

**操作系统大作业**

**实验报告**

**姓 名：姜才武**

**年 级：2017级**

**专 业：智能科学与技术1班**

**时 间：2019/10/23-2019/11/04**

**目录**

[**一、哲学家就餐问题** 3](#_Toc23711408)

[1.问题描述 3](#_Toc23711409)

[2.约束条件 3](#_Toc23711410)

[3.解决方法 3](#_Toc23711411)

[4.结果展示 3](#_Toc23711412)

[**二、生产者消费者问题** 4](#_Toc23711413)

[1.问题描述 4](#_Toc23711414)

[2.问题背景 5](#_Toc23711415)

[3.解决方法 5](#_Toc23711416)

[4.结果展示 5](#_Toc23711417)

[**三、Linux内核实验** 6](#_Toc23711418)

[a. 6](#_Toc23711419)

一、哲学家就餐问题

****1.问题描述****

问题是五个哲学家围在一张圆桌上，圆桌上有五个碗和五只筷子，平时哲学家思考，饥饿时就试图拿起两只筷子进餐，进餐完毕会放下筷子继续思考。

****2.约束条件****

①只有拿到两只筷子时，哲学家才能吃饭。

②如果筷子已被别人拿走，则必须等别人吃完之后才能拿到筷子。

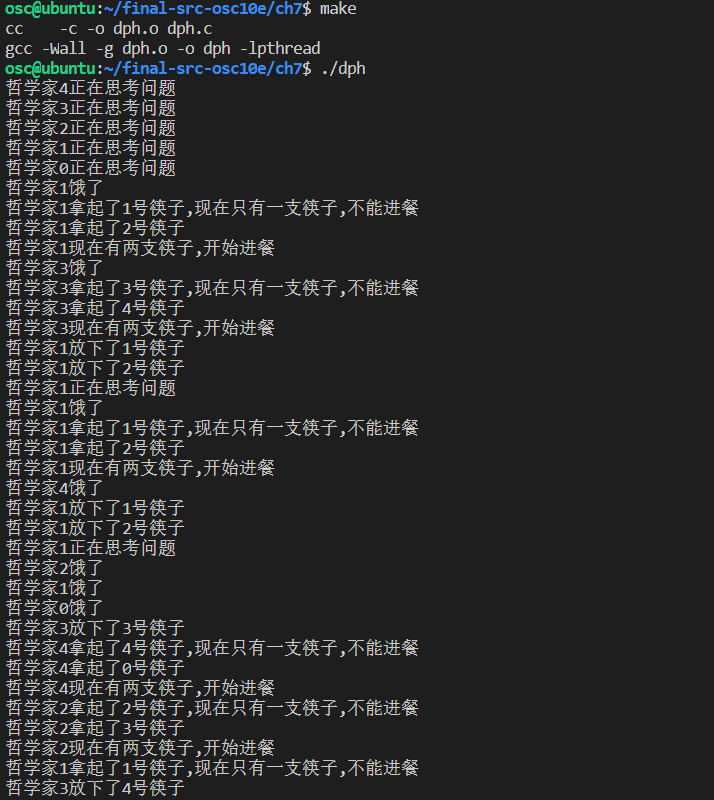
③任意一哲学家在自己未拿到两只筷子吃饭前，不会放下手中拿到的筷子。

****3.解决方法****

直接的解决方法就是五只筷子分别设置初始值为1的互斥信号量，保证不会有相邻的哲学家同时进餐，但是若五位哲学家同时拿起左筷子，则会因为无法拿到右筷子而引起死锁。

所以产生的解决死锁的方法如下：通过互斥信号量mutex对哲学家进餐之前取左侧和右侧筷子的操作进行保护，可以防止死锁的出现。（当第i个哲学家将左右筷子都拿到了才允许其他哲学家拿筷子）

****4.结果展示****



二、生产者消费者问题

****1.问题描述****

生产者消费者问题（英语：Producer-consumer problem），也称有限缓冲问题（英语：Bounded-buffer problem），是一个多线程同步问题的经典案例。该问题描述了共享固定大小缓冲区的两个线程——即所谓的“生产者”和“消费者”—— 在实际运行时会发生的问题。生产者的主要作用是生成一定量的数据放到缓冲区中，然后重复此过程。与此同时，消费者也在缓冲区消耗这些数据。该问题的关键就是要保证生产者不会在缓冲区满时加入数据，消费者也不会在缓冲区为空时消耗数据。

