深 圳 大 学 实 验 报 告

课程名称：操作系统

实验项目名称：综合实验 1——第二部分

学院：计算机与软件学院

专业：计算机科学与技术

指导教师：谭舜泉

报告人：冯海月 黎浩然 学号：2018191116 2018112061

实验时间：2021 年 4 月 25 日-2021 年 6 月 13 日

实验报告提交时间：2021 年 6 月 13 日

教务部制

实验目的与要求：

实验目的：

(1)、掌握计算机操作系统管理进程、处理机、存储器、文件系统的基本方法。

(2)、了解现代计算机操作系统的工作原理，具有初步分析、设计操作系统的能力。

(3)、通过阅读 xv6 操作系统代码，理解其是如何实现操作系统中的各种管理功能，在系统程序设计能力方面得到提升。

实验要求：

(1)、阅读“xv6 中文文档” 第 2 章：页表，回答以下问题：

a）(vm.c L11) kpgdir 被用于创建一个调度器所用的页表。 请问理论上这个页表所支持的最大虚拟地址空间是多大？

b）(vm.c L124) 在 kmap 中，DEVSPACE 的 phys_end 为 0？！请问在 mappages 函数中这一段的长度有多大。

c）(vm.c L151) kpgdir = setupkvm();

通过 setupkvm 函数，创建了调度器所用的页表。请深入 setupkvm 函数内部，确定在创建页表过程中，总共调用了多少次 kalloc 函数分配 4K 物理块用于存放页表项？

（2)、阅读“xv6 中文文档” 第 4 章：锁，回答以下问题：

a）我们可以看到 xv6 总是让持有锁的线程负责释放该锁。但有一个例外。阅读“xv6 中文手册”P38“代码：调度”，请回答在调度器和被调度的进程之间，如何确保 ptable.lock 的锁定(acquire)和释放(release)能够两两配对？

（3)、阅读“xv6 中文文档”第 3 章：陷入、中断和驱动程序，回答以下问题：

a）在 zombie.c 中，sleep(5); （L12）。sleep 是一个系统调用。请分析代码，阐述在代码中，这一系统调用如何一步步的转化为一个对核心函数 sleep（proc.c / L343）的调用？

b）sleep(5)代表 zombie 进程总共睡眠多少毫秒？请通过代码分析给出你的答案。

说明：

（1）本次实验课作业满分为 100 分，占总成绩的比例（待定）。

（2）本次实验课作业截至时间 2021 年 6 月 13 日（周日）23:59。

（3）报告正文：请在指定位置填写，本次实验不需要单独提交源程序文件。

（4）个人信息：WORD 文件名中的“姓名”、“学号”，请改为你的姓名和学号；实验报告的首页，请准确填写“学院”、“专业”、“报告人”、“学号”、“班级”、“实验报告提交时间”等信息。

（5）提交方式：请在 BLACKBOARD 平台中按时提交；延迟提交不得分。

（6）发现抄袭（包括复制&粘贴整句话、整张图），该次作业记零分。

（7）期末考试阶段补交无效。

(1)、阅读“xv6 中文文档” 第 2 章：页表，回答以下问题：

a）(vm.c L11) kpgdir 被用于创建一个调度器所用的页表。请问理论上这个页表所支持的最大虚拟地址空间是多大？

b）(vm.c L124) 在 kmap 中，DEVSPACE 的 phys_end 为 0？！请问在 mappages 函数中这一段的长度有多大。 32M

c）(vm.c L151) kpgdir = setupkvm();

通过 setupkvm 函数，创建了调度器所用的页表。请深入 setupkvm 函数内部，确定在创建页表过程中，总共调用了多少次 kalloc 函数分配 4K 物理块用于存放页表项？

a)

setupkvm 函数在前面的实验中已经解释过。

在 mmu.h 文件中可以看到如下注释，说明 xv6 虚拟地址的构成：

```
103  // A virtual address 'la' has a three-part structure as follows:
104  //
105  // +--------10------+-------10-------+---------12----------+
106  // | Page Directory |   Page Table   | Offset within Page  |
107  // |      Index     |      Index     |                     |
108  // +----------------+----------------+---------------------+
109  //  \--- PDX(va) --/ \--- PTX(va) --/
```

