

《操作系统课程实验》

实验报告

**学 院**  计算机学院

**姓 名**

**学 号**

**班 级**

**专 业** 计算机科学与技术

《操作系统课程实验》  
实验报告



|  |  |
| --- | --- |
| **学院：** | **计算机学院** |
| **班级：** |  |
| **姓名：** |  |
| **学号：** |  |

目录

[1. 实验目的 6](#_Toc92715965)

[2. 实验环境 6](#_Toc92715966)

[3. 实验内容 6](#_Toc92715967)

[3.1 进程创建与管理 7](#_Toc92715968)

[3.2 线程创建管理与通信 7](#_Toc92715969)

[1. 线程创建与管理 7](#_Toc92715970)

[2. 线程通信 8](#_Toc92715971)

[3.3 进程通信 8](#_Toc92715972)

[1. 基于消息队列的进程通信 8](#_Toc92715973)

[2. 共享内存通信 8](#_Toc92715974)

[3. 管道通信 9](#_Toc92715975)

[4. 信号signal通信 9](#_Toc92715976)

[3.4 进程/线程同步互斥 10](#_Toc92715977)

[3.4.1 实验题目 10](#_Toc92715978)

[3.4.2 进程/线程同步互斥的系统调用和API库函数 12](#_Toc92715979)

[3.5 多核多线程编程及性能分析 14](#_Toc92715980)

[4 第一组 进程创建与管理 14](#_Toc92715981)

[Linux进程和task\_struct结构分析 15](#_Toc92715982)

[任务状态 15](#_Toc92715983)

[任务ID 16](#_Toc92715984)

[任务存储结构 17](#_Toc92715985)

[进程关系 17](#_Toc92715986)

[进程权限 17](#_Toc92715987)

[内核态函数栈 19](#_Toc92715988)

[父子进程创建与控制 21](#_Toc92715989)

[实验设计原理 21](#_Toc92715990)

[父子进程创建与控制 22](#_Toc92715991)

[fork()和clone()的差异 25](#_Toc92715992)

[clone()参数的影响 27](#_Toc92715993)

[Linux命令功能和使用 28](#_Toc92715994)

[Ps 28](#_Toc92715995)

[Top 29](#_Toc92715996)

[Pstree 30](#_Toc92715997)

[Vmstat 31](#_Toc92715998)

[Strace 31](#_Toc92715999)

[ltrace 33](#_Toc92716000)

[Sleep 34](#_Toc92716001)

[Kill 34](#_Toc92716002)

[Jobs 35](#_Toc92716003)

[第二组 线程创建管理与通信 35](#_Toc92716004)

[线程创建管理 35](#_Toc92716005)

[实验设计原理（Pthread 线程库背景知识） 36](#_Toc92716006)

[程序代码 39](#_Toc92716007)

[实验结果及分析 43](#_Toc92716008)

[线程通信 45](#_Toc92716009)

[程序代码 45](#_Toc92716010)

[实验结果与分析 48](#_Toc92716011)

[第三组 进程通信 48](#_Toc92716012)

[1. 基于消息队列的进程通信 48](#_Toc92716013)

[实验设计原理 48](#_Toc92716014)

[程序代码 50](#_Toc92716015)

[实验结果及分析 52](#_Toc92716016)

[3. 共享内存通信 52](#_Toc92716017)

[实验设计原理 52](#_Toc92716018)

[程序代码 53](#_Toc92716019)

[实验结果及分析 55](#_Toc92716020)

[3. 管道通信 56](#_Toc92716021)

[实验设计原理 56](#_Toc92716022)

[程序代码 58](#_Toc92716023)

[实验结果及分析 60](#_Toc92716024)

[4. 信号signal通信 61](#_Toc92716025)

[实验设计原理 61](#_Toc92716026)

[程序代码 63](#_Toc92716027)

[实验结果及分析 65](#_Toc92716028)

[**第四组 进程/线程间同步与互斥** 65](#_Toc92716029)

[**实验题目** 65](#_Toc92716030)

[**实验原理** 65](#_Toc92716031)

[**进程/线程同步互斥的系统调用和 API 库函数** 65](#_Toc92716032)

[**实验题目及分析** 68](#_Toc92716033)

[**实验题目** 68](#_Toc92716034)

[**实验分析** 69](#_Toc92716035)

[**代码实现** 70](#_Toc92716036)

[**条件变量实现（3\_phr\_mutx\_cond.cpp）** 70](#_Toc92716037)

[**Pthread/POSIX信号量实现（3\_phr\_mutx.c）** 72](#_Toc92716038)

[**结果分析** 76](#_Toc92716039)

[**条件变量实现** 76](#_Toc92716040)

[**Pthread/POSIX信号量实现** 77](#_Toc92716041)

[第五组 多核多线程编程及性能分析 77](#_Toc92716042)

[观察实验平台物理cpu、CPU核和逻辑cpu的数目 78](#_Toc92716043)

[实验设计原理 78](#_Toc92716044)

[程序代码 78](#_Toc92716045)

[实验结果与分析 79](#_Toc92716046)

[单线程/进程串行vs 两线程并行vs 三线程加锁并行程序对比 79](#_Toc92716047)

[实验设计原理 79](#_Toc92716048)

[程序代码 80](#_Toc92716049)

[实验结果与分析 84](#_Toc92716050)

[三线程加锁vs 三线程不加锁对比 86](#_Toc92716051)

[实验设计原理 86](#_Toc92716052)

[程序代码 86](#_Toc92716053)

[实验结果与分析 88](#_Toc92716054)

[针对Cache的优化 89](#_Toc92716055)

[实验设计原理 89](#_Toc92716056)

[程序代码 90](#_Toc92716057)

[实验结果与分析 93](#_Toc92716058)

[CPU亲和力对并行程序影响 96](#_Toc92716059)

[实验设计原理 96](#_Toc92716060)

[程序代码 96](#_Toc92716061)

[实验结果与分析 99](#_Toc92716062)

[八种实现方案运行时间对比总结 102](#_Toc92716063)

[**第七组 内存管理** 102](#_Toc92716064)

[**一、实验目的** 102](#_Toc92716065)

[**二、实验环境** 102](#_Toc92716066)

[**三、实验内容** 103](#_Toc92716067)

[**3.1 写拷贝 COW** 103](#_Toc92716068)

[**3.2 内存映射文件 mmap** 103](#_Toc92716069)

[**实验原理** 104](#_Toc92716070)

[**4.1 写拷贝 COW 使用的系统调用和函数** 104](#_Toc92716071)

[**4.2 内存映射文件 mmap 使用的系统调用和函数** 105](#_Toc92716072)

[**实验过程** 107](#_Toc92716073)

[**写拷贝COW** 107](#_Toc92716074)

[**内存映射文件 mmap** 116](#_Toc92716075)

[**实验总结** 119](#_Toc92716076)

[**碰到的问题及解决** 119](#_Toc92716077)

[**mmap中的offset** 119](#_Toc92716078)

[**mmap和read执行时间相近** 120](#_Toc92716079)

# 实验目的

1. 以Linux操作系统为实验对象，理解Linux系统中任务（task）和进程、线程等概念，掌握利用Linux API函数/系统调用创建和管理进程的方法，认识进程生命周期、状态转换和并发执行的实质。
2. 理解进程和内核级、用户级线程的概念，掌握在Linux操作系统环境下，采用Pthread线程库创建和管理线程的方法，观察分析线程并发执行和通信行为。
3. 掌握Linux提供的消息队列、共享内存、管道、signal信号（软中断）四种进程间通信的原理和方法，采用系统调用和库函数编程实现这三种进程间通信机制。
4. 掌握Linux提供的内核信号量、用户态信号量的原理和方法，以及系统调用和API函数，采用这两类进程/线程同步互斥方式机制，针对生产者-消费者、哲学家就餐、睡眠理发师等经典同步互斥问题的扩展问题，设计并编程实现解决方案，观察验证方案正确性，加深对复杂进程线程同步互斥问题的理解。

或：了解C++11等程序设计语言提供的线程同步互斥机制（类似于教科书中管程monitor），掌握使用互斥变量和条件变量实现多线程间的同步互斥的方法，编程实现上述典型同步互斥问题。

1. 通过Linux环境下多核多线程编程，定量分析线程自身业务逻辑、并发线程数目、线程间同步互斥、CPU亲和、cache优化对多线程并发/并行线程执行效率和系统吞吐量的影响，初步掌握针对复杂任务的多核多线程编程及性能分析方法。

# 2. 实验环境

硬件：阿里云弹性计算服务ECS (Elastic Compute Service)，多核CPU，内存不少于8G；

或：微机，笔记本电脑，服务器

软件：Linux操作系统（内核4.0及以上），如Ubuntu、CentOS、AliOS等；

【在阿里云ECS实验环境下，操作系统为centOS 8.2或7.9，内核版本为3.？】

Pthread/POSIX线程库，System V；gcc编译；

C/C++，Java，Python等

# 3. 实验内容

在Linux环境下，采用C/C++/Java（或其它语言）编程，完成以下实验内容。

## 3.1 进程创建与管理

在Linux环境下，采用C/C++/Java（或其它语言）编程，完成以下实验：

1. 参照Linux内核源码结构，阅读Linux内核源码，分析Linux进程的组成，观察Linxu进程的task\_struc等进程管理数据结构；
2. 参照相关示例程序，查阅Linux系统调用等相关资料，设计父进程和子进程的业务处理逻辑；利用C++等高级程序设计语言和fork()、clone、exec()、wait()、exit()、kill()、getpid()等系统调用，创建管理进程，观察父子进程/线程的结构和并发行为，掌握等待、撤销等进程控制方法。

要求：

* 至少用到fork()、clone、exec、wait、exit、kill、getpid等6个系统调用；
* 所创建的父子进程各自具有不同的业务处理逻辑，父进程创建子进程后，子进程通过exec()调入自身执行代码；
* 进程可以自己通过exit()主动结束，也可以被父进程执行kill()命令来结束；
* 观察对比通过fork()和clone()创建的子任务/进程/线程的差异，分析clone()系统调用中设置与不设置CLONE\_FS、CLONE\_VFORK、CLONE\_FILES、CLONE\_FS、CLONE\_PID等参数对所创建的子进程/线程的影响；
* 在创建的父子进程/线程代码中的不同位置处增加随机延迟，使得进程执行横跨多个时间片，如通过增加数十到数百毫秒的延迟，保证进程执行时间不少于3个时间片。

1. 掌握ps、top、pstree –h、vmstat、strace、ltrace、sleep x、kill、jobs等命令的功能和使用方式，利用这些命令观察进程的行为和特征。

## 3.2 线程创建管理与通信

### 1. 线程创建与管理

查阅Linux和Pthread线程库相关资料，参照相关示例程序，设计父进程/线程和子线程的业务处理逻辑；利用pthread\_create、pthread\_exit、pthread\_cancel、pthread\_join、pthread\_self

等线程管理函数，创建和管理线程，观察父子进程/线程的结构和并发行为，掌握等待、退出、撤销等线程控制方法。

要求：

⚫ 在一个进程内创建多个子线程（如不少于3个子线程），父子子线程具有各自不同的业务处理逻辑，子线程执行自身特定的线程函数。父子线程间通过pthread\_join()函数实现同步和资源释放；

⚫ 一个进程内的多个子线程分别以不同方式终止或退出执行，如通过pthread\_exit()和return自己主动终止，或被被其它线程通过pthread\_cancel被动终止，观察对比以不同方式退出执行的子线程的行为差异；

⚫ 在创建的父子进程/线程代码中的不同位置处增加数十到数百毫秒的随机延迟（使用sleep()），使得进程和线程的执行横跨多个时间片，保证线程执行时间不少于3个时间片。

### 2. 线程通信

属于同一个进程中的多个线程使用相同的地址空间，共享大部分数据，一个线程的数据可以直接为其它线程所用，这些线程相互间可以方便快捷地利用共享数据结构进行通信。

编写程序，创建线程，实现主线程与子线程间、子线程相互间通过共享数据类型，如整型变量、字符串、结构体等传递信息，进行通信。

## 3.3 进程通信

查阅资料，学习掌握Linux系统提供的用于四种进程间通信的系统调用、库函数的使用方法和参数，参照样例程序，设计完成以下四组实验。

### 1. 基于消息队列的进程通信

Linux消息队列包括POSIX消息队列、System V消息队列。发送者进程向消息队列中写入数据，接收者进程从队列中接收数据，实现相互通信。消息队列克服了信号signal承载信息量少，管道pipe只能承载无格式字节流以及缓冲区大小受限等缺点。

要求：编程实现发送者和接收者两个并发进程：

（1）发送者和接收者使用msgget(key\_t, key, int msgflg)、msgctl(int msgid, int command, struct msgid\_ds \*buf)创建、管理消息队列。只有接收者在接收完最后一个消息之后，才删除消息。

（2）发送者使用msgsend(int msgid, const void \*msg\_ptr, size\_t msg\_sz, int msgflg)向消息队列不断写入数据，并打印提示信息；

（3）接收者使用msgrcv(int msgid, void \*msg\_ptr, size\_t msg\_st, long int msgtype, int msgflg)从消息队列中接收消息，并打印提示信息。

### 2. 共享内存通信

Linux内核支持多种共享内存方式，如mmap()系统调用，POSIX共享内存，以及System V共享内存。本实验采用System V共享内存实现方法。

要求：编程实现写者进程Writer和读者进程Reader，

（1）写者进程和读者进程使用shmget(key\_t key, int size, int shmflg)创建在内存中创建用于两者间通信的共享内存，使用shmat(int shmid, char\*shmaddr, int flag)将共享内存映射到进程地址空间中，以便访问共享内存内容；

（2）写者进程向共享内存写入多组数据，读者进程从共享内存d读出数据；

（3）进程间通信结束后，写者进程和读者进程使用shmdt(char\*shmaddr)解除共享内存映射，使用shmctl(int shmid, int cmd, struct shmid\_ds\*buf)删除共享内存。

原理：

进程间需要共享的数据存放于一个称为IPC共享内存的内存区域，需要访问该共享区域的进程需要将该共享内存区域映射到本进程的地址空间中。系统V共享内存通过shmget 获得或创建一个IPC共享内存区域，并返回相应的标识符。内核执行 shmget创建一个共享内存区，初始化该共享内存区对应的struct shmid\_kernel结构，同时在特殊文件系统shm中创建并打开一个同名文件，并在内存中建立起该文件对应的dentry和inode结构。新打开的文件不属于任何一个特定进程，任何进程都可以访问该共享内存区。

所创建的共享内存区有一个重要的控制结构 struct shmid\_kernel，该结构联系内存管理和文件系统的桥梁，该结构中最重要的域是shm\_file，存储了被映射文件的地址。每个共享内存区对象都对应特殊文件系统shm中的一个文件，当采取共享内存的方式将shm中的文件映射到进程地址空间后，可直接以访问内存的方式访问该文件。

### 3. 管道通信

管道是进程间共享的、用于相互间通信的文件，Linux/Unix提供了2种类型管道：

（1）用于父进程-子进程间通信的无名管道。无名管道是利用系统调用pipe(filedes)创建的无路径名的临时文件，该系统调用返回值是用于标识管道文件的文件描述符filedes，只有调用pipe(filedes)的进程及其子进程才能识别该文件描述符，并利用管道文件进行通信。通过无名管道进行通信要注意读写端需要通过lock锁对管道的访问进行互斥控制。

（2）用于任意两个进程间通信的命名管道named pipe。命名管道通过mknod(const char \*path, mode\_t mod, dev\_t dev)、mkfifo(const char \*path, mode\_t mode)创建，是一个具有路径名、在文件系统中长期存在的特殊类型的FIFO文件，读写遵循先进先出（first in first out），管道文件读是从文件开始处返回数据，对管道的写是将数据添加到管道末尾。命名管道的名字存在于文件系统中，内容存放在内存中。访问命名管道前，需要使用open()打开该文件。

管道是单向的、半双工的，数据只能向一个方向流动，双向通信时需要建立两个管道。一个命名管道的所有实例共享同一个路径名，但是每一个实例均拥有独立的缓存与句柄。只要可以访问正确的与之关联的路径，进程间就能够彼此相互通信。

要求：编程实现进程间的（无名）管道、命名管道通信，方式如下。

（1）管道通信：

父进程使用fork()系统调用创建子进程；

子进程作为写入端，首先将管道利用lockf()加锁，然后向管道中写入数据，并解锁管道；

父进程作为读出端，从管道中读出子进程写入的数据。

（2）命名管道通信：

读者进程使用mknod或mkfifo创建命名管道；

读者进程、写者进程使用open()打开创建的管道文件；

读者进程、写者进程分别使用read()、write()读写命名管道文件中的内容；

通信完毕，读者进程、写者进程使用close关闭命名管道文件。

### 4. 信号signal通信

使用信号相关的系统调用，参照后续样例程序展示的系统调用使用方法，设计实现基于信号signal的进程通信。例如：

父进程使用系统调用fork()函数创建两个子进程，再用系统调用signal()函数通知父进程捕捉键盘上传来的中断信号（即按Del 键），当父进程接收到这两个软中断的其中某一个后，父进程用系统调用kill()向两个子进程分别发送整数值为16 和17软中断信号，子进程获得对应软中断信号后，分别输出下列信息后终止。

## 3.4 进程/线程同步互斥

### 3.4.1 实验题目

Linux提供了内核信号量、用户态信号量两类进程/线程间同步互斥机制，用户态信号量又包括Pthread /POSIX信号量、System V信号量。

查阅资料，参照样例程序，采用内核信号量、Pthread信号量、System V信号量机制（三选一），为下面三个问题（三选一）设计正确的解决方案，并编程实现该方案。观察程序运行结果，分析并发进程/线程的运行行为和结果，验证方案正确性。

也可以采用高级程序设计语言提供的类似于教科书上管程monitor通过吧互斥机制，如C++中的互斥锁、条件变量，实现上述典型同步互斥问题。

#### 题目1. 生产流水线

An assembly line is to produce a product C with *n*1=4 part As, and *n*2=3 part Bs. The worker of machining A and worker of machining B produce *m*1=2 part As and *m*2=1 part B independently each time. Then the *m*1 part As or *m*2 part B will be moved to a station, which can hold at most *N*=12 of part As and part Bs altogether. The produced *m*1 part As must be put onto the station **simultaneously**. The workers must exclusively put a part on the station or get it from the station. In addition, the worker to make C must get **all part** of As and Bs, i.e. *n*1 part As and *n*2 part Bs, for one product C once.

Using semaphores to coordinate the three workers who are machining part A, part B and manufacturing the product C without deadlock.

It is required that

(1) definition and initial value of each semaphore, and

(2) the algorithm to coordinate the production process for the three workers

should be given.

实现原理：

本题目是生产者-消费者问题的扩展，需要注意题目要求：(a) worker A 一次生产*m1=2*个A，必须一次性放入工作台，(b) worker C必须一次性获得所需的*n1=4*个A和*n2=3*个B。因此，如果当前工作台空位小于*m1*，worker A被阻塞；如果当前工作台没有足够的A、B，worker C被阻塞。

采用类似于多次wait(empty)的操作，为worker A从工作台获取多个空位是不合理的。例如，假设当前工作台已经放置了*N=11*个零件，只有1个空位。worker A生产出2个A，worker，B生产出1个B。如果worker A先通过两次wait(empty)操作申请两个工作台空位，将被阻塞，生产的零件A没有放入工作台。随后worker B再执行wait(empty) 申请空位也将被阻塞，而此时工作台有1个空位。

因此，正确的解决方案是，

（1）利用用户空间计数变量countA、countB、Numempty，配合互斥信号量mutexA、mutexB、mutexNumempty，统计工作台中零件A、零件B、空单元的数目，当有足够多的零件A、B和工作台空位时，再一次性申请/获取多个所需的零件A、零件B、工作台空位，即将worker A所需的*m1*个工作台空位作为一个整体一次性地申请，worker C所需的工作台中的*n1*个A和*n2*个B作为一个整体一次性地申请；

（2）当所需工作台空位不满足时，worker A和worker B应主动阻塞自身(suspend/block)，防止忙等待。当worker C所需的工作台中A和B不满足时，也应主动阻塞自身；

（3）当worker A、worker B分别生产并放入新的零件A、B后，考虑唤醒由于所需零件不足而处于阻塞态的worker C；同样地，当worker C从工作台取出零件A、零件B后，考虑唤醒因没有足够空位而处于阻塞态的worker A、worker B。

#### 题目2. 进程多类资源申请

六个并发进程P1, P2, P3, P4, P5, P6，每个进程执行时需要同时使用*m*1=1 page 的内存资源和*m2=1*个I/O设备资源。系统内存资源总量为*N1=3* 个pages，I/O设备数目为*N2=3*。

对进程Pi，1≤ i ≤6 进程的业务流程为：（1）在进入段中，同时申请1个page 内存资源和1个I/O设备，（2）进入临界段，使用内存和I/O设备资源，（3）在退出段，释放内存和I/O设备资源。重复上述资源申请-使用-释放过程。

进程Pi在进入段中申请内存资源、I/O设备资源的先后顺序是不固定的，微观上申请步骤为：(i) 申请资源i∈{memory, I/O\_device}所要求的*m1*或*m2*个资源实例；(ii) 申请资源j∈{memory, I/O\_device}-{i}所要求的*m2*或*m1*个资源实例。

要求：设计合适的信号量机制，控制6个进程对内存资源、I/O设备资源的申请和使用，使得这6个进程能够获得各自所需全部资源，完成其业务流程。

说明：如果进程采用固定的资源申请顺序，如先申请内存资源，再申请I/O设备资源，则必

然不会发生死锁。因为这种资源申请机制相当于deadlock prevention中的资源有序申请法。

实现原理：

参照哲学家就餐问题的正确解决方案，设计基于信号量的进程同步互斥方案，应保证当一个进程所需的内存、I/O设备资源同时得到满足时，再将这两类资源分配给进程。引入描述当前系统内可用空闲内存资源数量的整型计数变量NumPages和可用空闲I/O设备数量的整型计数变量NumIOs，支持进程同时申请所需内存和I/O设备资源。

由于进程需要同时申请多类资源，应防止通过多次wait()操作依次申请多类资源，以避免死锁。例如，首先，将3个pages 内存资源依次分配给P1, P2, P3；然后，将3个I/O设备依次分配给P4, P5, P6。之后，P1, P2, P3中任一进程再申请I/O设备，或P4, P5, P6中任一进程再申请内存，将导致死锁。

#### 题目3. 医院门诊

校医院口腔科每天向患者提供*N=30*个挂号就诊名额。患者到达医院后，如果有号，则挂号，并在候诊室排队等待就医；如果号已满，则离开医院。

在诊疗室内，有*M=3*位医生为患者提供治疗服务。如果候诊室有患者等待并且诊疗室内有医生处于“休息”态，则从诊疗室挑选一位患者，安排一位医生为其治疗，医生转入“工作”状态；如果三位医生均处于“工作”态，候诊室内患者需等待。当无患者候诊时，医生转入“休息”状态。

要求：

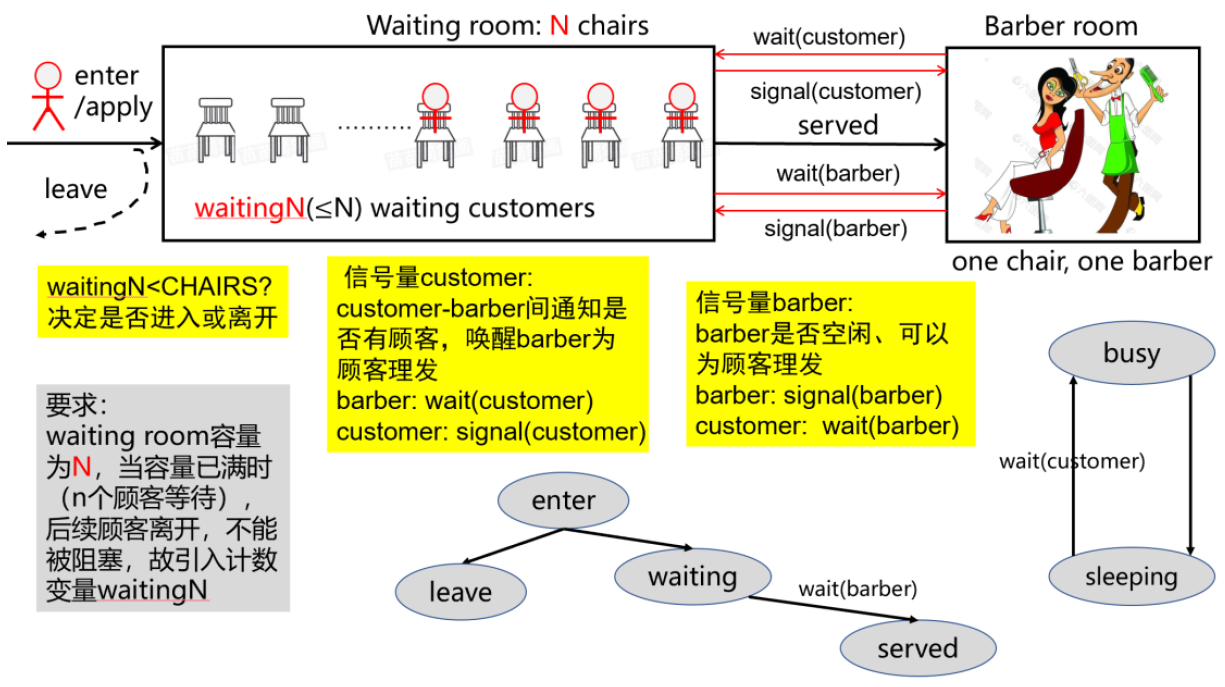
（1）采用信号量机制，描述患者、医生的行为；

（2）设置医生忙闲状态向量DState[M]，记录每位医生的“工作”、“休息”状态；

（3）设置患者就诊状态向量PState[N]，记录挂号成功后的患者的“候诊”、“就医”状态

实现原理：

本问题类似于睡眠理发师问题，既有多个进程互斥竞争使用资源，又有进程间同步。但与之不同之处在于：（1）睡眠理发师问题中只有一位理发师，本问题中有多位医生为患者服务；（2）引入两个作为共享数据结构的状态向量DState[M]、PState[N]。多个医生对DState[3]的访问必须互斥，多个患者对的访问必须互斥也进行互斥，为此，需要引入两个互斥信号量；（3）医院每天只放*N=30*个号，最先挂号的前*N*个患者可以获得号，并进入候诊室等待治疗。之后的患者由于拿不到当天号，没有必要等待，可以离开；而在‘睡眠理发师问题中，只要等候室有空位，顾客就可以进入等候室，等待理发服务。



### 3.4.2 进程/线程同步互斥的系统调用和API库函数

#### 3.4.2.1 内核信号量

内核信号量定义为struct semaphore类型的对象：

struct semaphore {undefined

atomic\_t count;

int sleepers;

wait\_queue\_head\_t wait;

}

count：信号量的值，大于0，信号量控制的资源空闲；等于0，资源忙，但没有进程等待这个资源；小于0，资源不可用，并至少有一个进程等待资源。

wait：存放等待队列链表的地址，当前等待资源的所有睡眠进程都会放在这个链表中。

sleepers：等待使用该资源的进程个数，也是当该资源空闲时需要被唤醒的等待进程个数。

对信号量的系统调用操作如下。

（1）内核信号量初始化：

void sema\_init (struct semaphore \*sem, int val);

void init\_MUTEX (struct semaphore \*sem); //将sem的值置为1，表示资源空闲

void init\_MUTEX\_LOCKED (struct semaphore \*sem); //将sem的值置为0，表示资源忙

（2）申请内核信号量所保护的资源，相当于wait()/P()操作：

void down(struct semaphore \* sem); // 可引起睡眠

int down\_interruptible(struct semaphore \* sem); // down\_interruptible能被信号打断

int down\_trylock(struct semaphore \* sem); // 非阻塞函数，不会睡眠。无法锁定资源则马上返回

（3）释放内核信号量所保护的资源，相当于signal()/V()操作：

void up(struct semaphore \* sem);

#### 3.4.2.2 Pthread/POSIX信号量

Pthread提供了两种线程同步互斥机制：互斥锁和信号量。

互斥锁相当于教科书中的binary semphore，取值为0、1，用于线程互斥。Pthread提供了以下操作互斥锁机的基本函数：

⚫ 互斥锁初始化：pthread\_mutex\_init()；

⚫ 互斥锁上锁：pthread\_mutex\_lock()；

⚫ 互斥锁判断上锁：pthread\_mutex\_trylock()；

⚫ 互斥锁解锁：pthread\_mutex\_unlock()；

⚫ 消除互斥锁：pthread\_mutex\_destroy()

信号量为教科书中所述的counting semaphore，即计数信号量，即可用于进程/线程间互斥、也可以应用于进程间同步。

进程/线程通过wait()、signal操作（或、PV操作），改变信号量的值，获取共享资源的访问权限，实现进程/线程同步互斥。Pthread线程库提供的信号量访问操作有：

⚫ sem\_init()，创建一个信号量，并初始化信号量的值；

⚫ sem\_wait()和sem\_trywait()，相当于wait/P操作，在信号量>0时，将信号量的值减1。两者的区别在于信号量<0时，sem\_wait将会阻塞进程/线程，而sem\_trywait则会立即返回；

⚫ sem\_post()，相当于signal/V操作，它将信号量的值加1，同时发出信号唤醒等待的进程/线程；

⚫ sem\_getvalue()，得到信号量的值；

⚫ sem\_destroy()，删除信号量。

#### 3.4.2.3 System V信号量

System V信号量是由一个或多个计数信号量组成的计数信号量集合(set of counting semaphores)。

System V 提供的信号量相关的函函数数原型为：

⚫ 创建一个新信号量或取得一个已有信号量

int semget(key\_t key, int num\_sems, int sem\_flags)

多个进程使用同一个信号量键值key来获得同一个信号量。

⚫ 信号量初始化和删除int semctl (int semid, int semnum, int cmd,...)