****2.问题背景****

1.共享内存：共享存储是进程间通信的一种手段，通常，使用信号量同步或互斥访问共享存储。共享存储的原理是将进程的地址空间映射到一个共享存储段。在LINUX下，通过使用 shmget 函数创建或者获取共享内存。

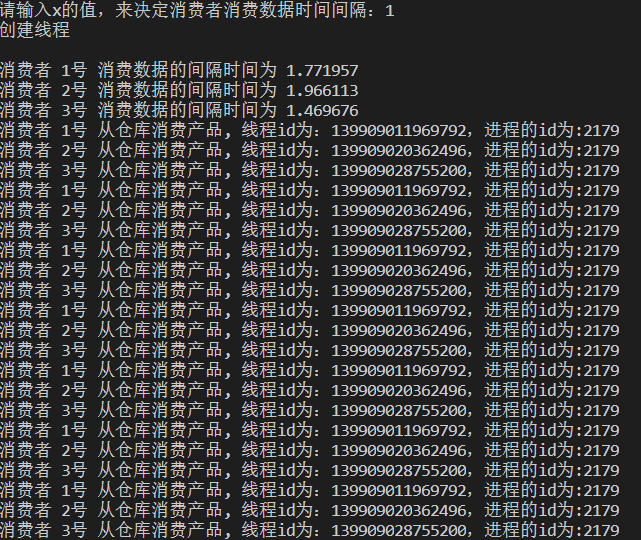
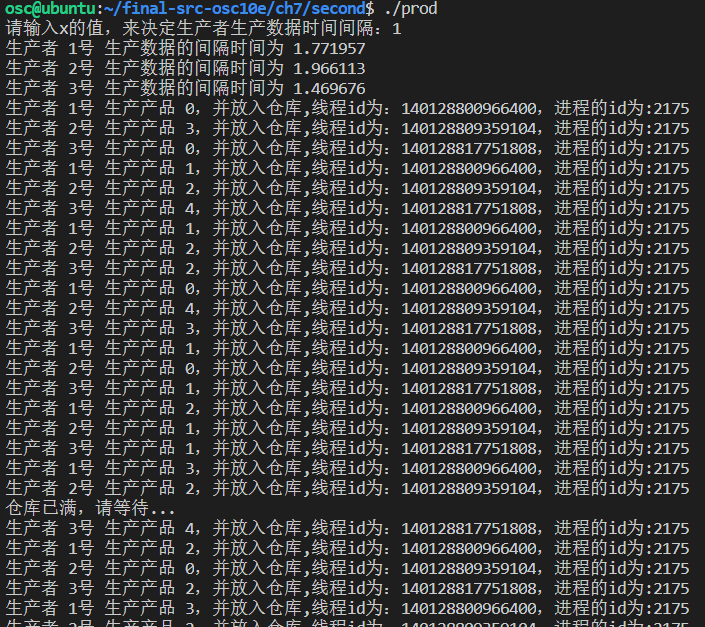
2.信号量：共需要三个信号量:第一个信号量用于限制生产者必须在缓冲区不满时才能生产，是同步信号量。第二个信号量用于限制消费者必须在缓冲区有产品时才消费，是同步信号量。第三个信号量用于限制生产者和消费者在访问缓冲区时必须互斥，是互斥信号量