由上面可以知道，xv6 使用的 32 位寻址。地址的前 10 比特由外层页表中的页表项决定，第 11-20 比特由内层页表中的页表项决定。表明每一个外层页表和内层页表都有 1024 项。

由于一个页面大小为 4KB，xv6 使用 32 位地址空间，指针的大小为 4B，说明一个页面能存放 4KB/4B=1K=1024 个内层页表项。每个页表项又指向一个内层页表，里面存放 1024 个 4KB 物理块的首地址，因此，理论上这个页表所支持的最大虚拟地址空间为：1024 * 1024 * 4KB = 4GB。

b)

如下所示，kmap[3]的 phys_end 为 0。乍一看似乎有一些违背常理，phys_end 不是应该比 phys_start 的值要大吗？

```
// This table defines the kernel's mappings, which are present in
// every process's page table.
static struct kmap {
  void *virt;
  uint phys_start;
  uint phys_end;
  int perm;
} kmap[] = {
 { (void*)KERNBASE, 0,             EXTMEM,    PTE_W}, // I/O space
 { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},     // kern text+rodata
 { (void*)data,     V2P(data),     PHYSTOP,   PTE_W}, // kern data+memory
 { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W}, // more devices
};
```

查找 phys_end 在 xv6 源代码中出现的地方，发现除了声明外，这个变量只在 vm.c 的 setupkvm 函数中被使用。

```
// Set up kernel part of a page table. 创建每个进程的内核部分的虚拟内存的页表
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE); // 4M
  if (p2v(PHYSTOP) > (void*)DEVSPACE)
    panic("PHYSTOP too high");
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0)
      return 0;
  return pgdir;
}
```

映射区域的大小（size）

如上图所示，（k->phys_end - k->phys_start）用来指示映射的物理地址空间大小。如此一来，我们用 0 – DEVSPACE = 0 – 0xFE00 0000 = 0x200 0000（利用 32 位地址空间溢出原理来计算），由于 size 为无符号整数，可以得到这块空间大小为 32MB。

```
struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE); // 4M
int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
```

所以在 mappages 函数中的 DEVSPACE 的大小为 32MB。

c)

我们先来看看 setupkvm 函数。

```
127  // Set up kernel part of a page table.
128  pde_t*
129  setupkvm(void)
130  {
131    pde_t *pgdir;
132    struct kmap *k;
133
134    if((pgdir = (pde_t*)kalloc()) == 0)
135      return 0;
136    memset(pgdir, 0, PGSIZE);
137    if (p2v(PHYSTOP) > (void*)DEVSPACE)
138      panic("PHYSTOP too high");
139    for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
140      if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
141                  (uint)k->phys_start, k->perm) < 0)
142        return 0;
143    return pgdir;
144  }
```

可以在 setupkvm 函数中看到，函数首先在 134 行调用一次 kalloc 分配一个外层页表所需的空间，并存放该页表地址到 pgdir 变量中。所以只有一个外层页表就是 pgdir 所指向的区域。**而 kalloc 函数的调用次数等于外层页表的个数+内层页表的**个数。

容易知道 mappages 是负责内层页表的分配和映射的，我们来看看这个函数：

```
70  static int
71  mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
72  {
73    char *a, *last;
74    pte_t *pte;
75
76    a = (char*)PGROUNDDOWN((uint)va)
77    last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
78    for(;;){
79      if((pte = walkpgdir(pgdir, a, 1)) == 0)
80        return -1;
81      if(*pte & PTE_P)
82        panic("remap");
83      *pte = pa | perm | PTE_P;
84      if(a == last)
85        break;
86      a += PGSIZE;
87      pa += PGSIZE;
88    }
89    return 0;
90  }
```

