存在物理内存中但是有映射，也不属于文件mmap部分，说明该部分被swap-out
- ④匿名页异常，通常是通过mmap分配的内存
- ⑤写操作访问只读权限的内存，执行COW流程，通常是fork、或者新分配的内存被映射到零页时会触发

vma->vm\_ops由mmap系统调用时绑定，如ext4文件系统绑定的是ext4\_file\_vm\_ops->fault = filemap\_fault，其会返回一个新的物理内存page(来源于文件的pagecache)随后会将该page与vma进行关联映射，则文件mmap时访问的就是文件的pagecache

当VMA类型是SHARED时，尽量共享物理内存，则需要调用page\_mkwrite通知具体的文件系统该区域被共享





■ Thanks for your listening