代码阅读3

蒲伟良 1600011338

代码阅读3

1. 什么是临界区？什么是同步和互斥？什么是竞争状态？临界区操作时中断是否应该开启？中断会有什么影响？以上概念在XV6中是否实现，是如何实现的?
2. XV6 的锁是如何实现的，有什么操作？xchg 是什么指令，该指令有何特性?
3. 基于 XV6 的 spinlock，请给出实现信号量、读写锁、信号机制的设计方案（三选一，请写出相应的伪代码）
4. XV6 初始化之后到执行 main.c 时，内存布局是怎样的（其中已有哪些内容）？
5. XV6 的动态内存管理是如何完成的？ 有一个 kmem（链表），用于管理可分配的物理内存页。（vend=0x00400000，也就是可分配的内存页最大为 4Mb）
6. XV6 的虚拟内存是如何初始化的？画出 XV6 的虚拟内存布局图，请说出每一部分对应的内容是什么。 见 memlayout.h 和 vm.c 的 kmap 上的注释
7. 关于 XV6 的内存页式管理。发生中断时，用哪个页表？一个内页是多大？页目录有多少项？页表有多少项？最大支持多大的内存？画出从虚拟地址到物理地址的转换图。在 XV6 中，是如何将虚拟地址与物理地址映射的（调用了哪些函数实现了哪些功能）？
8. 其他你认为有趣有价值的问题。

    再谈sleep

1. 什么是临界区？什么是同步和互斥？什么是竞争状态？临界区操作时中断是否应该开启？中断会有什么影响？以上概念在XV6中是否实现，是如何实现的?

- 临界区是进程中互斥执行的一段代码，如果两个进程同时执行会有冲突，造成一些不可预测的错误。
- 同步是指两个进程共享的资源、变量需要保持相同，不然就不是共享同一个资源了，并且两个进程能够按照一定的次序先后访问共享资源，比如说一个进程先写再让另一个进程读。因此两个进程不能同时访问某些资源，形成了互斥，一个访问的时候另一个不能访问。
- 竞争状态指两个进程同时意图进入临界区，但最终只有一个能够进入临界区，造成了竞争。
- 临界区操作时不应该开启中断，因为不知道中断处理程序是否需要共享相同的资源，如果中断处理程序也要进入同一个临界区，那么中断处理程序需要等待，但是已经进入临界区的进程因为中断阻塞无法退出临界区，最终死锁。
- 以上概念在XV6中有实现，利用锁机制，在`spinlock.h`和`spinlock.c`中实现。
- 实现的方法是每个共享资源都用一个锁变量来保护，任意进程要进入临界区首先要获得唯一的锁，否则会忙等待或者休眠，退出临界区时会释放锁，让其他等待的进程可以获得。因为一个锁只能被一个进程获得，所以两个进程同时意图获得锁时会竞争。获得锁后屏蔽中断，释放锁后会取消屏蔽。

2. XV6 的锁是如何实现的，有什么操作？xchg 是什么指令，该指令有何特性?

- XV6中有两种锁，spinlock（自旋锁）和sleeplock（睡眠锁），分别在`spinlock.h`，`spinlock.c`，`sleeplock.h`和`sleeplock.c`中。XV6中`sleeplock`是基于`spinlock`的，可以说是继承关系，区别在于`spinlock`在锁已经被其他进程获得的时候会一直尝试获得，处于忙等待，而`sleeplock`会在获取锁失败时睡眠，在获得锁的进程释放时再发出信号唤醒。

- `spinlock`的定义如下，主要的是一个整型变量`locked`，用于表示锁的状态（打开/关闭），其他的名字、持有该锁的CPU、获取了锁的函数的调用栈都是用于debug，不影响操作系统的运行。

```
// in spinlock.h

// Mutual exclusion lock.
struct spinlock {
  uint locked;       // Is the lock held?

  // For debugging:
  char *name;        // Name of lock.
  struct cpu *cpu;   // The cpu holding the lock.
  uint pcs[10];      // The call stack (an array of program counters)
                     // that locked the lock.
};
```

- `spinlock`有4个操作，分别是`initlock`（初始化锁），`acquire`（获得锁），`release`（释放锁）和`holding`（检查是否持有锁）。初始化锁非常简单，就是设置锁为打开（0），然后设置初始化名字和CPU。

```
// in spinlock.c
void
initlock(struct spinlock *lk, char *name)
{
  lk->name = name;
  lk->locked = 0;
  lk->cpu = 0;
}
```

- `holding()`用于检查当前的CPU是否持有指定的锁，由于要访问锁的状态，所以需要先屏蔽中断（防止访问中途其他进程改变锁的状态），访问后取消屏蔽，这里用`pushcli()`屏蔽中断，用`popcli()`取消屏蔽，两个函数是成对使用的，并且是汇编代码`cli`和`sti`的包装函数。

```
// in spinlock.c
// Check whether this cpu is holding the lock.
int
holding(struct spinlock *lock)
{
  int r;
  pushcli();
  r = lock->locked && lock->cpu == mycpu();
  popcli();
  return r;
}
```

- 下面的是`pushcli()`和`popcli()`，在`cli`和`sti`的基础上设置了标志位的检查，另外进行了计数，保证`pushcli()`和`popcli()`一一配对，即只有所有锁被释放才会允许中断。

```
//in spinlock.c
void
pushcli(void)
```

```
{
  int eflags;

  eflags = readeflags();
  cli();
  if(mycpu()->ncli == 0)
    mycpu()->intena = eflags & FL_IF;
  mycpu()->ncli += 1;
}

void
popcli(void)
{
  if(readeflags()&FL_IF)
    panic("popcli - interruptible");
  if(--mycpu()->ncli < 0)
    panic("popcli");
  if(mycpu()->ncli == 0 && mycpu()->intena)
    sti();
}
```