使用semctl()函数的SETVAL操作进行初始化，使用semctl()函数的IPC\_RMID操作删除信号量。

⚫ 对信号量组进行wait()/P()、signal()/V()操作，改变信号量的值

int semop(int semid, struct sembuf \* opsptr , size\_t numops)

## 3.5 多核多线程编程及性能分析

参照参考文献“利用多核多线程进行程序优化”，在Linux环境下，编写多线程程序，分析以下几个因素对程序运行时间的影响：

⚫ 程序并行化

⚫ 线程数目

⚫ 共享资源加锁

⚫ CPU亲和

⚫ cache优化

掌握多CPU、多核硬件环境下基本的多线程并行编程技术。

实验内容包括：

⚫ 实验5.1观察实验平台物理cpu、CPU核和逻辑cpu的数目

⚫ 实验5.2. 单线程/进程串行vs 两线程并行vs 三线程加锁并行程序对比

⚫ 实验5.3. 三线程加锁vs 三线程不加锁对比

⚫ 实验5.4. 针对Cache 的优化

⚫ 实验5.5. CPU亲和力对并行程序影响

⚫ 八种实现方案运行时间对比总结

# 4 第一组 进程创建与管理

在Linux环境下，采用C/C++/Java（或其它语言）编程，完成以下实验：

1. 参照Linux内核源码结构，阅读Linux内核源码，分析Linux进程的组成，观察Linxu进程的task\_struct等进程管理数据结构；
2. 参照相关示例程序，查阅Linux系统调用等相关资料，设计父进程和子进程的业务处理逻辑；利用C++等高级程序设计语言和fork()、clone、exec()、wait()、exit()、kill()、getpid()等系统调用，创建管理进程，观察父子进程/线程的结构和并发行为，掌握等待、撤销等进程控制方法。

要求：

* 至少用到fork()、clone、exec、wait、exit、kill、getpid等6个系统调用；
* 所创建的父子进程各自具有不同的业务处理逻辑，父进程创建子进程后，子进程通过exec()调入自身执行代码；
* 进程可以自己通过exit()主动结束，也可以被父进程执行kill()命令来结束；
* 观察对比通过fork()和clone()创建的子任务/进程/线程的差异，分析clone()系统调用中设置与不设置CLONE\_FS、CLONE\_VFORK、CLONE\_FILES、CLONE\_FS、CLONE\_PID等参数对所创建的子进程/线程的影响；
* 在创建的父子进程/线程代码中的不同位置处增加随机延迟，使得进程执行横跨多个时间片，如通过增加数十到数百毫秒的延迟，保证进程执行时间不少于3个时间片。

1. 掌握ps、top、pstree –h、vmstat、strace、ltrace、sleep x、kill、jobs等命令的功能和使用方式，利用这些命令观察进程的行为和特征。

## Linux进程和task\_struct结构分析

参照Linux内核源码结构，在/linux/include/linux/sched.h目录找到task\_struct结构源码。

### 任务状态

每个进程都有一个任务状态，涉及如下变量：

1. */\* -1 unrunnable, 0 runnable, >0 stopped: \*/*
2. volatile long   state;
3. */\* Per task flags (PF\_\*), defined further below: \*/*
4. unsigned int   flags;
5. int    exit\_state;

state状态的取值如下：

1. */\* Used in tsk->state: \*/*
2. #define TASK\_RUNNING   0x0000
3. #define TASK\_INTERRUPTIBLE  0x0001
4. #define TASK\_UNINTERRUPTIBLE  0x0002
5. #define \_\_TASK\_STOPPED   0x0004
6. #define \_\_TASK\_TRACED   0x0008
7. */\* Used in tsk->exit\_state: \*/*
8. #define EXIT\_DEAD   0x0010
9. #define EXIT\_ZOMBIE   0x0020
10. #define EXIT\_TRACE   (EXIT\_ZOMBIE | EXIT\_DEAD)
11. */\* Used in tsk->state again: \*/*
12. #define TASK\_PARKED   0x0040
13. #define TASK\_DEAD   0x0080
14. #define TASK\_WAKEKILL   0x0100
15. #define TASK\_WAKING   0x0200
16. #define TASK\_NOLOAD   0x0400
17. #define TASK\_NEW   0x0800
18. #define TASK\_STATE\_MAX   0x1000
19. */\* Convenience macros for the sake of set\_current\_state: \*/*
20. #define TASK\_KILLABLE   (TASK\_WAKEKILL | TASK\_UNINTERRUPTIBLE)
21. #define TASK\_STOPPED   (TASK\_WAKEKILL | \_\_TASK\_STOPPED)
22. #define TASK\_TRACED   (TASK\_WAKEKILL | \_\_TASK\_TRACED)
23. #define TASK\_IDLE   (TASK\_UNINTERRUPTIBLE | TASK\_NOLOAD)
24. */\* Convenience macros for the sake of wake\_up(): \*/*
25. #define TASK\_NORMAL   (TASK\_INTERRUPTIBLE | TASK\_UNINTERRUPTIBLE)
26. */\* get\_task\_state(): \*/*
27. #define TASK\_REPORT   (TASK\_RUNNING | TASK\_INTERRUPTIBLE | \
28. TASK\_UNINTERRUPTIBLE | \_\_TASK\_STOPPED | \
29. \_\_TASK\_TRACED | EXIT\_DEAD | EXIT\_ZOMBIE | \
30. TASK\_PARKED)

每个进程都处于以上状态中的一种：

1. **TASK\_RUNNING** 表示进程时刻准备着运行的状态。当处于这个状态的进程获得时间片的时候，就是运行中；如果没有获得时间片，那么说明被其他进程强占了，再等待再次分配时间片。
2. **TASK\_INTERRUPTIBLE，可中断的睡眠状态**。当前进程阻塞状态，虽然此时在睡眠状态，但可以被信号唤醒，唤醒后进行信号的处理，此时处理的函数可以由程序员来决定。
3. **TASK\_UNINTERRUPTIBLE，不可中断的睡眠状态**。此时在睡眠状态时，不可被信号唤醒，只能等待I/O操作完成。如果在I/O过程中发生意外无法完成，那么就会形成一个无法唤醒的进程。包括 kill 命令也无法唤醒，因为 kill 本身就是一种信号。
4. **TASK\_KILLABLE，可终止的睡眠状态**。从上面的注释可以发现为了唤醒而使用的宏，它设置的值是TASK\_UNINTERRUPTIBLE 和 TASK\_WAKEKILL ，即在 TASK\_UNINTERRUPTIBLE 的基础上 可以响应致命唤醒信号。
5. **\_\_TASK\_STOPPED ，被停止状态**。 在进程接收到 SIGTTIN、SIGSTOP、SIGTSTP、STGTTOU信号后会进入此状态。
6. **\_\_TASK\_TRACED** 进程被 debugger 等进程监视。
7. **EXIT\_ZOMBIE** 进程的执行被终止，但是其父进程还没有使用wait()等系统调用来获知它的终止信息。
8. **EXIT\_DEAD** 进程的最终状态。

### 任务ID

每个进程都有一个任务ID，涉及如下变量：

1. pid\_t    pid;
2. pid\_t    tgid;
3. struct task\_struct  \*group\_leader;

* pid 是 process id
* tgid是 thread group id
* group\_leader 指针用于快速访问

如果一个进程只有主线程，那么pid和tgid都是自己，group\_leader指向的也是自己。

如果进程创建了其他线程，线程有自己的pid，tgid就是主线程的pid，group\_leader指向的是进程的主线程。

查看/include/linux/threads.h发现pid 的取值范围：

1. #define PID\_MAX\_DEFAULT (CONFIG\_BASE\_SMALL ? 0x1000 : 0x8000)

在CONFIG\_BASE\_SMALL为0的情况下，pid的取值范围是0到32767。

### 任务存储结构

内核通过链表（list\_head结构体）任务串联起来的：

1. struct list\_head  tasks;

### 进程关系

在linux中，任何一个进程都拥有一个父进程，所有进程通过父子关系形成一颗进程树：

1. struct task\_struct \_\_rcu \*real\_parent;
2. struct task\_struct \_\_rcu \*parent;
3. struct list\_head  children;
4. struct list\_head  sibling;
5. struct task\_struct  \*group\_leader;

* real\_parent 指向其父进程，如果创建它的父进程不存在，则指向PID为1的init进程。
* parent 指向其父进程， 当它终止时必须向它的父进程发信号。值通常与 real\_parent 相同
* children 表示链表的头部，链表中的所有元素都是它的子进程。
* sibling 用于把当前进程插入到兄弟链表中。
* group\_leader 指向其所在进程组的 learder 进程。

### 进程权限

进程权限控制一个任务能访问哪些文件，能访问哪些其他的任务，以及可以被哪些任务所访问：

1. */\* Process credentials: \*/*
2. */\* Tracer's credentials at attach: \*/*
3. const struct cred \_\_rcu  \*ptracer\_cred;
4. */\* Objective and real subjective task credentials (COW): \*/*
5. const struct cred \_\_rcu  \*real\_cred;
6. */\* Effective (overridable) subjective task credentials (COW): \*/*
7. const struct cred \_\_rcu  \*cred;

* real\_cred是被操作的权限
* cred是这个进程操作其他的权限。
* \_\_rcu(Read-Copy update)是共享数据的同步机制

在cred.h存在cred 结构体：

1. kuid\_t  uid;  */\* real UID of the task \*/*
2. kgid\_t  gid;  */\* real GID of the task \*/*
3. kuid\_t  suid;  */\* saved UID of the task \*/*
4. kgid\_t  sgid;  */\* saved GID of the task \*/*
5. kuid\_t  euid;  */\* effective UID of the task \*/*
6. kgid\_t  egid;  */\* effective GID of the task \*/*
7. kuid\_t  fsuid;  */\* UID for VFS ops \*/*
8. kgid\_t  fsgid;  */\* GID for VFS ops \*/*
9. unsigned securebits; */\* SUID-less security management \*/*
10. kernel\_cap\_t cap\_inheritable; */\* caps our children can inherit \*/*
11. kernel\_cap\_t cap\_permitted; */\* caps we're permitted \*/*
12. kernel\_cap\_t cap\_effective; */\* caps we can actually use \*/*
13. kernel\_cap\_t cap\_bset; */\* capability bounding set \*/*
14. kernel\_cap\_t cap\_ambient; */\* Ambient capability set \*/*

* uid、gid 表示这个进程的启动者，谁启动的就是谁的id。
* euid、egid 当一个进程要访问资源时，比较的就是这个用户和组是否有权限。
* fsuid、fsgid 这个是对文件操作所需要的比较的权限。

一般来说，fsuid、euid和uid 是一样的， fsuid、egid和gid也是一样的。linux 通过SUID(Set User ID on execution) 来赋予权限的话，会带来安全隐患，那么为了解决这个问题，linux 引用了capabilities 能力机制。capabilities 机制将 root 用户的权限细分，可以分别的启动和禁用，在实际的操作当中，如果euid 不是root，便会检查该特权具有哪些capabilities。在linux 操作系统中可以查看/proc/{PID}/status来查看进程所属属性。capabilities定义了如下常见的权限：

1. #define CAP\_CHOWN            0
2. #define CAP\_KILL             5
3. #define CAP\_SETUID           7
4. #define CAP\_NET\_BIND\_SERVICE 10
5. #define CAP\_NET\_RAW          13
6. #define CAP\_SYS\_MODULE       16
7. #define CAP\_SYS\_RAWIO        17
8. #define CAP\_SYS\_PTRACE       19
9. #define CAP\_SYS\_BOOT         22
10. #define CAP\_SYS\_TIME         25
11. #define CAP\_AUDIT\_READ       37
12. #define CAP\_LAST\_CAP         CAP\_AUDIT\_READ

* cap\_permitted 表示进程能够使用的权限
* cap\_effective 实际起作用的权限
* cap\_inheritable 表示当可执行文件的设置了 inheritable 位时，调用exec执行该程序会继承调用者的inheritable 集合，并将其加入到permitted集合中。在非root用户下执行exec时 通常不会保存
* cap\_ambient 非root用户使用 exec 执行一个程序时，cap\_ambient会被加入到 cap\_permitted和cap\_effective中。
* cap\_bset 进程中所有进程允许保留的集合。

### 内核态函数栈

在task\_struct结构中，内核栈由stack变量管理：

1. void    \*stack;

内核栈位于pt\_regs和thread\_info之间：pt\_regs 、内核栈、thread\_info。内核栈是从该内存内存空间的自顶向下(从高位到地位)，而thread\_info 是从底向上的(从地位到高位)。内核栈的栈顶地址存储在esp中。所以当用户从用户态进入内核态后 esp会指向内核栈的底部。通过 thread\_info 可以获取task\_struct。在include/linux/thread\_info.h文件中，可以找到current\_thread\_info() 方法：

1. #define current\_thread\_info() ((struct thread\_info \*)current)

在arch/x86/include/asm/current.h：

1. static \_\_always\_inline struct task\_struct \*get\_current(void)
2. {
3. return this\_cpu\_read\_stable(current\_task);
4. }

在32位系统上内核栈的大小在arch/x86/include/asm/page\_32\_types.h：

1. #define THREAD\_SIZE\_ORDER 1
2. #define THREAD\_SIZE  (PAGE\_SIZE << THREAD\_SIZE\_ORDER)

在64位系统上内核栈的大小在 arch/x86/include/asm/page\_64\_types.h：

1. #ifdef CONFIG\_KASAN
2. #define KASAN\_STACK\_ORDER 1
3. #else
4. #define KASAN\_STACK\_ORDER 0
5. #endif
6. #define THREAD\_SIZE\_ORDER (2 + KASAN\_STACK\_ORDER)
7. #define THREAD\_SIZE  (PAGE\_SIZE << THREAD\_SIZE\_ORDER)

PAGE\_SIZE的值在page.h：

1. */\* PAGE\_SHIFT determines the page size \*/*
2. #define PAGE\_SHIFT 12
3. #define PAGE\_SIZE (\_AC(1, UL) << PAGE\_SHIFT)

\_AC和UL宏的定义在const.h：

1. #ifdef \_\_ASSEMBLY\_\_
2. #define \_AC(X,Y) X
3. #define \_AT(T,X) X
4. #else
5. #define \_\_AC(X,Y) (X##Y)
6. #define \_AC(X,Y) \_\_AC(X,Y)
7. #define \_AT(T,X) ((T)(X))
8. #endif
9. #define \_UL(x)  (\_AC(x, UL))
10. #define \_ULL(x)  (\_AC(x, ULL))
11. #define \_BITUL(x) (\_UL(1) << (x))
12. #define \_BITULL(x) (\_ULL(1) << (x))

从上面代码分析可以得出：对于32位系统 \_AC(1,UL) 是 1 ，左移12位后PAGE\_SIZE是4096。左移1位后THREAD\_SIZE为8192。对于64位系统，在非KASAN情况下，将PAGE\_SIZE左移2位后THREAD\_SIZE为16384。

我们知道了内核栈的大小之后，就会考虑当系统调用从用户态到内核态的时候，用户状态是如何保存的，此时需要用到一个pt\_regs的结构，将用户态运行过程中CPU的上下文信息保存在寄存器中，这样当内核态系统调用返回的时候，进程就会接着运行下去了。

内存栈的最高地址端，存放的是一个 pt\_regs 结构在/arch/x86/include/uapi/asm/ptrace.h：

1. *//32位*
2. struct pt\_regs {
3. long ebx;
4. long ecx;
5. long edx;
6. long esi;
7. long edi;
8. long ebp;
9. long eax;
10. int  xds;
11. int  xes;
12. int  xfs;
13. int  xgs;
14. long orig\_eax;
15. long eip;
16. int  xcs;
17. long eflags;
18. long esp;
19. int  xss;
20. };
21. *//64位*
22. struct pt\_regs {
23. unsigned long r15;
24. unsigned long r14;
25. unsigned long r13;
26. unsigned long r12;
27. unsigned long rbp;
28. unsigned long rbx;
29. unsigned long r11;
30. unsigned long r10;
31. unsigned long r9;
32. unsigned long r8;
33. unsigned long rax;
34. unsigned long rcx;
35. unsigned long rdx;
36. unsigned long rsi;
37. unsigned long rdi;
38. unsigned long orig\_rax;
39. unsigned long rip;
40. unsigned long cs;
41. unsigned long eflags;
42. unsigned long rsp;
43. unsigned long ss;
44. };

## 父子进程创建与控制

### 实验设计原理

参照相关示例程序，查阅Linux系统调用等相关资料，设计父进程和子进程的业务处理逻辑；利用C++等高级程序设计语言和fork()、clone、exec()、wait()、exit()、kill()、getpid()等系统调用，创建管理进程，观察父子进程/线程的结构和并发行为，掌握等待、撤销等进程控制方法。

至少使用到fork()、clone、exec、wait、exit、kill、getpid等6个系统调用。所创建的父子进程各自具有不同的业务处理逻辑，父进程创建子进程后，子进程通过exec()调入自身执行代码。进程可以自己通过exit()主动结束，也可以被父进程执行kill()命令来结束。观察对比通过fork()和clone()创建的子任务/进程/线程的差异，分析clone()系统调用中设置与不设置CLONE\_FS、CLONE\_VFORK、CLONE\_FILES、CLONE\_FS、CLONE\_PID等参数对所创建的子进程/线程的影响；在创建的父子进程/线程代码中的不同位置处增加随机延迟，使得进程执行横跨多个时间片，如通过增加数十到数百毫秒的延迟，保证进程执行时间不少于3个时间片。

涉及的系统调用用法如下：

* int fork()：用于创建一个新进程。

正确返回时，等于0表示创建子进程，从子进程返回的ID值；大于0表示从父进程返回的子进程的进程ID值。

错误返回时，等于-1表示创建失败。

* int clone(int (\*fn)(void \*), void \*child\_stack, int flags, void \*arg)：用于创建Linux线程。

其中，fn为函数指针，指向一个函数体，即所创建的子进程/线程的静态程序/代码；

child\_stack为给子进程分配系统堆栈的指针。 Linux系统中，系统堆栈空间大小为 2 pages 8K，并且该空间的低位地址中存放了进程控制块 task\_struct；

arg为传给子进程的参数，一般为0；

flags为被创建的子进程/线程需要从父进程复制/继承的资源的标志。

如果该调用执行成功，所创建的子进程的threadID将返回给调用者；如果该调用执行失败，-1将返回给调用者。

* int execl(char\* path,char\* arg0,…,char\* argn,(char\*)0)：用于将一个新的程序代码调入本进程所占的内存，并将其覆盖，产生新的内存进程映像.

其中，path为程序的路径，arg0到argn为参数。

新的程序可以是可执行文件或Shell批命令。

* (pid t)wait(int \*statloc)：用于控制父进程与子进程的同步。

其中，statloc为子进程的终止状态字。

正确返回时：大于0表示子进程的进程ID值；等于0表示其他。

错误返回时：等于-1表示调用失败。

父进程调用wait()函数，则父进程被阻塞，进入等待队列，等待子进程结束。当子进程结束时，会产生一个终止状态字，系族会向父进程发出SIGCHLD信号。当接到信号后，父进程提取子进程的终止状态字，从wait()函数返回继续执行原程序。

* exit(int status)：用于结束进程自身。

其中，status为进程结束状态。

* kill(int PID, int IID)：用于删除执行中的程序或者任务。

其中，PID为要被kill的进程号，ID为向将被kill的进程发送的中断号。

* signal(int sig, void(\*func)())：用于支持进程间基于信号signal（也称为软中断）的通信机制，允许调用进程控制软中断信号的处理。
* int getpid()：用于获取目前进程的进程标识pid

### 父子进程创建与控制

参照实验设计原理、实验要求和系统调用用法，编写实验源程序。

#### 程序代码

1. #include<stdio.h>
2. #include<stdlib.h>
3. #include<unistd.h>
4. #include<sched.h>
5. #include<sys/types.h>
6. #include<sys/wait.h>
7. #include<time.h>
8. #include<algorithm>
9. int main(){
10. srand(time(0));
11. int pid\_fork\_1, pid\_fork\_2, pid\_fork\_3, status;
12. if((pid\_fork\_1 = fork()) == 0){ *// fork 系统调用 （1）*
13. */\*子进程处理过程1：通过exec调入自身执行代码 \*/*
14. printf("Child process 1!\n");
15. int pid\_get = getpid(); *// getpid 系统调用*
16. printf("P1: pid\_fork\_1=%d pid\_get=%d\n", pid\_fork\_1, pid\_get);
17. *// 增加数十到数百毫秒的延迟，保证进程执行时间不少于3个时间片 (5)*
18. sleep(std::max(rand() % 1000, 10) / 1000.0);
19. *// 父进程创建子进程后，子进程通过exec()调入自身执行代码 （2）*
20. if(execl("/bin/echo", "echo", "P1: executed by execl", NULL) < 0) {
21. *// 异常处理*
22. printf("P1: Error on execl\n"); *// exec 系统调用*
23. exit(0); *// exit 系统调用*
24. }
25. }
26. else if (pid\_fork\_1 < 0){
27. *// 异常处理*
28. printf("P1: fork error\n");
29. exit(0);
30. }
31. if((pid\_fork\_2 = fork()) == 0){ *// fork 系统调用 （1）*
32. */\*子进程处理过程2：通过exit主动结束 \*/*
33. printf("Child process 2!\n");
34. int pid\_get = getpid(); *// getpid 系统调用*
35. printf("P2: pid\_fork\_2=%d pid\_get=%d\n", pid\_fork\_2, pid\_get);
36. *// 增加数十到数百毫秒的延迟，保证进程执行时间不少于3个时间片 (5)*
37. sleep(std::max(rand() % 1000, 10) / 1000.0);
38. *// 进程可以自己通过exit()主动结束 （3）*
39. printf("P2: Child process 2 exit(0)!\n");
40. exit(0); *// exit 系统调用*
41. }
42. else if (pid\_fork\_2 < 0){
43. printf("P2: fork error\n");
44. exit(0);
45. }
46. if((pid\_fork\_3 = fork()) == 0){ *// fork 系统调用 （1）*
47. */\*子进程处理过程3：通过父进程kill结束 \*/*
48. printf("Child process 3!\n");
49. int pid\_get = getpid(); *// getpid 系统调用*
50. printf("P3: pid\_fork\_3=%d pid\_get=%d\n", pid\_fork\_3, pid\_get);
51. *// 睡眠10秒*
52. sleep(10);
53. printf("P3: Child process 3 exit(0)!\n");
54. exit(0); *// exit 系统调用*
55. }
56. else if (pid\_fork\_3 < 0){
57. printf("P3: fork error\n");
58. exit(0);
59. }
60. */\*父进程处理过程 \*/*
61. printf("Parent process!\n");
62. int child\_pid = wait(&status); *// wait 系统调用*
63. printf("P: wait process %d with status %d\n", child\_pid, status);
64. *// 也可以被父进程执行kill()命令来结束*
65. kill(pid\_fork\_3,SIGABRT); *// kill 系统调用*
66. printf("P: kill process %d\n", pid\_fork\_3);
67. sleep(3);
68. printf("Parent process return!\n");
69. return 0;
70. }

#### 实验结果及分析

编译源程序并运行程序，得到运行结果。

1. Child process 1!
2. Parent process!
3. Child process 2!
4. P1: pid\_fork\_1=0 pid\_get=21579
5. Child process 3!
6. P2: pid\_fork\_2=0 pid\_get=21580
7. P3: pid\_fork\_3=0 pid\_get=21581
8. P2: Child process 2 exit(0)!
9. P: wait process 21580 with status 0
10. P: kill process 21581
11. P1: executed by execl
12. Parent process return!

分析运行结果发现：父进程和三个子进程并发运行，没有固定的运行顺序先后，子进程2主动结束之后父进程成功等待到结束信号并捕捉，父进程成功结束子进程3，子进程1成功调用exec，最后父进程成功结束。

### fork()和clone()的差异

#### 程序代码

1. #include<stdio.h>
2. #include<stdlib.h>
3. #include<unistd.h>
4. #include<sched.h>
5. #include<sys/types.h>
6. #include<sys/wait.h>
7. #include<time.h>
8. #include<algorithm>
9. int value = 0;
10. int child\_progress(void \*arg){
11. while(value < 10) {
12. printf("child\_progress 2,value = %d\n",value);
13. value++;
14. sleep(std::max(rand() % 1000, 10) / 1000.0);
15. }
16. return 0;
17. }
18. int main(){
19. srand(time(0));
20. int pid\_fork\_1, pid\_fork\_2, pid\_fork\_3, status;
21. if((pid\_fork\_1 = fork()) == 0){ *// fork 系统调用 （1）*
22. */\*子进程处理过程1：通过exec调入自身执行代码 \*/*
23. printf("Child process 1!\n");
24. int pid\_get = getpid(); *// getpid 系统调用*
25. printf("P1: pid\_fork\_1=%d pid\_get=%d\n", pid\_fork\_1, pid\_get);
26. *// 增加数十到数百毫秒的延迟，保证进程执行时间不少于3个时间片 (5)*
27. sleep(std::max(rand() % 1000, 10) / 1000.0);
28. *// 进程可以自己通过exit()主动结束 （3）*
29. printf("P1: Child process 1 exit(0)!\n");
30. exit(0); *// exit 系统调用*
31. }
32. else if (pid\_fork\_1 < 0){
33. *// 异常处理*
34. printf("P1: fork error\n");
35. exit(0);
36. }
37. char \*stack = (char\*)malloc(4096);
38. if (stack == NULL) {
39. printf("malloc fail\n");
40. }
41. *//子进程继承父进程的数据空间/在子进程结束后运行/将子进程的id存储到tid变量中*
42. int tid = 0, mask = CLONE\_VM|CLONE\_VFORK|CLONE\_CHILD\_SETTID;
43. if((pid\_fork\_2 = clone(child\_progress, stack + 4096, mask, NULL, NULL, NULL, &tid)) >= 0){ *// clone 系统调用 （1）*
44. */\*子进程处理过程2：通过exit主动结束 \*/*
45. printf("clone child process 2 pid %d, tid %d!\n", pid\_fork\_2, tid);
46. }
47. else if (pid\_fork\_2 < 0){
48. printf("P2: clone error\n");
49. exit(0);
50. }
51. */\*父进程处理过程 \*/*
52. printf("Parent process!\n");
53. while(value < 20){
54. printf("father, value = %d\r\n",value);
55. value ++;
56. sleep(std::max(rand() % 1000, 10) / 1000.0);
57. }
58. printf("Parent process return!\n");
59. return 0;
60. }

#### 实验结果及分析

1. Child process 1!
2. child\_progress 2,value = 0
3. P1: pid\_fork\_1=0 pid\_get=27128
4. child\_progress 2,value = 1
5. P1: Child process 1 exit(0)!
6. child\_progress 2,value = 2
7. child\_progress 2,value = 3
8. child\_progress 2,value = 4
9. child\_progress 2,value = 5
10. child\_progress 2,value = 6
11. child\_progress 2,value = 7
12. child\_progress 2,value = 8
13. child\_progress 2,value = 9
14. clone child process 2 pid 27129, tid 27129!
15. Parent process!
16. father, value = 10
17. father, value = 11
18. father, value = 12
19. father, value = 13
20. father, value = 14
21. father, value = 15
22. father, value = 16
23. father, value = 17
24. father, value = 18
25. father, value = 19
26. Parent process return!

