源代码阅读【Lab顺序】

姓名：吴悦欣

Lab1-syscall

xv6采用了RISC-V架构，共有三种执行指令的模式——机器模式、监督者模式和用户模式，在机器模式下的指令有完全的权限，xv6的启动就是在机器模式下进行，在执行完几条指令后，系统就会转为监督者模式。

xv6和多数Unix的OS一样，采用的宏内核的实现形式，整个操作系统都驻留在内核中，因此所有系统调用的实现都在监督者模式下运行。其内核源码都在kernel/子目录下，被分成多个模块文件，模块间的接口在defs.h中定义。

进程相关的结构

```
proc.h：主要给出了context/cpu/trapframe/proc结构体的定义
  1. context结构体：保存内核进程切换时需要保存的上下文(主要是被调用者保存的寄存器和ra/sp)
  2. cpu结构体：记录cpu的状态——当前运行的进程，保存的上下文，push_off()的depth(定量的关闭中断），以及interrupt enable变量表示是否可以被中断
  3. trapframe结构体：用于保存在用户态和内核态切换时的一些信息，占用一个单独的页，紧跟在trampoline的页后面
  4. proc结构体：xv6中的PCB块，记录每个进程的状态，包括，锁，进程运行状态，父进程指针，正在睡眠的channel，是否被kill，退出时的状态，pid，对应内核栈的虚拟地址，使用的内存空间大小，pagetable指针，trapframe指针，打开文件表，目前的目录（为inode指针），进程名字
```

xv6内核启动和第一个进程的运行

计算机开机时，会初始化自己，并运行一个存储在ROM中的boot loader，由它将xv6内核加载到内存（物理地址0x80000000）中；在机器模式下，CPU从_entry开始执行xv6，即entry.S文件，此时虚拟地址直接映射到物理地址

```
entry.s：被放在0x80000000地址，每个CPU都从这个地方开始执行；初始化栈的空间，为每个CPU留4096byte的空间，因此sp是从stack0+hartid*4096【stack0在start.c文件中定义】的地方开始的；然后调用start函数
```

RISC-V提供了mret指令，用来从上一次的调用中返回，从监督者模式到机器模式

```
start.c:
  1. start():
    start函数是entry跳转的地方，为了契合mret的使用，会先设置上一次的调用为特权模式，设置返回mepc地址main函数；将中断和异常都委托给特权模式下进行；
    调用timeinit()对时钟芯片进行编程初始化定时器中断；
    获取cpu的id并为每个cpu保存在tp寄存器中；
    调用mret跳转到main
  2. timeinit():
    每个CPU都有一个独立的时钟中断的源，初始化一个时钟周期为0.1s，设置时钟中断的处理程序的入口timervec到相对应的寄存器，开启机器模式下的中断和时钟中断
------------------------------------------------------------------------------------------
kernelvec.S/timervec:
  进行机器模式的时钟中断，把mtimecmp加上一个interval设置下一个中断的时间，引发一个软中断，在进入trap的处理程序后会引发进程的调度等
```

所有的CPU在执行完start函数后都会因为mret跳转到main函数的执行

```
main.c/main():
    如果是第一个CPU(cpuid()==0)，就要对整个系统各个部分进行初始化(包括内存，页表，进程表，trap相关的处理，中断控制器，缓冲区，文件系统，然后调用user_init创建第一个进程)；第一个CPU的初始化成功之后，后续的CPU要等待第一个CPU完成必要的初始化之后才能够对自己独立的设备进行初始化；结束之后进入scheduler开始进程的调用
```

关于第一个进程的运行，是通过proc.c/userinit()函数实现，执行的是initcode.S汇编代码，该代码通过exec("./init")在init进程中返回，可以看到init.c/init()会创建一个新的consoler设备文件，占用文件描述符0/1/2，启动shell

```
proc.c
    userinit():创建初始进程init，为initproc分配pid存在proc.c的全局变量，调用uvminit将初始化的指令/数据复制到用户空间分配的一个页表，调用了mappages将物理地址映射到p->pagetable的虚拟地址；这个init进程在main函数最后调用scheduler之后被运行

user/initcode.S: 是initcode 的机器语言，主要是通过exec调用/init
user/init.c: 第一个用户级的进程，创建一个新的console文件，用文件描述符0/1/2打开它，之后在控制台上启动一个shell（fork之后子进程执行sh，父进程wait直到sh结束）
```

系统至此启动完毕。

系统调用

进程通过执行RISC-V的ecall指令进行系统调用（正如在user/usys.pl脚本中写的，在运行时，该脚本会生成对应的汇编代码），由此作为系统调用的入口

user > usys.pl

```perl
print "# generated by usys.pl - do not edit\n";

print "#include \"kernel/syscall.h\"\n";

sub entry {
    my $name = shift;
    print ".global $name\n";
    print "${name}:\n";
    print " li a7, SYS_${name}\n";
    print " ecall\n";
    print " ret\n";
}
```

