深圳大学
Shenzhen University

操作系统

xv6实验题目解答-3

计算机与软件学院

大作业Part I-4

- (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？
- Kpgdir指向了一个能存放1024个页表项的页面。每个页表项指向一个内层页表，里面能存放1024个4K物理块的首字节地址。因此：
  - 1K*1K*4K=4G虚拟地址空间。

大作业Part I-4

- (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。
- 注意在kmap中，phys_start和phys_end都是uint类型的。
  - #define DEVSPACE 0xFE000000
  - 在mappages中，size=k->phys_end - k->phys_start
  - 可以理解为0x100000000-0xFE000000=0x2000000=32M
  - 也即是从DEVSPACE指向的那个字节开始直到4G虚拟内存末尾的32M空间。

大作业Part I-4

- (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？

- L134，setupkvm()中调用kalloc分配了一个页面用于存放外层页表。跟着遍历kmap，对于每个内层页表都调用了一次kalloc。因此调用了几次kalloc就和kmap中到底指定了多少虚拟内存有关。

大作业Part I-4

- 注意#define KERNLINK (KERNBASE+EXTMEM)
- 因此前三个段在虚拟空间中实际上是连在一起的。其空间尺寸为#define PHYSTOP 0xE000000。
- 0xE000000=224MB。
- 一个内层页表对应于1024*4K=4MB。
- (224+32)/4=64个内层页表，所以共有64+1=65个kalloc。

```
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
};
```

Lab4实验题目1

- 回答以下问题：
  - kmem中的freelist指针指向空闲物理块链表。空闲物理块链表中的节点为run结构体。但是：

```
struct run {
  struct run *next;
};
```

  - 可以看到这个结构体只有指向下一个节点的指针。请解释这个链表中的空闲物理块保存在哪里呢？

Lab4实验题目解答

- kmem结构体定义在kalloc.c L23；kmem.freelist是空闲物理块链表。看kfree函数。

```
void kfree(char *v)//参数为指向空闲物理块第一个字节的指针
{
  struct run *r;
  //...
  memset(v, 1, PGSIZE);//这个物理块的所有内容置1，抹去原有内容
  //...
  //注意！通过强制类型转换，这个v指针所指向的内存被用来存放run结构体的内容
  //也即使用空闲物理块本身来存放run结构体！
  //next这个指针，也即存放下一个空闲物理块首地址的
  //next变量是放在当前空闲物理块中的。
  r = (struct run*)v;
  //next指针被赋值为freelist指针的值，也即原来的空闲物理块链表中的第一个物理块的首字节地址
  r->next = kmem.freelist;
  //freelist指向r所在的位置，也即v指针指向的物理块的首地址
  kmem.freelist = r;
  if(kmem.use_lock)
    release(&kmem.lock);
}
```

Lab4实验题目2

- 大作业partI-4其中一个问题：
- (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？
- 请编程验证大作业partI-4的理论推导结果，在终端中输出在setupkvm函数中调用kalloc函数的次数。

Lab4实验题目解答

```
xv6...
cpu1: starting
The number of the calls of kalloc in setupkvm is 65
cpu0: starting
The number of the calls of kalloc in setupkvm is 65
init: starting sh
The number of the calls of kalloc in setupkvm is 65
The number of the calls of kalloc in setupkvm is 65
```

Lab4实验题目解答（vm.c）

```
@@ -10,6 +10,8 @@
 extern char data[];  // defined by kernel.ld
 pde_t *kpgdir;  // for use in scheduler()
 struct segdesc gdt[NSEGS];
+//accumulate the number of the calls of kalloc in setupkvm
+uint no_of_calls_of_kalloc;

 // Set up CPU's kernel segment descriptors.
 // Run once on entry on each CPU.
@@ -131,6 +133,9 @@
   pde_t *pgdir;
   struct kmap *k;

+  //initialize the accumulator. --tansq
+  no_of_calls_of_kalloc=0;
+
   if((pgdir = (pde_t*)kalloc()) == 0)
     return 0;
   memset(pgdir, 0, PGSIZE);
@@ -140,6 +145,7 @@
     if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                 (uint)k->phys_start, k->perm) < 0)
       return 0;
+  cprintf("The number of the calls of kalloc in setupkvm is %d\n", no_of_calls_of_kalloc);
   return pgdir;
 }
```

Lab4实验题目解答（kalloc.c）

```
@@ -9,9 +9,14 @@
 #include "mmu.h"
 #include "spinlock.h"

+
 void freerange(void *vstart, void *vend);
 extern char end[]; // first address after kernel loaded from ELF file

+//accumulate the number of the calls of kalloc in setupkvm
+extern uint no_of_calls_of_kalloc;
+
+
 struct run {
   struct run *next;
 };
@@ -91,6 +96,8 @@
     kmem.freelist = r->next;
   if(kmem.use_lock)
     release(&kmem.lock);
+
+  no_of_calls_of_kalloc++;
   return (char*)r;
 }
```