观察对比通过fork()和clone()创建的子任务/进程/线程的差异：fork创造的子进程复制了父亲进程的资源（写时复制技术），子进程一般会比父进程优先运行，只有在子进程发生写的时候，子进程才会真正复制父进程的资源。fork调用一次，返回两次，父进程返回值为子进程pid，子进程返回值为0，然后父子进程各自继续执行程序。clone创建的子任务/进程/线程需要传入子任务的函数，分配给子任务的堆栈空间，调用函数输入参数（如果有），还有clone标志。

### clone()参数的影响

分析clone()系统调用中设置与不设置CLONE\_FS、CLONE\_VFORK、CLONE\_FILES、CLONE\_VM、CLONE\_PID等参数对所创建的子进程/线程的影响。

* CLONE\_FS：

子进程与父进程共享相同的文件系统，包括root、当前目录、umask。

如果设置该参数，调用者父进程对于chroot(2), chdir(2), umask(2)的调用，同样会影响到创建的子进程，反之亦然；

如果没有设置CLONE\_FS，子进程将工作在调用者父进程的文件系统信息的拷贝上，父进程对于chroot(2), chdir(2), umask(2)的调用不会影响子进程，反之亦然。

* CLONE\_VFORK：

如果设置该参数，调用者父进程被挂起，直至所创建的子进程通过execve(2) or \_exit(2) (as with vfork(2))释放虚拟内存资源后，父进程才重新激活。

如果没有设置该参数，父进程和子进程并发执行，都是可调度的。

* CLONE\_FILES：

如果设置该参数，子进程与父进程共享相同的文件描述符（file descriptor）表。

调用者父进程创建的文件描述符对于被创建的子进程同样有效，反之亦然。例如，父进程或子进程中的一个关闭了文件描述符，另一个进程也会受影响。

* CLONE\_VM：

如果设置该参数，子进程与父进程运行于相同的内存空间中。对调用者父进程或被创建子进程，其中一个进程的内存写操作对另一个进程是可见的，即会影响到另一个进程。而且，由父进程或子进程通过mmap or munmap执行的内存映射或解映射操作，同样会影响到另一个进程。

如果没有设置 CLONE\_VM，子进程在 clone()时在调用进程的内存空间的单独副本中运行 其中一个进程执行的内存写入不会影响另一个进程，就像 fork一样。

* CLONE\_PID：

如果设置该参数，子进程创建时得到的PID与父进程PID一致，可用于hacking the system。

## Linux命令功能和使用

掌握ps、top、pstree、vmstat、strace、ltrace、sleep x、kill、jobs等命令的功能和使用方式，利用这些命令观察进程的行为和特征。

### Ps

Linux中的ps命令是Process Status的缩写。ps命令用来列出系统中当前运行的那些进程。ps命令列出的是当前那些进程的快照，就是执行ps命令的那个时刻的那些进程，如果想要动态的显示进程信息，就可以使用top命令。

语法：ps [options] [--help]

参数：

* a 显示所有进程
* -a 显示同一终端下的所有程序
* -A 显示所有进程
* c 显示进程的真实名称
* -N 反向选择
* -e 等于“-A”
* e 显示环境变量
* f 显示程序间的关系
* -H 显示树状结构
* r 显示当前终端的进程
* T 显示当前终端的所有程序
* u 指定用户的所有进程
* -au 显示较详细的资讯
* -aux 显示所有包含其他使用者的行程
* -C<命令> 列出指定命令的状况
* –lines<行数> 每页显示的行数
* –width<字符数> 每页显示的字符数
* –help 显示帮助信息
* –version 显示版本显示

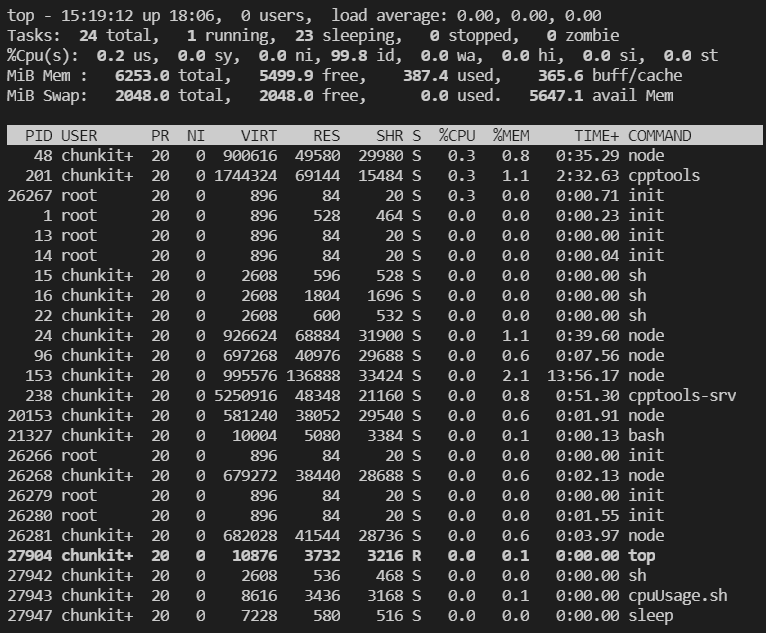


### Top

top命令是Linux下常用的性能分析工具，能够实时显示系统中各个进程的资源占用状况，类似于Windows的任务管理器。top是一个动态显示过程,即可以通过用户按键来不断刷新当前状态.如果在前台执行该命令,它将独占前台,直到用户终止该程序为止.比较准确的说,top命令提供了实时的对系统处理器的状态监视.它将显示系统中CPU最“敏感”的任务列表.该命令可以按CPU使用.内存使用和执行时间对任务进行排序；而且该命令的很多特性都可以通过交互式命令或者在个人定制文件中进行设定。语法：top [-] [d delay] [q] [c] [S] [s] [i] [n] [b]

参数：

* d : 改变显示的更新速度，或是在交谈式指令列( interactive command)按 s
* q : 没有任何延迟的显示速度，如果使用者是有 superuser 的权限，则 top 将会以最高的优先序执行
* c : 切换显示模式，共有两种模式，一是只显示执行档的名称，另一种是显示完整的路径与名称
* S : 累积模式，会将己完成或消失的子进程 ( dead child process ) 的 CPU time 累积起来
* s : 安全模式，将交谈式指令取消, 避免潜在的危机
* i : 不显示任何闲置 (idle) 或无用 (zombie) 的进程
* n : 更新的次数，完成后将会退出 top
* b : 批次档模式，搭配 "n" 参数一起使用，可以用来将 top 的结果输出到档案内



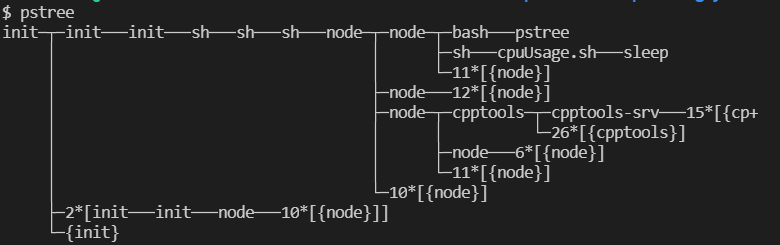
### Pstree

Linux pstree(英文全称：display a tree of processes）) 命令将所有进程以树状图显示，树状图将会以 pid (如果有指定) 或是以 init 这个基本进程为根 (root)，如果有指定使用者 id，则树状图会只显示该使用者所拥有的进程。

语法：pstree [-a] [-c] [-h|-Hpid] [-l] [-n] [-p] [-u] [-G|-U] [pid|user]

参数：

* -a 显示该进程的完整指令及参数, 如果是被记忆体置换出去的进程则会加上括号
* -c 如果有重覆的进程名, 则分开列出（预设值是会在前面加上 \*）



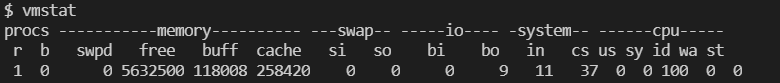
### Vmstat

vmstat是Virtual Meomory Statistics（虚拟内存统计）的缩写，可实时动态监视操作系统的虚拟内存、进程、CPU活动。

语法：vmstat [-V] [-n] [delay [count]]

参数：

* -V表示打印出版本信息；
* -n表示在周期性循环输出时，输出的头部信息仅显示一次；
* delay是两次输出之间的延迟时间；
* count是指按照这个时间间隔统计的次数。



### Strace

strace常用来跟踪进程执行时的系统调用和所接收的信号。 在Linux世界，进程不能直接访问硬件设备，当进程需要访问硬件设备(比如读取磁盘文件，接收网络数据等等)时，必须由用户态模式切换至内核态模式，通过系统调用访问硬件设备。strace可以跟踪到一个进程产生的系统调用,包括参数，返回值，执行消耗的时间。

示例：strace cat /dev/null

参数：

* -c 统计每一系统调用的所执行的时间,次数和出错的次数等.
* -d 输出strace关于标准错误的调试信息.
* -f 跟踪由fork调用所产生的子进程.
* -ff 如果提供-o filename,则所有进程的跟踪结果输出到相应的filename.pid中,pid是各进程的进程号.
* -F 尝试跟踪vfork调用.在-f时,vfork不被跟踪.
* -h 输出简要的帮助信息.
* -i 输出系统调用的入口指针.
* -q 禁止输出关于脱离的消息.
* -r 打印出相对时间关于,,每一个系统调用.
* -t 在输出中的每一行前加上时间信息.
* -tt 在输出中的每一行前加上时间信息,微秒级.
* -ttt 微秒级输出,以秒了表示时间.
* -T 显示每一调用所耗的时间.
* -v 输出所有的系统调用.一些调用关于环境变量,状态,输入输出等调用由于使用频繁,默认不输出.
* -V 输出strace的版本信息.
* -x 以十六进制形式输出非标准字符串
* -xx 所有字符串以十六进制形式输出.
* -a column
* 设置返回值的输出位置.默认 为40.
* -e expr
* 指定一个表达式,用来控制如何跟踪.格式如下:
* [qualifier=][!]value1[,value2]...
* qualifier只能是 trace,abbrev,verbose,raw,signal,read,write其中之一.value是用来限定的符号或数字.默认的 qualifier是 trace.感叹号是否定符号.例如:
* -eopen等价于 -e trace=open,表示只跟踪open调用.而-etrace!=open表示跟踪除了open以外的其他调用.有两个特殊的符号 all 和 none.
* 注意有些shell使用!来执行历史记录里的命令,所以要使用\\.
* -e trace=set
* 只跟踪指定的系统 调用.例如:-e trace=open,close,rean,write表示只跟踪这四个系统调用.默认的为set=all.
* -e trace=file
* 只跟踪有关文件操作的系统调用.
* -e trace=process
* 只跟踪有关进程控制的系统调用.
* -e trace=network
* 跟踪与网络有关的所有系统调用.
* -e strace=signal
* 跟踪所有与系统信号有关的 系统调用
* -e trace=ipc
* 跟踪所有与进程通讯有关的系统调用
* -e abbrev=set
* 设定 strace输出的系统调用的结果集.-v 等与 abbrev=none.默认为abbrev=all.
* -e raw=set
* 将指 定的系统调用的参数以十六进制显示.
* -e signal=set
* 指定跟踪的系统信号.默认为all.如 signal=!SIGIO(或者signal=!io),表示不跟踪SIGIO信号.
* -e read=set
* 输出从指定文件中读出 的数据.例如:
* -e read=3,5
* -e write=set
* 输出写入到指定文件中的数据.
* -o filename
* 将strace的输出写入文件filename
* -p pid
* 跟踪指定的进程pid.
* -s strsize
* 指定输出的字符串的最大长度.默认为32.文件名一直全部输出.
* -u username
* 以username 的UID和GID执行被跟踪的命令



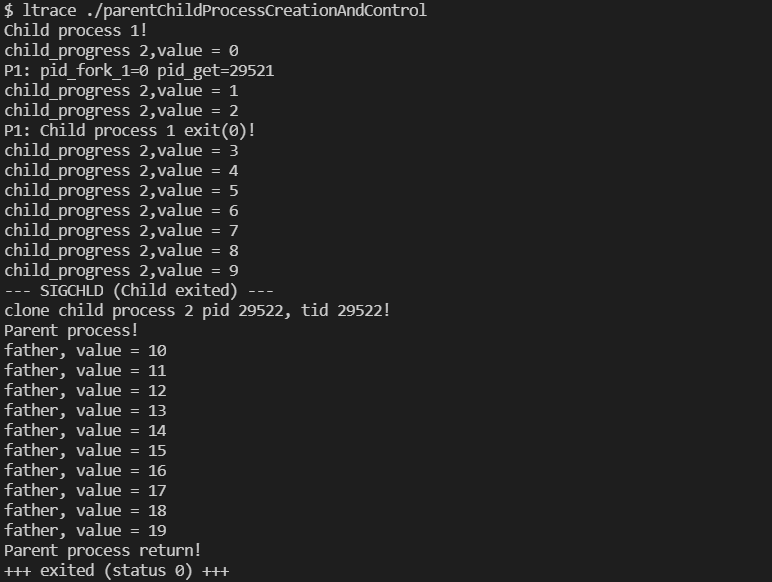
### ltrace

ltrace命令是用来跟踪进程调用库函数的情况。

语法：ltrace [option ...] [command [arg ...]]

参数：

* -a 对齐具体某个列的返回值。
* -c 计算时间和调用，并在程序退出时打印摘要。
* -C 解码低级别名称（内核级）为用户级名称。
* -d 打印调试信息。
* -e 改变跟踪的事件。
* -f 跟踪子进程。
* -h 打印帮助信息。
* -i 打印指令指针，当库调用时。
* -l 只打印某个库中的调用。
* -L 不打印库调用。
* -n, --indent=NR 对每个调用级别嵌套以NR个空格进行缩进输出。
* -o, --output=file 把输出定向到文件。
* -p PID 附着在值为PID的进程号上进行ltrace。
* -r 打印相对时间戳。
* -s STRLEN 设置打印的字符串最大长度。
* -S 显示系统调用。
* -t, -tt, -ttt 打印绝对时间戳。
* -T 输出每个调用过程的时间开销。
* -u USERNAME 使用某个用户id或组ID来运行命令。
* -V, --version 打印版本信息，然后退出。
* -x NAME treat the global NAME like a library subroutine.



### Sleep

Linux sleep命令可以用来将目前动作延迟一段时间。

语法：sleep [--help] [--version] number[smhd]

参数：

* --help : 显示辅助讯息
* --version : 显示版本编号
* number : 时间长度，后面可接 s、m、h 或 d
* 其中 s 为秒，m 为 分钟，h 为小时，d 为日数



### Kill

Linux kill 命令用于删除执行中的程序或工作。

kill 可将指定的信息送至程序。预设的信息为 SIGTERM(15)，可将指定程序终止。若仍无法终止该程序，可使用 SIGKILL(9) 信息尝试强制删除程序。程序或工作的编号可利用 ps 指令或 jobs 指令查看。

语法：kill [-s <信息名称或编号>][程序]　或　kill [-l <信息编号>]

参数：

* -l <信息编号> 　若不加<信息编号>选项，则 -l 参数会列出全部的信息名称。
* -s <信息名称或编号> 　指定要送出的信息。
* [程序] 　[程序]可以是程序的PID或是PGID，也可以是工作编号。



### Jobs

jobs 查看当前终端有多少在后台运行的进程。

语法：jobs [-l]

参数：

* -l：选项可显示所有任务的进程号 pid



# 第二组 线程创建管理与通信

## 线程创建管理

查阅Linux和Pthread线程库相关资料，参照相关示例程序，设计父进程/线程和子线程的业务处理逻辑；利用pthread\_create、pthread\_exit、pthread\_cancel、pthread\_join、pthread\_self

等线程管理函数，创建和管理线程，观察父子进程/线程的结构和并发行为，掌握等待、退出、撤销等线程控制方法。

要求：

* 在一个进程内创建多个子线程（如不少于3个子线程），父子子线程具有各自不同的业务处理逻辑，子线程执行自身特定的线程函数。父子线程间通过pthread\_join()函数实现同步和资源释放；
* 一个进程内的多个子线程分别以不同方式终止或退出执行，如通过pthread\_exit()和return自己主动终止，或被被其它线程通过pthread\_cancel被动终止，观察对比以不同方式退出执行的子线程的行为差异；
* 在创建的父子进程/线程代码中的不同位置处增加数十到数百毫秒的随机延迟（使用sleep()），使得进程和线程的执行横跨多个时间片，保证线程执行时间不少于3个时间片。

### 实验设计原理（Pthread 线程库背景知识）

线程也称为轻质进程、轻量级进程LWP。线程是多线程程序运行中的基本调度单位，是进程中一个单一顺序的控制流，一个进程内部可以有很多线程。同一进程中的多个线程共享分配给该进程的系统资源，比如文件描述符和信号处理等。

Linux系统中的多线程遵循POSIX线程接口标准，称为pthread，通过Pthread 线程库来实现线程管理。编写Linux下的多线程程序，需要使用头文件pthread.h，连接时需要使用库libpthread.a。

Linux下pthread的实现是通过系统调用clone()来实现的，clone()是 Linux所特有的系统调用，其使用方式类似fork。

线程与进程比较：

* 在Linux系统中，启动一个新的进程必须分配给它独立的地址空间，建立数据结构以维护其代码段、堆栈段和数据段，管理代价非常“昂贵”。而运行于一个进程中的多个线程，它们彼此之间使用相同的地址空间，共享大部分数据，启动一个线程所花费的空间远远小于启动一个进程所花费的空间，而且线程间彼此切换所需要时间也远远小于进程间切换所需要的时间。
* 线程间方便的通信机制。对不同进程来说它们具有独立的数据空间，要进行数据的传递只能通过通信的方式进行，费时且不方便。线程则不然，由于同一进程下的线程之间共享数据空间，所以一个线程的数据可以直接为其他线程所用，方便快捷。

Pthread线程库提供的基本线程管理函数有：

* 创建线程的pthread\_create
* 线程退出pthread\_exit，线程取消pthread\_cancel，线程脱离pthread\_detach
* 线程等待pthread\_join
* 线程标识获取pthread\_self

#### 线程创建pthread\_create()

调用格式：

#include<pthread.h>

int pthread\_create(pthread\_t \*thread，pthread\_attr\_t\*attr，void\*(\*start\_routine)(void \*), void \*arg)

参数：

thread：指向所创建线程的标识符的指针，线程创建成功时，返回被创建线程的ID

attr：用于指定被创建线程的属性，NULL表示使用默认属性

start\_routine: 函数指针，指向线程创建后要调用的函数，是一个以指向void的指针作 为参数和返回值的函数指针，这个被线程调用的函数也被称为线程函数

arg：指向传递给线程函数的参数

返回值：

创建成功：0

创建失败：返回错误码

#### 线程退出pthread\_exit()

调用格式：

#include<pthread.h>

void pthread\_exit(void \*retval)

参数：

retval：线程退出时的返回值，可被pthread\_join()等其它函数获取

说明：

当一个进程中包括多个线程时，使用进程退出函数exit()将终止进程内全部线程，而使用pthread\_exit()可以终止进程内特定线程。

如果被退出/终止的线程未脱离，则其线程ID和退出状态将一直保留到某个线程调pthread\_join为止。

#### 线程脱离pthread\_detach()

调用格式：

#include<pthread.h>

int pthread\_detach(pthread\_t thread)

参数：

thread：被脱离的线程的ID

说明：

函数pthread\_detach将指定的线程脱离，脱离的线程类似于守护进程。线程可以通过执行pthread detach(pthread self())将自己主动脱离。

#### 线程取消pthread\_cancel()

调用格式：

#include<pthread.h>

int pthread\_cancel(pthread\_t thread)

参数：

thread：要取消的线程的标识符ID

返回值：

取消成功：0

取消失败：返回错误码

说明：

在一个线程中调用pthread\_cancel()函数可以终止另一个线程的执行，但在被取消的线程的内部需要调用pthread\_setcancel()函数和pthread\_setcanceltype()函数设置自己的取消状态。

例如，被取消的线程接收到来自另一个线程的取消请求之后，可以忽略或接受这个请求；如果是接受取消请求，再判断立刻采取终止操作还是等待某个函数的调用等。

#### 线程清除pthread\_cleanup\_push()/pthread\_cleanup\_pop()

##### Pthread\_cleanup\_push()

调用格式：

#include<pthread.h>

void pthread\_cleanup\_push(void (\*rtn)(void \*), void \*arg)

参数：

rtn：清除函数

arg：清除函数的参数

说明：

将清除函数压入清除栈

##### pthread\_cleanup\_pop()

调用格式：

#include<pthread.h>

void pthread\_cleanup\_pop(int execute)

参数：

execute：执行到pthread\_cleanup\_pop()时是否在弹出清除函数的同时执行该函数。 非0：执行；0：不执行。

说明：

线程终止有两种情况：正常终止和非正常终止。线程主动调用 pthread\_exit()或者从线程 函数中 return 都将使线程正常退出，属于正常的、可预见的退出方式；

如果线程在其它线程的干预下，或者由于自身运行出错（例如访问非法地址）而退出， 对线程自身而言，这种退出方式是非正常、不可预见的。

对线程可预见的正常终止和不可预见异常终止，通过两个线程清除函数，可以保证线程 终止时能顺利地释放掉自己所占用的资源。

在线程函数体内先后调用pthread\_cleanup\_push()、pthread\_cleanup\_pop()后，在从 pthread\_cleanup\_push()的调用点到pthread\_cleanup\_pop()之间的程序段中的各种线程终止操 作（包括 pthread\_exit()和异常终止，但不包括 return）都将执行 pthread\_cleanup\_push()所指 定的清理函数。

#### 线程等待 pthread\_join()

调用格式：

#include<pthread.h>

int pthread\_join(pthread\_t thread，void \*\*thread\_return)

参数：

thread：等待退出的线程 ID

thread\_return：用于定义的指针，存储被等待线程结束时的返回值（不为 NULL 时）

返回值：

等待成功：0

等待失败：返回错误码。

说明：

一个进程内部的多个线程共享进程内的数据段。当进程内一个线程退出后，退出线程所 占用的资源并不会随着该线程的终止而得到释放。类似于进程之间使用wait()系统调用来同 步终止并释放资源，线程之间pthread\_join()函数实现线程同步和资源释放。

pthread\_join()将当前进程挂起来等待线程的结束。这个函数是一个线程阻塞的函数，调用它的函数将一直等待到被等待的线程结束为止，当函数返回时，被等待线程的资源被收回。

#### 线程标识获取 pthread\_self()

调用格式：

#include<pthread.h>

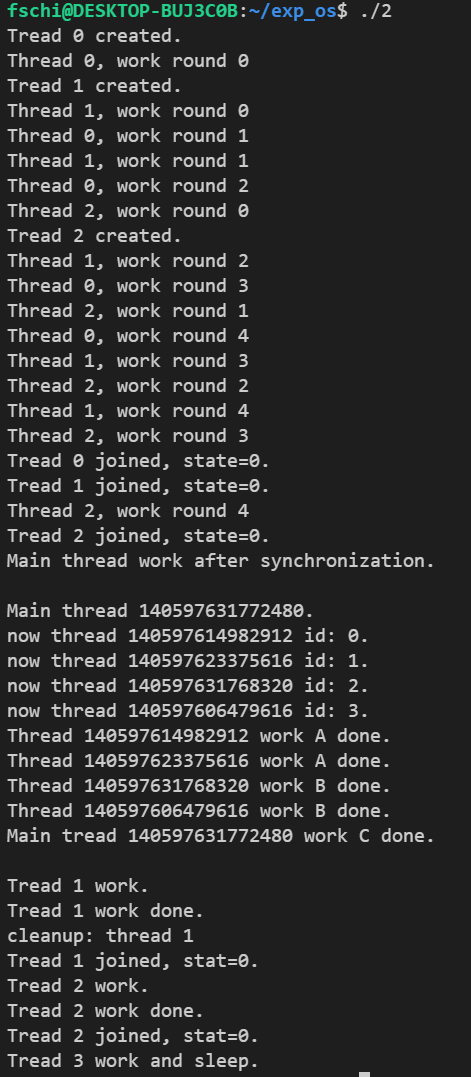
Pthread\_t pthread\_self(void)

返回值： 返回调用该函数的线程的标识 ID。

### 程序代码

1. #include <pthread.h>
2. #include <stdio.h>
3. #include <stdlib.h>
4. #include <time.h>
5. #include <unistd.h>
6. **const** **int** PTHEAR\_NUMBER = 4;
7. pthread\_t threads[4];
8. **int** res;
9. **void** \*ret;
10. **int** max(**int** x, **int** y) { **return** x > y ? x : y; }
11. **void** \*thread\_work(**void** \*arg) {
12. **int** tid = \*(**int** \*)arg;
13. **if** (tid >= PTHEAR\_NUMBER) pthread\_exit((**void** \*)1);
14. **for** (**int** x = 0; x < 5; x++) {
15. printf("Thread %d, work round %d\n", tid, x);
16. sleep(max(rand() % 1000, 10) / 1000.0);
17. }
18. pthread\_exit((**void** \*)0);
19. }
21. **void** \*threadA(**void** \*arg) {
22. **if** ((**int**)arg > PTHEAR\_NUMBER) pthread\_exit((**void** \*)1);
23. printf("now thread %lu id: %d.\n", pthread\_self(), (**int**)arg);
24. sleep(3.0);
25. printf("Thread %lu work A done.\n", pthread\_self());
26. pthread\_exit((**void** \*)0);
27. }
28. **void** \*threadB(**void** \*arg) {
29. printf("now thread %lu id: %d.\n", pthread\_self(), (**int**)arg);
30. res = pthread\_join(threads[(**int**)arg - 2], &ret);
31. **if** (ret) pthread\_exit((**void** \*)ret);  // 返回值不为0说明
32. sleep(3.0);
33. printf("Thread %lu work B done.\n", pthread\_self());
34. pthread\_exit((**void** \*)0);
35. }
37. // thread 1
38. **void** \*thread\_work\_exit(**void** \*arg) {
39. **int** tid = (**int**)arg;
40. **if** (tid >= PTHEAR\_NUMBER) pthread\_exit((**void** \*)1);
41. printf("Tread %d work.\n", tid);
42. pthread\_cleanup\_push((**void** \*)clean, tid);
43. sleep(3.0);
44. printf("Tread %d work done.\n", tid);
45. // pthread\_cleanup\_pop(0);
46. pthread\_exit((**void** \*)0);
47. pthread\_cleanup\_pop(0);
48. }
49. // thread 2
50. **void** \*thread\_work\_return(**void** \*arg) {
51. **int** tid = (**int**)arg;
52. **if** (tid >= PTHEAR\_NUMBER) **return** (**void** \*)1;
53. printf("Tread %d work.\n", tid);
54. pthread\_cleanup\_push((**void** \*)clean, tid);
55. sleep(3.0);
56. printf("Tread %d work done.\n", tid);
57. **return** (**void** \*)0;
58. pthread\_cleanup\_pop(1);
59. }
60. **void** \*clean(**void** \*arg) {
61. printf("cleanup: thread %d\n", (**int** \*)arg);
62. **return** (**void** \*)0;
63. }
64. // thread 3
65. **void** \*thread\_work\_sleep(**void** \*arg) {
66. **int** tid = (**int**)arg;
67. **if** (arg >= PTHEAR\_NUMBER) pthread\_exit((**void** \*)1);
68. printf("Tread %d work and sleep.\n", tid);
69. pthread\_cleanup\_push((**void** \*)clean, tid);
70. sleep(20.0);
71. pthread\_cleanup\_pop(1);
72. **return** (**void** \*)0;
73. }
74. **int** main() {
75. srand(time(NULL));
76. // 1: 父子进程同步
77. // 创建父子进程
78. **for** (**int** i = 0; i < 3; i++) {
79. // printf("%d %d ", &i, i);
80. res = pthread\_create(&threads[i], NULL, thread\_work, (**void** \*)&i);
81. **if** (!res)
82. printf("Tread %d created.\n", i);
83. **else**
84. printf("Tread %d create failed.\n", i);
85. sleep(max(rand() % 1000, 10) / 1000.0);
86. }
87. // 进程同步
88. **for** (**int** i = 0; i < 3; i++) {
89. res = pthread\_join(threads[i], &ret);
90. **if** (!res)
91. printf("Tread %d joined, state=%d.\n", i, ret);
92. **else**
93. printf("Tread %d join failed.\n", i);
94. }
95. printf("Main thread work after synchronization.\n\n");
96. sleep(3.0);
98. /\*\* 线程同步
99. \* A(0)---B(2)---\
100. \*                C(Main)
101. \* A(1)---B(3)---/
102. \*/
103. printf("Main thread %lu.\n", pthread\_self());
104. res = pthread\_create(&threads[0], NULL, threadA, (**void** \*)0);
105. sleep(max(rand() % 1000, 10) / 1000.0);
106. res = pthread\_create(&threads[1], NULL, threadA, (**void** \*)1);
107. sleep(max(rand() % 1000, 10) / 1000.0);
108. res = pthread\_create(&threads[2], NULL, threadB, (**void** \*)2);
109. sleep(max(rand() % 1000, 10) / 1000.0);
110. res = pthread\_create(&threads[3], NULL, threadB, (**void** \*)3);
111. res = pthread\_join(threads[2], &ret);
112. res = pthread\_join(threads[3], &ret);
113. sleep(3.0);
114. printf("Main tread %lu work C done.\n\n", pthread\_self());
116. // 2. 退出执行
117. res = pthread\_create(&threads[1], NULL, thread\_work\_exit, (**void** \*)1); // pthread\_exit()退出
118. **if** (res) printf("Tread %d create failed.\n", 1);
119. res = pthread\_join(threads[1], &ret);
120. **if** (res)
121. printf("Tread 1 join failed.\n");
122. **else**
123. printf("Tread 1 joined, stat=%d.\n", ret);
125. res = pthread\_create(&threads[2], NULL, thread\_work\_return, (**void** \*)2); // return 退出
126. **if** (res) printf("Tread 2 create failed.\n");
127. res = pthread\_join(threads[2], &ret);
128. **if** (res)
129. printf("Tread 2 join failed.\n");
130. **else**
131. printf("Tread 2 joined, stat=%d.\n", ret);
133. res = pthread\_create(&threads[3], NULL, thread\_work\_sleep, (**void** \*)3); // pthread\_cancel()退出
134. **if** (res) printf("Tread 3 create failed.\n");
135. sleep(3.0);
136. pthread\_cancel(threads[3]);
137. }

