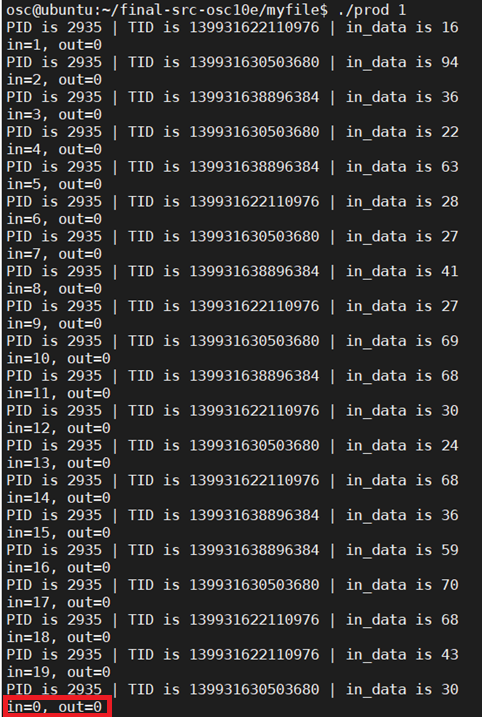
### 操作系统第一次大作业 （18364020刁琪）

1. 生产者和消费者的进程执行结果如下

运行程序时，先执行prod.c文件，生产者进程会产生数据到共享内存ShareBuffer1中，由于，缓冲区的大小为20，当in加到19时（图1红框显示），缓冲区满了，即下图1所示。此时生产者不能产生数据，不再打印输出。

然后，在另外一个窗口运行cons.c消费者程序，此时消费者开始消费缓冲区的数据（如图3所示），缓冲区有空位，由于在本次代码中设置的循环为15，生产者还会在消费者消费之前生产出的数据的同时，继续生产新的数据到缓冲区中，如图2所示。

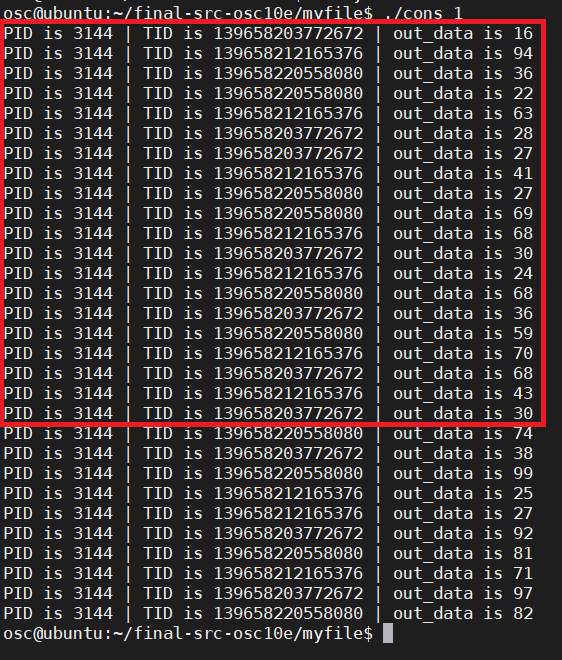


**图1 生产者先产生20个数据**

图片包含 文本

描述已自动生成

**图2 有空位后，生产者继续产生新数据**



**图3 消费者开始消费缓冲区中的数据**

**（先消费生产者已经生产好的前20的数据，后面继续消费生产者后生产的剩余数据）**

上面的测试，生产者和消费者的速度都是一样的。下面视频展示了当生产者和消费者的执行速度不一样时的情况。

在“prod速度更快”的视频中，我输入prod的λ为20，cons的λ为1，让生产者生产数据的速度大于消费者消费的数据，随后运行时发现，生产者很快就把数据生产好了，右面的消费者再还在慢慢地消费缓冲区的数据，等缓冲区有空位后，生产者才会进一步生产新数据。

在“cons速度更快”的视频中，输入cons的λ为20，prod的λ为1，让消费者消费的速度大于生产者生产的速度，运行时可看出，即使先运行生产者，再运行消费者，消费者的速度还是很快就赶上了生产者的速度。后半部分基本上就是生产者生产一个数据，消费者就消费一个数据。



1. 哲学家问题

文本

描述已自动生成

三、课程实验

1、sleep.代码如下：

文本

描述已自动生成

sleep.c代码运行情况如下：

文本

描述已自动生成

2、阅读xv6内核代码的进程和调度相关文件，围绕swtch.S、proc.h/proc.c，理解进程的基本数据结构，组织方式，以及调度方法

a)修改proc.c中的procdump函数，打印出各个进程的扩展信息，通^p来查看，代码及运行结果如下图所示：

文本

描述已自动生成

**procdump函数修改部分**

图片包含 文本

描述已自动生成

**使用^p查看到的结果**

b)