ecall指令进入内核后，通过usertrap判断中断的原因是系统调用(scause寄存器中是8)后会关闭中断调用syscall.c/syscall()函数，通过被放在a7寄存器的系统调用号匹配系统调用函数的指针数组进行调用。参数的传递通过固定的寄存器实现。

```
syscall.c
  1. fetchaddr(): 从当前进程的指定地址拷贝数据到指定的地址
  2. fetchstr(): 从当前进程指定地址拷贝有长度上限的字符串到指定位置
  3. argraw(): 获取进程的trapframe中a0-a5指定寄存器的值
  4. argint(): 给整型参数赋值
  5. argaddr(): 给指针参数赋值
  6. argstr(): 给字符串参数赋值
  7. syscalls数组：为系统函数的指针
  8. syscall(): 根据用户寄存器a7的值调用对应的系统函数，将返回值放在用户寄存器a0中
syscall.h: 定义系统调用号
user/user.h: 声明系统调用在用户空间调用的原型
user/usys.pl: 生成系统调用的汇编代码的脚本
```

```
fcntl.h
  定义open的第二个参数，是关于文件读写的标志【可读/写/创建/截断】
stat.h
  对文件相关的结构体的定义，记录了目录/文件/设备所在的磁盘设备、inode、文件类型、文件的链接数、文件大小

vm.c
  copyinstr将用户页表pagetable中虚拟地址srcva复制到dst
  walkaddr调用walk函数在软件中模拟分页硬件的操作获取物理地址pa
* push_off()是类似于intr_off()的函数，但是不同点在于，多次intr_off()效果一样，但是多次push_off()会计数，并且需要对应数量的push_on()才能取消
```

在Lab1中主要是需要利用原有内核代码对系统调用的支持，添加新的系统调用函数。

Lab2-Page Table

页表是操作系统为每个进程提供自己的私有地址空间的机制，决定了虚拟内存和物理内存的对应关系和访问状态。xv6只使用64位虚拟地址的低39位，在逻辑上，页表由2^{27}个页表项组成，每个页表项PTE都包含一个44位的物理页号和其他标识位。利用39位中的高27位所引导页表中找到一个PTE来将虚拟地址转化为物理页号，加上偏移值，得到一个56位的物理地址。实际上在将PTE转换为页框号的过程分为三层页表的查询，每9位查询一级页表的页表项，得到下一级页表的起始地址。如果有无效的PTE则会产生缺页故障pagefault，陷入内核处理该异常。硬件satp

寄存器保存查询的一级页表的起始地址（每个CPU独立），由此实现独立的用户空间

```
riscv.h
  1. r_xx和w_xx函数为一系列使用嵌入的汇编代码读写寄存器的函数
  2. sfence_vma(): 更新tlb（使用嵌入的汇编代码）
  3. 定义了和页表相关的一系列宏，包括页表的大小，偏移量，PTE的标志位(PTE_V为有效位，PTE_R表示是否允许读取页，PTE_W表示是否允许写，PTE_X表示是否可执行，PTE_U表示是否允许用户态下的指令访问)，PTE查询过程中需要的特定位的提取
```

```
memlayout.h (xv6内核内存布局)
  1. 注释中指明了xv6的物理内存的分布
  2. 给定部分设备(磁盘，内中断控制器，外中断控制器)物理内存地址
  3. KSTACK(p)给出第p个CPU的内核栈的地址(这里内核栈的虚拟地址分布在trampoline下面，并且两两之间通过一个无效的保护页来防止越界访问)
```

内核对RAM和内存映射设备寄存器采用直接映射，但是trampoline和内核栈页并不采用直接映射，trampoline页被映射在虚拟地址空间的顶部，同时也有一次直接映射的拷贝；内核栈页是每个进程分别都有的，被映射到高地址处（trampoline后面），两两之间有一个无效的保护页进行保护。

大部分用于操作地址空间和页表的xv6代码都在vm.c中

```
vm.c （kvm开头的函数作用于内核页表；uvm开头的作用于用户页表；剩余的同时作用于两种）
  1. pagetable_t: 指向页表页的指针 实例化的kernel_pagetable变量 指向根页表页的指针
  2. kvminit(): 创建内核页表，调用kvmmap将需要的硬件资源映射到物理地址
  3. kvminithart(): 映射内核页表，将根页表页的物理地址写入寄存器satp
  4. walk(): 根据给定的页表和虚拟地址模拟多几页表得到PTE，参数alloc表明如果找到无效的PTE是否分配一个有效的PTE
  5. walkaddr(): 调用walk函数将虚拟地址翻译成物理地址
  6. kvmmap(): 在内核页表添加映射，只在系统启动的时候被调用
  7. kvmpa(): 将内核栈的虚拟地址翻译成物理地址
  8. mappages(): 将虚拟地址映射到物理地址，创建对应的PTE表项，成功返回0，否则返回-1
  9. uvmunmap(): 解除指定虚拟地址与其物理地址的映射
  10. uvmcreate(): 创建一个空的用户页表
  11. uvminit(): 将用户的initcode加载到指定页表的起始地址，来运行第一个进程，被main调用
  12. uvmalloc(): 为进程指定的内存扩展分配PTE和物理内存
  13. uvmdealloc(): 取消进程地址空间部分映射，将进程占用的内存缩小
  13. freewalk(): 递归释放页表页
  14. uvmfree(): 释放用户内存页，然后调用freewalk来释放对应的页表页
  15. uvmcopy(): 将父进程的页表拷贝一份给子进程，被fork调用
  16. uvmclear(): 标记某页是用户不可访问的
  17. copyout(): 将数据从内核复制到用户空间
  18. copyin(): 将数据从用户空间复制到内核
```