### 实验结果及分析



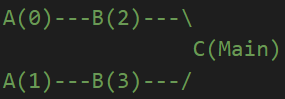
#### 线程同步

* 父子线程同步

1. 父线程创建0，1，2三个子进程，交付初始运行函数thread\_work()。
2. thread\_work()中，子进程进行5轮的work后，执行pthread\_exit()退出子进程。
3. 父线程依次join每个进程，执行父线程独有业务。

* 多级线程同步（类似同步作业题）

1. 共有3种工作线程A（线程0、1），B（线程2、3），C（主线程），工作逻辑如下图：



1. 0、1线程执行完成后，线程2、3通过pthread\_join分别收到0、1的退出信号，检查0、1的完成状态后开始工作。主线程收到2、3的退出信号后开始工作，完成线程间的同步。

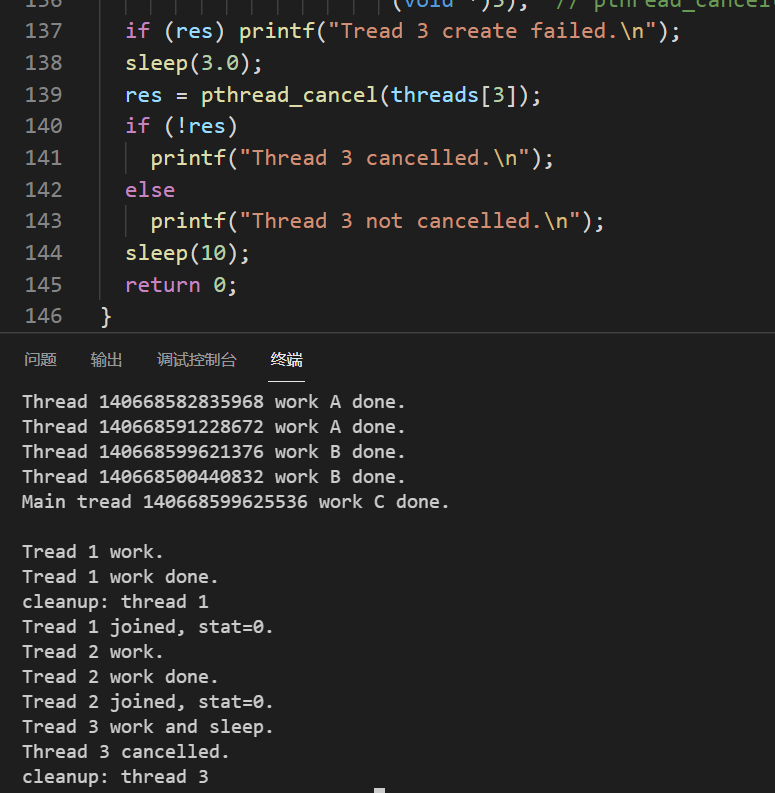
#### 退出方式分析

* pthread\_eixt()

1. pthread\_exit()函数终止了调用的线程，并使状态值value\_ptr可用于任何与终止的线程成功的join进行连接。
2. 任何已经被推送但尚未被弹出的取消清理处理程序都会按照它们被推送的相反顺序被弹出，然后被执行，本程序中只有一个cleanup handle。
3. 在所有取消清理处理程序被执行后，如果线程有任何线程特定的数据，适当的析构器函数将以未指定的顺序被调用。
4. 在一个线程终止后，对该线程的局部（自动）变量的访问结果是未定义的。因此，对退出线程的局部变量的引用不应该被用于pthread\_exit() value\_ptr参数值。
5. 线程终止不会释放任何应用程序可见的进程资源，包括但不限于互斥和文件描述符，也不会执行任何进程级的清理动作，包括但不限于调用可能存在的任何atexit()例程。
6. 在最后一个线程被终止后，进程退出时的退出状态为0。其行为就像实现在线程终止时调用exit()的参数为0一样。

* pthread\_cancel()

1. pthread\_cancel()函数请求消除某线程。该操作的生效时间取决于目标线程的可取消状态和类型。当取消被执行时，线程的取消清理处理程序被调用。当最后一个取消清理处理程序返回时，线程的特定数据析构函数被调用。当最后一个析构器函数返回时，线程被终止。但是在本程序种，没有观察到线程3的cleanup操作，即cleanup: thread 3.
2. 对于从pthread\_cancel()返回的调用线程来说，目标线程的取消处理是异步运行的。因此，对于上述问题而言，在pthread\_cancel()后主线程sleep()，则能成功进入线程3的cleanup种。结果如图：



* return

1. 当main()被首次调用的线程以外的线程从用于创建它的启动例程中返回时，会隐式调用pthread\_exit()。该函数的返回值是该线程的退出状态。
2. return后线程将不会执行任何在处理栈中的清理程序。

## 线程通信

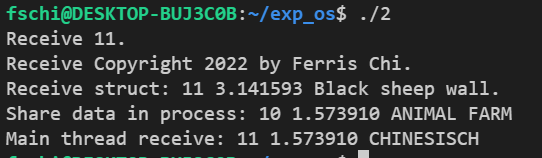
属于同一个进程中的多个线程使用相同的地址空间，共享大部分数据，一个线程的数据可以直接为其它线程所用，这些线程相互间可以方便快捷地利用共享数据结构进行通信。

编写程序，创建线程，实现主线程与子线程间、子线程相互间通过共享数据类型，如整型变量、字符串、结构体等传递信息，进行通信。

### 程序代码

1. #include <pthread.h>
2. #include <stdio.h>
3. #include <stdlib.h>
4. **struct** member {
5. **int** a;
6. **double** f;
7. **char** \*s;
8. };
9. **static** **struct** member st\_mem = {10, 1.57391, "ANIMAL FARM"};
11. // 1. 整数
12. **void** \*communi\_1(**void** \*arg) {
13. printf("Receive %d.\n", \*(**int** \*)arg);
14. pthread\_exit(0);
15. }
17. // 2. 字符串
18. **void** \*communi\_2(**void** \*arg) {
19. printf("Receive %s\n", (**char** \*)arg);
20. pthread\_exit(0);
21. }
23. // 3. 结构体
24. **void** \*communi\_3(**void** \*arg) {
25. **struct** member \*mem = (**struct** member \*)arg;
26. printf("Receive struct: %d %lf %s\n", mem->a, mem->f, mem->s);
27. pthread\_exit(0);
28. }
30. // 4. 参数回传
31. **void** \*communi\_4(**void** \*arg) {
32. printf("Share data in process: %d %lf %s\n", st\_mem.a, st\_mem.f, st\_mem.s);
33. **struct** member \*newmem = (**struct** member \*)malloc(**sizeof**(st\_mem));
34. \*newmem = st\_mem;
35. newmem->a = 11;
36. newmem->s = "CHINESISCH";
37. pthread\_exit((**void** \*)newmem);
38. }
40. **int** main() {
41. pthread\_t tid;
42. **int** res;
43. **void** \*ret;
44. **void** \*arg;
46. // 1. 整数
47. **int** a = 11;
48. arg = (**void** \*)&a;
49. res = pthread\_create(&tid, NULL, communi\_1, arg);
50. **if** (res) {
51. printf("Thread not created.\n");
52. exit(1);
53. }
54. pthread\_join(tid, &ret);
56. // 2. 字符串
57. **char** \*s = "Copyright 2022 by Ferris Chi.";
58. arg = (**void** \*)s;
59. res = pthread\_create(&tid, NULL, communi\_2, arg);
60. **if** (res) {
61. printf("Thread not created.\n");
62. exit(1);
63. }
64. pthread\_join(tid, &ret);
66. // 3. 结构体
67. **struct** member \*mem;
68. mem->a = 11;
69. mem->f = 3.1415926535;
70. mem->s = "Black sheep wall.";
71. arg = (**void** \*)mem;
72. res = pthread\_create(&tid, NULL, communi\_3, arg);
73. **if** (res) {
74. printf("Thread not created.\n");
75. exit(1);
76. }
77. pthread\_join(tid, &ret);
79. // 线程结束回传
80. **struct** member \*retmem;
81. res = pthread\_create(&tid, NULL, communi\_4, NULL);
82. **if** (res) {
83. printf("Thread not created.\n");
84. exit(1);
85. }
86. pthread\_join(tid, (**void** \*)&retmem);
87. printf("Main thread receive: %d %lf %s\n", retmem->a, retmem->f, retmem->s);
88. free(retmem);
89. **return** 0;
90. }

### 实验结果与分析



1. 可以看出，创建线程，实现主线程与子线程间、子线程相互间可以通过共享数据类型，如整型变量、字符串、结构体等传递信息，进行通信。
2. 第一次测试了整数的数据类型，子线程成功收到。
3. 第二次测试了字符串的数据类型，子线程成功收到。
4. 第三次测试了数据结构 member(int, double, char \*s) 的通信，子线程成功收到。
5. 第四次验证了线程对进程资源的共享，成功拿到了静态的st\_mem对象。另外还测试了子线程到父线程进行的参数传递机制。父线程成功收到经修改后的ret\_mem对象。注意在子线程种创建的new\_mem指针在线程结束后消亡，而其空间被ret\_mem指针引用，因此需要注意内存泄漏的问题，及时释放。

# 第三组 进程通信

查阅资料，学习掌握Linux系统提供的用于四种进程间通信的系统调用、库函数的使用方法和参数，参照样例程序，设计完成以下四组实验。

## 1. 基于消息队列的进程通信

### 实验设计原理

Linux消息队列包括POSIX消息队列、System V消息队列。具有一定权限的进程通过向消息队列中写入组织成消息的数据、从队列中读取数据，实现相互间通信。消息队列克服了信号signal承载信息量少，管道pipe只能承载无格式字节流以及缓冲区大小受限等缺点。

Linux提供的消息队列系统调用如下所示。

* 创建和访问一个消息队列，msgget

原型为：int msgget(key\_t, key, int msgflg);

与其他的IPC机制一样，程序必须提供一个键来命名某个特定的消息队列。msgflg是一个权限标志，表示消息队列的访问权限，它与文件的访问权限一样。msgflg可以与IPC\_CREAT做或操作，表示当key所命名的消息队列不存在时创建一个消息队列，如果key所命名的消息队列存在时，IPC\_CREAT标志会被忽略，而只返回一个标识符。

它返回一个以key命名的消息队列的标识符（非零整数），失败时返回-1.

* 将消息添加到消息队列，即消息发送，msgsnd

原型为：int msgsend(int msgid, const void \*msg\_ptr, size\_t msg\_sz, int msgflg);

msgid是由msgget函数返回的消息队列标识符。msg\_ptr是一个指向准备发送消息的指针，但是消息的数据结构却有一定的要求，指针msg\_ptr所指向的消息结构一定要是以一个长整型成员变量开始的结构体，接收函数将用这个成员来确定消息的类型。所以消息结构要定义成这样：

struct my\_message{ long int message\_type; /\* The data you wish to transfer\*/};

msg\_sz是msg\_ptr指向的消息的长度，注意是消息的长度，而不是整个结构体的长度，也就是说msg\_sz是不包括长整型消息类型成员变量的长度。

msgflg用于控制当前消息队列满或队列消息到达系统范围的限制时将要发生的事情。

如果调用成功，消息数据的一分副本将被放到消息队列中，并返回0，失败时返回-1。

* 从一个消息队列获取消息，即消息接收，msgrcv

原型为：int msgrcv(int msgid, void \*msg\_ptr, size\_t msg\_st, long int msgtype, int msgflg);

msgid, msg\_ptr, msg\_st的作用与在msgsnd中一样；

msgtype可以实现一种简单的接收优先级。如果msgtype为0，就获取队列中的

第一个消息。如果它的值大于零，将获取具有相同消息类型的第一个信息。如果它小于零，就获取类型等于或小于msgtype的绝对值的第一个消息；

msgflg用于控制当队列中没有相应类型的消息可以接收时将发生的事情。

调用成功时，该函数返回放到接收缓存区中的字节数，消息被复制到由msg\_ptr指向的用户分配的缓存区中，然后删除消息队列中的对应消息。失败时返回-1.

* 控制消息队列，msgctl

与共享内存的shmctl函数相似，原型为：int msgctl(int msgid, int command, struct msgid\_ds \*buf);

command是将要采取的动作，可以取3个值：（1）IPC\_STAT：把msgid\_ds结构中的数据设置为消息队列的当前关联值，即用消息队列的当前关联值覆盖msgid\_ds的值；（2）IPC\_SET：如果进程有足够的权限，就把消息列队的当前关联值设置为msgid\_ds结构中给出的值；（3）IPC\_RMID：删除消息队列

buf是指向msgid\_ds结构的指针，它指向消息队列模式和访问权限的结构。msgid\_ds结构至少包括以下成员：

struct msgid\_ds{uid\_t shm\_perm.uid; uid\_t shm\_perm.gid; mode\_t shm\_perm.mode;};

成功时返回0，失败时返回-1。

要求：编程实现发送者和接收者两个并发进程：

（1）发送者和接收者使用msgget(key\_t, key, int msgflg)、msgctl(int msgid, int command, struct msgid\_ds \*buf)创建、管理消息队列。只有接收者在接收完最后一个消息之后，才删除消息。

（2）发送者使用msgsend(int msgid, const void \*msg\_ptr, size\_t msg\_sz, int msgflg)向消息队列不断写入数据，并打印提示信息。

（3）接收者使用msgrcv(int msgid, void \*msg\_ptr, size\_t msg\_st, long int msgtype, int msgflg)从消息队列中接收消息，并打印提示信息。

### 程序代码

#### msgsend.c

1. #include <unistd.h>
2. #include <stdlib.h>
3. #include <stdio.h>
4. #include <string.h>
5. #include <sys/msg.h>
6. #include <errno.h>
7. #define MAX\_TEXT 512
8. struct msg\_st{
9. long int msg\_type;
10. char text[MAX\_TEXT];
11. };
12. int main(){
13. int running = 1;
14. struct msg\_st data;
15. char buffer[BUFSIZ];
16. int msgid = 1;
17. //建立消息队列
18. msgid = msgget((key\_t)1234, 0666 | IPC\_CREAT);
19. if(msgid == 1){
20. fprintf(stderr, "msgget failed with error: %d\n", errno);
21. exit(EXIT\_FAILURE);
22. }
23. //向消息队列中写消息，直到写入end
24. while(running){
25. //输入数据
26. printf("Enter some text: ");
27. fgets(buffer, BUFSIZ, stdin);
28. data.msg\_type = 1; //注意2
29. strcpy(data.text, buffer);
30. //向队列发送数据
31. if (msgsnd(msgid, (void\*)&data, MAX\_TEXT, 0) == -1){
32. fprintf(stderr, "msgsnd failed\n");
33. exit(EXIT\_FAILURE);
34. }
35. //输入end结束输入
36. if(strncmp(buffer, "end", 3) == 0)
37. running = 0;
38. sleep(1);
39. }
40. exit(EXIT\_SUCCESS);
41. }

#### msgreceive.c

1. #include <unistd.h>
2. #include <stdlib.h>
3. #include <stdio.h>
4. #include <string.h>
5. #include <errno.h>
6. #include <sys/msg.h>
7. struct msg\_st{
8. long int msg\_type;
9. char text[BUFSIZ];
10. };
11. int main(){
12. int running = 1;
13. int msgid = -1;
14. struct msg\_st data;
15. long int msgtype = 0; //注意1
16. //建立消息队列
17. msgid = msgget((key\_t)1234, 0666 | IPC\_CREAT);
18. if(msgid == -1){
19. fprintf(stderr, "msgget failed with error: %d\n", errno);
20. exit(EXIT\_FAILURE);
21. }
22. //从队列中获取消息，直到遇到end消息为止
23. while(running){
24. if(msgrcv(msgid, (void\*)&data, BUFSIZ, msgtype, 0) == -1){
25. fprintf(stderr, "msgrcv failed with errno: %d\n", errno);
26. exit(EXIT\_FAILURE);
27. }
28. printf("You wrote: %s\n",data.text);
29. //遇到end结束
30. if(strncmp(data.text, "end", 3) == 0){
31. running = 0;
32. }
33. }
34. //删除消息队列
35. if(msgctl(msgid, IPC\_RMID, 0) == -1){
36. fprintf(stderr, "msgctl(IPC\_RMID) failed\n");
37. exit(EXIT\_FAILURE);
38. }
39. exit(EXIT\_SUCCESS);
40. }

### 实验结果及分析

#### 实验结果





#### 分析

发送进程建立消息队列，依次向消息队列发送：Linux、Programming和end消息。当发送完end消息后结束进程。

接收进程建立消息队列，依次从消息队列接收：Linux、Programming和end消息。当接收到end消息后停止轮询消息队列，最后删除消息队列后结束进程。

## 共享内存通信

### 实验设计原理

共享内存是多个进程通过访问同一块内存空间，实现快速的进程间通信是最快的可用IPC形式。是针对其他通信机制运行效率较低而设计的。往往与其它通信机制，如信号量结合使用，来达到进程间的同步及互斥。Linux内核支持多种共享内存方式，如mmap()系统调用，POSIX共享内存，以及System V共享内存。本实验采用System V共享内存实现方法。

原理：进程间需要共享的数据存放于一个称为IPC共享内存的内存区域，需要访问该共享区域的进程需要将该共享内存区域映射到本进程的地址空间中。系统V共享内存通过shmget 获得或创建一个IPC共享内存区域，并返回相应的标识符。内核执行 shmget创建一个共享内存区，初始化该共享内存区对应的struct shmid\_kernel结构，同时在特殊文件系统shm中创建并打开一个同名文件，并在内存中建立起该文件对应的dentry和inode结构。新打开的文件不属于任何一个特定进程，任何进程都可以访问该共享内存区。

所创建的共享内存区有一个重要的控制结构 struct shmid\_kernel，该结构联系内存管理和文件系统的桥梁，该结构中最重要的域是shm\_file，存储了被映射文件的地址。每个共享内存区对象都对应特殊文件系统shm中的一个文件，当采取共享内存的方式将shm中的文件映射到进程地址空间后，可直接以访问内存的方式访问该文件。

共享内存通信采用的系统调用有：

* 创建共享内存

int shmget(key\_t key, int size, int shmflg)

key标识共享内存的键值：0/IPC\_PRIVATE。当key的取值为IPC\_PRIVATE，则函数shmget将创建一块新的共享内存；如果key的取值为0，而参数中又设置了IPC\_PRIVATE这个标志，则同样会创建一块新的共享内存。

返回值。如果成功，返回共享内存表示符，如果失败，返回-1。

* 映射共享内存

int shmat(int shmid, char\*shmaddr, int flag)

参数shmid：shmget函数返回的共享存储标识符，flag：决定以什么样的方式来确定映射的地址（通常为0）。

返回值。如果成功，则返回共享内存映射到进程地址空间中的地址；如果失败，则返回-1。

* 共享内存解除映射

int shmdt(char\*shmaddr)

当一个进程不再需要共享内存时，从进程的地址空间中删除共享内存。

* 控制释放

int shmctl(int shmid, int cmd, struct shmid\_ds\*buf);

int shmid是共享内存的ID。int cmd是控制命令，取值如下：IPC\_STAT得到共享内存的状态，IPC\_SET改变共享内存的状态，IPC\_RMID删除共享内存。

要求：编程实现写者进程Writer和读者进程Reader，

（1）写者进程和读者进程使用shmget(key\_t key, int size, int shmflg)创建在内存中创建用于两者间通信的共享内存，使用shmat(int shmid, char\*shmaddr, int flag)将共享内存映射到进程地址空间中，以便访问共享内存内容；

（2）写者进程向共享内存写入多组数据，读者进程从共享内存d读出数据；

（3）进程间通信结束后，写者进程和读者进程使用shmdt(char\*shmaddr)解除共享内存映射，使用shmctl(int shmid, int cmd, struct shmid\_ds\*buf)删除共享内存。

### 程序代码

#### write.c

1. #include <stdio.h>
2. #include <sys/ipc.h>
3. #include <sys/shm.h>
4. #include <sys/types.h>
5. #include <unistd.h>
6. #include <string.h>
7. typedef struct{
8. char name[8];
9. int age;
10. }people;
11. int main(int argc,char\*\* argv){
12. int shm\_id,i;
13. key\_t key;
14. char temp[8];
15. people\* p\_map;
16. char pathname[30];
17. strcpy(pathname,"/tmp");
18. key=ftok(pathname,0x03);//利用ftok()函数创建key。
19. if(key==-1){
20. perror("ftok error");
21. return -1;
22. }
23. printf("key=%d\n",key);
24. shm\_id=shmget(key,4096,IPC\_CREAT|IPC\_EXCL|0600); //shmid是共享内存标识符。Shmget()创建共享内存
25. if(shm\_id==-1){
26. perror("shmgct error");
27. return -1;
28. }
29. printf("shm\_id=%d\n",shm\_id);
30. p\_map=(people\*)shmat(shm\_id,NULL,0); //映射共享内存
31. memset(temp,0x00,sizeof(temp));
32. strcpy(temp,"test");
33. temp[4]='0';
34. for(i=0;i<3;i++){
35. temp[4]+=1;
36. strncpy((p\_map+i)->name,temp,5);
37. (p\_map+i)->age=0+i;
38. }
39. shmdt(p\_map);//用来断开与共享内存附加点的地址,禁本进程访问此片共享内存。
40. return 0;
41. }

#### read.c

1. #include <stdio.h>
2. #include <sys/ipc.h>
3. #include <sys/shm.h>
4. #include <sys/types.h>
5. #include <unistd.h>
6. #include <string.h>
7. typedef struct{
8. char name[8];
9. int age;
10. }people;
11. int main(int argc,char\*\*argv){
12. int shm\_id,i;
13. key\_t key;
14. people\* p\_map;
15. char pathname[30];
16. strcpy(pathname,"/tmp");
17. key=ftok(pathname,0x03);
18. if(key==-1){
19. perror("ftok error");
20. return -1;
21. }
22. printf("key=%d\n",key);
23. shm\_id=shmget(key,0,0);
24. if(shm\_id==-1){
25. perror("shmget error");
26. return -1;
27. }
28. printf("shm\_id=%d\n", shm\_id);
29. p\_map=(people\*)shmat(shm\_id,NULL,0);
30. for(i=0;i<3;i++){
31. printf("name:%s\n",(\*(p\_map+i)).name);
32. printf("age:%d\n",(\*(p\_map+i)).age);
33. }
34. if(shmdt(p\_map)==-1){
35. perror("detach error");
36. return -1;
37. }
38. return 0;
39. }

### 实验结果及分析

#### 实验结果





#### 分析

写进程使用路径创建key，然后使用key获取共享内存标识符，接着映射共享内存，向共享内存写入数据，最后断开与共享内存的连接。

读进程使用路径创建key，然后使用key获取共享内存标识符，接着映射共享内存，从共享内存取出数据，最后断开与共享内存的连接。

## 3. 管道通信

### 实验设计原理

管道（**Pipe**）及命名管道（**named pipe**）用于具有亲缘关系进程间的通信，命名管道克服了管道没有名字的限制，因此，除具有管道所具有的功能外，它还支持无亲缘关系进程间的通信。

管道是进程间共享的、用于相互间通信的文件，Linux/Unix提供了2种类型管道：

（1）用于父进程-子进程间通信的无名管道。无名管道是利用系统调用pipe(filedes)创建的无路径名的临时文件，该系统调用返回值是用于标识管道文件的文件描述符filedes，只有调用pipe(filedes)的进程及其子进程才能识别该文件描述符，并利用管道文件进行通信。通过无名管道进行通信要注意读写端需要通过lock锁对管道的访问进行互斥控制。

（2）用于任意两个进程间通信的命名管道named pipe。命名管道通过mknod(const char \*path, mode\_t mod, dev\_t dev)、mkfifo(const char \*path, mode\_t mode)创建，是一个具有路径名、在文件系统中长期存在的特殊类型的FIFO文件，读写遵循先进先出（first in first out），管道文件读是从文件开始处返回数据，对管道的写是将数据添加到管道末尾。命名管道的名字存在于文件系统中，内容存放在内存中。访问命名管道前，需要使用open()打开该文件。

管道是单向的、半双工的，数据只能向一个方向流动，双向通信时需要建立两个管道。一个命名管道的所有实例共享同一个路径名，但是每一个实例均拥有独立的缓存与句柄。只要可以访问正确的与之关联的路径，进程间就能够彼此相互通信。

管道相关系统调用：

* pipe()

建立一无名管道的格式：pipe(filedes)

参数定义如下：

int pipe(filedes);

int filedes[2];

其中，filedes[1]是写入端，filedes[0]是读出端。

* read()

系统调用格式：read(fd, buf, nbyte)

功能：从fd所指示的文件中读出nbyte个字节的数据，并将它们送至由指针buf所指示的缓冲区中。如该文件被加锁、等待，直到锁打开为止。

参数定义如下：

int read(fd,buf,nbyte);

int fd：

char \*buf;

unsigned nbyte;

* write()

系统调用格式：read((fd,buf,nbyte)

功能：把nbyte个字节的数据，从buf所指向的缓冲区写到由fd所指向的文件中。如文件加锁、暂停写入，直至开锁。

参数定义同read()。进程间通过管道用write和read来传递数据，但write和read不能同时进行，在管道中只能有4096字节数据被缓冲。

* Sleep()

系统调用格式：sleep(second);

功能：使现行进程暂停执行由自变量规定的秒数，用于进程的同步与互斥。

* lockf()

系统调用格式：lockf(fd,mode,size);

功能：对指定文件的指定区域(由size指示)进行加锁或解锁，以实现进程的同步与互斥。其中fd是文件描述字，mode是锁定方式，=1表示加锁，=0表示解锁，size是指定文件fd的指定区域，用0表示从当前位置到文件尾。

* int mknod(const char \*path,mode\_t mod,dev\_t dev);

int mkfifo(const char \*path,mode\_t mode);

参数：path为创建的命名管道的全路径名

mod为创建的命名管道的模式，指明其存取权限；

dev为设备值，该值取决于⽂件创建的种类，它只在创建设备⽂件时才会⽤到

返回值：成功返回0，失败返回－1

要求：编程实现进程间的（无名）管道、命名管道通信，方式如下。

（1）管道通信：

父进程使用fork()系统调用创建子进程；

子进程作为写入端，首先将管道利用lockf()加锁，然后向管道中写入数据，并解锁管道；

父进程作为读出端，从管道中读出子进程写入的数据。

（2）命名管道通信：

读者进程使用mknod或mkfifo创建命名管道；

读者进程、写者进程使用open()打开创建的管道文件；

读者进程、写者进程分别使用read()、write()读写命名管道文件中的内容；

通信完毕，读者进程、写者进程使用close关闭命名管道文件。

### 程序代码

#### pipeline.c

1. #include <unistd.h>
2. #include <signal.h>
3. #include <stdio.h>
4. #include <stdlib.h>
5. #include <sys/types.h>
6. #include <sys/wait.h>
7. int pid; /\*定义进程变量\*/
8. int main(){
9. int fd[2];
10. char outpipe[100],inpipe[100]; /\*定义两个字符数组\*/
11. pipe(fd); /\*创建一个管道\*/
12. while((pid=fork())==-1); /\*如果进程创建不成功，则空循环\*/
13. if(pid==0){ /\*如果子进程创建成功，pid为进程号\*/
14. lockf(fd[1],1,0); /\*锁定管道\*/
15. sprintf(outpipe,"child process is sending message!");/\*把串放入数组outpipe中\*/
16. write(fd[1],outpipe,50); /\*向管道写长为50字节的串\*/
17. sleep(5); /\*自我阻塞5秒\*/
18. lockf(fd[1],0,0); /\*解除管道的锁定\*/
19. exit(0);
20. }
21. else{
22. wait(0); /\*等待子进程结束\*/
23. read(fd[0],inpipe,50); /\*从管道中读长为50字节的串\*/
24. printf("%s\n",inpipe); /\*显示读出的数据\*/
25. exit(0); /\*父进程结束\*/
26. }
27. }

#### fifo\_reader.c

1. #include<stdlib.h>
2. #include<stdio.h>
3. #include<sys/types.h>
4. #include<sys/stat.h>
5. #include<fcntl.h>
6. #include<errno.h>
7. #include<unistd.h>
8. #define PATH "./fifo"
9. #define SIZE 128
10. int main() {
11. if (mkfifo(PATH,0777) == -1) {
12. perror("mkefifo error");
13. exit(0);
14. }
15. int fd = open(PATH,O\_RDONLY);
16. if (fd<0) {
17. printf("open fd is error\n");
18. return 0;
19. }
20. char Buf[SIZE];
21. while(1){
22. ssize\_t s = read(fd,Buf,sizeof(Buf));
23. if (s<0) {
24. perror("read error");
25. exit(1);
26. }
27. else if (s == 0) {
28. printf("client quit! i shoud quit!\n");
29. break;
30. } else {
31. Buf[s] = '\0';
32. printf("client# %s ",Buf);
33. fflush(stdout);
34. }
35. }
36. close(fd);
37. return 3;
38. }