****3.解决方法****

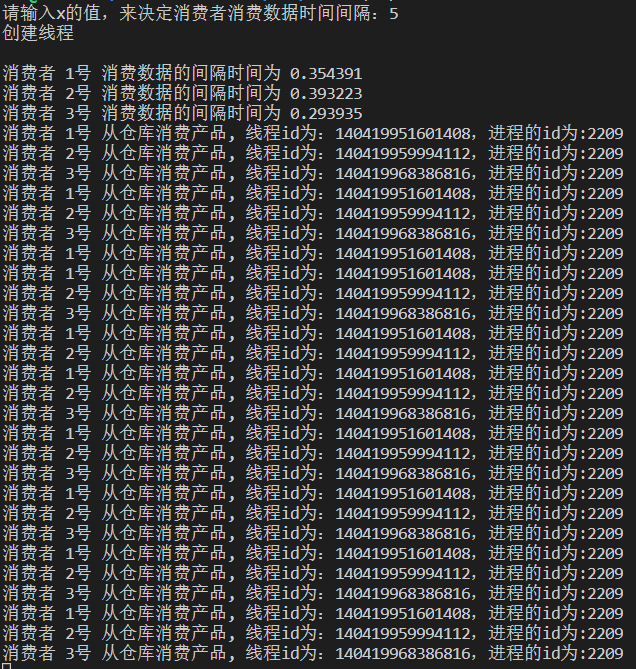
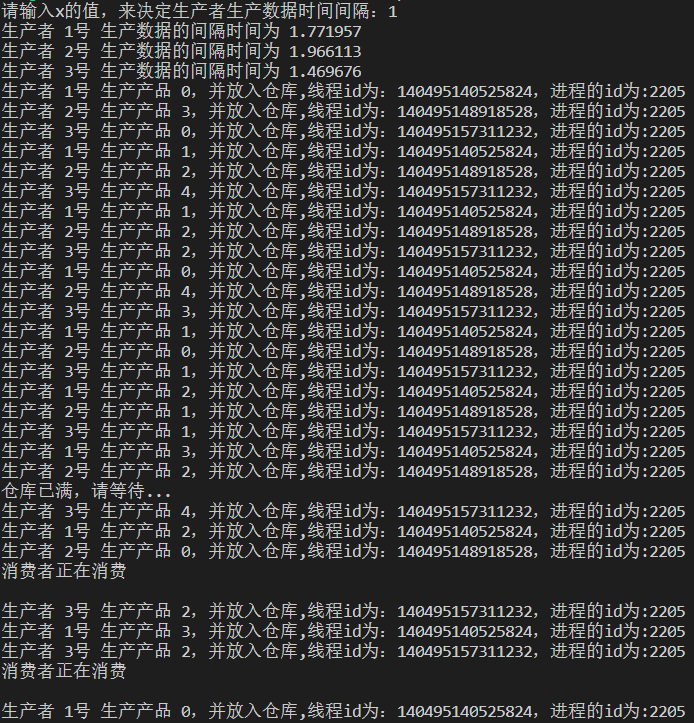
要解决该问题，就必须让生产者在缓冲区满时休眠（要么干脆就放弃数据），等到下次消费者消耗缓冲区中的数据的时候，生产者才能被唤醒，开始往缓冲区添加数据。同样，也可以让消费者在缓冲区空时进入休眠，等到生产者往缓冲区添加数据之后，再唤醒消费者。通常采用进程间通信的方法解决该问题。如果解决方法不够完善，则容易出现死锁的情况。出现死锁时，两个线程都会陷入休眠，等待对方唤醒自己。该问题也能被推广到多个生产者和消费者的情形。