- 接下来是`acquire()`，用于获得锁，该函数逻辑为
  - `pushcli()`屏蔽中断
  - 检查是否已经持有锁（如果是即程序有bug）
  - 用汇编指令`xchg`获得锁，如果获取失败会一直循环获取
  - `__sync_synchronize()`是一条特殊的指令，可以理解为一个栅栏，作用是告诉编译器和硬件（CPU），这条语句上下的`store`和`load`指令在改变顺序（因为优化）时不允许越过这条语句，即上面的只能留在上面，下面的只能留在下面，这样做保证了`acquire()`是先获得锁再改变锁中记录的持有锁的CPU编号。
  - 在锁中记录持有锁的CPU
  - 通过`%ebp`链获取当前的调用者栈，记录在`pcs[]`中，主要用于debug
- `xchg`是一条特殊的原子操作指令，专门用于进程同步，实现诸如锁、信号量等机制。`xchg`交换两个变量（寄存器）的值，因为是原子操作，所以交换过程不会被打断。这里`xchg`交换后返回第一个变量原来的值，XV6利用这个特性来用`xchg`获得锁，即循环交换锁的状态和1，然后对返回值进行判断：
  - 如果锁原来的状态为0（打开），交换后锁的状态为1，变为关闭，即进程已经获得锁，因为返回值为0，所以`while`循环会停下。
  - 如果锁原来的状态是1（关闭），即其他进程持有锁，那么交换后锁的状态不变，同时返回值为1，`while`循环继续，当前进程没有获得锁，一直尝试直到获得锁。

```
// in spinlock.c
// Acquire the lock.
// Loops (spins) until the lock is acquired.
// Holding a lock for a long time may cause
// other CPUs to waste time spinning to acquire it.
void
acquire(struct spinlock *lk)
{
  pushcli(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");

```

```
  // The xchg is atomic.
  while(xchg(&lk->locked, 1) != 0)
    ;

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that the critical section's memory
  // references happen after the lock is acquired.
  __sync_synchronize();

  // Record info about lock acquisition for debugging.
  lk->cpu = mycpu();
  getcallerpcs(&lk, lk->pcs);
}
```

- 释放锁`release()`，函数逻辑为
  - 检查是否已持有锁，避免出错
  - 初始化`pcs[]`和`cpu`，因为不需要了
  - 设置栅栏`__sync_synchronize()`
  - 用汇编指令`mov`设置锁状态为0，即释放锁，使用汇编指令而不是赋值语句保证这是原子操作，不会被打断或者调整顺序。
  - `popcli()`允许中断（如果是最后一把锁）

```
// in spinlock.c
// Release the lock.
void
release(struct spinlock *lk)
{
  if(!holding(lk))
    panic("release");

  lk->pcs[0] = 0;
  lk->cpu = 0;

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that all the stores in the critical
  // section are visible to other cores before the lock is released.
  // Both the C compiler and the hardware may re-order loads and
  // stores; __sync_synchronize() tells them both not to.
  __sync_synchronize();

  // Release the lock, equivalent to lk->locked = 0.
  // This code can't use a C assignment, since it might
  // not be atomic. A real OS would use C atomics here.
  asm volatile("movl $0, %0" : "+m" (lk->locked) : );

  popcli();
}
```

- 从锁的获得和释放可以看到XV6不允许在中断的时候有持有任何锁，所以在获得第一把锁之前就要屏蔽中断，只有最后一把锁释放后才允许中断，这样非常简单粗暴地避免了中断处理程序和原来的程序的临界区冲突，避免了死锁。

- 除了`spinlock`，XV6中还有`sleeplock`，其在获得锁失败时睡眠，从而将CPU让给其他进程，避免空转，对称地，释放锁时需要唤醒睡眠的进程。

```
//in sleeplock.h
// Long-term locks for processes
struct sleeplock {
  uint locked;       // Is the lock held?
  struct spinlock lk; // spinlock protecting this sleep lock

  // For debugging:
  char *name;        // Name of lock.
  int pid;           // Process holding lock
};
//
```

- `sleeplock`定义如上，其中包含了一个`spinlock`，同时和`spinlock`类似，有表示锁的状态的变量`locked`和用于debug的`name`和`pid`。
- `sleeplock`同样是有4个操作，其中`initsleeplock`和`holdingsleep`非常简单，和`spinlock`中相似，不展开。

```
//in sleeplock.c
void
acquiresleep(struct sleeplock *lk)
{
  acquire(&lk->lk);
  while (lk->locked) {
    sleep(lk, &lk->lk);
  }
  lk->locked = 1;
  lk->pid = myproc()->pid;
  release(&lk->lk);
}

void
releasesleep(struct sleeplock *lk)
{
  acquire(&lk->lk);
  lk->locked = 0;
  lk->pid = 0;
  wakeup(lk);
  release(&lk->lk);
}
```

- 上面的代码是`sleeplock`的获取与释放，`sleeplock`中包含的`spinlock` `lk`用来保护变量`locked`，即睡眠锁的状态，`acquiresleep()`循环检查`locked`的值（这里不需要保证是原子操作，因为已经进入了临界区），如果锁已经被其他进程持有则进程休眠。与之相对的是`releasesleep()`在释放锁后会唤醒需要该睡眠锁的其他进程。
- 这里虽然是睡眠锁，但其实依然会在获取`spinlock`时忙等待，但是`spinlock`和`sleeplock`在不同的情景下使用，一般认为一个进程获取自旋锁后执行几条指令就会释放锁，而睡眠锁可能会执行大量的指令才释放，所以相比于等待其他进程释放睡眠锁，获取自旋锁的等待时间非常短，可忽略不计。