#### fifo\_writer.c

1. #include<stdlib.h>
2. #include<stdio.h>
3. #include<unistd.h>
4. #include<sys/types.h>
5. #include<sys/stat.h>
6. #include<string.h>
7. #include<errno.h>
8. #include<fcntl.h>
9. #include<unistd.h>
10. #define PATH "./fifo"
11. #define SIZE 128
12. int main() {
13. int fd = open(PATH,O\_WRONLY);
14. if (fd < 0) {
15. perror("open error");
16. exit(0);
17. }
18. char Buf[SIZE];
19. while(1) {
20. printf("please Enter#:");
21. fflush(stdout);
22. ssize\_t s = read(0,Buf,sizeof(Buf));
23. if (s<0) {
24. perror("read is failed");
25. exit(1);
26. }
27. else if(s==0) {
28. printf("read is closed!");
29. return 1;
30. }
31. else{
32. Buf[s]= '\0';
33. write(fd,Buf,strlen(Buf));
34. }
35. }
36. return 0;
37. }

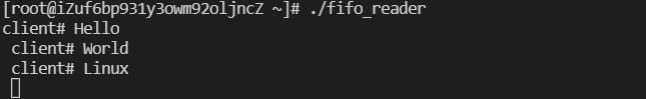
### 实验结果及分析

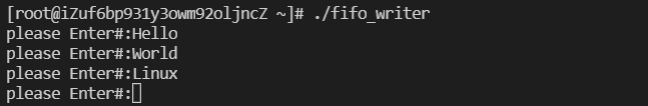
#### 实验结果

管道：



命名管道：





#### 分析

管道：父进程定义两个字符缓冲区并创建一个管道，然后创建子进程。子进程锁定管道并向管道写入字符串，阻塞一段时间后解除管道锁定并结束进程。父进程等待子进程结束后从管道读出字符串并结束进程。

命名管道：读进程创建命名管道并用读方式打开，然后不断从管道中读取数据；写进程用写方式打开管道，然后不断从键盘读取数据写入到管道中。

## 4. 信号signal通信

### 实验设计原理

信号（Signal）：也称为软中断，是一种基于事件的通信机制，用于通知接受进程有某种事件发生。除了用于进程间通信外，进程还可以发送信号给进程本身；linux除了支持Unix早期信号语义函数sigal外，还支持语义符合Posix.1标准的信号函数sigaction。信号只是用来通知某进程发生了什么事件，并不给该进程传递任何数据。

收到信号的进程对各种信号有不同的处理方法。处理方法可以分为三类：第一种是类似中断的处理程序，对于需要处理的信号，进程可以指定处理函数，由该函数来处理。第二种方法是，忽略某个信号，对该信号不做任何处理，就象未发生过一样。第三种方法是，对该信号的处理保留系统的默认值，这种缺省操作，对大部分的信号的缺省操作是使得进程终止。进程通过系统调用signal来指定进程对某个信号的处理行为。

在进程表的PCB表项中有一个软中断信号域，该域中每一位对应一个信号，当有信号发送给进程时，对应位置位。由此可以看出，进程对不同的信号可以同时保留，但对于同一个信号，进程并不知道在处理之前来过多少个。

有关信号的系统调用：系统调用signal是进程用来设定某个信号的处理方法，系统调用kill是用来发送信号给指定进程的。这两个调用可以形成信号的基本操作。后两个调用pause和alarm是通过信号实现的进程暂停和定时器，调用alarm是通过信号通知进程定时器到时。

* signal 系统调用

用来设定某个信号的处理方法。该调用声明的格式如下：

void (\*signal(int signum, void (\*handler)(int)))(int);

在使用该调用的进程中加入以下头文件：

#include <signal.h>

上述声明格式比较复杂，如果不清楚如何使用，也可以通过下面这种类型定义的格式来使用（POSIX的定义）：

typedef void (\*sighandler\_t)(int);

sighandler\_t signal(int signum, sighandler\_t handler);

但这种格式在不同的系统中有不同的类型定义，所以要使用这种格式，最好还是参考一下联机手册。

其中，参数signum指出要设置处理方法的信号。参数handler是一个处理函数，或者是

SIG\_IGN：忽略参数signum所指的信号。

SIG\_DFL：恢复参数signum所指信号的处理方法为默认值。

传递给信号处理例程的整数参数是信号值，这样可以使得一个信号处理例程处理多个信号。系统调用signal返回值是指定信号signum前一次的处理例程或者错误时返回错误代码SIG\_ERR。

* kill 系统调用

系统调用kill用来向进程发送一个信号。该调用声明的格式如下：

int kill(pid\_t pid, int sig);

在使用该调用的进程中加入以下头文件：

#include <sys/types.h>

#include <signal.h>

该系统调用可以用来向任何进程或进程组发送任何信号。如果参数pid是正数，那么该调用将信号sig发送到进程号为pid的进程。如果pid等于0，那么信号sig将发送给当前进程所属进程组里的所有进程。如果参数pid等于-1，信号sig将发送给除了进程1和自身以外的所有进程。如果参数pid小于-1，信号sig将发送给属于进程组-pid的所有进程。如果参数sig为0，将不发送信号。该调用执行成功时，返回值为0；错误时，返回-1，并设置相应的错误代码errno。下面是一些可能返回的错误代码：

EINVAL：指定的信号sig无效。

ESRCH：参数pid指定的进程或进程组不存在。注意，在进程表项中存在的进程，可能是一个还没有被wait收回，但已经终止执行的僵死进程。

EPERM：进程没有权力将这个信号发送到指定接收信号的进程。因为，一个进程被允许将信号发送到进程pid时，必须拥有root权力，或者是发出调用的进程的UID 或EUID与指定接收的进程的UID或保存用户ID（savedset-user-ID）相同。如果参数pid小于-1，即该信号发送给一个组，则该错误表示组中有成员进程不能接收该信号。

* pause系统调用

系统调用pause的作用是等待一个信号。该调用的声明格式如下：

int pause(void);

在使用该调用的进程中加入以下头文件：

#include <unistd.h>

该调用使得发出调用的进程进入睡眠，直到接收到一个信号为止。该调用总是返回-1，并设置错误代码为EINTR（接收到一个信号）。

* alarm和setitimer系统调用

系统调用alarm的功能是设置一个定时器，当定时器计时到达时，将发出一个信号给进程。该调用的声明格式如下：

unsigned int alarm(unsigned int seconds);

在使用该调用的进程中加入以下头文件：

#include <unistd.h>

系统调用alarm安排内核为调用进程在指定的seconds秒后发出一个SIGALRM的信号。如果指定的参数seconds为0，则不再发送SIGALRM信号。后一次设定将取消前一次的设定。该调用返回值为上次定时调用到发送之间剩余的时间，或者因为没有前一次定时调用而返回0。

注意，在使用时，alarm只设定为发送一次信号，如果要多次发送，就要多次使用alarm调用。

现在的系统中很多程序不再使用alarm调用，而是使用setitimer调用来设置定时器，用getitimer来得到定时器的状态，这两个调用的声明格式如下：

int getitimer(int which, struct itimerval \*value);

int setitimer(int which, const struct itimerval \*value, struct itimerval \*ovalue);

在使用这两个调用的进程中加入以下头文件：

#include <sys/time.h>

该系统调用给进程提供了三个定时器，它们各自有其独有的计时域，当其中任何一个到达，就发送一个相应的信号给进程，并使得计时器重新开始。三个计时器由参数which指定，如下所示：

TIMER\_REAL：按实际时间计时，计时到达将给进程发送SIGALRM信号。

ITIMER\_VIRTUAL：仅当进程执行时才进行计时。计时到达将发送SIGVTALRM信号给进程。

ITIMER\_PROF：当进程执行时和系统为该进程执行动作时都计时。与ITIMER\_VIR-TUAL是一对，该定时器经常用来统计进程在用户态和内核态花费的时间。计时到达将发送SIGPROF信号给进程。

* wait()函数

父进程处于阻塞状态,等待子进程终止,其返回值为所等待子进程的进程号

* exit()函数

进程自我终止,释放所占资源,通知父进程可以删除自己,此时它的状态变为P\_state= SZOMB,即僵死状态.如果调用进程在执行exit时其父进程正在等待它的中止，则父进程可立即得到该子进程的ID号。

使用系统调用fork()函数创建两个子进程，再用系统调用signal()函数让父进程捕捉键盘上来的中断信号(即按Del 键)，当父进程接收到这两个软中断的其中某一个后，父进程用系统调用kill()向两个子进程分别发送整数值为16 和17软中断信号，子进程获得对应软中断信号后，分别输出下列信息后终止。

Child process 1 is killed by parent!!

Child process 2 is killed by parent!!

父进程调用wait() 函数等待两个子进程终止后，输出以下信息后终止。

Parent process is killed! !

多运行几次编写的程序，简略分析出现不同结果的原因。

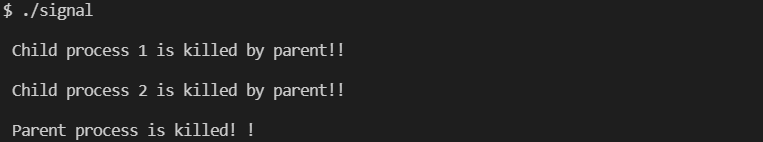
### 程序代码

#### signal.c

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <signal.h>
4. #include<sys/wait.h>
5. #include <unistd.h>
6. #include <sys/types.h>
7. int wait\_flag;
8. void stop() {
9. printf("\ninterruption\n");
10. }
11. int main() {
12. int pid1, pid2;
13. signal (3, stop);// 或者signal(14, stop);
14. while( (pid1 = fork())==-1);
15. if(pid1 > 0) {
16. while( (pid2 = fork())==-1);
17. if(pid2 > 0) {
18. wait\_flag = 1;
19. sleep(5);
20. kill(pid1, 16);
21. kill(pid2, 17);
22. wait(0);
23. wait(0);
24. printf("\n Parent process is killed! !\n");
25. exit(0);
26. }
27. else {
28. wait\_flag = 1;
29. signal (17, stop);
30. printf("\n Child process 2 is killed by parent!!\n");
31. exit(0);
32. }
33. }
34. else {
35. wait\_flag = 1;
36. signal (16, stop);
37. printf("\n Child process 1 is killed by parent!!\n");
38. exit(0);
39. }
40. }

### 实验结果及分析

#### 实验结果



#### 分析

父进程创建了两个子进程，使用kill函数分别向两个子进程发送信号，两个子进程收到信号后调用相应的函数处理，然后结束子进程，最后父进程等待子进程结束后结束。

**第四组 进程/线程间同步与互斥**

**实验题目**

Linux 提供了内核信号量、用户态信号量两类进程/线程间同步互斥机制，用户态信号量 又包括 Pthread /POSIX 信号量、System V 信号量。

查阅资料，参照样例程序，采用内核信号量、Pthread 信号量、System V 信号量机制（三选一），为下面三个问题（三选一）设计正确的解决方案，并编程实现该方案。观察程序运行结果，分析并发进程/线程的运行行为和结果，验证方案正确性。 也可以采用高级程序设计语言提供的类似于教科书上管程 monitor 通过互斥机制，如 C++中的互斥锁、条件变量，实现上述典型同步互斥问题。

**实验原理**

**进程/线程同步互斥的系统调用和 API 库函数**

* 内核信号量
  + 内核信号量定义为 struct semaphore 类型的对象：

struct semaphore {undefined

atomic\_t count;

int sleepers;

wait\_queue\_head\_t wait; }

count：信号量的值，大于 0，信号量控制的资源空闲；等于 0，资源忙，但没有进程等待这个资源；小于 0，资源不可用，并至少有一个进程等待资源。

wait：存放等待队列链表的地址，当前等待资源的所有睡眠进程都会放在这个链表中。

sleepers：等待使用该资源的进程个数，也是当该资源空闲时需要被唤醒的等待进程个数。

* + 对信号量的系统调用操作如下。

（1）内核信号量初始化：

void sema\_init (struct semaphore \*sem, int val);

void init\_MUTEX (struct semaphore \*sem); //将 sem 的值置为 1，表示资源空闲

void init\_MUTEX\_LOCKED (struct semaphore \*sem); //将 sem 的值置为 0，表示资源忙

（2）申请内核信号量所保护的资源，相当于wait()/P() 操作：

void down(struct semaphore \* sem); // 可引起睡眠

int down\_interruptible(struct semaphore \* sem); // down\_interruptible 能被信号打断

int down\_trylock(struct semaphore \* sem); // 非阻塞函数，不会睡眠。无法锁定资源则马上返回。

（3）释放内核信号量所保护的资源，相当于 signal()/V()操作：

void up(struct semaphore \* sem);

* Pthread/POSIX 信号量
  + Pthread 提供了两种线程同步互斥机制：互斥锁和信号量。互斥锁相当于教科书中的 binary semphore，取值为 0、1，用于线程互斥。
  + Pthread 提供了以下操作互斥锁机的基本函数：

⚫ 互斥锁初始化：pthread\_mutex\_init()；

⚫ 互斥锁上锁：pthread\_mutex\_lock()；

⚫ 互斥锁判断上锁：pthread\_mutex\_trylock()；

⚫ 互斥锁解锁：pthread\_mutex\_unlock()；

⚫ 消除互斥锁：pthread\_mutex\_destroy()

* 信号量为教科书中所述的 counting semaphore，即计数信号量，即可用于进程/线程间互斥、也可以应用于进程间同步。

进程/线程通过 wait()、signal 操作（或、PV 操作），改变信号量的值，获取共享资源的访问权限，实现进程/线程同步互斥。

* Pthread 线程库提供的信号量访问操作有：

⚫ sem\_init()，创建一个信号量，并初始化信号量的值；

⚫ sem\_wait()和 sem\_trywait()，相当于 wait/P 操作，在信号量>0 时，将信号量的值减1。两者的区别在于信号量<0 时，sem\_wait 将会阻塞进程/线程，而 sem\_trywait 则会立即返回；

⚫ sem\_post()，相当于 signal/V 操作，它将信号量的值加 1，同时发出信号唤醒等待的进程/线程；

⚫ sem\_getvalue()，得到信号量的值；

⚫ sem\_destroy()，删除信号量。

* System V 信号量
  + System V 信号量是由一个或多个计数信号量组成的计数信号量集合(set of counting semaphores)。
  + System V 提供的信号量相关的函函数数原型为：

⚫ 创建一个新信号量或取得一个已有信号量

int semget(key\_t key, int num\_sems, int sem\_flags)

多个进程使用同一个信号量键值 key 来获得同一个信号量。

⚫ 信号量初始化和删除 int semctl (int semid, int semnum, int cmd,...) 使用 semctl()函数的 SETVAL 操作进行初始化，使用 semctl()函数的 IPC\_RMID操作删除信号量。

⚫ 对信号量组进行 wait()/P()、signal()/V()操作，改变信号量的值int semop(int semid, struct sembuf \* opsptr , size\_t numops)

* 互斥锁/互斥量 mutex
  + C++中的互斥锁属于教科书中的初值为 1 的二元互斥信号量，用于控制对资源、共享变量的互斥访问，包括：

（1）std::mutex：最基本的互斥量，不支持递归地对 std::mutex 对象上锁；

（2）std::recursive\_lock：允许递归地对互斥量对象上锁。

* + 对互斥锁的操作有 lock()、unlock()、try\_lock()。其成员函数包括：

⚫ 构造函数

std::mutex 不允许拷贝构造和 move 拷贝，刚创建的 mutex 对象处于 unlocked 状态。

⚫ lock()

调用 lock()的线程将锁住互斥锁，线程调用该函数会发生 3 种情况：

(i)如果该互斥量当前没有被锁住，表示互斥锁对应的共享资源空闲，则调用线程将该互斥量锁住，拥有该锁，直到调用 unlock 去释放锁。

(ii)如果互斥量已经被其它线程锁住，表示其它线程已经占有共享资源，则当前的调用线程被阻塞。

(iii)如果当前互斥量被当前调用线程锁住，将产生死锁。

⚫ unlock()

解锁，释放对互斥量的所有权，即释放对共享资源的控制权。

⚫ try\_lock()

尝试锁住互斥量，如果互斥量已经被其它线程占有，当前线程不会被阻塞。线程调用该函数会发生 3 种情况：

(i)如果该互斥量当前没有被锁住，表示互斥锁对应的共享资源空闲，则调用线程将该互斥量锁住，拥有该锁，直到调用 unlock 去释放锁。

(ii)如果互斥量已经被其它线程锁住，表示其它线程已经占有共享资源，则当前的调用线程返回 false，并不会被阻塞。

(iii)如果当前互斥量被当前调用线程锁住，将产生死锁。

⚫ lock\_guard

在 lock\_guard 对象构造时，传入的 Mutex 对象（即它所管理的 Mutex 对象）将被当前线程锁住。在 lock\_guard 对象被析构时，它所管理的 Mutex 对象会自动解锁。由于不需要程序显示调用 lock 和 unlock 对 Mutex 进行上锁和解锁操作，因此是最简单

* + 安全的上锁和解锁方式，尤其是在程序抛出异常后先前已被上锁的 Mutex 对象可以正确进行解锁操作，大大简化了编写与 Mutex 相关的异常处理代码。
* 条件变量
  + 在 C++11 中，线程使用条件变量（condition\_variable）实现相互间的同步操作。
  + 条件变量利用线程间共享全局变量（即条件变量）进行同步，主要包括两个动作：

（1）一个线程等待“条件变量的条件成立”，当该条件不满足时，被阻塞、挂起，即进入等待状态；

（2）其它线程使“定义在条件变量上的条件成立”，给出信号，通知唤醒等待的线程。条件变量与互斥锁/互斥量 mutex、用户提供的判定条件相互配合，一起组合使用，可以实现生产者-消费者等复杂的同步互斥问题。条件变量可以原子地使得线程在唤醒时检查用户定义的判定条件，如果条件不满足就释放互斥锁，并入阻塞。

* 线程修改条件变量的动作为：

⚫ （1）获得一个定义在条件变量 std::condition\_variable 上的互斥锁 std::mutex；

⚫（2）当持有互斥锁后，修改条件变量；

⚫（3）对条件变量执行 notify\_one 或 notify\_all。说明：当执行 notify 动作时，不必持有锁共享变量是原子性的，必须在 mutex 的保护下被修改，以便能够将对条件变量的修改改动正确 notify 发布到正在等待的线程。

* 等待条件变量 std::condition\_variable 的线程必须：

⚫ （1）获取 std::unique\_lock<std::mutex>，此处的互斥量 mutex 用于保护作为条件的共享变量；

⚫ （2）执行 wait, wait\_for 或者 wait\_until. 这些等待动作原子性地释放 mutex，并使得线程的执行暂停；

⚫ （3）当获得条件变量的通知 notify，或者超时，或者一个虚假的唤醒，线程被唤醒，并且获得 mutex. 然后线程检查条件是否成立，如果是虚假唤醒，就继续等待。

* 与条件变量相关的函数如下。

1. 等待函数 wait：

(1) wait(unique\_lock <mutex>＆lck) 当前线程的执行会被阻塞，直到收到 notify 为止。

(2) wait(unique\_lock <mutex>＆lck，Predicate pred) 当前线程仅在 pred=false 时阻塞；如果 pred=true 时，不阻塞。wait()执行包括三个步骤：释放互斥锁、等待在条件变量上、再次获取互斥锁。

2. 通知函数 notify\_one：

notify\_one()：没有参数，也没有返回值。解除阻塞当前正在等待此条件的线程之一。如果没有线程在等待，则函数不执行任何操作。如果超过一个，不会指定具体哪一线程。

**实验题目及分析**

**实验题目**

生产流水线

An assembly line is to produce a product C with n1=4 part As, and n2=3 part Bs. The worker of

machining A and worker of machining B produce m1=2 part As and m2=1 part B independently

each time. Then the m1 part As or m2 part B will be moved to a station, which can hold at most

N=12 of part As and part Bs altogether. The produced m1 part As must be put onto the station

simultaneously. The workers must exclusively put a part on the station or get it from the station.

In addition, the worker to make C must get all part of As and Bs, i.e. n1 part As and n2 part Bs, for

one product C once.

Using semaphores to coordinate the three workers who are machining part A, part B and

manufacturing the product C without deadlock.

It is required that

(1) definition and initial value of each semaphore, and

(2) the algorithm to coordinate the production process for the three workers

should be given.

**实验分析**

**实验原理**

本题目是生产者-消费者问题的扩展，需要注意题目要求：(a) worker A 一次生产 m1=2个 A，必须一次性放入工作台，(b) worker C 必须一次性获得所需的 n1=4 个 A 和 n2=3 个 B。因此，如果当前工作台空位小于 m1，worker A 被阻塞；如果当前工作台没有足够的 A、B，worker C 被阻塞。

采用类似于多次 wait(empty)的操作，为 worker A 从工作台获取多个空位是不合理的。例如，假设当前工作台已经放置了 N=11 个零件，只有 1 个空位。worker A 生产出 2 个 A，worker，B 生产出 1 个 B。如果 worker A 先通过两次 wait(empty)操作申请两个工作台空位，将被阻塞，生产的零件 A 没有放入工作台。随后 worker B 再执行 wait(empty) 申请空位也将被阻塞，而此时工作台有 1 个空位。

因此，正确的解决方案是：

（1）利用用户空间计数变量 countA、countB、Numempty，配合互斥信号量 mutexA、mutexB、mutexNumempty，统计工作台中零件 A、零件 B、空单元的数目，当有足够多的零件 A、B和工作台空位时，再一次性申请/获取多个所需的零件 A、零件 B、工作台空位，即将 worker A 所需的 m1 个工作台空位作为一个整体一次性地申请，worker C 所需的工作台中的 n1个 A和 n2个 B 作为一个整体一次性地申请；

（2）当所需工作台空位不满足时，worker A 和 worker B 应主动阻塞自身(suspend/block)，防止忙等待。当 worker C 所需的工作台中 A 和 B 不满足时，也应主动阻塞自身；

（3）当 worker A、worker B 分别生产并放入新的零件 A、B 后，考虑唤醒由于所需零件不足而处于阻塞态的 worker C；同样地，当 worker C 从工作台取出零件 A、零件 B 后，考虑唤醒因没有足够空位而处于阻塞态的 worker A、worker B。

**实现分析**

**利用c++11条件变量实现**

（1） 设置缓冲区互斥锁access\_mutex实现对countA, countB, count的互斥访问。设置条件变量cond实现workA, workB, workC之间的同步操作。

（2） 对于workA，首先请求缓冲区互斥锁access\_mutex，然后判断缓冲区是否满足workA的工作要求，即countA<=7 && count <= 10。若不满足，则wait()利用条件变量cond将其挂起并阻塞，同时释放缓冲区互斥锁access\_mutex。

（3） 当workA的工作条件满足时，workA线程被唤醒，并拿到互斥锁access\_mutex，开始工作。工作完毕后，调用条件变量的notify\_all()函数通知其他阻塞线程，同时释放互斥锁access\_mutex。

（4）对workB，workC同理。

**利用Pthread/POSIX信号量实现**

（1） 利用类似管程的方式对workA，workB，workC进行管理。设置缓冲区互斥锁mutex实现对countA，countB，count的互斥访问。设置二元信号量goA，goB，goC实现对workA，workB，workC的唤醒。

（2） 对于workA，首先调用testA()函数，判断workA能否工作。

（3） workA在testA()后，等待信号量goA以继续执行。之后调用testC()，测试workC能否进行工作，然后结束workA的一个工作周期，开始下一轮的工作循环。

（4） 在testA()中，先取得互斥锁mutex，再进行workA的工作条件判断，即countA<=7 && count <= 10。若满足条件，则workA立即进行工作，并唤醒二元信号量goA，使得workA进行下一次的工作条件判断，之后释放互斥锁mutex；若不满足条件，则直接释放互斥锁mutex，使workA进入等待goA的阻塞状态。

（5） workB和workC的工作流程相似。但是在workC等待信号量goC后，应调用testA()和testB()，以开始workA和workB的工作条件判断，避免阻塞。

**代码实现**

**条件变量实现（3\_phr\_mutx\_cond.cpp）**

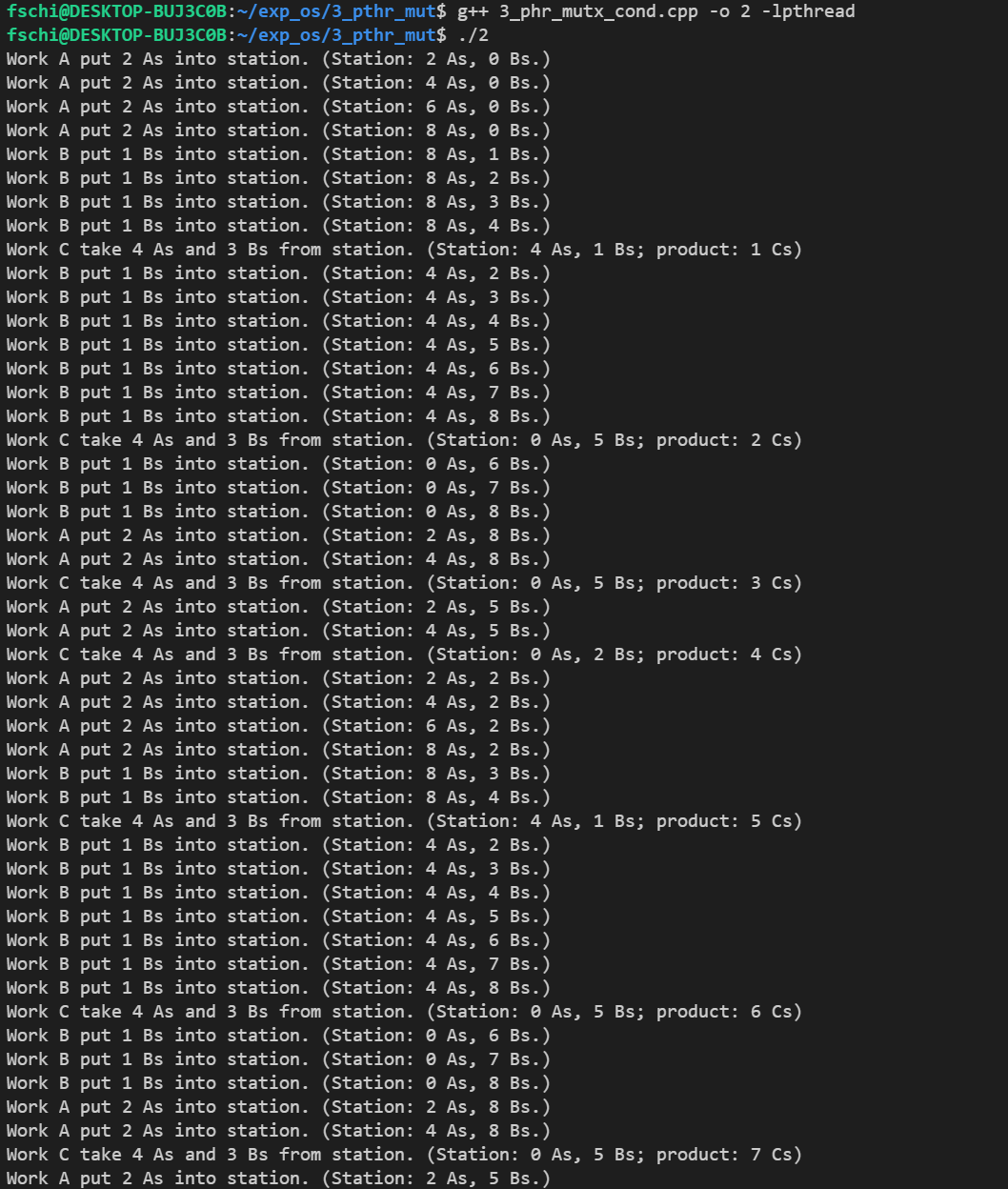
1. #include <condition\_variable>
2. #include <cstdio>
3. #include <iostream>
4. #include <mutex>
5. #include <queue>
6. #include <thread>
8. **const** **int** MAX\_STATION\_SIZE = 12;
9. **const** **int** CAPA\_A = 2;
10. **const** **int** CAPA\_B = 1;
12. // 缓冲区
13. **int** countA, countB, count;
15. std::mutex access\_mtx;         // 缓冲区互斥锁
16. std::condition\_variable cond;  // 条件变量（countA, countB, count）
18. **void** workA\_fun(**int** goal) {
19. **int** now\_prods = 0;
20. **while** (now\_prods < goal) {
21. std::unique\_lock<std::mutex> lk(access\_mtx);  //对缓冲区加锁
22. cond.wait(lk, []() {
23. **return** countA <= 7 && count <= 10;
24. });  //工人A可以继续生产，继续添加A
25. countA += CAPA\_A;
26. now\_prods += CAPA\_A;
27. count += CAPA\_A;
28. printf("Work A put %d As into station.", CAPA\_A);
29. printf(" (Station: %d As, %d Bs.)\n", countA, countB);
30. cond.notify\_all();  //唤醒其他线程，同时自动释放锁lk
31. }
32. }
34. **void** workB\_fun(**int** goal) {
35. **int** now\_prods = 0;
36. **while** (now\_prods < goal) {
37. std::unique\_lock<std::mutex> lk(access\_mtx);
38. cond.wait(lk, []() { **return** countB <= 7 && count <= 11; });
39. countB += CAPA\_B;
40. now\_prods += CAPA\_B;
41. count += CAPA\_B;
42. printf("Work B put %d Bs into station.", CAPA\_B);
43. printf(" (Station: %d As, %d Bs.)\n", countA, countB);
44. cond.notify\_all();
45. }
46. }
48. **void** workC\_fun(**int** goal) {
49. **int** now\_prods = 0;
50. **while** (now\_prods < goal) {
51. std::unique\_lock<std::mutex> lk(access\_mtx);  //对缓冲区加锁
52. cond.wait(lk, []() {
53. **return** countA >= 4 && countB >= 3;
54. });  //工人C达到组装条件，开始组装
55. countA -= 4;
56. countB -= 3;
57. count -= 7;
58. now\_prods += 1;
59. printf("Work C take 4 As and 3 Bs from station.");
60. printf(" (Station: %d As, %d Bs; product: %d Cs)\n", countA, countB,
61. now\_prods);
62. cond.notify\_all();  //唤醒其他线程，同时自动释放锁lk
63. }
64. }
65. **int** main() {
66. //   freopen("res.txt", "w", stdout);
67. countA = countB = count = 0;
68. std::**thread** workA, workB, workC;
69. workA = std::**thread**(workA\_fun, 40000);
70. workB = std::**thread**(workB\_fun, 30000);
71. workC = std::**thread**(workC\_fun, 10000);
72. workA.join();
73. workB.join();
74. workC.join();
75. **return** 0;
76. }