每一次调用 walkpgdir 应该就是分配一个内层页表了。因为每一次调用 walkpgdir 如果无错误的话最多只会调用一次 kalloc，并且由其返回的值所在的变量名 pte（page table entry）可以推测。因此只需要考虑调用了多少次 walkpgdir 即可。

值得注意的是 mappages 函数的第 2、3 个参数。这两个参数指示了这次调用分配的内层页表需要映射/管控的虚拟内存范围。由前面的实验我们可以知道：一个内层页表有 4096 个字节，每个条目占 4 个字节，所以共 1024 个条目。而每个条目管控/映射一个内存页面，也就是 4096 字节。

所以一个内层页表负责映射/管控 1024 * 4096 B = 4 MB 的内存。

所以我们只要知道到底有多少内存是需要被映射的，然后将其除以 4MB，就能得到内层页表的数量!!!

```
115 static struct kmap {
116   void *virt;
117   uint phys_start;
118   uint phys_end;
119   int perm;
120 } kmap[] = {
121  { (void*)KERNBASE, 0,             EXTMEM,    PTE_W}, // I/O space
122  { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},     // kern text+rodata
123  { (void*)data,     V2P(data),     PHYSTOP,   PTE_W}, // kern data+memory
124  { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W}, // more devices
125 };
```

我们观察 kmap 这个全局结构数组，共有 4 个元素。这 4 个元素就是内层页表需要负责映射的区域。Setupkvm 通过一个 for 循环遍历 kmap 的 4 个元素，然后根据其值调用前面提到的 mappages 函数分配页表进行映射。

```
1  // Memory layout
2
3  #define EXTMEM  0x100000            // Start of extended memory
4  #define PHYSTOP 0xE000000           // Top physical memory
5  #define DEVSPACE 0xFE000000         // Other devices are at high addresses
6
7  // Key addresses for address space layout (see kmap in vm.c for layout)
8  #define KERNBASE 0ex80000000         // First kernel virtual address
9  #define KERNLINK (KERNBASE+EXTMEM)  // Address where kernel is linked
10
11 #ifndef __ASSEMBLER__
12
13 static inline uint v2p(void *a) { return ((uint) (a))  - KERNBASE; }
14 static inline void *p2v(uint a) { return (void *) ((a) + KERNBASE); }
15
16 #endif
17
18 #define V2P(a) (((uint) (a)) - KERNBASE)
19 #define P2V(a) (((void *) (a)) + KERNBASE)
20
21 #define V2P_WO(x) ((x) - KERNBASE)    // same as V2P, but without casts
22 #define P2V_WO(x) ((x) + KERNBASE)    // same as V2P, but without casts
```

结合 def.h 文件中 EXTMEM，KERNLINK 以及 V2P 的宏定义可以知道：kmap[0]、kmap[1]和 kmap[2]所指示的内存区域是连续的，其大小之和为 PHYSTOP 字节，也就是 0xE000000 B = 224 MB；而 kmap[4] 从 0xFE000000 到 0x0 共有 0x2000000 B = 32MB

于是需要映射的内存区域的大小为 224 MB + 32 MB = 256 MB；需要分配的内层页表的数量为 256MB / 4MB = 64。加上外层页表调用的一次 kalloc，所以**总共调用 65 次 kalloc 函数分配 4K 物理块用于存放页表项。**

（2）、阅读“xv6 中文文档” 第 4 章：锁，回答以下问题：

a）我们可以看到 xv6 总是让持有锁的线程负责释放该锁。但有一个例外。阅读“xv6 中文手册”P38“代码：调度”，请回答在调度器和被调度的进程之间，如何确保 ptable.lock 的锁定(acquire)和释放(release)能够两两配对？