3. 基于 XV6 的 spinlock，请给出实现信号量、读写锁、信号机制的设计方案（三选一，请写出相应的伪代码）

- 这里给出基于`spinlock`实现的信号量

```
class semaphore {
private:
  int sem;
  spinlock lock;                          //protect sem
  Waitqueue q;
public:
  semaphore() {                           //contructor
    initlock(&lock, "semaphore_xxxx");
    sem = N;
    q.clear();
  }

  void P() {
    acquire(&lock);
    sem--;
    if (sem < 0) {
      add this process to q;
      sleep();
    }
    release(&lock);
  }

  void V() {
    acquire(&lock);
    sem++;
    if (sem <= 0) {
      remove t from q;
      wakeup(t);
    }
    release(&lock);
  }
};
```

- 主要思路就是用`spinlock`保护信号量sem，同一时间只有一个进程可以获得锁，即可以访问`sem`，然后修改`sem`，显然`P()`和`V()`可以看成原子操作。但是因为同一时间只有一个进程可以获得锁，而且当`sem`小于0时进程休眠，而看起来不会释放锁，所以实际上需要在`sleep()`中释放锁，放弃CPU，重新调度回来后再获得锁。所以现在不难理解XV6中`sleep()`需要两个参数，一个是用于唤醒的标志，另一个是一个锁。XV6的锁最多只有两层，所以只需要传递一个锁就足够了。下面的代码是考虑了XV6中提供的进程接口`sleep()`和`wakeup()`后的实现。

```
class semaphore {
private:
  int sem;
```

```
  spinlock lock;                          //protect sem
public:
  semaphore() {                           //contructor
    initlock(&lock, "semaphore_xxxx");
    sem = N;
  }

  void P() {
    acquire(&lock);
    sem--;
    if (sem < 0) sleep(&sem, &lock);
    release(&lock);
  }

  void V() {
    acquire(&lock);
    sem++;
    if (sem <= 0) wakeup(&sem);
    release(&lock);
  }
};
```

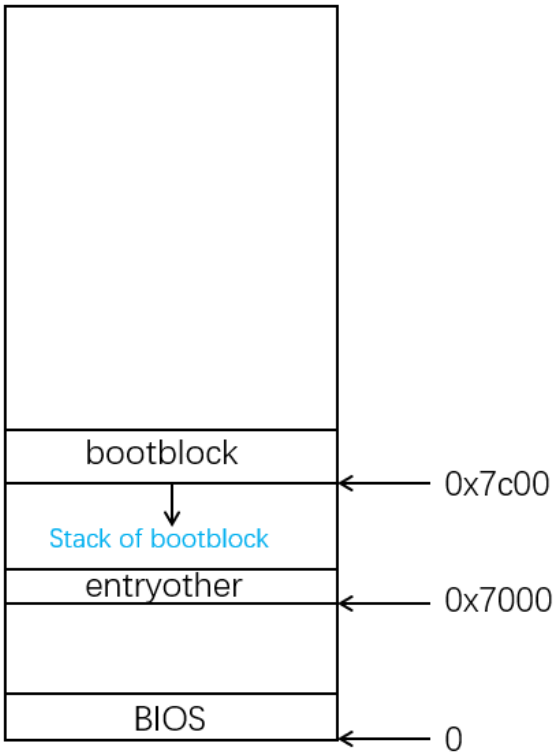
4. XV6 初始化之后到执行 main.c 时，内存布局是怎样的（其中已有哪些内容）？

- xv6初始化后到执行`main.c`前经历了三个文件，分别是`bootasm.S`，`bootmain.c`和`entry.S`。另一个和内存布局相关的是文件`Makefile`，该文件指导了链接器如何工作。

```
#in Makefile
bootblock: bootasm.S bootmain.c
    $(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c bootmain.c
    $(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S
    $(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o bootmain.o
    $(OBJDUMP) -S bootblock.o > bootblock.asm
    $(OBJCOPY) -S -O binary -j .text bootblock.o bootblock
    ./sign.pl bootblock

entryother: entryother.S
    $(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c entryother.S
    $(LD) $(LDFLAGS) -N -e start -Ttext 0x7000 -o bootblockother.o entryother.o
    $(OBJCOPY) -S -O binary -j .text bootblockother.o entryother
    $(OBJDUMP) -S bootblockother.o > entryother.asm
```

- 这段代码节选展现了`Makefile`对编译器和链接器的指导，之前常说的bootloader在这里被称为`bootblock`，由`bootasm.S`和`bootmain.c`组成，被链接器链接后保存在`0x7c00`的地址，前面说了这事一个约定俗成的结果，`entryother`用于多核启动时其他CPU的初始化，由`entryother.S`编译来，放在`0x7000`的地址。所以到