这一Lab实现了vmprint函数打印出多级的页目录情况，只需要类似freewalk调用walk函数查询页表即可；第二/三部分则是希望改变xv6最初将所有进程的内核页表统一映射到trampoline后的方式，真正让每个进程的地址空间有一个独立的内核页表，同时为了能够让内核态的进程能够直接访问到用户空间的数据，需要将用户空间的页表拷贝一份给内核页表。这一过程多数是在vm.c和proc.c中修改实现，proc.c中多是和进程相关的函数

proc.c
  1. procinit(): 在系统启动的时候被调用，初始化进程表，并且为每个进程预留内核页表，按照顺序排列在trampoline后
  2. cpuid(): 返回当前cpu号，读取寄存器获取，需要在关中断的时候被调用
  3. mycpu(): 返回当前的cpu结构体
  4. myproc(): 返回当前的进程结构体
  5. allocpid(): 分配下一个可使用的进程id，使用全局的nextpid记录
  6. allocproc(): 分配一个可使用的进程，初始化该进程结构体(包括其trapframe，页表，内核栈和上下文)
  7. freeproc(): 释放一个进程，需要释放为其分配的空间，恢复成可使用进程表中的空闲进程
  8. proc_pagetable(): 为指定进程创建用户页表
  9. initcode二进制数组：是一段可执行代码，实现第一个用户进程运行
  10. userinit(): 创建第一个用户进程
  11. growproc(): 将进程空间扩展/缩小指定的大小
  12. fork(): 创建子进程，进行适当的proc结构体的初始化，拷贝用户内存
  13. reparent(): 将僵尸进程的子进程移交给init进程管理
  14. exit(): 推出当前进程，保持僵尸进程的状态，直到其父进程调用wait
  15. wait(): 等待子进程退出并返回它的pid
  16. scheduler(): 进程调度，从进程列表中选择一个就绪的进程给它上CPU的机会，负责上下文的切换和部分必要寄存器值的切换，包括内核页表，sched（不是显式的调用，而是通过切换上下文，更新cp直接改变逻辑流）
  17. sched(): 被scheduler调用（同样不是显式的调用）进行调度
  18. yield(): 进程主动放弃cpu(包括时间片或是中断的原因)要求进行调度
  19. sleep(): 当前进程在指定的channel上等待，释放持有的锁，主动放弃CPU
  20. wakeup1(): 唤醒在wait等待的指定的进程，被exit调用
  21. kill(): kill指定pid的进程
  22. either_copyout(): 将指定地址的数据拷贝到用户或者内存地址
  23. either_copyin(): 将数据从用户或者内存地址拷贝到指定地址
  24. procdump(): 向控制台打印进程的列表及其状态相关信息，调试的时候使用

exec.c
  1. loadseg(): 将程序段加载到指定虚拟地址的页表中
  2. exec(): 函数是exec函数系统调用的具体实现， exec 读取文件来初始化一个地址空间的用户部分；分配新的页表，为每个ELF段分配内存，并加载到内存
  调用namei来打开二进制文件路径，读取elf文件头和程序段头获取并检查文件信息的合法性，程序段头描述了必须加载到内存中的程序段，根据这些头文件信息将每个段加载到内存中，其中需要合法性的检查来防止恶意的指针引发系统崩溃

kernel/elf.h
  elfhdr结构体 描述文件elf头
  proghdr结构体 描述了一个必须加载到内存中的程序段

物理内存的分配主要是通过调用kalloc.c中的函数实现的

```
kernel/kalloc.c 进行物理内存分配（end到PHYSTOP之间），维护一个空闲链表kmem.freelist
  1. run结构体：指针链表，用来维护空闲链表
  2. kmem结构体实例：一个自旋锁和freelist的run结构指针组成
  3. kinit()：初始化锁，将RAM的内存部分4096字节为一页加入空闲链表
  4. freerange()：释放start到end的物理内存
  5. kfree()：释放指定物理页，并放入空闲链表中
  6. kalloc()：分配一个页，从空闲链表中移出
```

进程内存的收缩和增长主要通过sbrk系统调用实现

```
sysproc.c/sys_sbrk()：系统调用，调用growproc函数使得进程内存收缩
```

Lab3-Traps

CPU从原本的进程执行过程中停止执行指令，而将控制权转移给内核或者对应的handler主要有以下情况：系统调用，异常和设备中断。这一过程统一称为trap。trap的实现需要硬件的支持、软件提供的处理程序的入口和具体的处理程序的实现。

其中硬件的支持主要是CPU的一组控制寄存器，来表明trap发生的原因

```
stvec：trap处理程序的地址，跳转到该地址处理trap
sepc：当trap发生时，保存PC的地方（原本的PC被stvec覆盖），sret将sepc复制到PC中返回
scause：描述trap的原因
sscratch：可以暂时存储一个值
sstatus：SIE位控制设备中断是否被启用，如果未被启用，将会推迟设备中断到内核设置SIE；SPP表明trap来自用户还是内核态，控制sret返回的状态
satp：每个CPU有一个自己的页表，satp指向页表的起始地址
```