****4.结果展示****

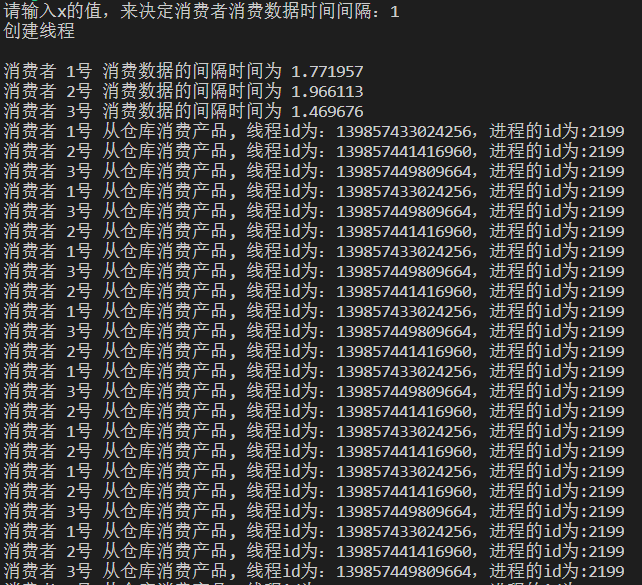
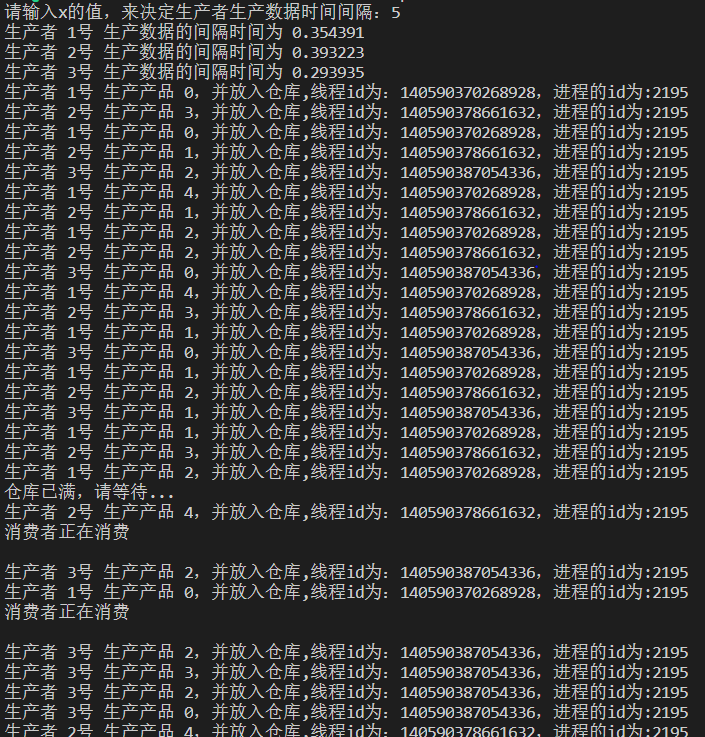
①当生产者的x输入1，消费者x输入1时：



②当生产者的x输入1，消费者x输入5时：



③当生产者的x输入5，消费者x输入1时：



三、Linux内核实验

下载了与虚拟机种linux系统版本一致的内核：“linux-source-4.4.0.tar.bz2，并且解压安装。

****a.****

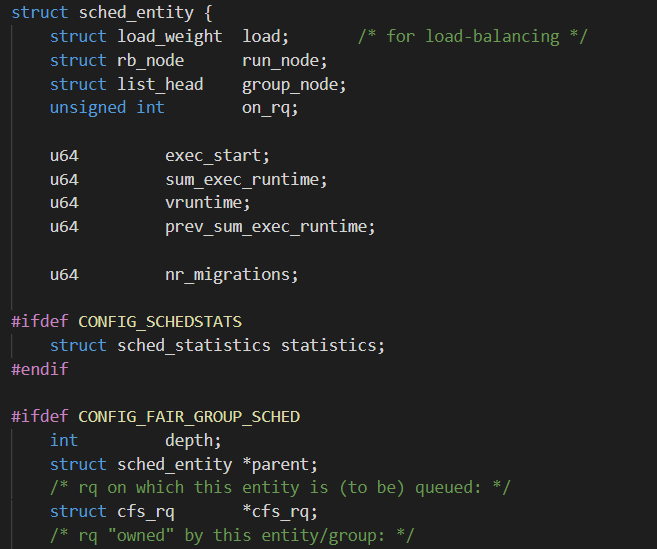
Linux系统所定义操控的一个特殊的数据结构实例叫做进程。它连接了用户代码，拥有代码运行所需的独立内存空间，在调度器的调度下使用分配给它的处理器时间片来运行。

Linux系统一个被创建的进程的状态可分为5种：就绪状态和运行状态、可中断等待状态、不可中断等待状态、停止状态、中止状态。进程的当前状态记录在进程控制块的state成员中。不同状态标志的state的值各不相同，在进程的整个生命周期中，它可在5种状态之间转换。