大作业Part I-5

- 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？
- 在对 swtch 的调用的整个过程中,xv6 都持有锁 ptable.lock : swtch 的调用者必须持有该锁,并将锁的控制权转移给切换代码。
- ptable.lock 会保证进程的 state 和 context 在运行 swtch 时保持不变。

大作业Part I-5 (proc.c)

```
void
scheduler(void)
{
  struct proc *p;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, proc->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      proc = 0;
    }
    release(&ptable.lock);

  }
}
```

大作业Part I-5 (proc.c)

- 注意在yield里面，sched之后CPU已经让给调度器了，因此对于ptable.lock的release要等到当前进程重新获得CPU使用权之后。

```
// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  proc->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

大作业Part I-5 (proc.c)

- 在exit函数中，只有ptable.lock的锁定。

```
iput(proc->cwd);
proc->cwd = 0;

acquire(&ptable.lock);

// Parent might be sleeping in wait().
wakeup1(proc->parent);

// Pass abandoned children to init.
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->parent == proc){
    p->parent = initproc;
    if(p->state == ZOMBIE)
      wakeup1(initproc);
  }
}

// Jump into the scheduler, never to return.
proc->state = ZOMBIE;
sched();
panic("zombie exit");
```

大作业Part I-5 (proc.c)

- 在wait函数中，也是这样。进入sleep之后并没有释放ptable.lock。

```
int
wait(void)
{
  struct proc *p;
  int havekids, pid;

  acquire(&ptable.lock);

  。 。 。 。 。 。

    // Wait for children to exit.  (See wakeup1 call in proc_exit.)
    sleep(proc, &ptable.lock);  //DOC: wait-sleep
  }
}
```

大作业Part I-5 (proc.c)

- scheduler在内层遍历ptable的for循环前锁定ptable.lock，在结束循环后释放ptable.lock。
- 它在选定一个RUNNALBE的进程，让其获得CPU的使用权的过程中会一直保持ptable.lock的锁定，在被选定的进程获得CPU使用权后：
  - 如果进程之前是通过yield进入RUNNABLE状态的话，它将重新进入L317的代码，在此释放ptable.lock。
  - 如果进程之前是通过sleep进入SLEEPING状态的话，它将重新进入L365往下的代码，释放ptable.lock。
  - 如果进程是新进入CPU的RUNNABLE进程呢？

大作业Part I-5 (proc.c)

- L71，在allocproc，创建新进程时，已经指定了scheduler第一调用这个进程的入口为forkret。
- L327，在forkret中释放掉对于ptable.lock的锁定。
- 对于进程而言，在把CPU的使用权还回给scheduler之前，锁定ptable.lock。如yield函数(L314)，sleep函数(L358)。

大作业Part I-5 (proc.c)

- Thus the first 6 kB (NDIRECT×BSIZE) bytes of a file can be loaded from blocks listed in the inode, while the next 64kB (NINDIRECT×BSIZE) bytes can only be loaded after consulting the indirect block.
  - #define BSIZE 512 // block size
  - #define NDIRECT 12
  - #define NINDIRECT (BSIZE / sizeof(uint))
- 512*12=6KB；512/4*512=128*512=64KB。

大作业Part I-6

- 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？
- sleep函数的声明在user.h(L23)，定义在usys.S(L30)。
- SYSCALL(sleep)对应的宏展开为：
- .globl sleep; sleep: movl $SYS_sleep, %eax; int $T_SYSCALL; ret
- 这相当于定义了sleep函数的代码，这份代码进行了系统调用，调用码为SYS_sleep (定义在syscall.h)

大作业Part I-6

- 系统调用发起后，进入alltraps标号指向的代码(trapasm.S)，然后调用trap函数(trap.c)，因为陷入类型为T_SYSCALL，进入syscall函数。
- 看一下syscall函数上方的函数指针数组，调用的是sys_sleep函数(sysproc.c)。在里面调用了核心函数sleep。

大作业Part I-6

- 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。
- 大致睡眠50毫秒。

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

  if(argint(0, &n) < 0)//取出参数，这里n=5
    return -1;
  acquire(&tickslock);//获取锁
  //时钟中断每秒100次，每次10毫秒
  //ticks是当前计时，每10毫秒+1，ticks0记录了运行到这一行时的当前计时
  ticks0 = ticks;
  while(ticks - ticks0 < n){//只有ticks比ticks0大于等于5才退出
    if(proc->killed){
      release(&tickslock);
      return -1;
    }//期间不断睡眠，也即让出CPU，每次时钟中断都会被wakeup
    sleep(&ticks, &tickslock);
  }
  release(&tickslock);
  return 0;
}
```