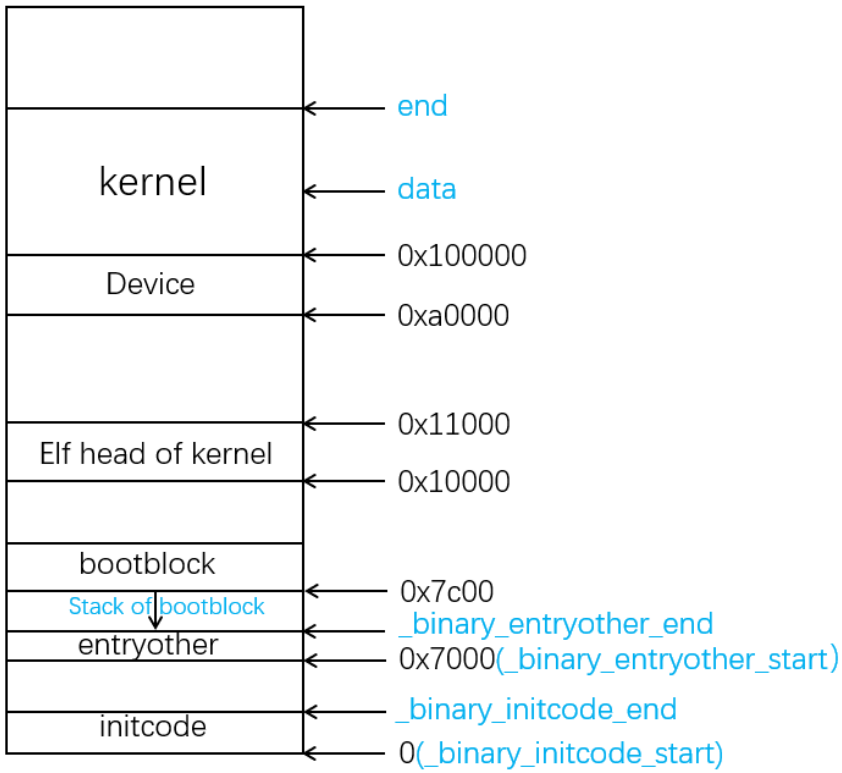
这里内存分布如下图。




```
initcode: initcode.S
    $(CC) $(CFLAGS) -nostdinc -I. -c initcode.S
    $(LD) $(LDFLAGS) -N -e start -Ttext 0 -o initcode.out initcode.o
    $(OBJCOPY) -S -O binary initcode.out initcode
    $(OBJDUMP) -S initcode.o > initcode.asm

kernel: $(OBJS) entry.o entryother initcode kernel.ld
    $(LD) $(LDFLAGS) -T kernel.ld -o kernel entry.o $(OBJS) -b binary initcode
entryother
    $(OBJDUMP) -S kernel > kernel.asm
    $(OBJDUMP) -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > kernel.sym
```

- 同时链接器还会将`initcode`保存到地址0处，最后将其他所有的文件（`$(OBJS)`）链接生成`kernel`，在`bootmain.c`中会复制`kernel`的ELF头部复制到内存地址`0x10000`处，然后通过ELF头将`kernel`的其他部分读到相应的内存地址中（在ELF头中的program header中记录了相应的虚拟地址和物理地址，由于还没有开启分页，会读取到物理地址中）。所以最终到`main.c`前的内存布局如下图（图中还添加了表示`kernel`的`.data`段

开头的`data`指针和表示`kernel`结尾的`end`指针，两者都是在`kernel.ld`中定义）。



- XV6同时使用了段机制和分页机制，但是基本上是使用分页机制管理，分段基本没怎么用，个人感觉只是使用了分段里的权限保护，对于内存管理其实没有什么贡献。在运行到`main.c`前在`bootasm.S`中载入了一个段表。

```
# in bootasm.S
  lgdt    gdtdesc

# Bootstrap GDT
.p2align 2                                # force 4 byte alignment
gdt:
  SEG_NULLASM                             # null seg
  SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
  SEG_ASM(STA_W, 0x0, 0xffffffff)         # data seg

gdtdesc:
  .word   (gdtdesc - gdt - 1)             # sizeof(gdt) - 1
  .long   gdt                             # address gdt
```

- `SEG_ASM()`在`asm.h`中定义，根据给出的type，base_address和limited填写相应的段描述符，由于只需要填写内核的代码段和内核数据段，所以非常简单地设置了一些参数（注意这里是小端）。
  - `lim>>12`是因为段是以4kb为单位（与后面提到的granularity_flag有关）
  - `type|0x90`让前四位为`1001`
    - 高位是present，置1表示在内存中
    - 中间两位是DPL，内核设置为0
    - 低位是区分system和code/data，因为是code或者data，置为1（system为0）
  - type有4位，在code/data的前提下
    - 高位1为code，即`STA_X`，0为data

        - code的前提下，剩下3位是conforming（通用保护有关），read，access（在内存中?）
        - data的前提下，剩下3为是expand_down，write，access
    - `(lim>>28)|0xc0)`设置了前四位为`1101
        - 高位为granularity_flag，置为1表示段的单位是4kbytes（0表示单位为1bytes）
        - 第二位为D/B，大概是置为1表示是32位的段
        - 第三位是L位，置1表示64位，XV6是32位的，所以置0
        - 低位是Available and reserved bits，留给用户定义，这里定义为1

```
// The 0xC0 means the limit is in 4096-byte units
// and (for executable segments) 32-bit mode.
#define SEG_ASM(type,base,lim)                                  \
        .word (((lim) >> 12) & 0xffff), ((base) & 0xffff);      \
        .byte (((base) >> 16) & 0xff), (0x90 | (type)),         \
                (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)

#define STA_X     0x8       // Executable segment
#define STA_W     0x2       // Writeable (non-executable segments)
#define STA_R     0x2       // Readable (executable segments)
```

- 这里段的映射非常简单，第一段是空的（似乎是intel的体系决定的），第二段是内核代码段，起始地址是0，段界限是最大值4GB，第三段是内核数据段，同样是起始地址0，段界限4GB。这说明两者共用一段，同时32位机器在一个进程中最大的地址空间即为4GB，所以该段还使用了所有的空间，所以在其实有且仅有一段，即XV6的分段机制没有怎么利用起来。
- 因为XV6的分段非常简单，这里补充一点运行到`main`后的段表的初始化，初始化函数为`seginit()`

```
// in vm.c
void
seginit(void)
{
  struct cpu *c;

  // Map "logical" addresses to virtual addresses using identity map.
  // Cannot share a CODE descriptor for both kernel and user
  // because it would have to have DPL_USR, but the CPU forbids
  // an interrupt from CPL=0 to DPL=3.
  c = &cpus[cpuid()];
  c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
  c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
  c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
  c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
  lgdt(c->gdt, sizeof(c->gdt));
}
```