**Pthread/POSIX信号量实现（3\_phr\_mutx.c）**

1. #include <pthread.h>
2. #include <semaphore.h>
3. #include <stdio.h>
4. #include <stdlib.h>
5. #include <unistd.h>
7. **const** **int** MAX\_STATION\_SIZE = 12;
8. **const** **int** CAPA\_A = 2;
9. **const** **int** CAPA\_B = 1;
10. **const** **int** CAPA\_C = 1;
12. **int** countA, countB, count;  //缓冲区
13. pthread\_mutex\_t mutex;      // 对Station的互斥访问锁
14. sem\_t goA, goB, goC;
16. **void** testA() {
17. pthread\_mutex\_lock(&mutex);        // 取得互斥锁
18. **if** (countA <= 7 && count <= 10) {  // 判断是否满足工作条件
19. countA += CAPA\_A;                // 工作
20. count += CAPA\_A;
21. printf("Work A put %d As into station.", CAPA\_A);
22. printf(" (Station: %d prods: %d As, %d Bs.)\n", count, countA, countB);
23. sem\_post(&goA);  // 唤醒goA信号量，允许下一次工作条件判断
24. }
25. pthread\_mutex\_unlock(&mutex);  //归还互斥锁
26. }
27. **void** \*workA\_fun(**void** \*arg) {
28. **while** (1) {
29. testA();         // workA工作条件判断
30. sem\_wait(&goA);  // workA刚工作了一次，允许下一次工作条件判断
31. testC();         //测试workC能否工作
32. }
33. }
35. **void** testB() {
36. pthread\_mutex\_lock(&mutex);
37. **if** (countB <= 7 && count <= 11) {
38. countB += CAPA\_B;
39. count += CAPA\_B;
40. printf("Work B put %d Bs into station.", CAPA\_B);
41. printf(" (Station: %d prods: %d As, %d Bs.)\n", count, countA, countB);
42. sem\_post(&goB);
43. }
44. pthread\_mutex\_unlock(&mutex);
45. }
46. **void** \*workB\_fun(**void** \*arg) {
47. **while** (1) {
48. testB();
49. sem\_wait(&goB);
50. testC();
51. }
52. }
54. **void** testC() {
55. pthread\_mutex\_lock(&mutex);
56. **if** (countA >= 4 && countB >= 3) {
57. countA -= 4;
58. countB -= 3;
59. count -= 7;
60. printf("Work C take 4 As and 3 Bs from station.");
61. printf(" (Station: %d = %d As, %d Bs)\n", count, countA, countB);
62. sem\_post(&goC);
63. }
64. pthread\_mutex\_unlock(&mutex);
65. }
66. **void** \*workC\_fun(**void** \*arg) {
67. **while** (1) {
68. testC();
69. sem\_wait(&goC);
70. testA();
71. testB();
72. }
73. }
75. **int** main() {
76. pthread\_t workA\_id, workB\_id, workC\_id;
77. **int** ret, arg;
78. pthread\_mutex\_init(&mutex, NULL);
79. ret = sem\_init(&goA, 0, 0);
80. **if** (ret != 0) {
81. printf("sem\_init error");
82. exit(0);
83. }
84. ret = sem\_init(&goB, 0, 0);
85. **if** (ret != 0) {
86. printf("sem\_init error");
87. exit(0);
88. }
89. ret = sem\_init(&goC, 0, 0);
90. **if** (ret != 0) {
91. printf("sem\_init error");
92. exit(0);
93. }
95. ret = pthread\_create(&workA\_id, NULL, workA\_fun, NULL);
96. **if** (ret != 0) {
97. printf("work A create error");
98. exit(0);
99. }
100. ret = pthread\_create(&workB\_id, NULL, workB\_fun, NULL);
101. **if** (ret != 0) {
102. printf("work B create error");
103. exit(0);
104. }
105. ret = pthread\_create(&workC\_id, NULL, workC\_fun, NULL);
106. **if** (ret != 0) {
107. printf("work C create error");
108. exit(0);
109. }
110. pthread\_join(workA\_id, NULL);
111. pthread\_join(workB\_id, NULL);
112. pthread\_join(workC\_id, NULL);
113. **return** 0;
114. }

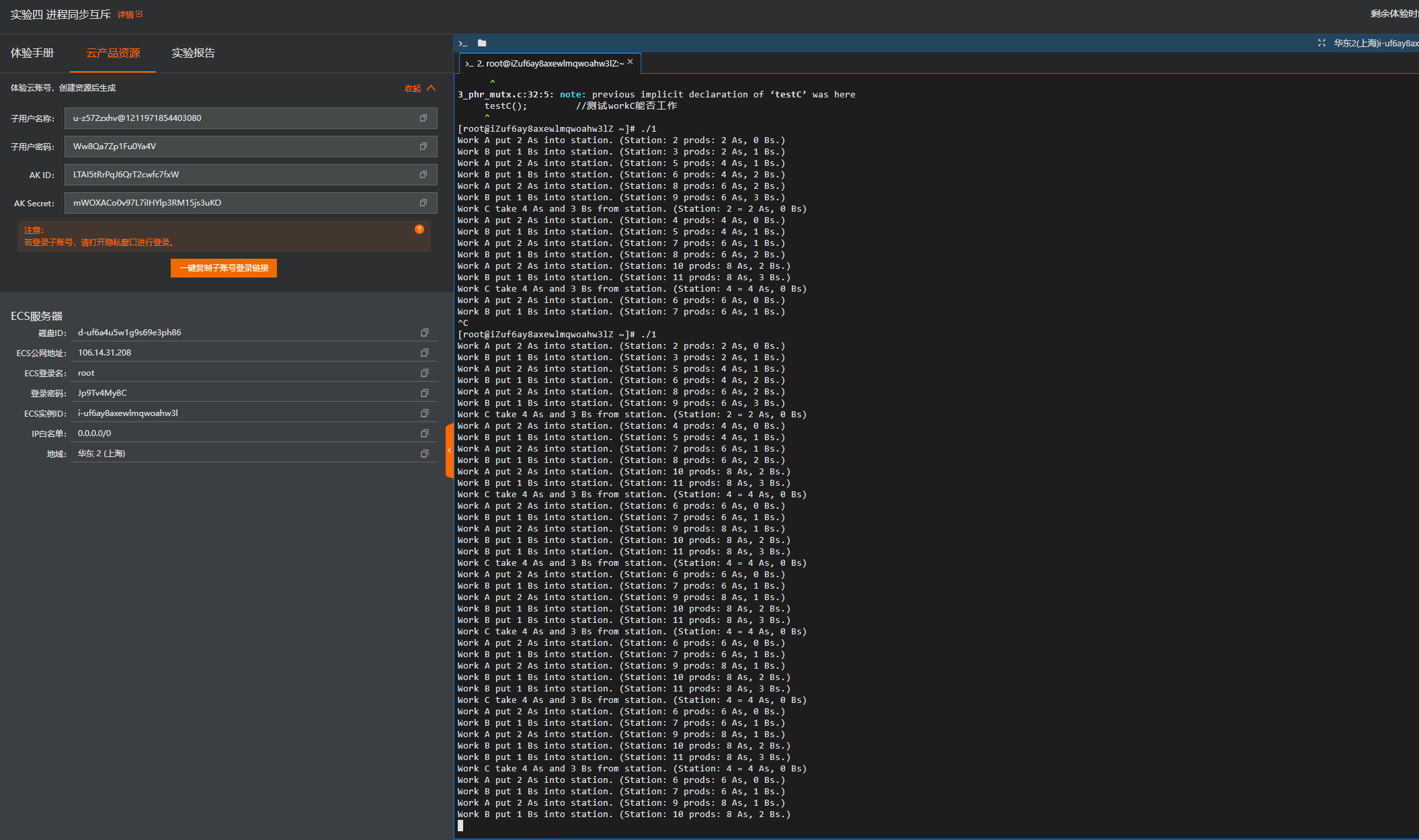
**结果分析**

**条件变量实现**



程序正常运行5min，无死锁现象。由于条件变量的引入，系统避免了空等待的问题，使得线程间的同步更有效地执行。

**Pthread/POSIX信号量实现**



程序正常运行5min，无死锁现象。通过管程的实现，保证了每个工人多次的读取、存储操作为原子操作，保证了线程间的同步互斥关系的正确性。

# 第五组 多核多线程编程及性能分析

参照参考文献“利用多核多线程进行程序优化”，在Linux环境下，编写多线程程序，分析以下几个因素对程序运行时间的影响：

⚫ 程序并行化

⚫ 线程数目

⚫ 共享资源加锁

⚫ CPU亲和

⚫ cache优化

掌握多CPU、多核硬件环境下基本的多线程并行编程技术。

实验内容包括：

⚫ 实验5.1观察实验平台物理cpu、CPU核和逻辑cpu的数目

⚫ 实验5.2. 单线程/进程串行vs 两线程并行vs 三线程加锁并行程序对比

⚫ 实验5.3. 三线程加锁vs 三线程不加锁对比

⚫ 实验5.4. 针对Cache 的优化

⚫ 实验5.5. CPU亲和力对并行程序影响

⚫ 八种实现方案运行时间对比总结

要求：

⚫ 以柱状图形式表示上述5 种情况下程序运行时间的定量测试结果（运行时间）

⚫ 分析对比程序并行化、线程数目、共享资源加、CPU 亲和、cache 优化对程序运行时间的影响，结合程序/进程自身业务逻辑、相互间同步互斥关系等分析解释运行时间产生差异的原因。

## 观察实验平台物理cpu、CPU核和逻辑cpu的数目

### 实验设计原理

目的：观察实验所采用的计算机（微机、笔记本电脑）物理cpu、CPU核和逻辑cpu的数目。

测试环境(示例)：

硬件：Intel(R) Core(TM) i5-8250U CPU，主频1.60GHz，内存8.00 GB

软件：Windows 10 家庭中文版 64 位操作系统，Windows Subsystem for Linux 2，Ubuntu 20.04.3 LTS，内核版本：Linux version 5.4.72-microsoft-standard-WSL2 (oe-user@oe-host) (gcc version 8.2.0 (GCC))

* 物理cpu数目：主板上实际插入的cpu数量，可以数不重复的physical id 有几个（physical id）。多路服务器、大型主机系统、集群系统一般可以配置多个物理CPU；常规微机、笔记本电脑一般只配备1个物理CPU；
* cpu核（cpu cores）的数目：单块CPU上面能处理数据的芯片组的数量，如双核、四核等；
* 逻辑cpu数目：

对不支持超线程HT的CPU，逻辑cpu数目=物理CPU个数×每颗CPU核数，

对支持超线程HT的CPU，逻辑cpu数目=物理CPU个数×每颗CPU核数\*2

通过下列命令查看**cpu**相关信息

* 物理cpu数：

[XXXX@server ~]$ grep 'physical id' /proc/cpuinfo|sort|uniq|wc -l

* cpu核数：

[XXXX@server ~]$ grep 'cpu cores' /proc/cpuinfo|uniq|awk -F ':' '{print $2}'

* 逻辑cpu：

[XXXX@server ~]$ cat /proc/cpuinfo| grep "processor"|wc -l

### 程序代码

1. grep 'physical id' /proc/cpuinfo|sort|uniq|wc -l
2. grep 'cpu cores' /proc/cpuinfo|uniq|awk -F ':' '{print $2}'
3. cat /proc/cpuinfo| grep "processor"|wc -l

### 实验结果与分析

#### 实验结果







#### 分析

实验平台有1个物理CPU，4个CPU核，8个逻辑CPU。

## 单线程/进程串行vs 两线程并行vs 三线程加锁并行程序对比

### 实验设计原理

程序功能：

求从1一直到 APPLE\_MAX\_VALUE (100000000) 相加累计的和，并赋值给 apple 的 a 和 b ；求 orange 数据结构中的 a[i]+b[i ] 的和，循环 ORANGE\_MAX\_VALUE(1000000) 次。

（**1**） 步骤**1.** 单线程程序

（**2**） 步骤**2. 2**线程并行程序

采用任务分解的方法，将互不相关的计算apple 值和计算orange 值的2部分代码分解为2个线程，实现线程级并行执行。

（**3**） 步骤**3. 3** 线程加锁并行程序

通过数据分解的方法，还可以发现，计算apple 的值可以分解为两个线程，一个用于计算apple a 的值，另外一个线程用于计算apple b 的值。

但两个线程存在同时访问apple 的可能性，需要加锁访问该数据结构。

（4） 步骤**4.** 参照参考文献，采用K-Best 测量方法，对比分析单线程/进程、2线程、3线程加锁程序的运行时间差异，以表格或图形方式给出对比结果,并分析导致差异的原因，判断多线程并行编程是否达到预期。

K**-Best** 测量方法

在检测程序运行时间这个复杂问题上，采用 Randal E.Bryant 和 David R. O’Hallaron 提出的K次最优测量方法。

假设重复执行一个程序，并纪录该程序的K次最快执行时间，如果发现测量的误差ε很小，那么用测量得到的最快值表示程序的真正执行时间， 这种方法称为“ K 次最优（K-Best）方法”，要求设置三个参数：

（1）k：要求在某个接近最快执行时间值范围内的测量值数量；

（2）ε：测量值必须多大程度地接近，即测量值按照升序标号V1, V2, V3 ,… , Vi，… ，同时必须满足(1+ ε)Vi ≥ Vk

（3）M: 在结束测试之前，测量值的最大数量，即最大测量次数

按照升序的方式维护一个由k个最快执行时间组成的数组KTEST，数组KTEST的长度为k，记录了k个测量得到的程序执行时间，且按照升序排列。

在多轮程序执行时间测量过程中，对于每一轮测试得到的每一个新的程序执行时间测量值，如果该值比当前数组KTEST中的最大值KTEST[k]更小，则用该最新测量值替换数组中k处的元素KTEST[k]，然后再按照升序重新排列数组KTEST[k]。

持续不断进行该过程，并满足误差标准，此时就称测量值已经收敛。如果M次后，不能满足误差标准，则称为测量值不能收敛。

在接下来的所有试验中，采用K=10，ε=2%，M=200来获取程序运行时间，同时也对K次最优测量方法进行了改进：不是采用最小值来表示程序执行的时间，而是采用K次测量值的平均值来表示程序的真正运行时间。由于采用的误差ε比较大，在所有试验程序的时间收集过程中，均能收敛，但也能说明问题。

为了可移植性，采用gettimeofday()来获取系统时钟（system clock）时间，可以精确到微秒。

### 程序代码

#### demo1.c

1. #define ORANGE\_MAX\_VALUE 1000000
2. #define APPLE\_MAX\_VALUE 100000000
3. #define MSECOND 1000000
4. struct apple{
5. unsigned long long a;
6. unsigned long long b;
7. };
8. struct orange{
9. int a[ORANGE\_MAX\_VALUE];
10. int b[ORANGE\_MAX\_VALUE];
11. };
12. int main(int argc, const char\* argv[]) {
13. struct apple test;
14. struct orange test1;
15. int sum;
16. for(sum = 0; sum < APPLE\_MAX\_VALUE; sum++){
17. test.a += sum;
18. test.b += sum;
19. }
20. sum = 0;
21. for(int index = 0; index < ORANGE\_MAX\_VALUE; index++){
22. sum += test1.a[index] + test1.b[index];
23. }
24. return 0;
25. }

#### demo2.c

1. #include <pthread.h>
2. #include <stdio.h>
3. #define ORANGE\_MAX\_VALUE 1000000
4. #define APPLE\_MAX\_VALUE 100000000
5. #define MSECOND 1000000
6. #define NUM\_THREADS 1
7. struct apple{
8. unsigned long long a;
9. unsigned long long b;
10. };
11. struct orange{
12. int a[ORANGE\_MAX\_VALUE];
13. int b[ORANGE\_MAX\_VALUE];
14. };
15. void\* p\_apple(void \*test) {
16. int sum;
17. for(sum = 0; sum < APPLE\_MAX\_VALUE; sum++){
18. ((struct apple \*)test)->a += sum;
19. ((struct apple \*)test)->b += sum;
20. }
21. }
22. void p\_orange(void \*test1) {
23. int sum = 0;
24. for(int index = 0; index < ORANGE\_MAX\_VALUE; index++){
25. sum += ((struct orange \*)test1)->a[index] + ((struct orange \*)test1)->b[index];
26. }
27. }
28. int main(int argc, const char\* argv[]) {
29. struct apple test;
30. struct orange test1;
31. // 定义线程的 id 变量，多个变量使用数组
32. pthread\_t tids[NUM\_THREADS];
33. //参数依次是：创建的线程id，线程参数，调用的函数，传入的函数参数
34. int ret0 = pthread\_create(&tids[0], NULL, p\_apple, &test);
35. if (ret0 != 0){
36. printf("pthread\_create error: error\_code=%d\n", ret0);
37. }
38. p\_orange(&test1);
39. pthread\_join(tids[0],NULL);
40. return 0;
41. }

#### demo3.c

1. #include <pthread.h>
2. #include <stdio.h>
3. #define ORANGE\_MAX\_VALUE 1000000
4. #define APPLE\_MAX\_VALUE 100000000
5. #define MSECOND 1000000
6. #define NUM\_THREADS 3
7. struct apple{
8. unsigned long long a;
9. unsigned long long b;
10. };
11. struct orange{
12. int a[ORANGE\_MAX\_VALUE];
13. int b[ORANGE\_MAX\_VALUE];
14. };
15. pthread\_mutex\_t mutex;
16. void\* p\_apple\_a(void \*test) {
17. int sum;
18. for(sum = 0; sum < APPLE\_MAX\_VALUE; sum++){
19. pthread\_mutex\_lock(&mutex);
20. ((struct apple \*)test)->a += sum;
21. pthread\_mutex\_unlock(&mutex);
22. }
23. }
24. void\* p\_apple\_b(void \*test) {
25. int sum;
26. for(sum = 0; sum < APPLE\_MAX\_VALUE; sum++){
27. pthread\_mutex\_lock(&mutex);
28. ((struct apple \*)test)->b += sum;
29. pthread\_mutex\_unlock(&mutex);
30. }
31. }
32. void p\_orange(void \*test1) {
33. int sum = 0;
34. for(int index = 0; index < ORANGE\_MAX\_VALUE; index++){
35. sum += ((struct orange \*)test1)->a[index] + ((struct orange \*)test1)->b[index];
36. }
37. }
38. int main(int argc, const char\* argv[]) {
39. struct apple test;
40. struct orange test1;
41. // 定义线程的 id 变量，多个变量使用数组
42. pthread\_t tids[NUM\_THREADS];
43. pthread\_mutex\_init(&mutex, NULL); //初始化互斥量
44. //参数依次是：创建的线程id，线程参数，调用的函数，传入的函数参数
45. int ret0 = pthread\_create(&tids[0], NULL, p\_apple\_a, &test);
46. if (ret0 != 0){
47. printf("pthread\_create error: error\_code=%d\n", ret0);
48. }
49. int ret1 = pthread\_create(&tids[1], NULL, p\_apple\_b, &test);
50. if (ret1 != 0){
51. printf("pthread\_create error: error\_code=%d\n", ret1);
52. }
53. p\_orange(&test1);
54. pthread\_join(tids[0],NULL);
55. pthread\_join(tids[1],NULL);
56. return 0;
57. }

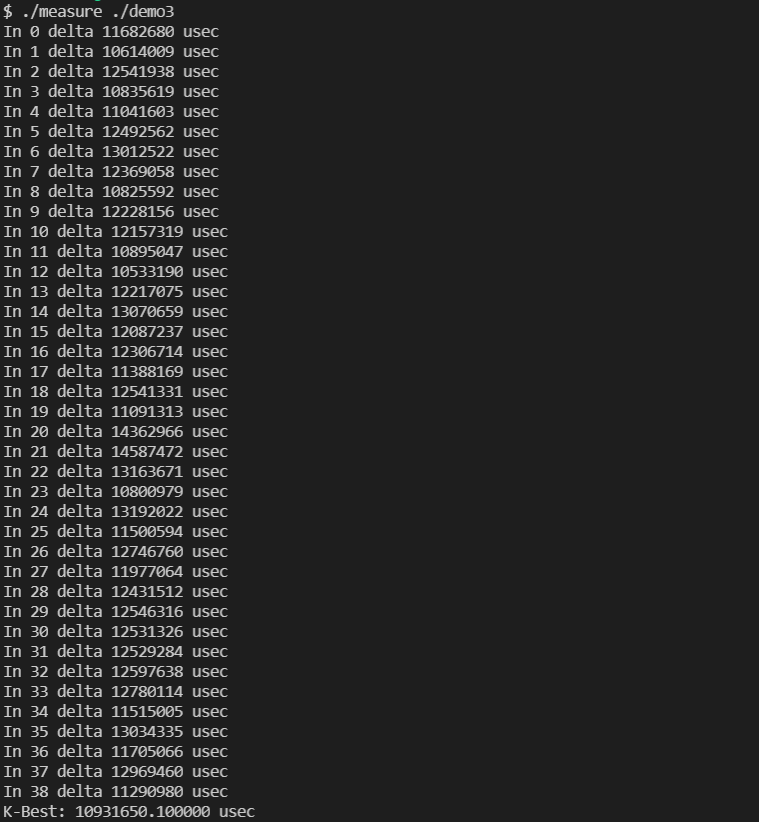
#### measure.c

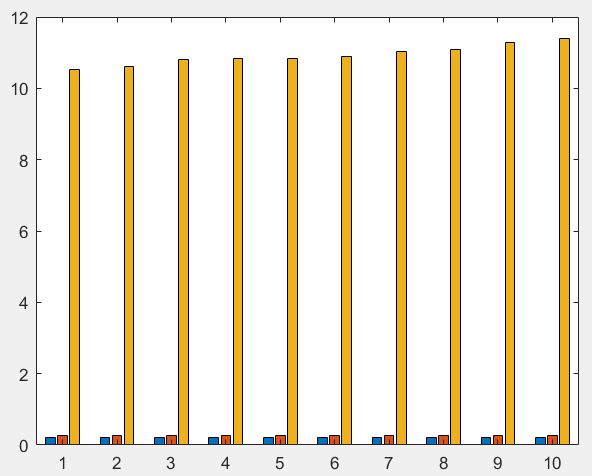
1. #include<stdio.h>
2. #include<sys/time.h>
3. #include<unistd.h>
4. #include<stdlib.h>
5. #define K 10
6. #define EPS 0.02
7. #define M 200
8. #define INF 1000000000
9. int main(int argc, char\* argv[]) {
10. struct timeval tv\_begin,tv\_end;
11. int ktest[K];
12. for (int k = 0; k < K; ++k) ktest[k] = INF;
13. for (int round = 0; round < M; ++round) {
14. gettimeofday(&tv\_begin,NULL);
15. //system("./demo1");
16. system(argv[1]);
17. gettimeofday(&tv\_end,NULL);
18. int delta = tv\_end.tv\_sec \* 1e6 + tv\_end.tv\_usec - (tv\_begin.tv\_sec \* 1e6 + tv\_begin.tv\_usec);
19. printf("In %d delta %d usec\n", round, delta);
20. for (int k = 0; k < K; ++k)
21. if (delta < ktest[k]) {
22. for (int p = K - 1; p > k; --p)
23. ktest[p] = ktest[p - 1];
24. ktest[k] = delta;
25. break;
26. }
27. int isFinish = 1;
28. for (int k = 1; k < K; ++k)
29. if (ktest[k - 1] \* (1 + EPS) < ktest[k]) {
30. isFinish = 0;
31. break;
32. }
33. if (isFinish) {
34. int sum = 0;
35. for (int k = 0; k < K; ++k)
36. sum += ktest[k];
37. printf("K-Best: %lf usec\n", 1.0 \* sum / K);
38. exit(0);
39. }
40. }
41. printf("Can't find K-Best\n");
42. //printf(“tv\_begin: %d sec %d usec\n”,tv\_begin.tv\_sec,tv\_begin.tv\_usec);
43. //printf(“tv\_end: %d sec %d usec\n”,tv\_end.tv\_sec,tv\_end.tv\_usec);
44. }

### 实验结果与分析

#### 实验结果







#### 分析

单线程需要204毫秒，两线程并行需要271毫秒，三线程加锁需要10931毫秒。发现两线程比单线程耗时还多，这是因为线程初始化和上下文切换需要消耗额外的时间。三线程加锁比两线程耗时还多，这是因为加锁解锁及等待消耗额外的时间。

## 三线程加锁vs 三线程不加锁对比

### 实验设计原理

在前述3线程加锁方案，计算Apple的两个并行线程访问的是 apple 的不同元素，没有加锁的必要。所以修改 apple 的数据结构，删除读写锁代码，通过不加锁来提高性能。

比较分析3线程程序，在加锁与不加锁耗时的运行时间差异，以表格或图形方式给出对比结果,并分析导致差异的原因，判断加锁机制是否达到预期。

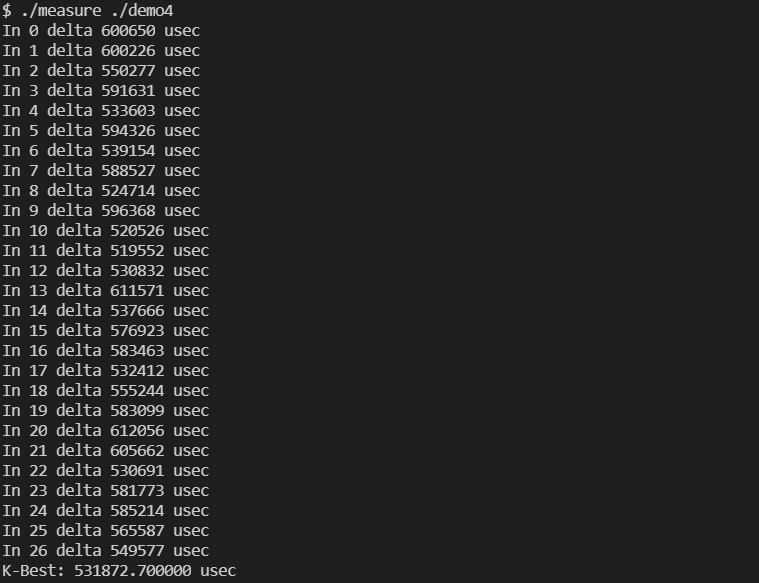
### 程序代码

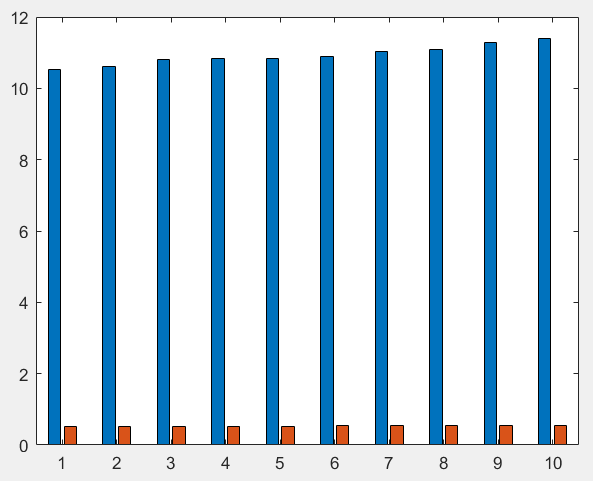
#### demo4.c

1. #include <pthread.h>
2. #include <stdio.h>
3. #define ORANGE\_MAX\_VALUE 1000000
4. #define APPLE\_MAX\_VALUE 100000000
5. #define MSECOND 1000000
6. #define NUM\_THREADS 3
7. struct apple{
8. unsigned long long a;
9. unsigned long long b;
10. };
11. struct orange{
12. int a[ORANGE\_MAX\_VALUE];
13. int b[ORANGE\_MAX\_VALUE];
14. };
15. void\* p\_apple\_a(void \*test) {
16. int sum;
17. for(sum = 0; sum < APPLE\_MAX\_VALUE; sum++){
18. ((struct apple \*)test)->a += sum;
19. }
20. }
21. void\* p\_apple\_b(void \*test) {
22. int sum;
23. for(sum = 0; sum < APPLE\_MAX\_VALUE; sum++){
24. ((struct apple \*)test)->b += sum;
25. }
26. }
27. void p\_orange(void \*test1) {
28. int sum = 0;
29. for(int index = 0; index < ORANGE\_MAX\_VALUE; index++){
30. sum += ((struct orange \*)test1)->a[index] + ((struct orange \*)test1)->b[index];
31. }
32. }
33. int main(int argc, const char\* argv[]) {
34. struct apple test;
35. struct orange test1;
36. // 定义线程的 id 变量，多个变量使用数组
37. pthread\_t tids[NUM\_THREADS];
38. //参数依次是：创建的线程id，线程参数，调用的函数，传入的函数参数
39. int ret0 = pthread\_create(&tids[0], NULL, p\_apple\_a, &test);
40. if (ret0 != 0){
41. printf("pthread\_create error: error\_code=%d\n", ret0);
42. }
43. int ret1 = pthread\_create(&tids[1], NULL, p\_apple\_b, &test);
44. if (ret1 != 0){
45. printf("pthread\_create error: error\_code=%d\n", ret1);
46. }
47. p\_orange(&test1);
48. pthread\_join(tids[0],NULL);
49. pthread\_join(tids[1],NULL);
50. return 0;
51. }