回到 scheduler 函数中：

```
257  void
258  scheduler(void)
259  {
260    struct proc *p;
261
262    for(;;){
263      // Enable interrupts on this processor.
264      sti();
265
266      // Loop over process table looking for process to run.
267      acquire(&ptable.lock);
268      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
269        if(p->state != RUNNABLE)
270          continue;
271
272        // Switch to chosen process.  It is the process's job
273        // to release ptable.lock and then reacquire it
274        // before jumping back to us.
275        proc = p;
276        switchuvm(p);
277        p->state = RUNNING;
278        swtch(&cpu->scheduler, proc->context);
279        switchkvm();
280
281        // Process is done running for now.
282        // It should have changed its p->state before coming back.
283        proc = 0;
284      }
285      release(&ptable.lock);
286
287    }
288  }
```

可以看到：在 scheduler 函数核心的调度代码中，按时间片轮转法调度每一个进程的 for 循环中，并没有任何获取或释放 p->lock 的操作（假设总是存在就绪状态的进程），而是将这项工作交给了被调度的进程。

前面的实验我们曾经提到，无论是基于硬件中断还是进程自发的放弃 cpu 的行为，最终都会进入 yield 函数，如下：

```
310  // Give up the CPU for one scheduling round.
311  void
312  yield(void)
313  {
314    acquire(&ptable.lock);  //DOC: yieldlock
315    proc->state = RUNNABLE;
316    sched();
317    release(&ptable.lock);
318  }
```

可以看到，进程在进入调度器之前，先获取了 p->lock；从调度器回来之后（被选中作为获得 CPU 的进程），又释放了 p->lock。因此，在整个 scheduler 函数的过程中，p->lock 是处于已经被持有的状态。很明显，除非一个进程连续被调度两次，获取 p->lock 和释放 p->lock 的并不是同一个进程。

顺便提一下：在 xv6 的文档中解释了这样处理 p->lock 是出于 xv6 可能运行在多 CPU 机器上的考量。这样做的目的是使得在整个 swtch 调用切换进程期间，p->lock 都是处于被持有的状态。p->lock 的目的是为了使得对进程的 state 和 context 的修改(在 p->lock 持有期间)成为临界区。虽然处理机调度(器)是基于 CPU 的，但是考虑到运行 xv6 操作系统的机器上可能会有其它的 CPU 的存在，所以需要保证调度期间只有一个 CPU 能修改一个进程的核心元数据，例如 state 和 context。

当然，仅仅在 yield 中成对出现还不太够，还需要考虑 sleep，exit 等可能让进程失去 CPU 的情况。来看看 sleep：

```
342  void
343  sleep(void *chan, struct spinlock *lk)
344  {
345    if(proc == 0)
346      panic("sleep");
347
348    if(lk == 0)
349      panic("sleep without lk");
350
351    // Must acquire ptable.lock in order to
352    // change p->state and then call sched.
353    // Once we hold ptable.lock, we can be
354    // guaranteed that we won't miss any wakeup
355    // (wakeup runs with ptable.lock locked),
356    // so it's okay to release lk.
357    if(lk != &ptable.lock){  //DOC: sleeplock0
358      acquire(&ptable.lock);  //DOC: sleeplock1
359      release(lk);
360    }
361
362    // Go to sleep.
363    proc->chan = chan;
364    proc->state = SLEEPING;
365    sched();
366
367    // Tidy up.
368    proc->chan = 0;
369
370    // Reacquire original lock.
371    if(lk != &ptable.lock){  //DOC: sleeplock2
372      release(&ptable.lock);
373      acquire(lk);
374    }
375  }
```