1. **swtch.S文件头部注释理解：**

保存当前进程的上下文

加载新选定的进程的上下文到机器的寄存器中

这里要说明的一点是，不管是sched函数还是scheduler函数调用swtch实现上下文切换之前，都会先拥有进程的锁p->lock。正常来说，持有锁的线程应该释放该锁，但是对于上下文切换来说，我们必须保证swtch调用者持有该锁，并将锁的控制权移交给切换代码。这样的话，我们才能保证进程的state和context在运行swtch时保持不变。

表格

描述已自动生成

Swtch.S文件具体代码分析：

第10~38行：把a0和a1都分成了14个小内存区域，每个小区域的大小为8个字节。

每个进程都有一个指向自己context结构的指针，当发生进程切换的时候，通过调用swtch(struct context \*old, struct \*new)来将当前（old）进程的context存到a0中，也将新进程（new）的context加载到a1。

1. **Scheduler函数的的头部注释的直译信息：**

每个CPU有相同的scheduler

每个CPU在初始化自身之后就会调用scheduler函数

每个scheduler调度器永远不返回，不断循环，循环的内容是：

* 选择一个进程来运行
* 有swtch函数来开始运行被选择的进程
* 最后，该进程同样调用swtch函数来将CPU的控制权交还给scheduler

文本

描述已自动生成

Scheduler()函数代码具体解释：

第470~490行：最外层的循环是一个无限循环，使得scheduler函数永远不会返回；

第472行：打开中断。是为了防止scheduler进入死锁后能去中断进程的调度；

第474行：进入第二层循环，调度器按顺序遍历进程表中的进程

第475行：获得锁，防止其他CPU对进程表中的进程状态做出修改，避免数据冲突；xv6使用acquire()来获得锁，并且在acquire函数中使用x86架构下的xchg指令来实现，xchg原子地交换一个寄存器和内存字的值，通过循环反复xchg，如果返回1，代表由CPU或者进程占用锁，继续循环（忙等待）；如果返回0，代表目前没有人占用锁，通过将锁置1占用，然后跳出循环，获得锁；

第476行：找到一个进程状态为RUNNABLE的进程p

第480行：将上面找到的进程p状态改为RUNNING

第481行：为这个进程p分配CPU，定为被选中的进程指针

第482行：调用swtch来切换上下文，将当前CPU各个寄存器的信息从scheduler的上下文切换为选定的进程p的上下文信息，直到进程p因为某些原因（比如时间片用完）调用swtch切换回scheduler的上下文

第486行：完成上面的内容后，将CPU上运行的进程置为0，进入下一次的循环；

第487行：表示通过scheduler完成了一个进程的执行

第488行：释放p的锁

第491-494行：这段代码是当遍历完整个进程队列后，找不到一个RUNNABLE的进程来执行，我们就会进入等待中断状态。asm表示后面的代码是汇编语言；wfi是指wait for interruption，进入等待中断状态

1. **sched函数头部注释直译信息：**

在进入scheduler函数之前，我们必须获得一个进程p的锁p->lock，并且当前进程的状态不能是RUNNING；

存储了当前进程的intena。因为intena是判断上锁之前系统中断是否已经被阻止，这是进程释放所有锁后是否允许系统中断的很重要的判断依据，而切换进程之后，这个值将会被新进程的intena顶替，所有要先存储intena，再进入scheduler。从scheduler回来后，再将intena赋值给mycpu（）\_>intena。保证数据不被破坏。

文本

描述已自动生成

我们再看sched具体的实现：

第506~511行：基本上就是检查了注释里面提到的几个条件有没有满足，

第512~513行：确保CPU在终端关闭的情况下运行

第515行：存储进入swtch之前，当前运行的进程的最后的信息intena。

第516行：然后直接调用swtch函数将当前进程的上下文与scheduler上下文进行切换。将当前的上下文保存在p->context，然后切换到调度器上下文cpu->scheduler。这里提醒以下，在调用swtch函数之前必须保证已经获得p->lock，为了保证当进程把自己的状态设置为RUNNABLE后，在切换上下文过程中没有其他CPU上的调度器scheduler可以执行它；