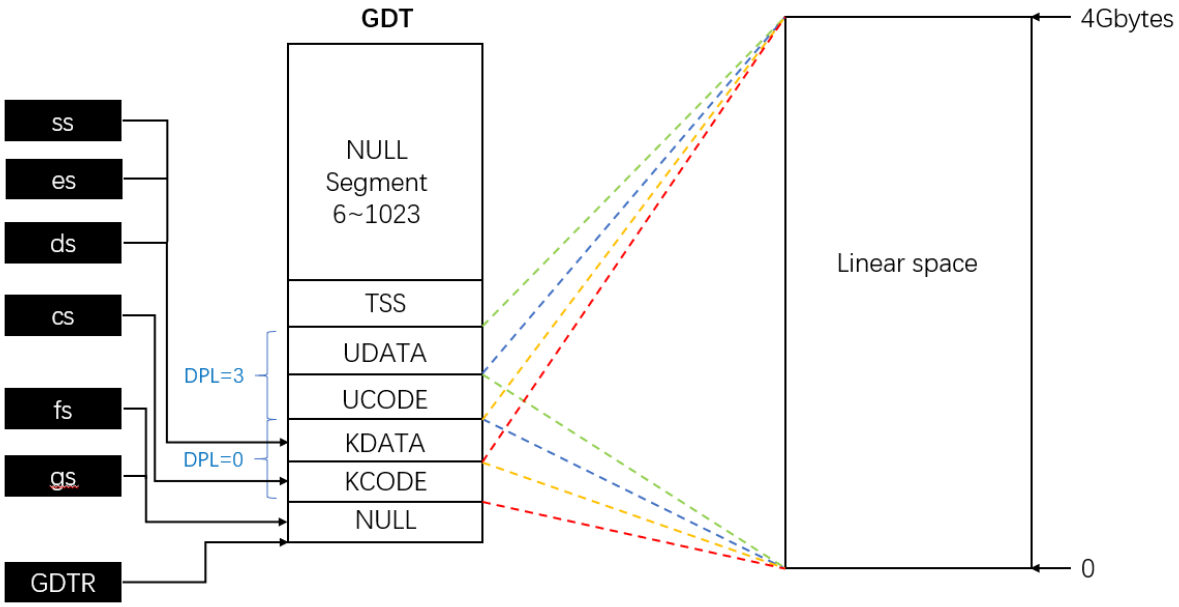
- 这里初始化了4段，在之前内核代码、数据段的基础上多了用户的代码和数据段，可以看到几乎一样，内核段和用户端的区别在与最后的参数DPL，内核为0，用户为`DPL_USER`即3。
- `SEG()`的作用和之前的`SEG_ASM()`几乎一致，区别是多了一个参数`dpl`，可以设置不同的权限。代码如下，可以看到其中定义了所有的段，总共只有6段，一头一尾的空的和任务状态段，中间的是内核、用户的代码、数据段。并且中间4段的起始地址、段界限一模一样。

```
// in mmu.h
```

```
// various segment selectors.
#define SEG_KCODE 1  // kernel code
#define SEG_KDATA 2  // kernel data+stack
#define SEG_UCODE 3  // user code
#define SEG_UDATA 4  // user data+stack
#define SEG_TSS   5  // this process's task state

// cpu->gdt[NSEGS] holds the above segments.
#define NSEGS     6

#ifndef __ASSEMBLER__
// 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
};

// Normal segment
#define SEG(type, base, lim, dpl) (struct segdesc)    \
{ ((lim) >> 12) & 0xffff, (uint)(base) & 0xffff,      \
  ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1,       \
  (uint)(lim) >> 28, 0, 0, 1, 1, (uint)(base) >> 24 }
#endif
```

- 因为起始地址都是0，翻译的时候都是逻辑的地址的偏移+0（如下图），即将逻辑地址直接转换为线性地址，所以其实XV6的分段机制基本是个幌子，个人感觉只是设置了权限保护，并且在中断的时候有用。对于内存管理起作用的主要是分页机制。下图给出了`seginit()`后的段的布局，如果是在`main.c`之前的话，会只有前面三段，即`NULL`的0段，`KCODE`和`KCODE`，在内核中`%cs`寄存器会一直指向`KCODE`，而其他的`%ds`,`%ss`和`%es`会