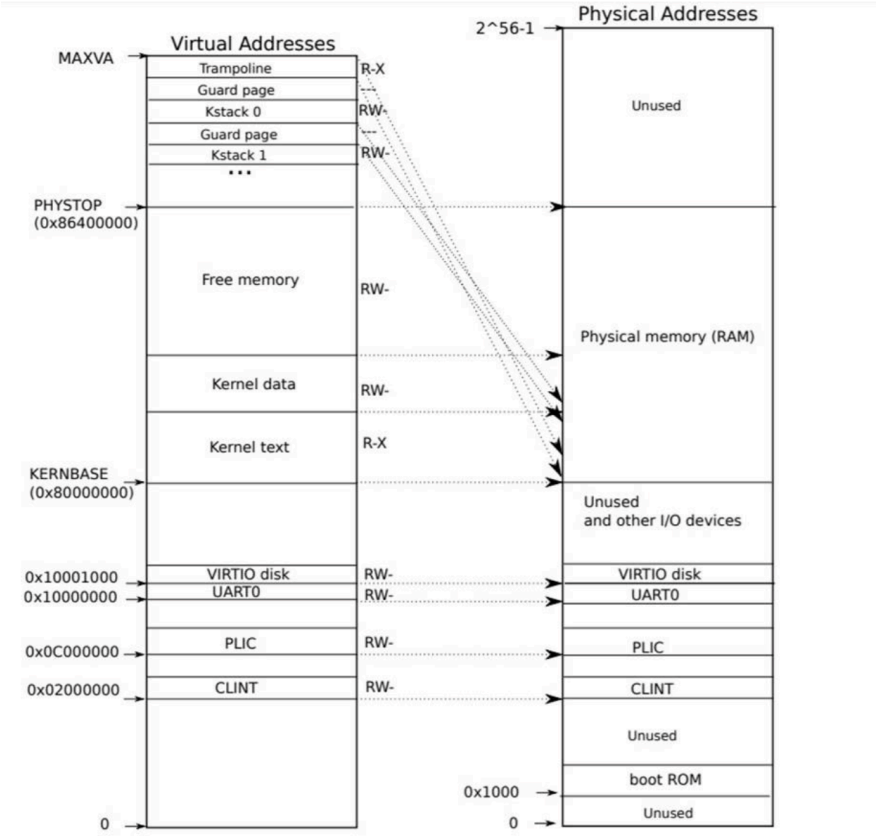
来自用户空间的trap处理路径是uservec-->usertrap-->usertrapret-->userret

```
trampoline.S：汇编代码 用于切换用户和内核空间
  uservec:从用户进程调用usertrap()跳转进来：保存user regs in trapframe；跳转回usertrap()
  userret:从内核态切换到用户态usertrapret()，恢复保存在trapframe中的寄存器；跳转回去
  * sscratch：supervisor scratch register，用来给用户trapframe的地址
  * sfence.vma r1, r2：通知处理器页表的修改，r1指定虚页,r2给出被修改页表的进程的地址空间标志符
                                均为0时，会刷新整个TLB
trap.c
  1. usertrap()：处理用户空间的中断/异常/系统调用，被trampoline调用，保存原PC到sepc寄存器，通过scause寄存器的值判断trap的原因，如果是系统调用则调用syscall函数，返回执行下一条指令；如果是时钟中断，则主动调度，下次上CPU的时候通过usertrapret恢复上下文，返回。
  2. usertrapret()：trap处理完毕后返回用户空间，需要恢复上下文
  3. kerneltrap()：处理内核的中断和异常
  4. clockintr()：时钟中断，时钟计数++
  5. devintr()：检查中断来源是外部还是软件，如果是时钟中断则返回2，其他设备中断返回1，未知原因返回0
```

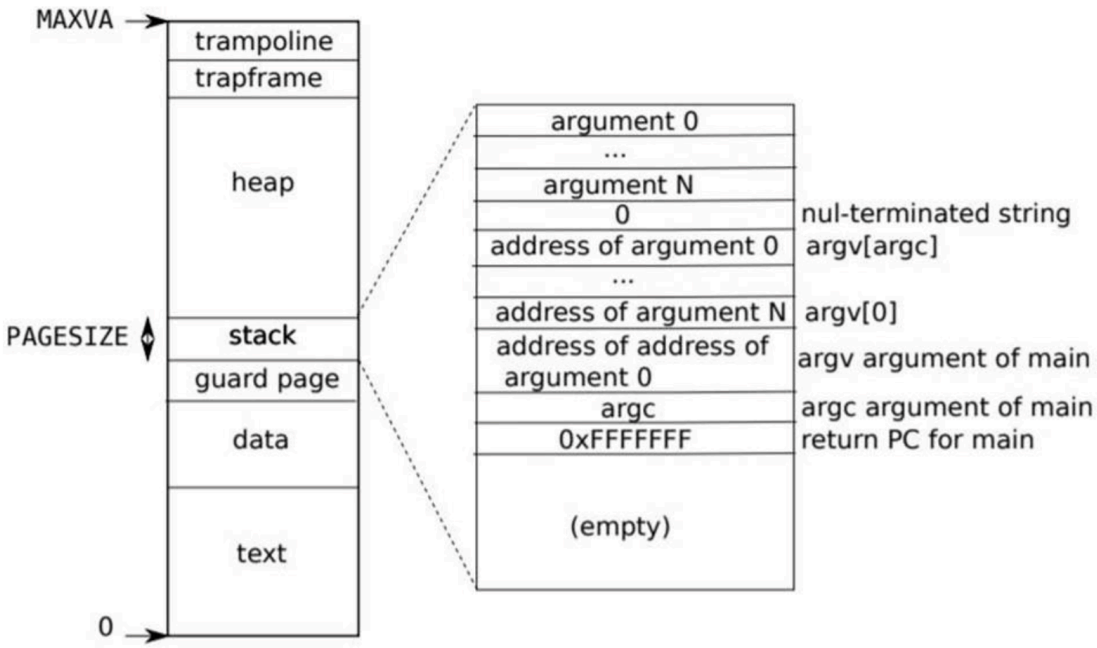
lab3中较难实现的alarm实际上就是将trap处理时钟中断时的代码进行修改，使之调用指定函数即可。