### 实验结果与分析

#### 实验结果





#### 分析

三线程加锁需要10931毫秒，三线程不加锁需要531毫秒。三线程不加锁耗时远小于三线程加锁，可见加锁解锁消耗了相当一部分的时间。

## 针对Cache的优化

### 实验设计原理

在串行程序设计过程中，为了节约带宽或者存储空间，比较直接的方法，就是对数据结构做一些针对性的设计，将数据压缩 (pack) 的更紧凑，减少数据的移动，以此来提高程序的性能。但在多核多线程程序中，这种方法往往有时会适得其反。

数据不仅在执行核和存储器之间移动，还会在执行核之间传输。根据数据相关性，其中有两种读写模式会涉及到数据的移动：写后读和写后写 ，因为这两种模式会引发数据的竞争，表面上是并行执行，但实际只能串行执行，进而影响到性能。

处理器交换的最小单元是 cache 行，或称 cache 块。在多核体系中，对于不共享 cache 的架构来说，两个独立的 cache 在需要读取同一 cache 行时，会共享该 cache 行，如果在其中一个 cache 中，该 cache 行被写入，而在另一个 cache 中该 cache 行被读取，那么即使读写的地址不相交，也需要在这两个 cache 之间移动数据，这就被称为 cache 伪共享，导致执行核必须在存储总线上来回传递这个 cache 行，这种现象被称为“乒乓效应”。

同样地，当两个线程写入同一个 cache 的不同部分时，也会互相竞争该 cache 行，也就是写后写的问题。上文曾提到，不加锁的方案反而比加锁的方案更慢，就是互相竞争 cache 的原因。

在 X86 机器上，某些处理器的一个 cache 行是64字节，具体可以参看 Intel 的参考手册。

不加锁三线程程序的瓶颈在于 cache，针对不加锁的3线程程序： 让 apple 的两个成员 a 和 b 位于不同的 cache 行中，观察增加Cache后的运行时间，分析是否达到预期要求。

针对加锁三线程程序，对apple 数据结构也增加一行类似功能的代码，判断效率是否提升？

预期结论：性能不会有所提升，其原因是加锁的三线程方案效率低下的原因不是 Cache 失效造成的，而是那把锁。

### 程序代码

#### demo3\_p.c

1. #include <pthread.h>
2. #include <stdio.h>
3. #define ORANGE\_MAX\_VALUE 1000000
4. #define APPLE\_MAX\_VALUE 100000000
5. #define MSECOND 1000000
6. #define NUM\_THREADS 3
7. struct apple{
8. unsigned long long a;
9. char c[128]; /\*32,64,128\*/
10. unsigned long long b;
11. };
12. struct orange{
13. int a[ORANGE\_MAX\_VALUE];
14. int b[ORANGE\_MAX\_VALUE];
15. };
16. pthread\_mutex\_t mutex;
17. void\* p\_apple\_a(void \*test) {
18. int sum;
19. for(sum = 0; sum < APPLE\_MAX\_VALUE; sum++){
20. pthread\_mutex\_lock(&mutex);
21. ((struct apple \*)test)->a += sum;
22. pthread\_mutex\_unlock(&mutex);
23. }
24. }
25. void\* p\_apple\_b(void \*test) {
26. int sum;
27. for(sum = 0; sum < APPLE\_MAX\_VALUE; sum++){
28. pthread\_mutex\_lock(&mutex);
29. ((struct apple \*)test)->b += sum;
30. pthread\_mutex\_unlock(&mutex);
31. }
32. }
33. void p\_orange(void \*test1) {
34. int sum = 0;
35. for(int index = 0; index < ORANGE\_MAX\_VALUE; index++){
36. sum += ((struct orange \*)test1)->a[index] + ((struct orange \*)test1)->b[index];
37. }
38. }
39. int main(int argc, const char\* argv[]) {
40. struct apple test;
41. struct orange test1;
42. // 定义线程的 id 变量，多个变量使用数组
43. pthread\_t tids[NUM\_THREADS];
44. pthread\_mutex\_init(&mutex, NULL); //初始化互斥量
45. //参数依次是：创建的线程id，线程参数，调用的函数，传入的函数参数
46. int ret0 = pthread\_create(&tids[0], NULL, p\_apple\_a, &test);
47. if (ret0 != 0){
48. printf("pthread\_create error: error\_code=%d\n", ret0);
49. }
50. int ret1 = pthread\_create(&tids[1], NULL, p\_apple\_b, &test);
51. if (ret1 != 0){
52. printf("pthread\_create error: error\_code=%d\n", ret1);
53. }
54. p\_orange(&test1);
55. pthread\_join(tids[0],NULL);
56. pthread\_join(tids[1],NULL);
57. return 0;
58. }

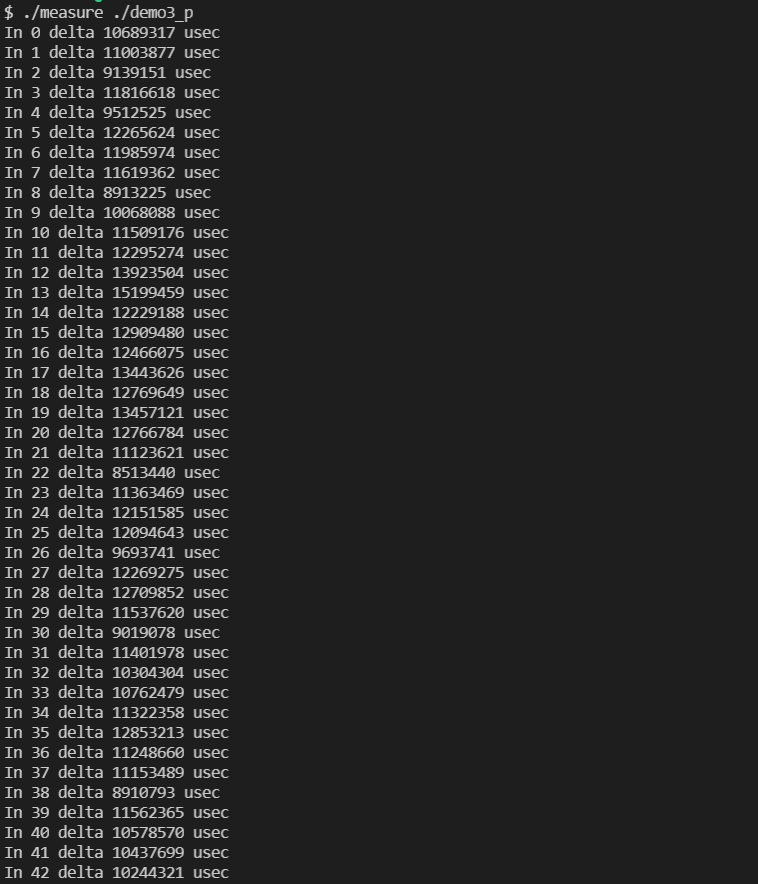
#### demo4\_p.c

1. #include <pthread.h>
2. #include <stdio.h>
3. #define ORANGE\_MAX\_VALUE 1000000
4. #define APPLE\_MAX\_VALUE 100000000
5. #define MSECOND 1000000
6. #define NUM\_THREADS 3
7. struct apple{
8. unsigned long long a;
9. char c[128]; /\*32,64,128\*/
10. unsigned long long b;
11. };
12. struct orange{
13. int a[ORANGE\_MAX\_VALUE];
14. int b[ORANGE\_MAX\_VALUE];
15. };
16. void\* p\_apple\_a(void \*test) {
17. int sum;
18. for(sum = 0; sum < APPLE\_MAX\_VALUE; sum++){
19. ((struct apple \*)test)->a += sum;
20. }
21. }
22. void\* p\_apple\_b(void \*test) {
23. int sum;
24. for(sum = 0; sum < APPLE\_MAX\_VALUE; sum++){
25. ((struct apple \*)test)->b += sum;
26. }
27. }
28. void p\_orange(void \*test1) {
29. int sum = 0;
30. for(int index = 0; index < ORANGE\_MAX\_VALUE; index++){
31. sum += ((struct orange \*)test1)->a[index] + ((struct orange \*)test1)->b[index];
32. }
33. }
34. int main(int argc, const char\* argv[]) {
35. struct apple test;
36. struct orange test1;
37. // 定义线程的 id 变量，多个变量使用数组
38. pthread\_t tids[NUM\_THREADS];
39. //参数依次是：创建的线程id，线程参数，调用的函数，传入的函数参数
40. int ret0 = pthread\_create(&tids[0], NULL, p\_apple\_a, &test);
41. if (ret0 != 0){
42. printf("pthread\_create error: error\_code=%d\n", ret0);
43. }
44. int ret1 = pthread\_create(&tids[1], NULL, p\_apple\_b, &test);
45. if (ret1 != 0){
46. printf("pthread\_create error: error\_code=%d\n", ret1);
47. }
48. p\_orange(&test1);
49. pthread\_join(tids[0],NULL);
50. pthread\_join(tids[1],NULL);
51. return 0;
52. }

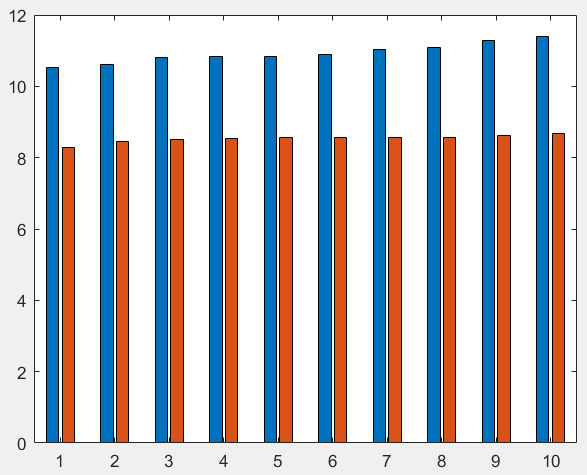
### 实验结果与分析

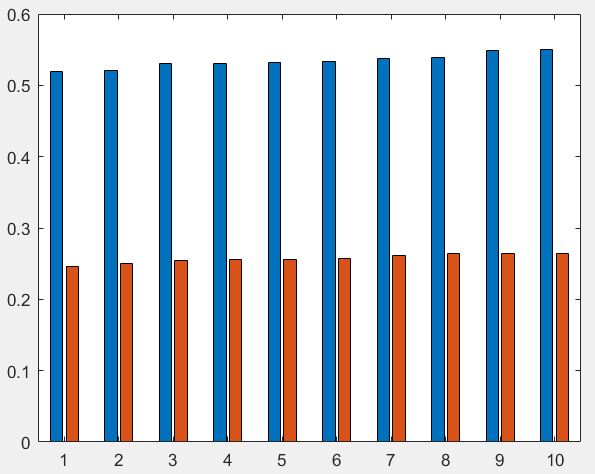
#### 实验结果











#### 分析

三线程不加锁需要531毫秒，Cache优化后需要257毫秒，需要的时间大大缩短。三线程加锁需要10931毫秒，Cache优化后需要8535毫秒，提升不明显因为主要是锁消耗时间。

## CPU亲和力对并行程序影响

### 实验设计原理

CPU 亲和力可分为两大类：软亲和力和硬亲和力。

Linux 内核进程调度器天生就具有被称为 CPU 软亲和力（affinity） 的特性，这意味着进程通常不会在处理器之间频繁迁移。这种状态正是我们希望的，因为进程迁移的频率小就意味着产生的负载小。但不代表不会进行小范围的迁移。

CPU 硬亲和力是指进程固定在某个处理器上运行，而不是在不同的处理器之间进行频繁的迁移。这样不仅改善了程序的性能，还提高了程序的可靠性。

在某种程度上硬亲和力比软亲和力具有一定的优势。但在内核开发者不断的努力下， 2.6内核软亲和力的缺陷已经比2.4的内核有了很大的改善。

（1） 步骤1.

在多核机器上，针对两线程的方案，将计算 apple 的线程绑定到一个 CPU 上，将计算 orange 的线程绑定到另外一个 CPU 上，分析对比与单线程/进程程序、2线程（无CPU 亲和）程序的运行时间。

（2） 步骤2

进一步分析不难发现，样例程序大部分时间都消耗在计算 apple 上。为此，将计算 a 和 b 的值，分布到不同的 CPU 上进行计算，同时考虑 Cache 的影响。观察程序运行时间，并与采用 Cache 的三线程方案进行对比。

### 程序代码

#### demo2\_p.c

1. #define \_GNU\_SOURCE
2. #include <pthread.h>
3. #include <stdio.h>
4. #include <sched.h>
5. #include <stdlib.h>
6. #include <string.h>
7. #include <unistd.h>
8. #define ORANGE\_MAX\_VALUE 1000000
9. #define APPLE\_MAX\_VALUE 100000000
10. #define MSECOND 1000000
11. #define NUM\_THREADS 1
12. struct apple{
13. unsigned long long a;
14. unsigned long long b;
15. };
16. struct orange{
17. int a[ORANGE\_MAX\_VALUE];
18. int b[ORANGE\_MAX\_VALUE];
19. };
20. cpu\_set\_t mask;
21. void\* p\_apple(void \*test) {
22. CPU\_ZERO(&mask);
23. CPU\_SET(1,&mask);
24. int sum;
25. for(sum = 0; sum < APPLE\_MAX\_VALUE; sum++){
26. ((struct apple \*)test)->a += sum;
27. ((struct apple \*)test)->b += sum;
28. }
29. }
30. void p\_orange(void \*test1) {
31. int sum = 0;
32. for(int index = 0; index < ORANGE\_MAX\_VALUE; index++){
33. sum += ((struct orange \*)test1)->a[index] + ((struct orange \*)test1)->b[index];
34. }
35. }
36. int main(int argc, const char\* argv[]) {
37. struct apple test;
38. struct orange test1;
39. CPU\_ZERO(&mask);
40. CPU\_SET(0,&mask);
41. // 定义线程的 id 变量，多个变量使用数组
42. pthread\_t tids[NUM\_THREADS];
43. //参数依次是：创建的线程id，线程参数，调用的函数，传入的函数参数
44. int ret0 = pthread\_create(&tids[0], NULL, p\_apple, &test);
45. if (ret0 != 0){
46. printf("pthread\_create error: error\_code=%d\n", ret0);
47. }
48. p\_orange(&test1);
49. pthread\_join(tids[0],NULL);
50. return 0;
51. }

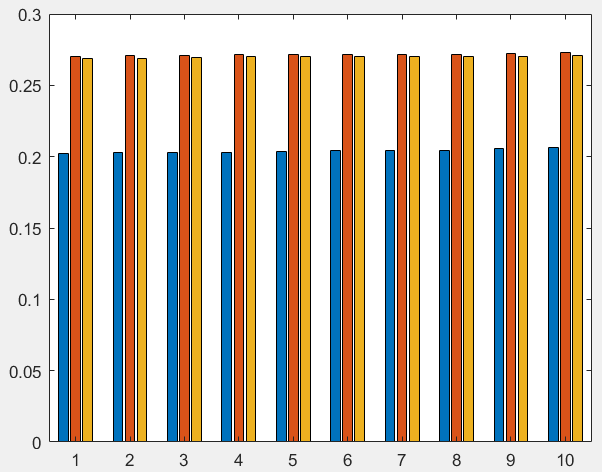
#### demo4\_p\_p.c

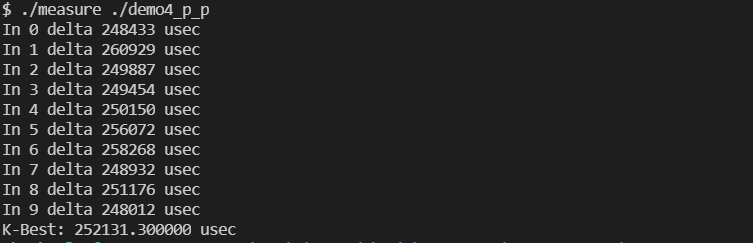
1. #define \_GNU\_SOURCE
2. #include <pthread.h>
3. #include <stdio.h>
4. #include <sched.h>
5. #include <stdlib.h>
6. #include <string.h>
7. #include <unistd.h>
8. #define ORANGE\_MAX\_VALUE 1000000
9. #define APPLE\_MAX\_VALUE 100000000
10. #define MSECOND 1000000
11. #define NUM\_THREADS 3
12. struct apple{
13. unsigned long long a;
14. char c[128]; /\*32,64,128\*/
15. unsigned long long b;
16. };
17. struct orange{
18. int a[ORANGE\_MAX\_VALUE];
19. int b[ORANGE\_MAX\_VALUE];
20. };
21. cpu\_set\_t mask;
22. void\* p\_apple\_a(void \*test) {
23. CPU\_ZERO(&mask);
24. CPU\_SET(1,&mask);
25. int sum;
26. for(sum = 0; sum < APPLE\_MAX\_VALUE; sum++){
27. ((struct apple \*)test)->a += sum;
28. }
29. }
30. void\* p\_apple\_b(void \*test) {
31. CPU\_ZERO(&mask);
32. CPU\_SET(2,&mask);
33. int sum;
34. for(sum = 0; sum < APPLE\_MAX\_VALUE; sum++){
35. ((struct apple \*)test)->b += sum;
36. }
37. }
38. void p\_orange(void \*test1) {
39. int sum = 0;
40. for(int index = 0; index < ORANGE\_MAX\_VALUE; index++){
41. sum += ((struct orange \*)test1)->a[index] + ((struct orange \*)test1)->b[index];
42. }
43. }
44. int main(int argc, const char\* argv[]) {
45. struct apple test;
46. struct orange test1;
47. CPU\_ZERO(&mask);
48. CPU\_SET(0,&mask);
49. // 定义线程的 id 变量，多个变量使用数组
50. pthread\_t tids[NUM\_THREADS];
51. //参数依次是：创建的线程id，线程参数，调用的函数，传入的函数参数
52. int ret0 = pthread\_create(&tids[0], NULL, p\_apple\_a, &test);
53. if (ret0 != 0){
54. printf("pthread\_create error: error\_code=%d\n", ret0);
55. }
56. int ret1 = pthread\_create(&tids[1], NULL, p\_apple\_b, &test);
57. if (ret1 != 0){
58. printf("pthread\_create error: error\_code=%d\n", ret1);
59. }
60. p\_orange(&test1);
61. pthread\_join(tids[0],NULL);
62. pthread\_join(tids[1],NULL);
63. return 0;
64. }

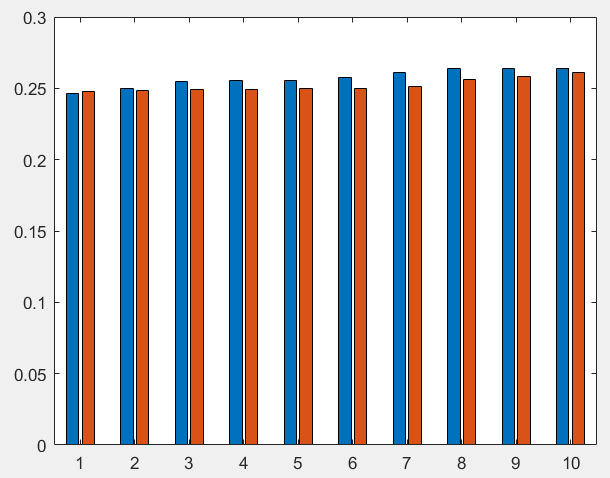
### 实验结果与分析

#### 实验结果







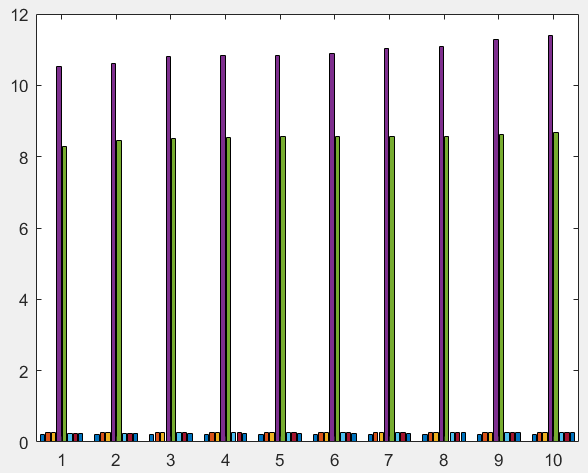


#### 分析

两线程并行需要271毫秒，使用硬亲和后需要269毫秒，单线程需要204毫秒，硬亲和能提高效率但是还是没有单进程效率高。

三线程无锁Cache优化硬亲和后需要252毫秒，无硬亲和需要257毫秒，硬亲和提高了效率。

## 八种实现方案运行时间对比总结



单线程效率最高，三线程要比两线程效率要高，加锁花费大量时间，Cache优化可以明显提高效率，使用CPU硬亲和也可以提高效率。

**第七组 内存管理**

**一、实验目的**

理解 Linux 系统逻辑地址空间、物理地址空间的概念，采用相关系统调用设计并编程实

现写拷贝 COW 和内存映射文件 mmap 实验，观察分析实验结果，加深对 Linux 系统中逻辑地址、物理地址、写拷贝和内存映射文件机制的理解。

**二、实验环境**

硬件：阿里云弹性计算服务，多核 CPU，核数不少于 2，内存不少于 8G；

软件：Linux 操作系统（内核 3.0 及以上），如 Ubuntu、CentOS、AliOS 等；

Pthread/POSIX 线程库，System V；gcc 编译； C/C++，Java

**三、实验内容**

**3.1 写拷贝 COW**

第 1 步。编写程序，采用 fork()系统调用，由父进程创建子进程，父进程、子进程共享同一全局变量，且对该变量只读、不修改。在父进程、子进程中分别打印输出共享变量的值和变量的地址，例如 printf("parent process num val=%d, addr=%x\n", num, &num)，观察父进程、子进程读到的是否为同一变量。

第 2 步。在前一步程序基础上，父进程创建子进程后，允许子进程修改与父进程共享的全局变量，触发写拷贝 COW 机制。在父进程、子进程中分别打印输出共享变量的值和变量的地址，观察父进程、子进程读到的是否为同一变量，分析解释观察结果；

第 3 步。父进程创建子进程后，父进程、子进程均进入各自 while 循环体。在循环体内部，

父进程读取共享变量的值，子进程修改共享变量后再读取变量值，之后父进程、子进程中分

别打印输出共享变量的值和变量的地址。观察父进程、子进程的打印输出结果，并从 COW

角度，分析解释观察结果。

第 4 步。在第 2 步、第 3 步程序基础上，使用 linux 中的/proc/self/pagemap 文件，编写进程逻辑/虚拟地址→物理地址转换函数 mem\_addr\_vir2phy，观察父进程、子进程共享变量的值，以及变量的逻辑地址和物理地址，并从 COW 角度，分析解释观察结果。

**3.2 内存映射文件 mmap**

编写程序，实现如下功能，分析验证 Linux 内存映射文件机制。

（1）使用 open()系统调用/函数，创建一个文本文件；使用 write()系统调用/函数，向该文件中写入一定数量的字符。

（2）使用内存映射函数 mmap()，将上述文件映射至进程的虚拟/逻辑地址空间，返回被映射区的指针。

（3）创建内存缓冲区 mapbuffer，使用函数 mmset()将缓冲区清零。

（4）指定读写偏移 offset，使用函数 memcpy，将文件中从 offset 开始的一定长度 L 的文件内容，读入缓冲区 mapbuffer，并统计文件读取时间 Tm。

（5）用 putc()输出 mapbuffer 中的内容，用 munmap()删除/关闭前面建立的文件对象映射。

（6）使用 read()函数，从偏移 offset 处开始，将长度 L 的文件内容读入缓冲区 mapbuffer，修改后（如将原值加 1）再写回文件，并统计文件读取时间 Tf。

（7）对比分析两种文件读取模式下的文件读写时间 Tm 和 Tf 的差异。

**实验原理**

**4.1 写拷贝 COW 使用的系统调用和函数**

**4.1.1 fork()**

功能：父进程使用该系统调用，全部复制、创建出与父进程几乎完全一样的子进程

返回值：执行成功时，该函数被调用一次，但返回两次：对创建出的子进程返回 0，对 父进程返回非零的子进程 pid；执行不超过，对父进程返回-1，对子进程无返回值。

**4.1.2 mem\_addr\_vir2phy**

在前 3 步实验中，父进程、子进程打印的共享变量地址是变量在各自逻辑地址空间中的

逻辑/虚拟地址地址，例如 printf("parent process num val=%d, addr=%x\n", num, &num)。利用

Linux 的 pagemap 文件，可以查看进程逻辑地址空间中的各个虚拟页（page）所映射到的物理页（page frame），包括物理页的基址。因此，将变量 num 在进程逻辑地址空间中的逻辑/虚拟地址加上该变量所在的物理页的起始地址（即基址），可以得到该变量的物理地址。

函数 mem\_addr\_vir2phy 如下：

1. **int** mem\_addr\_vir2phy(unsigned **long** vir, unsigned **long** \*phy) {
2. **int** fd;
3. **int** page\_size = getpagesize();
4. unsigned **long** vir\_page\_idx = vir / page\_size;
5. unsigned **long** pfn\_item\_offset = vir\_page\_idx \* **sizeof**(uint64\_t);
6. uint64\_t pfn\_item;
8. fd = open(page\_map\_file, O\_RDONLY);
9. **if** (fd < 0) {
10. printf("open %s failed", page\_map\_file);
11. **return** -1;
12. }
14. **if** ((off\_t)-1 == lseek(fd, pfn\_item\_offset, SEEK\_SET)) {
15. printf("lseek %s failed", page\_map\_file);
16. **return** -1;
17. }
19. **if** (**sizeof**(uint64\_t) != read(fd, &pfn\_item, **sizeof**(uint64\_t))) {
20. printf("read %s failed", page\_map\_file);
21. **return** -1;
22. }
24. **if** (0 == (pfn\_item & PFN\_PRESENT\_FLAG)) {
25. printf("page is not present");
26. **return** -1;
27. }
29. \*phy = (pfn\_item & PFN\_MASK) \* page\_size + vir % page\_size;
30. **return** 0;
31. }

**4.2 内存映射文件 mmap 使用的系统调用和函数**

**4.2.1 建立内存映射函数 mmap(start, length, prot, flags, fd, offset)**

头文件 <sys/mman.h>

函数原型void\* mmap(void\* start, size\_t length, int prot, int flags, int fd, off\_t offset);