主动进入 sleep 函数时，**sleep 函数对 p->lock 的处理与 yield 函数相似**；再看看 exit 函数：

```
166  void
167  exit(void)
168  {
169    struct proc *p;
170    int fd;
171
172    if(proc == initproc)
173      panic("init exiting");
174
175    // Close all open files.
176    for(fd = 0; fd < NOFILE; fd++){
177      if(proc->ofile[fd]){
178        fileclose(proc->ofile[fd]);
179        proc->ofile[fd] = 0;
180      }
181    }
182
183    iput(proc->cwd);
184    proc->cwd = 0;
185
186    acquire(&ptable.lock);
187
188    // Parent might be sleeping in wait().
189    wakeup1(proc->parent);
190
191    // Pass abandoned children to init.
192    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
193      if(p->parent == proc){
194        p->parent = initproc;
195        if(p->state == ZOMBIE)
196          wakeup1(initproc);
197      }
198    }
199
200    // Jump into the scheduler, never to return.
201    proc->state = ZOMBIE;
202    sched();
203    panic("zombie exit");
204  }
```

由于 exit 的进程不会再被调度，因此在 sched 以后，就没有以后了……最后的那一句 panic 当然也是不会被运行的。我们仍然能保证整个调度期间 p->lock 是持有状态！

```
208  int
209  wait(void)
210  {
211    struct proc *p;
212    int havekids, pid;
213
214    acquire(&ptable.lock);
215    for(;;){
216      // Scan through table looking for zombie children.
217      havekids = 0;
218      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
219        if(p->parent != proc)
220          continue;
221        havekids = 1;
222        if(p->state == ZOMBIE){
223          // Found one.
224          pid = p->pid;
225          kfree(p->kstack);
226          p->kstack = 0;
227          freevm(p->pgdir);
228          p->state = UNUSED;
229          p->pid = 0;
230          p->parent = 0;
231          p->name[0] = 0;
232          p->killed = 0;
233          release(&ptable.lock);
234          return pid;
235        }
236      }
237
238      // No point waiting if we don't have any children.
239      if(!havekids || proc->killed){
240        release(&ptable.lock);
241        return -1;
242      }
243
244      // Wait for children to exit.  (See wakeup1 call in proc_exit.)
245      sleep(proc, &ptable.lock);  //DOC: wait-sleep
246    }
247  }
```

在 wait 函数中就比较麻烦了。在 wait 函数的开始就获取 p->lock，然后分两种情况，如果 wait 函数能够直接返回则立刻释放 p->lock。否则然后调用 sleep 进入调度器。

```
357    if(lk != &ptable.lock){  //DOC: sleeplock0
358      acquire(&ptable.lock);  //DOC: sleeplock1
359      release(lk);
360    }
361
362    // Go to sleep.
363    proc->chan = chan;
364    proc->state = SLEEPING;
365    sched();
366
367    // Tidy up.
368    proc->chan = 0;
369
370    // Reacquire original lock.
371    if(lk != &ptable.lock){  //DOC: sleeplock2
372      release(&ptable.lock);
373      acquire(lk);
374    }
375  }
```

有趣的是，由于此时 sleep 的第二个参数导致 sleep **中的条件判断未通过从而防止二次获取 p->lock**；但无论如何，wait 函数总是在释放 p->lock 之后才回返回！

因此：

（1） wait 函数的调用、主动进入 sleep 以及主动或被动进入 yield 时，会进入调度器，都要先获取 p->lock，返回时都要释放 p->lock。

（2） exit 之前要先获取 p->lock，因为 exit 的进程不会返回，所以不需要释放 p->lock。

因此 p->lock 的获取和释放总是成对出现的，尽管它们运行在不同的进程中。

（3）、阅读“xv6 中文文档”第 3 章：陷入、中断和驱动程序，回答以下问题：

a）在 zombie.c 中，sleep(5); （L12）。sleep 是一个系统调用。请分析代码，阐述在代码中，这一系统调用如何一步步的转化为一个对核心函数 sleep（proc.c / L343）的调用？

b）sleep(5)代表 zombie 进程总共睡眠多少毫秒？请通过代码分析给出你的答案。

a)

zombie.c 代码如下。其中 sleep 函数在 user.h 头文件中声明。

zombie.c > ...

```c
// Create a zombie process that
// must be reparented at exit.

#include "types.h"
#include "stat.h"
#include "user.h"

int
main(void)
{
  if(fork() > 0)
    sleep(5);  // Let child exit before parent.
  exit();
}
```