Lab4-Lazy allocation

这一部分是实现内存的懒分配：在“分配”的时候并不进行真正的分配，直到使用到该部分内容的时候，产生page fault才分配物理内存



内核不采用直接映射：否则guard page对应的物理内存中会出现很多空着的部分，内存管理变得困难。所以直接映射到RAM并且将guard page设置为PTE_V=0



限制进程地址空间的最大长度的因素：指针是64位的，硬件在页表中查找虚拟地址时只使用低39位，xv6只使用其中的38位。因此最大地址是$2^{38}-1$ =0x3fffffff，即在riscv.h中定义的MAXVA，在地址空间的顶端，保留了一页用作trampoline和映射进程trapframe的页，便于切换到内核。

与lab相关的vm.c/trap.c的代码在前几个lab中都进行了详细的阐述。具体的实现过程依赖于page fault的处理，需要在usertrap中处理的异常类型中加入scause为13或15的页相关的错误，调用vm.c中和内存管理相关的函数分配空间。进一步的工作是修改其他相关的部分，Lab5则是将这种思想运用到了Fork函数中，实现COW写时复制。

Lab5-Copy-on-Write Fork

COW是懒分配思想的体现，直到真正需要用到部分内存的时候才进行内存的分配。和懒分配不同的是，COW是通过将页表项PTE的标识为改为只读（修改uvmcopy函数对父子进程内存拷贝的实现），类似的是，通过对page fault的处理完成完整的内存映射。

还有不同的一点是由于父子进程可能同时映射同一个内存空间，即使子进程退出要释放空间，也不能直接将其free掉，否则还在运行的父进程在使用这部分内存的时候就会报错。因此需要修改kalloc.c的内容，为每一页增加引用计数，直到没有进程引用该页才能释放占用的内存空间。这一部分锁的使用也比较微妙需要注意。

Lab6-Multithreading

操作系统需要运行的进程数量通常是很多的，而CPU却只有几个，因此为了实现多个进程“同时”在一个CPU上运行并且独占它的假象，操作系统需要通过适当的调度手段实现CPU的复用。

首先需要完成的是进程的切换，xv6系统通过sleep和wakeup系统调用进行进程的切换，进程切换的时机包括进程等待设备或管道I/O、等待子进程的、时钟中断的时候。

进程的切换意味着上下文的切换，需要把部分寄存器（callee-saved），由swtch函数执行

```
swtch.S
  传入参数a0和a1分别为旧/新进程的context，将旧的寄存器存储到这一页中，并从新进程的页中加载新的寄存器值，不保存pc值，但是会保存ra寄存器，保存swtch应该返回的地址
```

在之前的Lab相关代码阅读中提到过sched和schedular函数，二者配合相互切换，实现最终进程的切换，其中上下文的切换就是通过swtch函数实现。sched会调用swtch切换到cpu调度器的上下文，而这个上下文被scheduler的swtch调用保存，因此在执行完swtch之后，实际不会回到sched，而是回到scheduler继续执行CPU的调度器进程选择下一个被调度的进程。当选定后，如果这个进程之前是被sched挂起的，那么等scheduler调用玩swtch返回的会是那个进程上次完成swtch的地方，即sched函数继续执行。因此实际上不存在用户进程之间的切换，只有用户进程和调度器进程之间的切换。

进程的切换也意味着会有多个进程并发执行（一个进程开始的时机在另一个进程结束之前，一个CPU在一个时间点同时有多个进程在运行过程中），因此调度的细节需要锁的机制来避免竞争和同步失败。在调用sched之前进程必须获得自己的锁，来保证进程的状态和context上的不变量，到scheduler中再释放进程的锁，来防止一个进程同时被两个CPU运行的场景。

xv6的每个CPU有一个cpu结构体，正如在上文中介绍proc.h中cpu结构体中所说的，每个CPU的id在自己的tp寄存器上，这一寄存器在start.c中被初始化。