第517行：当scheduler调用swtch恢复当前进程的时候，当前进程会重新占有CPU，此时将intena归还给mycpu()->intena

1. **yield函数头部注释信息：**

yield函数的作用是结束当前时间片的运行，交出CPU的使用权。是进程主动放弃CPU时调用的函数

文本

描述已自动生成

Yield函数具体代码实现：

第525行：获得进程p的锁，因为进入sched()函数要求必须持有进程p的锁

第526行：将进程p的进程状态改为RUNNABLE，因为后面进入sched()函数要求进程的状态必须为RUNNABLE

第527行：进入sched()函数，交出CPU的使用权

第528行：CPU的使用权归还后，释放进程p的锁

1. **进程基本数据结构**

Xv6源代码中在proc.h头文件中声明了CPU、进程、进程上下文等数据结构。其中，每个进程都是使用结构体proc定义。这个结构体中包含了一个具体进程的所有信息，如进程号ID、进程状态、进程大小、该进程的父进程、打开的文件列表等等，具体代码如下图所示

文本

描述已自动生成

具体代码分析：（下面只列举出个别需要扩展说明的信息）

第87行：定义一个自旋锁lock；lock.locked代表锁的状态（1表示已锁住，0表示未锁住），为了防止在多核系统中，两个CPU同时获得锁，违反访问数据的互斥性。我们要将读写变量的过程连续进行不可打断，并在访问时只允许一个CPU进行，实现访问locked的原子操作。

Xv6实现自旋锁用于内核临界区的同步和互斥，看得出来自旋锁的效率很低，我们自然会想到使用更高效的基于等待队列的方法，让等待进程陷入阻塞而不是无限循环。但是，xv6允许同时运行多个CPU核，多核CPU上的等待队列实现起来相当复杂，因此使用自旋锁是相对比较简单且能正确执行的实现方案；

下面的信息需要获得锁才能使用：

第90行：进程状态，这些状态在枚举类型procstate中

第91行：进程的父进程指针，指向另外一个和该进程一样结构的父进程

第92行：如果不等于0，执行sleep（chan），让进程在任意的chan上休眠

为更好地了解进程调度框架，我们下面还要对sleep、wait和wakeup等函数进行分析，因为在多进程的系统中，进程间的相互协作、同步也是问题。Xv6系统中就是通过sleep与wakeup来完成进程之间的同步协作。

在xv6系统中，当一个消费者进程要去消费另一个生产者进程生产的信息时，如果此时没有信息可供消费者消费，这个消费者进程就会进入sleep函数中，直到生产者进程生产出产品，就会使用wakeup函数来唤醒消费者进程前来消费。而wait函数就是让消费者尝试去消费一个产品，如果消费成功（有可供消费的产品），那么wait就会直接返回，否则就会让消费者进入sleep，直到有产品生产出来。

1. **sleep函数头部注释信息**

sleep函数实现自动释放锁，并且让进程在chan上sleep，当被唤醒时重新获取被释放的锁。sleep函数中我们一定要保证我们获取到进程p的锁，才能去修改进程p的状态为SLEEPING。（关于为什么一定要获得锁会在后面解释完wakeup函数一起说明）

文本

描述已自动生成

具体代码实现：

第565~568行：获得进程p的锁

第571行：整理，让进程p在chan上沉睡

第572行：将进程p的状态改为SLEEPING

第574行：调用sched以释放CPU

1. **wakeup函数头部注释信息**

wakeup函数的作用是唤醒所有在chan上沉睡的进程（符合条件的进程）

文本, 信件

描述已自动生成

具体代码实现：

第593~597行：遍历进程表中的所有进程，找到进程状态为SLEEPING并且是在chan上沉睡的进程，将找到的进程唤醒（状态改为RNNNABLE）

第598行：释放进程p的锁

下面wakeup1函数也是为了唤醒进程，不过唤醒的是在wait()中沉睡的进程，由exit()调用

文本

描述已自动生成

**进程调度框架：**

进程的调度就是调度器scheduler不断地选择进程来执行。整个调度过程主要发生的就是进程上下文和调度器上下文的切换，通过普通的上下文切换来实现进程间的切换。

简单的调度流程就是：进程现根据当前时钟中断（或其他事件导致中断）调用yield函数来主动地放弃CPU。进入yield函数之后，主要调用sched函数；在sched函数中在保证获得进程p的锁后调用swtch函数。在sched执行完swtch后，就切换到之前保存的调度器程序（scheduler函数），在scheduler函数中不断遍历进程列表，找到可执行的进程，接着进行上下文切换。简单的切换顺序流程图如下图1所示：

图示

描述已自动生成

**图1 简单的调度流程**

总结一下，进程进入scheduler调度器主要会有两种方式。第一，进程自己要主动放弃CPU，会调用yield函数去进入到scheduler；第二是进程时间片用完或被其他事件导致中断，会直接进入scheduler选择下一个执行的进程。