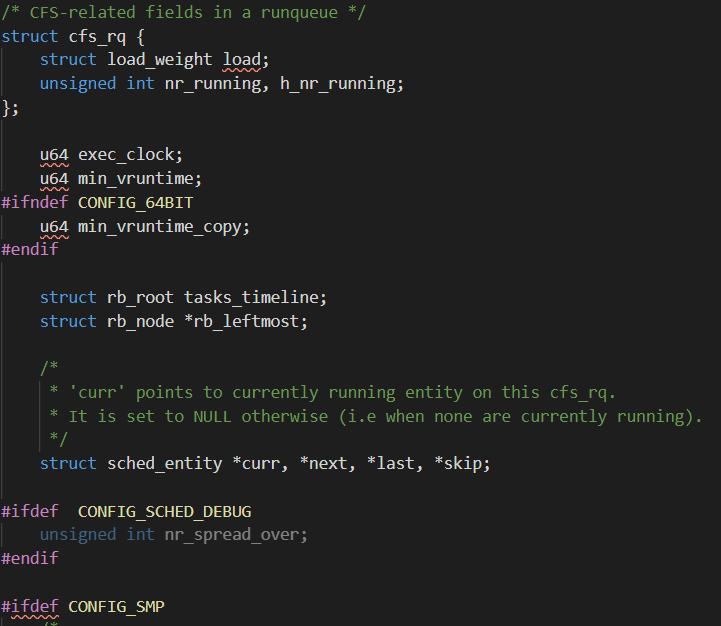
CPU调度就是当前进程需要进行I/O操作或者时间片结束了，如何从就绪队列中选择下一个执行的进程。它有FIFO、优先级等方法来判断进程执行的先后顺序。主要指标就是尽快结束任务：周转时间（任务进入到结束的时间）短。用户操作尽快响应：响应时间（从操作发生到响应）短。系统内耗时间少：吞吐量（任务的完成量）大。总原则：系统专注与任务执行，又能合理调配任务。

而CFS调度方式是思路就是当进程得到调度以后，根据各个进程的权重分配运行时间。或者可以这么理解：CFS的思想就是让每个调度实体（可以对应一个调度组也可以对应一个进程）的vruntime互相追赶，而每个调度实体的vruntime增加速度是不同的，权重越大的增加的越慢，这样就能获得更多的cpu执行时间。

下图为调度部分截图：



每一个进程或者调度组都对应一个调度的实体，每一个进程都通过调度实体与CFS运行对列建立联系，每次进行CFS调度的时候都会在CFS运行对列红黑树中选择一个进程。下图为CFS运行队列部分：



1）**进程优先级**指的是在进程cpu资源分配的过程中，先后执行该进程的权利。

一个进程的优先级与该进程的PRI值决定，PRI值会被一个NI值定期修正，即PRI(new)=PRI(old)+nice。而NI就是**nice**值

所以当nice为负值时，程序的PRI减小，优先级降低。反之则该进程的优先级升高。

但是nice值不能代表进程的优先级，它只是一个影响的参数。

而**权重**代表的进程对于CPU使用时间的占有率，所以它的大小跟进程优先级相关，优先级越高，抢占的CPU时间越多，因此它的权重就越大。

而权重与nice值之间存在一种映射关系，约定nice值为0的进程权重值为1024，其他nice值的权重值都是基于1024计算出来的，高一优先级的进程权重总是低一优先级进程的1.25倍，可以根据查表获取。

2）CFS调度器中的**vruntime基本思想**是用它来衡量哪个进程最值得被调度，CFS中的就绪队列是一棵以vruntime为键值的红黑树，虚拟时间越小的进程越靠近整个红黑树的最左端。因此，CFS每次选择位于红黑树最左端的那个进程，该进程的vruntime最小。

Vruntime也就是进程的虚拟运行时间，它是自进程诞生以来进行累加的，每个时钟周期内一个进程的虚拟运行时间 = 实际运行时间\*（NICE\_0\_LOAD/权重）。

NICE\_0\_LOAD是nice为0时的权重。也就是说，nice值为0的进程实际运行时间和虚拟运行时间相同。通过这个公式可以看到，权重越大的进程获得的虚拟运行时间越小，那么它将被CFS所调度的机会就越大。

每一个进程拥有一个vruntime, 每次需要调度的时候就选运行队列中拥有最小vruntime的那个进程来运行也就是所谓的**min\_vruntime**, 　创建新进程，加入就绪队列，调度tick等都会更新当前vruntime值。

而min\_vruntime在CFS就绪队列数据结构中，单步递增，用于跟踪该就绪队列红黑树中最小的vruntime。

b）