xv6使用了sleep和wakeup的机制解决进程实现同步机制。这一实现不会出现因为在sleep之前就wakeup过，而导致唤醒的丢失，因为sleep在释放CPU前和wakeup的时候都会先获取进程自己的锁。

```
proc.c
  sleep(): 进程调用sleep函数后会主动要求调度，在特定channel睡眠，等待唤醒，并且调度前释放持有的锁；在被唤醒的时候重新获得该锁
  wakeup(): 唤醒特定channel伤的进程，采取的是唤醒所有在这个channel上睡眠的进程，将他们的状态都修改为就绪态，如果它们运行时需要获得同一把锁，则让它们自己竞争需要的锁，不过这也要求sleep一定要在循环中被调用，防止它们被虚假唤醒
```

xv6的管道pipes实现就使用了sleep和wakeup进行生产者和消费者的同步。

```
pipe.c
  pipe结构体：有数据缓存数组，已读和已写字节数，文件描述符是否打开，自旋锁
  pipealloc(): 调用filealloc分配指定的读/写的文件描述符，恰当的设置文件描述符对象使得一个只读，一个只写
  pipeclose(): 唤醒等待的生产者或者消费者让它们把数据读/写完之后关闭管道
  pipewrite(): 当数据缓冲区满了的时候，唤醒读者来消费，然后再继续写入数据
  piperead(): 当数据缓冲区为空的时候，唤醒写者继续生产，然后再继续读
```

还有wait/exit/kill函数也是用了sleep和wakeup的机制实现等待

```
proc.c
  exit(): 进程退出前会先关闭所有打开文件，将自己的子进程托管给init进程，并且唤醒在wait中的自己的父进程，调用wakeup1来唤醒wait的进程，修改自己的状态为zombie，唤醒的时候必须持有自己的锁，防止父进程上CPU将自己释放掉
  wait(): 等待子进程结束才返回；扫描所有的进程，检查自己是不是该进程的父进程，如果是并且该子进程的状态是僵尸进程，则调用freeproc回收该进程，如果没有子进程则返回-1，如果有但是还没有要结束的子进程，则在自己的通道上sleep
  wakeup1(): 唤醒在wait中sleep的进程，专门唤醒父进程
  kill(): 允许一个进程指定pid的另一个进程终止
```

Lab6的具体实现内容其实跟sleep或wakeup还有进程的调度关系不大，不是使用内核的这些函数，而是使用用户状态的pthread库来实现等待和唤醒的过程，比较简单。

Lab7-Lock

Lab7主要是关于锁的使用，操作系统需要锁的根源在于它具有并发的特点，进程可能需要同时访问共享资源，顺序是其中很关键的问题，xv6系统根据不同的需要设计了两种锁——自旋锁和睡眠锁，以下是相关文件和它们特点的介绍。

```
spinlock.h: 自旋锁结构体定义
  struct spinlock: locked表示是否可获得该锁(可以为0); name; cpu表示拿着锁的cpu
spinlock.c:
  initlock: 初始化自旋锁
  acquire: 关中断（防止因为中断和原进程死锁，都试图获取同一个锁），用硬件支持的原子操作__sync_lock_test_and_set(本质是amoswap指令，返回值为锁的locked旧值)，循环直到获取该锁（修改locked变量；调用__sync_synchronize函数来“阻塞”（告诉编译器和CPU不要越过这个屏障重排任何内存读写操作；它们可能会重排来获得更高的性能，which会带来错误），设置持有锁的cpu为当前cpu
  realease: 调用c库的院子函数__sync_lock_release来释放锁，开中断
```

```
  holding：检查当前cpu是否已经持有该锁
  push_off：计数型的关中断，cpu struct中的noff变量记录
  pop_off：计数型的开中断

sleeplock.h：睡眠锁，改进自旋锁，在拥有锁的同时允许释放CPU并且开放中断，长操作的时候使用
  struct sleeplock：locked标记是否被拿了；有一个lk（spinlock）；name；pid（持有锁的进程）
  （不能用于中断处理例程和spinlock的核心代码中）
sleeplock.c：
  initsleeplock：初始化
  acquiresleep：获取lk，当lk被拿的时候在lk上sleep并且释放lk；被唤醒的时候重新拿lk锁，拿sleep锁，
释放lk
  releasesleep：获取lk，释放sleep锁，释放lk，唤醒其他在lk上等的进程
  holdingsleep：返回一个睡眠锁是否已经被进程自己持有了
```

锁和中断的处理需要注意一些潜在的危险，如果在持有锁的时候被中断，很可能出现死锁的情况，因此在一个CPU获取一个自旋锁的时候，xv6总是禁用该CPU上的中断。但是睡眠锁由于其设计需求，是在中断开启的时候被使用的，因此它们绝对不能被用在中断处理程序中。xv6系统根据各个组件的需要，设置了以下这么多种细粒度的锁。

Lock	Description
bcache.lock	Protects allocation of block buffer cache entries
cons.lock	Serializes access to console hardware, avoids intermixed output
ftable.lock	Serializes allocation of a struct file in file table
icache.lock	Protects allocation of inode cache entries
vdisk_lock	Serializes access to disk hardware and queue of DMA descriptors
kmem.lock	Serializes allocation of memory
log.lock	Serializes operations on the transaction log
pipe's pi->lock	Serializes operations on each pipe
pid_lock	Serializes increments of next_pid
proc's p->lock	Serializes changes to process's state
tickslock	Serializes operations on the ticks counter
inode's ip->lock	Serializes operations on each inode and its content
buf's b->lock	Serializes operations on each block buffer