指向`KDATA`。



- 在`entry.S`中会载入一个页目录`entrypgdir`，根据`CR4`和`CR0`的标志位，处于32位分页的模式，翻译机制如图。

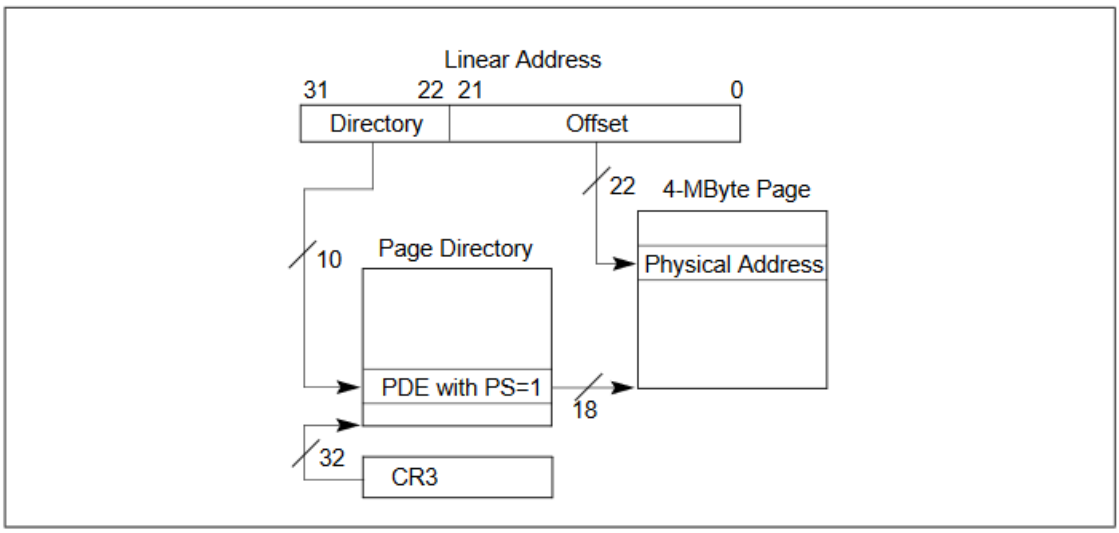


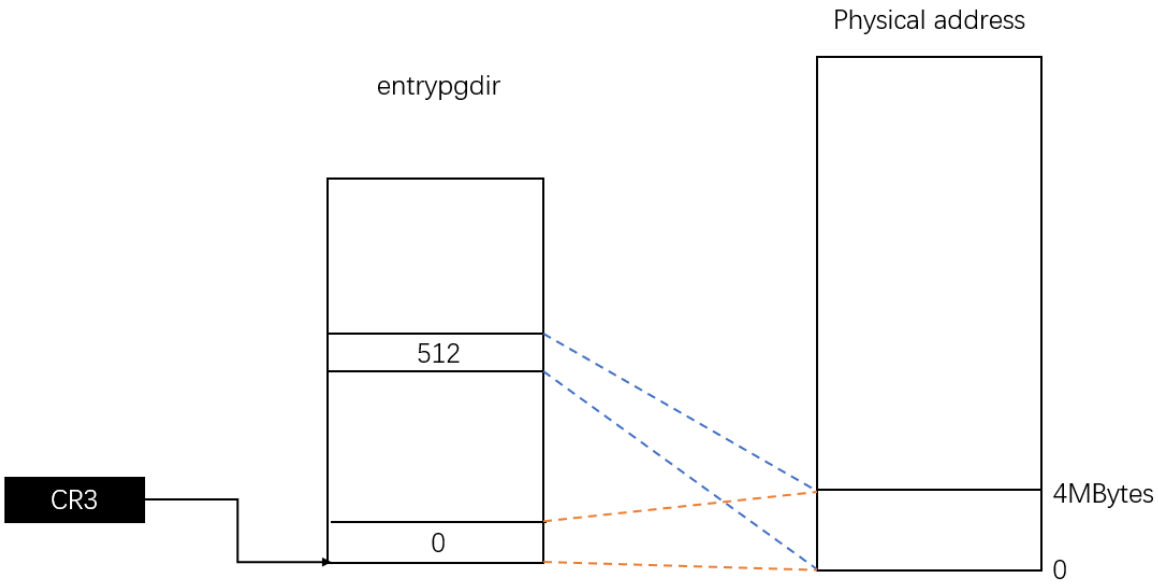
Figure 4-3. Linear-Address Translation to a 4-MByte Page using 32-Bit Paging

图片来源Intel® 64 and IA-32 Architectures Software Developer Manuals

- 而在`entrpgdir`在`main.c`中定义

```
// in main.c
// The boot page table used in entry.S and entryother.S.
// Page directories (and page tables) must start on page boundaries,
// hence the __aligned__ attribute.
// PTE_PS in a page directory entry enables 4Mbyte pages.

__attribute__((__aligned__(PGSIZE)))
pde_t entrypgdir[NPDENTRIES] = {
  // Map VA's [0, 4MB) to PA's [0, 4MB)
  [0] = (0) | PTE_P | PTE_W | PTE_PS,
  // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
  [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
};
```

- 所以最终的映射如图



- 在这里也可以看到页表的目录是存储在内存的，所以使用分页机制后读取一个地址需要访问两次内存，所以需要TLB缓存页表。

5. XV6 的动态内存管理是如何完成的？ 有一个 kmem（链表），用于管理可分配的物理内存页。（vend=0x00400000，也就是可分配的内存页最大为 4Mb）

- 用分页机制管理的虚拟内存的初始化在`main()`中主要是`kinit1()`，`kvmalloc()`和`kinit2()`。

```c
// in main.c
int
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
  // ........
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  // .......
}
```

- `kinit1()` 和 `kinit2()` 功能相似，都是释放一片内存，并连接到空闲页链表上，不同之处在与 `kinit2()` 最后开启了 `kmem` 的锁，之后再调用 `kalloc` 和 `kfree` 时需要获得锁。另一个不同点是实际上两者使用的页表不一样，`kinit1()` 使用的是上一题提到的 `entrypgdir`，而 `kinit2()` 使用的是新的页表，由 `kvmalloc()` 生成的 `kpgdir`。
- 在 `kalloc.c` 中定义了一个链表 `kmem` 用于存储所有的空闲页，然后用 `kfree` 和 `kalloc` 来释放和分配页。

```c
// in kalloc.c
struct run {
  struct run *next;
};

struct {
  struct spinlock lock;
  int use_lock;
  struct run *freelist;
} kmem;
```

- `kalloc` 用于分配空闲页，从链表头取下一页，`kfree` 则从释放一页，将其链接原来的表头成为新的表头。另外还有一个 `freerange` 用于释放大片内存。这里非常巧妙地使用了类型强制转换，将一个地址转换为 `run*`，再进行连接或取出。因为是非常典型的单向链表的应用，这里就不贴代码了。