用户开发程序所需的接口都在 user.h 中，包含系统调用接口和 ulib 库函数。可以看到，sleep 函数是一个系统调用。

user.h > uptime(void)

```c
struct stat;

// system calls
int fork(void);
int exit(void) __attribute__((noreturn));
int wait(void);
int pipe(int*);
int write(int, void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(char*, int);
int mknod(char*, short, short);
int unlink(char*);
int fstat(int fd, struct stat*);
int link(char*, char*);
int mkdir(char*);
int chdir(char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
```

但从 user.h 的声明出发，依然无法找到有关 sleep 函数的定义。实际上，sleep 一类

的系统调用是通过 usys.S 汇编实现的。
汇编文件如下。可以看到，文件里定义了一个宏，对于每个系统调用，都展开一段代码。

```
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
```

对 SYSCALL(sleep)，生成的汇编如下。

```
.globl sleep; \
sleep: \
  movl $SYS_sleep, %eax; \
  int $T_SYSCALL; \
  ret
```

其中 SYS_sleep 和 T_SYSCALL 分别在 syscall.h 和 traps.h 有宏定义，分别代表系统调用号和中断号。xv6 必须设置硬件在遇到 int 指令时进行一些特殊的操作，这些操作会使处理器产生一个软中断。x86 允许 256 个不同的中断。其中 64 作为系统调用的中断号。

```
syscall.h:14:#define SYS_sleep  13
traps.h:27:#define T_SYSCALL       64      // system call
```

实际上，最终生成的汇编是如下形式。这段汇编先将 sleep 函数的系统调用号 SYS_sleep 放入%eax 寄存器中，然后执行 int $0x40 陷入内核。

```
.globl sleep; \
sleep: \
  movl $0xD, %eax; \
  int $0x40; \
  ret
```

执行 int 指令后，硬件会使用任务段中指定的栈（这个栈在内核模式中建立），来保存寄存器的值。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。内陷发生时，如果处理器在用户模式下运行，它会从任务段描述符中加载 %esp 和 %ss，把老的%ss 和 %esp 压入新的栈中。接下来会把 %eflags，%cs，%eip 压栈。对于某些内陷来说，处理器会压入一个错误字。而后，处理器从相应 IDT 表项中加载新的 %eip 和 %cs。

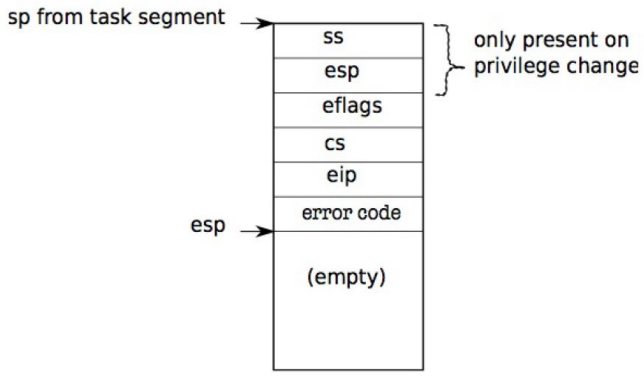

Figure 3-1. Kernel stack after an int instruction.

根据 xv6 的规定，此例中，硬件得到触发中断的原因是系统调用（64），**通过寻找在内存中保存的中断向量表来得到中断处理程序的入口地址，v 进而将控制权交给中断处理程序。**