Figure 6.3: Locks in xv6

Lab7第一部分是对kalloc中空闲链表的锁的粒度细分的改进，第二部分还涉及到对buffer cache使用锁的改进。总体的目标都是把一个全局的锁进一步细分为局部的锁，提高效率同时也要谨防死锁或是竞争的出现。buffer相关的结构如下

```
kernel/buf.h
  buf缓存的结构体，属性：有效位(是否从磁盘读入数据)，磁盘是否有缓存，设备号，块号，lock，引用计数，prev指针，next指针，数据[1024]
kernel/bio.c
  和磁盘缓存相关的函数
  binit 初始化bcache【lock初始化，初始化buf指针，形成next链:head->29...->0->head; prev链:head->0->1->...->29，这个顺序之后会改变，按照头部是最近使用的排序】
  bcache结构 有lock/buf[30]/head[buf]，用的spinlock，但是buf用的sleeplock;spinlock保护的是被缓存块bcache的信息，sleeplock保护的是buf块内容的读写
  bget 返回指定设备/块号的缓存bcache中的某个buf，如果没有缓存过就找一个最近没用过的buf，refcnt++
  bread 获取特定设备的某个block的缓存数据【如果invalid，即还没从磁盘读入，则调用virtio_disk_rw(b,0)读入】
  bwrite 写入特定设备的某个block，调用virtio_disk_rw(b,1)，每次写入就直接写回磁盘
  brelse 对缓存的操作完成[释放前面的操作默认会加的锁，refcnt--，将refcnt为0的块放到队列最前面，head后面]--在brelse之后不能使用buffer
  每次只能有一个进程能使用缓存
  bpin 将buffer的refcnt++
  bunpin refcnt--
```

Lab8-File System

Lab8是跟文件系统相关的lab，主要需要做的是将文件的数据索引结构进行扩展，引入二级索引，使得其支持更大的文件；还有在目录文件中添加符号连接。实际Lab的内容不多，对应的xv6课本内容比较多，以下是结合xv6课本对源代码对应部分的解读。

```
**文件系统共有七个层次：Disk读写virtio磁盘，buffer cache缓存Disk，Logging允许上层更新多个buffer，Inode组织文件Directory给出目录项序列，Pathname提供层次化的路径名，File descriptor给出

kernel/fs.h
  磁盘文件系统的格式
// [ boot block | super block | log | inode blocks |
//                                          free bit map | data blocks]
  superblock结构体:【文件系统的元数据，描述磁盘的分布，由mkfs单独写入】有magic/size/数据块个数/文件inode个数/log块个数/第一个log块号/第一个inode块号/第一个free map块号
  free bitmap: 位图块，记录那些数据块在使用
  data blocks: 数据块，要么在bitmap中标记空闲，要么就持有文件/目录的内容
  dinode结构体: 磁盘上的inode定义；文件类型，主要的设备号，次要设备号，引用数，文件大小，持有文件内容的磁盘块的块号数组
  dirent结构体:【目录是包含一些dirent结构的对象】inode号/名字

kernel/fs.c // low-level的文件系统implementation
  readsb(): 读指定磁盘设备的superblock[blockno为1的块]
  fsinit(): 初始化文件系统【通过读入设备的super block】，初始化日志log
  bzero(): 将一个指定的设备中的块清空并将修改提交磁盘
  balloc(): 申请一个新的磁盘块，每BPB个Bytes是一个块的磁盘，循环寻找位为0的空闲块，找到则更新bitmap并返回该块
  bfree(): 释放一个磁盘块，将对应的bitmap位改为0
```

****对inode典型的使用****

```
  ip=iget(dev, inum);
  ilock(ip);
  ...examine and modify
  iunlock(ip);
  iput(ip);
****
```

  icache结构体：缓存inode

  iinit()：初始化锁

  ialloc()：在磁盘上找一个空闲的inode分配，修改type域来使用它

  iupdate()：将被修改的inode从内存拷贝到磁盘上，每次被修改都要写回

  iget()：引用inode；先在icache里面找；否则使用icache中的空闲的inode缓存(找到或创建一个inode的缓存)

  idup()：增加inode的引用数

  ilock()：锁住inode，当inode无效的时候从磁盘加载对应的dinode；在读写inode之前一定要调用这个

  iunlock()：释放inode的锁，唤醒在等待这个锁的进程

  iput()：减少引用数，如果减为0且无其他的link则释放inode的cache，valid-->0；调用itrunc将文件截断为0字节和iupdate将内容写回磁盘

  iunlockput()：= iunlock+iput

  itrunc()：释放文件的块，将inode的大小重置为0，释放直接块然后释放间接块

  bmap()：包装直接块or间接块的表示方式，返回inode的第bn个数据块的磁盘块号，如果没有（对应的条目为0）就会分配一个

  stati()：将inode元数据复制到stat结构体中

  readi()：从inode中读取数据（要求偏移量和计数<=文件end，如果从末尾开始读或者在过程中读过末尾不会错误，只是返回的读入字节数少于期望的字节数）

  writei()：在inode写数据，可以超过文件末尾，增常文件，将数据复制到缓冲区，如果文件增长，需要更新size

  namecmp()：比较名字是否相同

  dirlookup()：在目录中搜索带特定名称的条目，找到则返回指向该inode的指针，更新目录中条目的字节偏移量*poff

  dirlink()：在当前目录dp中创建新的目录项，若已存在该名称会返回错误；循环目录项得到空闲的项，设置off；否则将off设置为dp->size，为dp增加一块