若一个进程需要等待某个特定事件发生后才能发生，就会调用sleep函数将自己睡眠（状态改为SLEEPING），这样在scheduler遍历寻找下个执行进程时就会忽略它。当它的特定事件发生后，会调用wakeup函数（由特定事件发生的进程调用）来唤醒这个沉睡的进程（将状态改为RUNNABLE），在下次的scheduler遍历中就可以选中它继续执行。这些函数之间的具体关系由下图2可看出。

图示, 示意图

描述已自动生成

**图2 一些函数之间的关系**

下面我总结了一些对调度框架的一些理解：

1. 每个xv6进程都有自己的内核栈以及寄存器集合，每个cpu都有一个单独的调度器线程，因此调度不会发生在进程的内核线程中，而是发生在调度器线程中。也就是说xv6永远不会直接从一个进程的上下文切换到另一个进程的上下文，它们是通过一个中间的内核线程实现，正如下图3所示。

这是因为进程退出时需要释放自己占有的内存和资源，但进程本身去释放资源的时候又需要使用资源来释放，所以不能由进程本身来释放自己占有的资源，需要借助中间线程来帮助进程释放资源。

图示

描述已自动生成

**图3 上下文切换具体实现**

1. 从上面对scheduler函数的代码分析可以看出scheduler函数的大致框架与RR调度算法相似，是一个基于时间片的调度算法。给每次选中执行的进程一定的运行时间，使用完时间片后就要交换CPU，回到就绪队列中等待下一次的分配。

但是xv6中调度算法的就绪队列不太一样。可以看出来xv6系统实现的所谓RR算法虽然满足了轮询的要求，但是并没有按照FCFS的原则来排就绪队列，而是按照固定的顺序（进程在进程表中的顺序）来排序。

也就是说，如果表中第i个进程正在运行，第i+2个进程处于就绪状态，那么该进程就必须等第i个进程交出CPU后才能运行。但是如果在i+2等待期间，第i+1的进程也完成了必要的工作，就会处于就绪状态，那么接下来调度器不会将CPU的控制权优先交给i+2，而是交给后面进入就绪队列的i+1进程。说明xv6中的就绪队列排队方式没有遵守FCFS的原则，而是按照进程表中的固定顺序排列。但是总体上，这样的偏差不会引起饥饿，这样的实现手段仍然能保证每个处于就绪状态的进程能在可以预期的时间内获取CPU的使用权。

1. 进程切换时的锁的问题：

为了对应同时有多个CPU切换进程的情况，我们要使用带锁的方案来避免错误。通过过观察scheduler和yield的代码可知，进程表的锁总是由旧进程获得，新进程释放，这样做是为了保护进程切换能够正常运行。

同时再scheduler函数中，为了防止当前的CPU一直占用锁，其他CPU无法调度运行导致的死锁，在外层循环释放进程表锁（有两个锁，一个是进程表锁，一个是进程锁，此时仍然拥有进程锁）。

1. 沉睡与唤醒：

沉睡和唤醒实际上提供了进程间通信的机制。

sleep和wakeup系列函数的存在主要是为了解决“丢失的唤醒“问题。因为有时候可能当某一进程检测到睡眠条件时，另一个进程触发了唤醒条件但是却没有唤醒任何进程，那么当前进程就会继续沉睡，永远无法被唤醒。这是因为沉睡操作和检测睡眠条件操作不是一个原子操作，可能检测到某一个进程不是沉睡状态，但是马上又将其沉睡，系统就会忽略它，一直不唤醒。

xv6系统为了解决这个问题，将等待队列加入锁来实现，进程在检测睡眠条件之前获得锁，并且在sleep中将锁释放。由于sleep修改进程表数据（如进程状态）也需要进程锁，进程锁和等待队列的锁中任意一个锁都可以将沉睡操作和检测沉睡条件操作置为原子操作，wakeup在调用前总是需要获得等待队列的锁，并且在wakeup操作中也需要使用进程表的锁。

c) 对照Linux的CFS进程调度算法，指出xv6的进程调度有何不足；设计一个更好的进程调度框架，可以用自然语言（可结合伪代码）描述，但不需要编码实现。

1、Linux的CFS（完全公平调度程序）进程调度算法：

Linux系统的调度基于调度类，内核针对不同的调度类采用不同的调度算法，这些调度算法是可以动态添加的。这里的CFS完全公平调度算法是针对一些普通进程的调度类。

CFS算法的主要思想是：不采用严格规则来为一个优先级分配某个长度的时间片，而是为每个任务分配一定比例的CPU处理时间。允许每个进程先运行一段时间、循环轮转，然后选择运行最少的进程作为下一个运行进程。所谓的运行最少并非单指实际物理运行时间最少，而是基于实际物理运行时间和虚拟运行时间vruntime。如果一个进程得以执行，随着时间的增长（一个个tick的到来），其虚拟运行时间vruntime不断增大，没有得到执行的进程vruntime不变。