中断向量表的初始化在 main 函数中调用 tvinit 函数完成。如下图所示，tvinit 设置了 idt 表（中断描述符表）中的 256 个表项。中断 i 被位于 vectors[i] 的代码处理。对于 T_SYSCALL，SETGATE 函数的第 2 个参数为 1 指示这是一个陷阱门。

```
17  void
18  tvinit(void)
19  {
20    int i;
21
22    for(i = 0; i < 256; i++)
23      SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
24    SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
25
26    initlock(&tickslock, "time");
27  }
```

```
#define SETGATE(gate, istra
{
  (gate).off_15_0 = (uint)(
  (gate).cs = (sel);
  (gate).args = 0;
  (gate).rsv1 = 0;
  (gate).type = (istrap) ? STS_TG32 : STS_IG32;
```

```
#define STS_TG32    0xF     // 32-bit Trap Gate

// A virtual address 'la' has a three-part structure as follows:

#define STS_TG32 0xF
32-bit Trap Gate
```

```
struct gatedesc {
  uint off_15_0 : 16;   // low 16 bits of offset in segment
  uint cs : 16;         // code segment selector
```

vector[i]

对于 vectors 数组的生成，xv6 使用一个 perl 脚本（2950—vectors.pl）来产生 IDT 表项指向的中断处理函数入口点。如前所述，当中断发生，寄存器被压入内核栈后，会跳转到 vectors64（vectors.pl 经过 make 后生成的中断向量表中）的地方开始执行，对应的汇编代码如下。

```
.globl vector64
vector64:
  pushl $0
  pushl $64
  jmp alltraps
```

可以看到，这段汇编先将一个错误码 0 压入栈中（如果 CPU 没有压入的话），再压入中断号（trapno），然后跳转到 alltraps。下面是 trapasm.S 的代码，即 alltraps 所在的地方。

```
#include "mmu.h"

  # vectors.S sends all traps here.
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal

  # Set up data and per-cpu segments.
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es
  movw $(SEG_KCPU<<3), %ax
  movw %ax, %fs
  movw %ax, %gs

  # Call trap(tf), where tf=%esp
  pushl %esp
  call trap
  addl $4, %esp

  # Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```

经过一系列的压栈操作后，调用 trap 函数，所有栈顶指针是 trap 函数（trap.c）的参数。可以看到，执行 trap 函数后，在第一个 if 判断中，调用 syscall()函数。

```
//PAGEBREAK: 41
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(proc->killed)
      exit();
    proc->tf = tf;
    syscall();
    if(proc->killed)
      exit();
    return;
  }
```

syscall 函 s 数在 syscall.c 中有定义，syscall.c 代码如下。

```
extern int sys_sleep(void);
extern int sys_unlink(void);
extern int sys_wait(void);
extern int sys_write(void);
extern int sys_uptime(void);

static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
};

void
syscall(void)
{
  int num;

  num = proc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    proc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            proc->pid, proc->name, num);
    proc->tf->eax = -1;
  }
}
```

首先拿到eax寄存器中的系统调用号

查对应函数的入口地址

接下来执行函数 sys_sleep，它在 sysproc.c 定义如下。Sys_sleep 主要功能是将用户栈上的参数——sleep 的时长，拉取到内核栈中（主要通过 argint 函数实现）。

```
int
sys_sleep(void)
{
  int n;
  uint ticks0;

  if(argint(0, &n) < 0)
    return -1;
  acquire(&tickslock);
  ticks0 = ticks;
  while(ticks - ticks0 < n){
    if(proc->killed){
      release(&tickslock);
      return -1;
    }
    sleep(&ticks, &tickslock);
  }
  release(&tickslock);
  return 0;
}
```

接着加锁，去实现内核具体的实现函数 sleep，它在 proc.c 中有如下定义。

```
// Atomically release lock and sleep on chan.
// Reacquires lock when awakened.
void
sleep(void *chan, struct spinlock *lk)
{
  if(proc == 0)
    panic("sleep");

  if(lk == 0)
    panic("sleep without lk");

  // Must acquire ptable.lock in order to
  // change p->state and then call sched.
  // Once we hold ptable.lock, we can be
  // guaranteed that we won't miss any wakeup
  // (wakeup runs with ptable.lock locked),
  // so it's okay to release lk.
  if(lk != &ptable.lock){  //DOC: sleeplock0
    acquire(&ptable.lock);  //DOC: sleeplock1
    release(lk);
  }