  // 路径相关

  skipelem()：将路径进行解析，忽略多个'/'，将下一层的文件名称放入传入的name指针中，更深层的传回path

  namex()：查找并返回指定路径的inode ——nameiparent是否查找其父目录文件

  namei()：返回path最后的目录文件

  nmaeiparent()：返回path最后的父目录的目录文件

kernel/proc.c

  either_copyout()：根据参数将指定位置的数据拷贝到内核/用户目标地址

  either_copyin()：根据参数将数据从内核/用户拷贝到目的

kernel/file.h

  file结构体:(每个打开的文件由一个结构体表示，调用open的时候都会创建一个新的file结构体) 有文件类型，引用量，可读性，可写性，pipe，inode，off(I/O偏移量)，major？？

inode结构体：dinode在内存中的备份，是正在使用的inode；设备号，inode号[与其在磁盘上的位置相关]，引用量，sleeplock，有效位[是否从磁盘加载]，种类，major，minor，nlink【链接该inode的目录项数】，大小，addrs数组:持有文件内容的磁盘块的块号数组（前NDIRECT个条目是直接块，最后一个条目给出放间接块的地址，NINDIRECT个间接块）
devsw结构体：封装两个函数，read和write

kernel/file.c
ftable:变量 全局文件表，由一个lock和file的数组组成
filealloc(): 分配文件
filedup(): 创建重复引用——ref++
fileclose(): 释放引用
filestat(): 获取特定file的元数据，只能对inode类型进行操作，存储在指定的stat结构体中
fileread/filewrite(): 读写数据

kernel/sysfile.c // high-level系统调用
fdalloc(): 给指定的文件分配一个文件描述符
sys_link/sys_unlink(): 创建或删除对inodes的引用，sys_link为一个现有的inode创建一个新的名字
create(): 为一个新的inode创建一个新的名字——被调用实现sys_open/sys_mkdir/sys_mknod
sys_open(): O_CREATE会调用create函数创建一个新的inode； 否则直接调用namei返回指定文件的inode

** Logging解决操作过程中的崩溃问题，系统调用不直接写磁盘上的文件系统数据结构，而是通过将写入的数据记录在磁盘的日志上，一旦系统调用记录了全部的写入数据，就会在磁盘上写一个特殊的提交记录，表明日志包含一个完整的操作，系统调用再将日志的写入数据写到磁盘上，完成后，系统调用清空日志
** 崩溃如果发生在操作提交前，那么日志不会被标记为完成，磁盘状态像没开始一样；发生在操作提交之后，恢复代码就会重新执行写操作（可能会重复执行）

系统调用对log的使用：
```
begin_op()
...
bp = bread(...);
bp->data[...] = ...;
log_write(bp);
...
end_op();
```

kernel/log.c
// 目前对于log的认知是当修改了文件内容后，需要用log记录这些修改的块号，提交到磁盘
// 文件修改的进程需要做的事每次开始写文件的时候调用begin_op，结束的时候调用end_op来将修改提交到磁盘
logheader结构体：日志块的数量，扇区号数组（每个号对应一个日志块）
log结构体：锁；outstanding是当前系统调用的个数，log.outstanding*MAXOPBLOCKS计算已使用的日志空间【假设每个系统调用至多写入MAXOPBLOCKS个块】
initlog(): 用superblock中的信息初始化log.start[块号]，log的数量，设备号
recover_from_log(): 将log从disk读入--包括header和log/lh/block，并将需要提交的数目清空
read_head(): 读入log头[从磁盘中，log.start块号的data]到log.lh
install_trans(): 加载log的内容[对块号的记录从磁盘到本地log.lh.blcok中
write_head(): 将对log.lh的修改提交到磁盘
write_log(): 将修改的块号从cache【log.lh.block】提交到buffer中
begin_op(): 在开始每个文件系统的系统调用前调用，当还在提交log或是使用log块数超过容量就sleep，否则正在运行的数+1

```
  end_op(): 正在运行数-1，如果正在运行的系统调用数为0则进行提交（提交+唤醒）；否则唤醒在begin_op等待的进程
  commit(): 调用write_log提交log到磁盘的日志槽中，write_head是提交点，将header块写到磁盘上，install_trans从日志中读取每个块，写到文件系统对应位置，并将修改数清空log.lh.n，重新读入header，必须在下个事务开始前修改，这样崩溃不会导致重启后的恢复使用这次的header和下次的日志块
  log_write(): 记录指定buffer块的修改，将blockno记录在log.lh.block[i]中，对应的buffer的refcnt++
```

Lab9-mmap

最后一个Lab是在上面对文件系统的理解基础上实现mmap相关的系统调用，将指定文件映射到内存中，并实行懒分配的机制。主要是要对地址空间的分布要有把握，还有适当的调用读写的函数。相关代码在上一个Lab已经叙述过了，Lab的具体实现在报告也已有呈现。