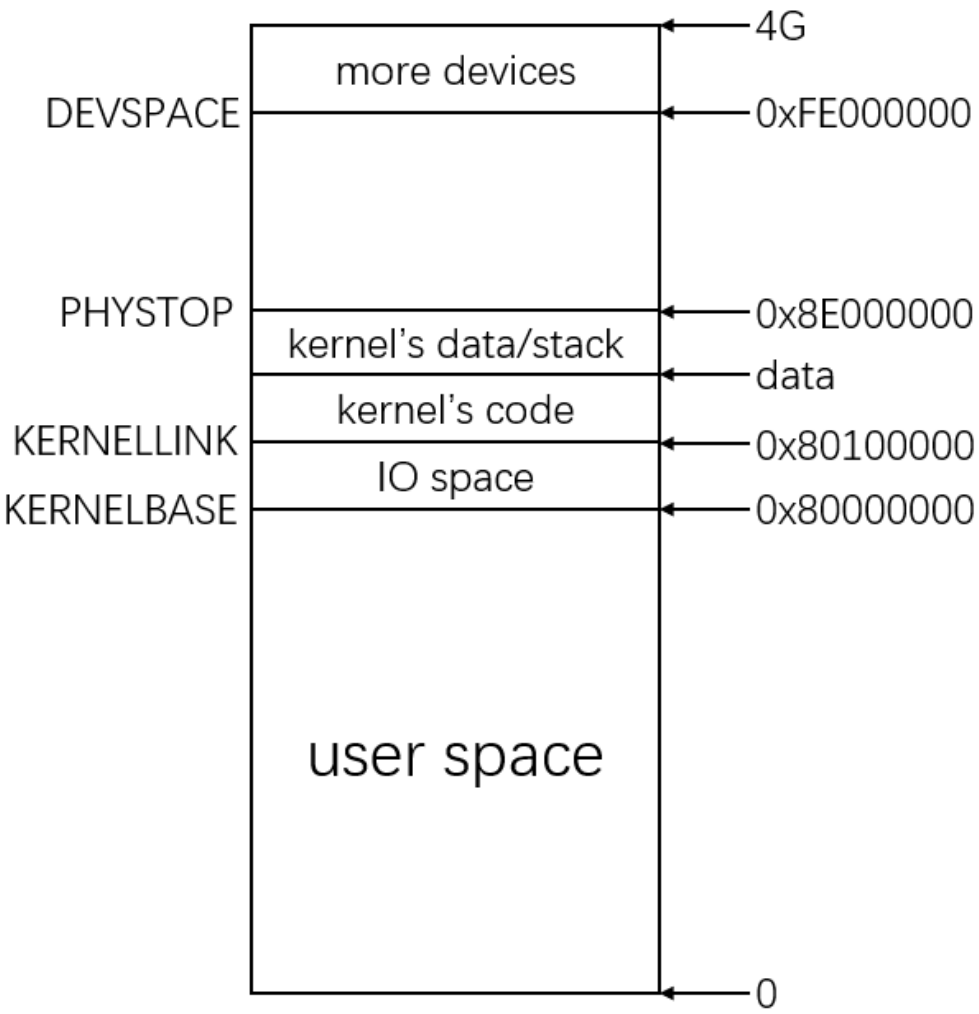
6. XV6 的虚拟内存是如何初始化的？画出 XV6 的虚拟内存布局图，请说出每一部分对应的内容是什么。 见 memlayout.h 和 vm.c 的 kmap 上的注释

- XV6使用了段页式管理，关于分段第4题已经提到过了 `seginit()`，这里不再重复。
- `kvmalloc()` 初始化了一个内核使用的页表 `kpgdir`，然后切换到该表。这个表是根据 `kamp` 初始化的，`kamp` 定义如下

```c
// in vm.c
static struct kmap {
  void *virt;
  uint phys_start;
  uint phys_end;
  int perm;
} kmap[] = {
 // I/O space
 { (void*)KERNBASE, 0,             EXTMEM,    PTE_W},
```

```
 // kern text+rodata
 { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},
 // kern data+memory
 { (void*)data,     V2P(data),     PHYSTOP,   PTE_W},
 // more devices
 { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W},
};
```

- 内核的内存初始化为4部分
  - 留给IO的空间，用于和IO设备通信
  - 留给内核的代码空间，只读不可写
  - 留给内核的数据和栈空间
  - 留给扩展设备的空间



- 下面的是`setupkvm()`及其调用的函数`mappages()`的代码，`setupkvm()`创建了一个新的页表，并通过`mappages()`将`kmap[]`中的虚拟地址映射到物理地址中，虚拟地址从`KERNBASE`到`PHYSTOP+KERNBASE`映射到`0`到`PHYSTOP`，将`DEVSPACE`映射到`DEVSPACE`到`0`。

```
// in vm.c
// Set up kernel part of a page table.
pde_t*
setupkvm(void)
```

```
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
  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) {
      freevm(pgdir);
      return 0;
    }
  return pgdir;
}

// Create PTEs for virtual addresses starting at va that refer to
// physical addresses starting at pa. va and size might not
// be page-aligned.
static int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

- `mappages` 会根据虚拟地址，在 `pgdir` 中搜索到相应到二级页表项，设置二级页表的描述符，映射到相应的物理地址上。可以看到这里的映射是线性、连续的一一对应的。

7. 关于 XV6 的内存页式管理。发生中断时，用哪个页表？一个内页是多大？页目录有多少项？页表有多少项？最大支持多大的内存？画出从虚拟地址到物理地址的转换图。在 XV6 中，是如何将虚拟地址与物理地址映射的（调用了哪些函数实现了哪些功能）？

- 发生中断时使用中断描述符表IDT（在`trap.c`中定义的`idt`）索引相应的中断处理程序，而中断处理程序在内核中，所以会接着索引内核的页表`kpgdir`
- 一个内页是4096字节，页目录有1024项，页表也是1024项，最大支持4Gbytes的内存。
- XV6中虚拟地址到物理地址的映射有直接的映射，也有通过页表转换的映射。直接的映射是将高位的虚拟地址映射到地位上，在`memlayout.h`中定义，这个其实类似于直接操作物理地址。

```
#define V2P(a) (((uint) (a)) - KERNBASE)
#define P2V(a) ((void *)(((char *) (a)) + KERNBASE))

#define V2P_WO(x) ((x) - KERNBASE)    // same as V2P, but without casts
#define P2V_WO(x) ((x) + KERNBASE)    // same as P2V, but without casts
```

- 另一个是利用分页机制的映射，一个虚拟地址转换成物理地址其实应该在硬件中完成，不过为了操作页表，XV6中用`walkpgdir()`模拟了这个过程，函数输入虚拟地址和页目录，返回一个在页表中的页描述符的指针。该页描述符的前20位指向了相应的4k物理页。