  // Go to sleep.
  proc->chan = chan;
  proc->state = SLEEPING;
  sched();

  // Tidy up.
  proc->chan = 0;

  // Reacquire original lock.
  if(lk != &ptable.lock){  //DOC: sleeplock2
    release(&ptable.lock);
    acquire(lk);
  }
}
```

检查是否存在当前进程

检查sleep是否持有锁

放入等待队列

当前进程转为SLEEPING状态

释放CPU

执行完一系列系统调用，从内核态返回，会调用 trapasm.S 的 trapret，恢复运行现场。

```
  # Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```

（2）

查阅 xv6 中文手册，可以发现时钟中断每次持续 1s/100 = 10ms.

我们来看一看分时硬件和时钟中断。我们希望分时硬件大约以每秒 100 次的速度产生一个中断，这样内核就可以对进程进行时钟分片。100 次每秒的速度足以提供良好的交互性能并且同时不会使处理器进入不断的中断处理中。

每次中断持续10ms

回到 sys_sleep 函数，可以推测睡眠持续了 5 个时钟周期，即 **50ms**。

```
int
sys_sleep(void)
{
  int n;
  uint ticks0;

  if(argint(0, &n) < 0)
    return -1;
  acquire(&tickslock);
  ticks0 = ticks;
  while(ticks - ticks0 < n){
    if(proc->killed){
      release(&tickslock);
      return -1;  持续5个时钟周期
    }
    sleep(&ticks, &tickslock);
  }
  release(&tickslock);
  return 0;
}
```

5

对于底层实现，中文文档有如下说明。

> 时钟芯片是在 LAPIC 中的，所以每一个处理器可以独立地接收时钟中断。xv6 在 lapicinit（6651）中设置它。关键的一行代码是 timer （6664）中的代码，这行代码告诉 LAPIC 周期性地在 IRQ_TIMER（也就是 IRQ 0）产生中断。第 6693 行打开 CPU 的 LAPIC 的中断，这使得 LAPIC 能够将中断传递给本地处理器。

```
#define TICR    (0x0380/4)   // Timer Initial Count
#define TCCR    (0x0390/4)   // Timer Current Count
#define TDCR    (0x03E0/4)   // Timer Divide Configuration

volatile uint *lapic;  // Initialized in mp.c

static void
lapicw(int index, int value)
{
  lapic[index] = value;
  lapic[ID];  // wait for write to finish, by reading
}
//PAGEBREAK!

void
lapicinit(void)
{
  if(!lapic)
    return;

  // Enable local APIC; set spurious interrupt vector.
  lapicw(SVR, ENABLE | (T_IRQ0 + IRQ_SPURIOUS));

  // The timer repeatedly counts down at bus frequency
  // from lapic[TICR] and then issues an interrupt.
  // If xv6 cared more about precise timekeeping,
  // TICR would be calibrated using an external time source.
  lapicw(TDCR, X1); // 设置时钟频率
  lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER)); 设置定期计数模式
  lapicw(TICR, 10000000); // 写入初始计数值，减为0时产生中断，再次计数
```

10000000 是指时钟周期数

系统调用	描述
fork()	创建进程
exit()	结束当前进程
wait()	等待子进程结束
kill(pid)	结束 pid 所指进程
getpid()	获得当前进程 pid
sleep(n)	睡眠 n 秒

++

其他（例如感想、建议等等）。

通过本次实验，我们围绕 scheduler 函数展开的一系列的讨论，更加深入了解了 xv6 内核处理机调度的实现以及其实现同步的方式。

本次实验还深入探究 sleep 函数的实现，说明如何通过 C 语言接口的 sleep 最终进入到 sleep 的核心实现。

本次实验进一步加强了我们分析、设计操作系统核心的能力。

指导教师批阅意见：   成绩评定：   指导教师签字：谭舜泉 2021 年 6 月 20 日
备注：

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。

2、教师批改学生实验报告时间应在学生提交实验报告时间后 10 日内。