在CFS中的友好值(nice value)不是用来计算分配给每个进程时间片，而是在所有可运行进程总数的基础上计算出一个进程运行时间。与此同时友好值用来计算进程获得CPU运行时间的比重，较低的友好值表示较高的相对优先级，获得更高比例的处理器处理时间。所以各个量的计算公式如下：

1. 一个进程在一个调度周期中的运行时间：

分配给进程的运行时间 = 调度周期 \* 进程权重 / 所有进程权重之和

调度周期是固定的，可以看出进程的权重越大，分到的运行时间越多

1. 一个进程的实际运行时间和虚拟运行时间的关系

vruntime = 实际运行时间 \* NICE\_0\_LOAD / 进程权重

=（调度周期 \* 进程权重 / 所有进程权重之和）\* NICE\_0\_LOAD / 进程权重

其中NICE\_0\_LOAD是友好值nice为0时的进程权重。从中可以看出，进程权重越大，运行相同的实际时间，vruntime增长地越慢。

总结来说，进程友好值nice越小，权重越大，所分配地运行时间越多；两个权重不同地进程，有相同的实际时间，权重大的进程vruntime增加的要少一些。所以vruntime可以反映该进程当前已经运行了多长时间。在CFS调度中，调度器每次选择vruntime最小的进程作为下一个进程，因为该进程的vruntime小说明之前使用CPU的时间短，因此可以让各个进程公平竞争CPU，也让优先级大的进程实际运行时间多一点。

2、xv6进程调度的不足

上面介绍的LINUX系统的CFS完全公平调度算法很好地解决了公平问题。xv6中的进程调度算法还是比较原始的算法，它虽然可以实现轮询，但是无法解决线程之间优先级不同的问题。每次轮询都是按照固定的进程表遍历，优先级高的线程（比如先进入就绪状态的线程）在面对表中顺序比自己前的线程时，没有优先运行的权力。

3、更好的调度框架

根据上面的分析，我们可以让xv6的调度算法仿照CFS算法，在优先级上做些优化，即让scheduler遍历选择下一个进程的之后优先选择优先级高的进程优先执行。这样就让原来的xv6调度算法完成摒弃时间片的概念，而是分配一个处理器使用比来度量

根据上面的CFS算法分析可知，虚拟时间vruntime越小的进程越应该给它分配CPU，即优先级越高。

**step1：**在每一个进程描述符(struct proc)里面添加一个调度实体sched\_entity，这个实体里面保留着进程的虚拟运行时间vruntime和权重，其记录着一个程序到底运行了多长时间以及应该再运行多久。而且这每个进程描述符里面还要为每个CPU维护一个调度队列cfs\_queen，例如下面所示的实体:

struct sched\_entity {

int vruntime;

struct cfs\_queen;

……

}

**Step2：**所有的sched\_entity实体以vruntime为key插到红黑树中，同时为了能够迅速选中vruntime最小的进程，缓存树的最左侧节点（vruntime最小的节点）。这里使用红黑树这个数据结构，是因为红黑树的典型用途是实现关联数组，存储有序的数据

**Step3：**在scheduler函数中遍历进程表选择vruntime最小的进程去实现进程切换。实现调度算法的优先级的考虑。其简单的伪代码实现如下：

**strcut proc \*p**

**struct cpu \*c <- mycpu()**

**c->proc <- 0**

**while 1 do**

**inter\_on()**

**for p &proc[0] to &proc[NPROC] by 1 do**

**acquire(p->lock)**

**p <- mycpu.tree\_get\_mintime()//在红黑树中找到vruntime值最小的进程来切换**

**c->proc <- p**

**swtch(&c->context, &p->context)//选中的进程进行切换，获得CPU**

**//计算运行时间**

**int exec\_time <- cul\_time(p, vruntime, nice)**

**//根据公式（调度周期 \* 进程权重 / 所有进程权重之和）\* NICE\_0\_LOAD / 进程权重**

**int new\_vruntime <- cul\_new\_vruntime(p, exec\_time)**

**//更新vruntime**

**vruntime <- new\_vruntime**

**//将更新后的进程加回到红黑树中**

**tree\_add\_newentity(&p)**

**//后面不断循环**