```
// in vm.c
// Return the address of the PTE in page table pgdir
// that corresponds to virtual address va.  If alloc!=0,
// create any required page table pages.
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  pde = &pgdir[PDX(va)];
  if(*pde & PTE_P){
    pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
  } else {
    if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
      return 0;
    // Make sure all those PTE_P bits are zero.
    memset(pgtab, 0, PGSIZE);
    // The permissions here are overly generous, but they can
    // be further restricted by the permissions in the page table
    // entries, if necessary.
    *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
  }
  return &pgtab[PTX(va)];
}
```

- XV6中使用分页机制管理内存，所有的页都是相同的大小4Kb（但是一开始分配了两页4Mb扩展页，在`entry.S`中分配的）。并且使用了2级页表，2级页表分为`page directory`和`page table`，32位的地址，高10位用于索引第一级`page directory`，`page directory`指向一个二级的目录`page table`，并以中间10位索引，最后得到4Kb为单位的物理页的起始地址，配合低12位地址`offset`寻址，如下图。这里每一个`page`

`directory`和`page table`都是4Kb大小， 每一条目都是32位4b，共有1024个条目。

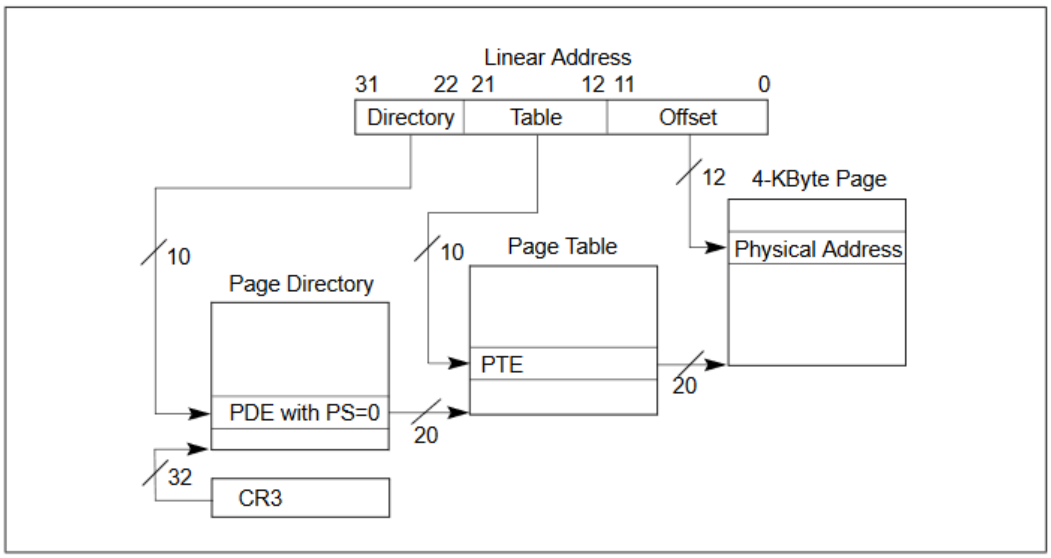


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

图片来源Intel® 64 and IA-32 Architectures Software Developer Manuals

8. 其他你认为有趣有价值的问题。

再谈sleep

- 关于sleep的代码，之前没有太理解为什么参数里还有一个锁，在写了第三题后我突然理解了

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

  if(p == 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.
```

```
  p->chan = chan;
  p->state = SLEEPING;

  sched();

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

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

- 在上面的题目中，第二题中的睡眠锁和第三题的信号量都是在已经获得一个锁的时候让进程睡眠，如果直接进入休眠状态，就会导致休眠的进程一直持有锁，导致其他的进程陷入忙等待甚至死锁。为了避免这种情况的发生，XV6规定了一个进程最多持有一个锁，同时在`sleep()`中，如果进程持有锁，会先持有锁`ptable.lock`再释放原来的锁，持有`ptable.lock`后就可以通过`sched()`回到`scheduler()`。
- 一个进程带着一个锁休眠，看起来似乎比价荒谬，这样其他进程就无法获得这个锁`ptable.lock`。 但是仔细看代码（注释里也提到了），这个锁是由被调度的进程来释放的。上一次的代码阅读已经提到了，`scheduler()`通过`swtch()`切换到其他进程，而其他进程都会通过`sched()`回到`scheduler()`（当然，`sched()`中也是通过调用`swtch()`进行上下文切换的）。所以从`scheduler()`切换上下文后其实只有两种情况
  - 第一是一个新创建的进程，这时会返回到`forkret()`，这个函数做的事情就是释放`ptable.lock`
  - 第二个是回到`sched()`中的`swtch()`后面，而`swtch()`后面只有一行代码，设置CPU的中断标志，然后返回到`sched()`的调用者。在`proc.c`中`sched()`被调用也只有三个函数
    - `exit()`，因为进程已经退出且设置为僵尸状态，所以其实不会返回到这里
    - `yield()`，一般的情况，返回后的下一行就是释放`ptable.lock`
    - `sleep()`，即这个休眠的进程被唤醒后回到的地方，如果原来持有的锁不是`ptable.lock`，同样会释放，否则会继续持有，因为这个锁将在`sleep()`的调度函数那里释放，比如`sleeplock`中的`acquiresleep()`
  - 综上，`scheduler()`调度到任意一个就绪的进程都会释放`ptable.lock`，所以其实不会造成死锁。这里也可以看出一把锁其实不一定属于一个进程，在不同进程共享某些资源的时候，锁会只属于一个进程，但是像`ptable.lock`这种锁更像是属于一个CPU，因为任意一个进程都能释放它，这个进程持有了，却在另一个进程那里释放了非常正常。同时可以看到要保证锁机制的正常运行，需要非常仔细地思考各种可能。