参数如下：

（1）start：映射区开始地址，即要映射到的内存区域的起始地址。

一般设置为 0/null，表示由 Linux 内核决定映射区起始地址。

（2）length：要映射的内存区域长度，以字节为单位，不足一内存页（page）按一内存页处理。

（3）prot：期望的内存保护标志，不能与文件的打开模式冲突。prot 取值如下，可以通过 or 运算组合在一起：

PROT\_EXEC //页内容可以被执行

PROT\_READ //页内容可以被读取

PROT\_WRITE //页可以被写入

PROT\_NONE //页不可访问

（4）flags：指定映射对象的类型，决定映射选项和映射页是否可以共享。flags 取值如下，可以是一个或者多个以下值的组合体：

MAP\_FIXED：使用指定的映射起始地址，如果由 start 和 length 参数指定的内存区重叠于现存的映射空间，重叠部分将会被丢弃。如果指定的起始地址不可用，操作将会失败。并且起始地址必须落在页的边界上。

MAP\_SHARED：对映射区域的写入数据会复制回文件内, 而且允许其他映射该文件的进程共享。

MAP\_PRIVATE：建立一个写入时拷贝的私有映射。内存区域的写入不会影响到原文件。这个标志和以上标志是互斥的，只能使用其中一个。

MAP\_DENYWRITE：这个标志被忽略。

MAP\_EXECUTABLE：同上

MAP\_NORESERVE：不要为这个映射保留交换（swap）空间。当交换空间被保留，对映射区修改的可能会得到保证。当交换空间不被保留，同时内存不足，对映射区的修改会引起段违例信号。

MAP\_LOCKED：锁定映射区的页面，从而防止页面被交换出内存。

MAP\_GROWSDOWN：用于堆栈，告诉内核虚拟内存 VM 系统，映射区可以向下扩展。

MAP\_ANONYMOUS：匿名映射，映射区不与任何文件关联。

MAP\_ANON：MAP\_ANONYMOUS 的别称，不再被使用。

MAP\_FILE：兼容标志，被忽略。

MAP\_32BIT：将映射区放在进程地址空间的低 2GB，MAP\_FIXED 指定时会被忽略。当前这个标志只在 x86-64 平台上得到支持。

MAP\_POPULATE：为文件映射通过预读的方式准备好页表。随后对映射区的访问不会被页违例阻塞。

MAP\_NONBLOCK：仅和 MAP\_POPULATE 一起使用时才有意义。不执行预读，只为已存在于内存中的页面建立页表入口。

（5）fd：被映射的文件描述符，由 open()函数返回。

（6）offset：被映射的文件对象的内容起点，即从文件中何处开始进行映射。一般取值为 0，即从文件开始处进行映射；当该值大小非零时，为 PAGE\_SIZE 的整数倍。

mmap()返回说明：

（1）成功执行时，mmap()返回被映射区的指针，munmap()返回 0。

（2）执行失败时，返回 MAP\_FAILED[其值为(void \*)-1]，错误码 errno 被设为以下的某个

值：

EACCES：访问出错。

EAGAIN：文件已被锁定，或者太多的内存已被锁定。

EBADF：fd 不是有效的文件描述词

EINVAL：一个或者多个参数无效

ENFILE：已达到系统对打开文件的限制

ENODEV：指定文件所在的文件系统不支持内存映射

ENOMEM：内存不足，或者进程已超出最大内存映射数量

EPERM：权能不足，操作不允许

ETXTBSY：已写的方式打开文件，同时指定 MAP\_DENYWRITE 标志

SIGSEGV：试着向只读区写入

SIGBUS：试着访问不属于进程的内存区

注意：mmap 必须以 PAGE\_SIZE 为单位进行映射，而内存也只能以页为单位进行映射，若要映射非 PAGE\_SIZE 整数倍的地址范围，要先进行内存对齐，强行以 PAGE\_SIZE 的倍数大小进行映射。

**4.2.2 删除/关闭内存映射 munmap(void \*start, size\_t length)**

头文件 <sys/mman.h>

函数原型int munmap(void\* start, size\_t length);

参数如下：

（1）start：映射区开始地址，即要映射到的内存区域的起始地址。一般设置为 0/null，表示由 Linux 内核决定映射区起始地址。

（2）length：要映射的内存区域长度，以字节为单位，不足一内存页（page）按一内存页处理。

munmap()返回说明：

（1）成功执行，返回 0。

（2）执行失败，返回-1。

**实验过程**

**写拷贝COW**

**mem\_addr\_vir2phy()函数编写**

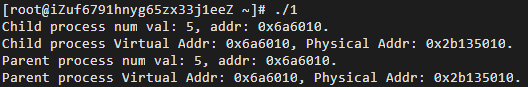
1. //计算虚拟地址对应的地址，传入虚拟地址vaddr，通过paddr传出物理地址
2. **int** mem\_addr\_vir2phy(unsigned **long** vir, unsigned **long** \*phy) {
3. **int** fd;
4. // 获取系统设定的页面大小
5. **int** page\_size = getpagesize();
6. //计算此虚拟地址相对于0x0的经过的页面数
7. unsigned **long** vir\_page\_idx = vir / page\_size;
8. //计算在/proc/pid/page\_map文件中的偏移量
9. unsigned **long** pfn\_item\_offset = vir\_page\_idx \* **sizeof**(uint64\_t);
10. //计算虚拟地址在页面中的偏移量
11. unsigned **long** page\_offset = vir % page\_size;
12. uint64\_t pfn\_item;
14. //以只读方式打开
15. fd = open(page\_map\_file, O\_RDONLY);
16. **if** (fd < 0) {
17. printf("open %s failed", page\_map\_file);
18. **return** -1;
19. }
20. //将游标移动到相应位置，即对应项的起始地址且判断是否移动失败
21. **if** ((off\_t)-1 == lseek(fd, pfn\_item\_offset, SEEK\_SET)) {
22. printf("lseek %s failed", page\_map\_file);
23. **return** -1;
24. }
25. //读取对应项的值，并存入item中，且判断读取数据位数是否正确
26. **if** (**sizeof**(uint64\_t) != read(fd, &pfn\_item, **sizeof**(uint64\_t))) {
27. printf("read %s failed", page\_map\_file);
28. **return** -1;
29. }
30. //判断present是否为0
31. **if** (0 == (pfn\_item & PFN\_PRESENT\_FLAG)) {
32. printf("page is not present");
33. **return** -1;
34. }
35. //计算物理页号，即取item的bit0-54
36. uint64\_t phy\_pageIndex = pfn\_item & PFN\_MASK;
37. //加上页内偏移量得到了物理地址
38. \*phy = phy\_pageIndex \* page\_size + page\_offset;
39. **return** 0;
40. }

**步骤一**

代码：

1. #include <fcntl.h>
2. #include <stdint.h>
3. #include <stdio.h>
4. #include <stdlib.h>
5. #include <sys/stat.h>
6. #include <sys/types.h>
7. #include <sys/wait.h>
8. #include <unistd.h>
9. #define page\_map\_file "/proc/self/pagemap"
10. #define PFN\_MASK ((((uint64\_t)1) << 55) - 1)
11. #define PFN\_PRESENT\_FLAG (((uint64\_t)1) << 63)
13. **int** mem\_addr\_vir2phy(unsigned **long** vir, unsigned **long** \*phy) {
14. **int** fd;
15. **int** page\_size = getpagesize();
16. unsigned **long** vir\_page\_idx = vir / page\_size;
17. unsigned **long** pfn\_item\_offset = vir\_page\_idx \* **sizeof**(uint64\_t);
18. unsigned **long** page\_offset = vir % page\_size;
19. uint64\_t pfn\_item;
21. fd = open(page\_map\_file, O\_RDONLY);
22. **if** (fd < 0) {
23. printf("open %s failed", page\_map\_file);
24. **return** -1;
25. }
26. **if** ((off\_t)-1 == lseek(fd, pfn\_item\_offset, SEEK\_SET)) {
27. printf("lseek %s failed", page\_map\_file);
28. **return** -1;
29. }
30. **if** (**sizeof**(uint64\_t) != read(fd, &pfn\_item, **sizeof**(uint64\_t))) {
31. printf("read %s failed", page\_map\_file);
32. **return** -1;
33. }
34. **if** (0 == (pfn\_item & PFN\_PRESENT\_FLAG)) {
35. printf("page is not present");
36. **return** -1;
37. }
38. uint64\_t phy\_pageIndex = pfn\_item & PFN\_MASK;
39. \*phy = phy\_pageIndex \* page\_size + page\_offset;
40. **return** 0;
41. }
43. **int** main() {
44. pid\_t pid;
45. **int** \*num = (**int** \*)malloc(**sizeof**(**int**));
46. \*num = 5;
47. **int** status;
48. //   pid = fork();
49. unsigned **long** phy;
50. **if** (fork()) {
51. wait(&status);
52. printf("Parent process num val: %d, addr: 0x%x.\n", \*num, num);
53. **if** (!mem\_addr\_vir2phy(num, &phy)) {
54. printf("Parent process Virtual Addr: 0x%x, Physical Addr: 0x%x.\n", num,
55. phy);
56. };
57. } **else** {
58. printf("Child process num val: %d, addr: 0x%x.\n", \*num, num);
59. **if** (!mem\_addr\_vir2phy(num, &phy)) {
60. printf("Child process Virtual Addr: 0x%x, Physical Addr: 0x%x.\n", num,
61. phy);
62. };
63. exit(0);
64. }
65. **return** 0;
66. }

结果分析：



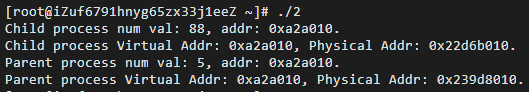
父进程在创建子进程后，变量并未发生修改的情况下，父进程与子进程在访问相同全局变量时有着相同的值，逻辑地址和物理地址。

**步骤二**

代码：

1. #include <fcntl.h>
2. #include <stdint.h>
3. #include <stdio.h>
4. #include <stdlib.h>
5. #include <sys/stat.h>
6. #include <sys/types.h>
7. #include <sys/wait.h>
8. #include <unistd.h>
9. #define page\_map\_file "/proc/self/pagemap"
10. #define PFN\_MASK ((((uint64\_t)1) << 55) - 1)
11. #define PFN\_PRESENT\_FLAG (((uint64\_t)1) << 63)
13. **int** mem\_addr\_vir2phy(unsigned **long** vir, unsigned **long** \*phy) {
14. **int** fd;
15. **int** page\_size = getpagesize();
16. unsigned **long** vir\_page\_idx = vir / page\_size;
17. unsigned **long** pfn\_item\_offset = vir\_page\_idx \* **sizeof**(uint64\_t);
18. uint64\_t pfn\_item;
20. fd = open(page\_map\_file, O\_RDONLY);
21. **if** (fd < 0) {
22. printf("open %s failed", page\_map\_file);
23. **return** -1;
24. }
25. **if** ((off\_t)-1 == lseek(fd, pfn\_item\_offset, SEEK\_SET)) {
26. printf("lseek %s failed", page\_map\_file);
27. **return** -1;
28. }
29. **if** (**sizeof**(uint64\_t) != read(fd, &pfn\_item, **sizeof**(uint64\_t))) {
30. printf("read %s failed", page\_map\_file);
31. **return** -1;
32. }
33. **if** (0 == (pfn\_item & PFN\_PRESENT\_FLAG)) {
34. printf("page is not present");
35. **return** -1;
36. }
38. \*phy = (pfn\_item & PFN\_MASK) \* page\_size + vir % page\_size;
39. **return** 0;
40. }
41. **int** main() {
42. pid\_t pid;
43. **int** \*num = (**int** \*)malloc(**sizeof**(**int**));  //在用户进程堆上为变量 num 分配空间
44. \*num = 5;
45. **int** status;
46. unsigned **long** phy;
47. **if** (fork()) {
48. wait(&status);
49. printf("Parent process num val: %d, addr: 0x%x.\n", \*num, num);
50. **if** (!mem\_addr\_vir2phy(num, &phy)) {
51. printf("Parent process Virtual Addr: 0x%x, Physical Addr: 0x%x.\n", num, phy);
52. };
53. } **else** {
54. \*num = 88;
55. printf("Child process num val: %d, addr: 0x%x.\n", \*num, num);
56. **if** (!mem\_addr\_vir2phy(num, &phy)) {
57. printf("Child process Virtual Addr: 0x%x, Physical Addr: 0x%x.\n", num, phy);
58. };
59. exit(0);
60. }
61. **return** 0;
62. }

结果分析：



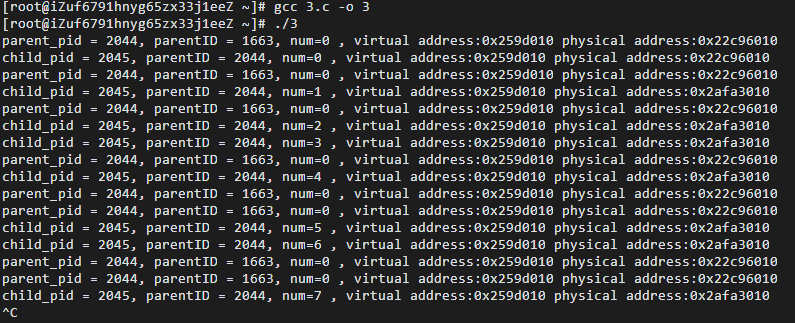
父进程在创建子进程后，由于子进程对num值进行了修改，触发写拷贝COW，进程和父进程不再共享同一物理地址中的变量 num。此时父进程和子进程各自读写不同的num 变量，这些变量的逻辑地址虽然相同，但值和物理地址已经不同了。

**步骤三**

代码：

1. #include <fcntl.h>
2. #include <stdint.h>
3. #include <stdio.h>
4. #include <stdlib.h>
5. #include <sys/stat.h>
6. #include <sys/types.h>
7. #include <sys/wait.h>
8. #include <unistd.h>
9. #define page\_map\_file "/proc/self/pagemap"
10. #define PFN\_MASK ((((uint64\_t)1) << 55) - 1)
11. #define PFN\_PRESENT\_FLAG (((uint64\_t)1) << 63)
13. **int** mem\_addr\_vir2phy(unsigned **long** vir, unsigned **long** \*phy) {
14. **int** fd;
15. **int** page\_size = getpagesize();
16. unsigned **long** vir\_page\_idx = vir / page\_size;
17. unsigned **long** pfn\_item\_offset = vir\_page\_idx \* **sizeof**(uint64\_t);
18. uint64\_t pfn\_item;
20. fd = open(page\_map\_file, O\_RDONLY);
21. **if** (fd < 0) {
22. printf("open %s failed", page\_map\_file);
23. **return** -1;
24. }
25. **if** ((off\_t)-1 == lseek(fd, pfn\_item\_offset, SEEK\_SET)) {
26. printf("lseek %s failed", page\_map\_file);
27. **return** -1;
28. }
29. **if** (**sizeof**(uint64\_t) != read(fd, &pfn\_item, **sizeof**(uint64\_t))) {
30. printf("read %s failed", page\_map\_file);
31. **return** -1;
32. }
33. **if** (0 == (pfn\_item & PFN\_PRESENT\_FLAG)) {
34. printf("page is not present");
35. **return** -1;
36. }
38. \*phy = (pfn\_item & PFN\_MASK) \* page\_size + vir % page\_size;
39. **return** 0;
40. }
41. **int** main() {
42. **int** \*num = (**int** \*)malloc(**sizeof**(**int**));
43. \*num = 0;
44. unsigned **long** phy;
45. **if** (fork() > 0) {  // parent
46. **while** (1) {
47. printf("parent\_pid = %d, parentID = %d, num=%d , ", getpid(), getppid(),
48. \*num);
49. **if** (mem\_addr\_vir2phy((unsigned **long**)num, &phy) == 0) {
50. printf("virtual address:0x%x physical address:0x%x\n", num, phy);
51. }
52. sleep(1);
53. }
54. } **else** {  // child
55. sleep(0.5); // parent first
56. **while** (1) {
57. printf("child\_pid = %d, parentID = %d, num=%d , ", getpid(), getppid(),
58. \*num);
59. **if** (mem\_addr\_vir2phy((unsigned **long**)num, &phy) == 0) {
60. printf("virtual address:0x%x physical address:0x%x\n", num, phy);
61. }
62. (\*num)++;
63. sleep(1);
64. }
65. }
66. printf("This is an end\n");
67. **return** 0;
68. }

结果分析：



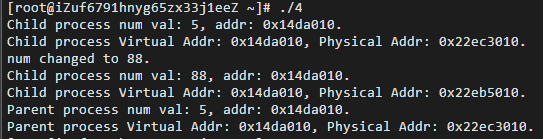
父进程在创建子进程后，子进程和父进程在 num=0 未发生修改时，有着相同的逻辑地址和物理地址，而随着子进程对 num值的修改，子进程 num值随着循环累加，而父进程 count 值并未改变，而且此时子进程与父进程访问的 num 变量的逻辑地址相同，但物理地址已经不相同了。

**步骤四**

代码：

1. #include <fcntl.h>
2. #include <stdint.h>
3. #include <stdio.h>
4. #include <stdlib.h>
5. #include <sys/stat.h>
6. #include <sys/types.h>
7. #include <sys/wait.h>
8. #include <unistd.h>
9. #define page\_map\_file "/proc/self/pagemap"
10. #define PFN\_MASK ((((uint64\_t)1) << 55) - 1)
11. #define PFN\_PRESENT\_FLAG (((uint64\_t)1) << 63)
13. **int** mem\_addr\_vir2phy(unsigned **long** vir, unsigned **long** \*phy) {
14. **int** fd;
15. **int** page\_size = getpagesize();
16. unsigned **long** vir\_page\_idx = vir / page\_size;
17. unsigned **long** pfn\_item\_offset = vir\_page\_idx \* **sizeof**(uint64\_t);
18. uint64\_t pfn\_item;
20. fd = open(page\_map\_file, O\_RDONLY);
21. **if** (fd < 0) {
22. printf("open %s failed", page\_map\_file);
23. **return** -1;
24. }
25. **if** ((off\_t)-1 == lseek(fd, pfn\_item\_offset, SEEK\_SET)) {
26. printf("lseek %s failed", page\_map\_file);
27. **return** -1;
28. }
29. **if** (**sizeof**(uint64\_t) != read(fd, &pfn\_item, **sizeof**(uint64\_t))) {
30. printf("read %s failed", page\_map\_file);
31. **return** -1;
32. }
33. **if** (0 == (pfn\_item & PFN\_PRESENT\_FLAG)) {
34. printf("page is not present");
35. **return** -1;
36. }
38. \*phy = (pfn\_item & PFN\_MASK) \* page\_size + vir % page\_size;
39. **return** 0;
40. }
41. **int** main() {
42. pid\_t pid;
43. **int** \*num = (**int** \*)malloc(**sizeof**(**int**));  //在用户进程堆上为变量 num 分配空间
44. \*num = 5;
45. **int** status;
46. unsigned **long** phy;
47. **if** (fork()) {
48. wait(&status);
49. printf("Parent process num val: %d, addr: 0x%x.\n", \*num, num);
50. **if** (!mem\_addr\_vir2phy(num, &phy)) {
51. printf("Parent process Virtual Addr: 0x%x, Physical Addr: 0x%x.\n", num,
52. phy);
53. };
54. } **else** {
55. printf("Child process num val: %d, addr: 0x%x.\n", \*num, num);
56. **if** (!mem\_addr\_vir2phy(num, &phy)) {
57. printf("Child process Virtual Addr: 0x%x, Physical Addr: 0x%x.\n", num,
58. phy);
59. };
60. \*num = 88;
61. printf("num changed to 88.\n");
62. printf("Child process num val: %d, addr: 0x%x.\n", \*num, num);
63. **if** (!mem\_addr\_vir2phy(num, &phy)) {
64. printf("Child process Virtual Addr: 0x%x, Physical Addr: 0x%x.\n", num,
65. phy);
66. };
67. exit(0);
68. }
69. **return** 0;
70. }

结果分析：



在 num 值改变前，子进程和父进程具有相同的逻辑地址和物理地址，而在num 值改变后，父子进程逻辑地址仍然相同，而子进程物理地址已经改变。

**内存映射文件 mmap**

**内存映射**

代码：

1. #include <fcntl.h>
2. #include <stdio.h>
3. #include <string.h>
4. #include <sys/mman.h>
5. #include <sys/stat.h>
6. #include <sys/types.h>
7. #include <unistd.h>
8. **int** main() {
9. // Step1: 写入字符
10. **int** fd = open("buptscs.txt", O\_CREAT | O\_RDWR, 0644);
11. write(fd, "1145142019211301987654", 22);
13. // Step2: 映射至进程的逻辑地址空间，被映射区指针share
14. **struct** stat sbuff;
15. stat("buptscs.txt", &sbuff);
16. **char** \*share = mmap(NULL, sbuff.st\_size, PROT\_READ, MAP\_PRIVATE, fd, 0);
17. **if** (share == MAP\_FAILED) {
18. perror("map error");
19. }
20. close(fd);
22. // Step3: 读取至缓冲区mapbuffer中
23. **char** mapbuffer[10];
24. **void** \*offset = share + 6;
25. memset(mapbuffer, 0, **sizeof**(mapbuffer));
26. memcpy(mapbuffer, offset, 10);
28. // Step4: putc()输出
29. **for** (**int** i = 0; i < 10; i++) {
30. putc(mapbuffer[i], stdout);
31. }
32. munmap(share, sbuff.st\_size);
33. }

结果：



**读写时间对比**

代码：

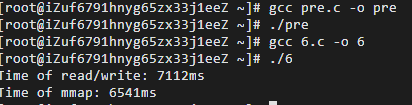
**pre6.c**

1. #include <fcntl.h>
2. #include <stdio.h>
3. #include <stdlib.h>
4. #include <string.h>
5. #include <sys/mman.h>
6. #include <sys/stat.h>
7. #include <sys/time.h>
8. #include <sys/types.h>
9. #include <unistd.h>
10. #define MAX 1000000
11. **int** main() {
12. **int** i = 0;
13. **int** fd = 0, offset = 10;
14. **struct** timeval tv1, tv2;
15. **int** \*array = (**int** \*)malloc(**sizeof**(**int**) \* MAX);
17. fd = open("time\_test.txt", O\_CREAT | O\_RDWR, 0644);
18. **for** (i = 0; i < offset; i++) {
19. array[i] = 0;
20. }
21. **for** (i = offset; i < MAX; i++) array[i] = 1;
22. write(fd, array, **sizeof**(**int**) \* MAX);
23. memset(array, 0, **sizeof**(**int**) \* MAX);
24. close(fd);
25. }

**6.c**

1. #include <fcntl.h>
2. #include <stdio.h>
3. #include <stdlib.h>
4. #include <string.h>
5. #include <sys/mman.h>
6. #include <sys/stat.h>
7. #include <sys/time.h>
8. #include <sys/types.h>
9. #include <unistd.h>
10. #define MAX 1000000
11. **int** main() {
12. **int** i = 0;
13. **int** fd = 0, offset = 10;
14. **struct** timeval tv1, tv2;
15. **int** \*array = (**int** \*)malloc(**sizeof**(**int**) \* MAX);
17. /\*read\*/
18. **int** size = MAX - offset;
19. gettimeofday(&tv1, NULL);
20. fd = open("time\_test.txt", O\_RDWR);
21. lseek(fd, offset \* **sizeof**(**int**), SEEK\_SET);
22. **if** (**sizeof**(**int**) \* size != read(fd, (**void** \*)array, **sizeof**(**int**) \* size)) {
23. printf("Reading data failed...\n");
24. **return** -1;
25. }
26. **for** (i = 0; i < size; ++i) ++array[i];
27. lseek(fd, offset \* **sizeof**(**int**), SEEK\_SET);
28. **if** (**sizeof**(**int**) \* size != write(fd, (**void** \*)array, **sizeof**(**int**) \* size)) {
29. printf("Writing data failed...\n");
30. **return** -1;
31. }
32. close(fd);
33. gettimeofday(&tv2, NULL);
34. printf("Time of read/write: %ldms\n", tv2.tv\_usec - tv1.tv\_usec);
36. memset(array, 0, **sizeof**(**int**) \* MAX);
37. free(array);
39. /\*mmap\*/
40. gettimeofday(&tv1, NULL);
41. fd = open("time\_test.txt", O\_RDWR);
42. array =
43. mmap(NULL, **sizeof**(**int**) \* MAX, PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd, 0);
44. **for** (i = offset; i < MAX; ++i) ++array[i];
45. // 关闭映射
46. munmap(array, **sizeof**(**int**) \* MAX);
47. // 将从ADDR开始并延长LEN字节的区域与它所映射的文件同步。
48. msync(array + offset \* **sizeof**(**int**), **sizeof**(**int**) \* MAX, MS\_SYNC);
49. close(fd);
50. gettimeofday(&tv2, NULL);
51. printf("Time of mmap: %ldms\n", tv2.tv\_usec - tv1.tv\_usec);
52. **return** 0;
53. }

结果分析：

  
通过 read 和 mmap 两种方法访问硬盘的文件“time\_test.txt”，该文件存有 1000000 个整数。程序分别使用两种方法读出这些整数，加 1 后再写回文件中。通过对比能够看到，read 消耗的时间长于 mmap 方式需要的时间。

**实验总结**

COW（CopyOnWrite，写时复制）机制是一种无锁的快速数据访问机制，针对随时变化的数据进行备份。当父子进程对数据只读时，共享数据；而其他进程需要修改时，数据的逻辑地址不变，物理地址发生变化，即复制一份再进行修改。父子进程访问相同名称的数据时，尽管逻辑地址相同，但对应的物理地址不同，将会访问到独立的不同值。

mmap将一个文件或者其它对象映射进内存,在用户空间映射调用系统中作用很大。mmap操作提供了一种机制，让用户程序直接访问设备内存，这种机制，相比较在用户空间和内核空间互相拷贝数据，效率更高。

**碰到的问题及解决**

**mmap中的offset**

由于对mmap的映射机制理解不到位，尝试如下写法：

array = mmap(NULL, sizeof(int) \* MAX, PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd, offset \* sizeof(int));

其中offset是需要跳过的文件开头的整数个数。

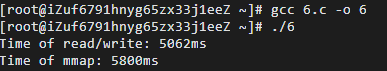
错误原因在于， mmap()必须以PAGE\_SIZE为单位进行映射，而内存也只能以页为单位进行映射，若要映射非PAGE\_SIZE整数倍的地址范围，要先进行内存对齐，强行以PAGE\_SIZE的倍数大小进行映射。因此，mmap()中的offset参数含义是被映射对象文件的页起点。

解决：在没有分析整数位于第几页时，只能传入offset=0，得到所有的数据。之后再通过下标寻找。

array = mmap(NULL, sizeof(int) \* MAX, PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd, 0);

**mmap和read执行时间相近**

一开始将数据准备程序pre6.c和性能比较程序6.c写在一起，导致的read的计时器和mmap的计时器结果相似，甚至常常出现mmap用时大于read的情况。



源程序代码：

1. #include <fcntl.h>
2. #include <stdio.h>
3. #include <stdlib.h>
4. #include <string.h>
5. #include <sys/mman.h>
6. #include <sys/stat.h>
7. #include <sys/time.h>
8. #include <sys/types.h>
9. #include <unistd.h>
10. #define MAX 1000000
11. **int** main() {
12. **int** i = 0;
13. **int** fd = 0, offset = 10;
14. **struct** timeval tv1, tv2;
15. **int** \*array = (**int** \*)malloc(**sizeof**(**int**) \* MAX);
17. // prepare
18. fd = open("time\_test.txt", O\_CREAT | O\_RDWR, 0644);
19. **for** (i = 0; i < offset; i++) {
20. array[i] = 0;
21. }
22. **for** (i = offset; i < MAX; i++) array[i] = 1;
23. write(fd, array, **sizeof**(**int**) \* MAX);
24. memset(array, 0, **sizeof**(**int**) \* MAX);
25. close(fd);
27. /\*read\*/
28. **int** size = MAX - offset;
29. gettimeofday(&tv1, NULL);
30. fd = open("time\_test.txt", O\_RDWR);
31. lseek(fd, offset \* **sizeof**(**int**), SEEK\_SET);
32. **if** (**sizeof**(**int**) \* size != read(fd, (**void** \*)array, **sizeof**(**int**) \* size)) {
33. printf("Reading data failed...\n");
34. **return** -1;
35. }
36. **for** (i = 0; i < size; ++i) ++array[i];
37. lseek(fd, offset \* **sizeof**(**int**), SEEK\_SET);
38. **if** (**sizeof**(**int**) \* size != write(fd, (**void** \*)array, **sizeof**(**int**) \* size)) {
39. printf("Writing data failed...\n");
40. **return** -1;
41. }
42. close(fd);
43. gettimeofday(&tv2, NULL);
44. printf("Time of read/write: %ldms\n", tv2.tv\_usec - tv1.tv\_usec);
46. memset(array, 0, **sizeof**(**int**) \* MAX);
47. free(array);
49. /\*mmap\*/
50. gettimeofday(&tv1, NULL);
51. fd = open("time\_test.txt", O\_RDWR);
52. array =
53. mmap(NULL, **sizeof**(**int**) \* MAX, PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd, 0);
54. **for** (i = offset; i < MAX; ++i) ++array[i];
55. // 关闭映射
56. munmap(array, **sizeof**(**int**) \* MAX);
57. // 将从ADDR开始并延长LEN字节的区域与它所映射的文件同步。
58. msync(array+offset\***sizeof**(**int**) , **sizeof**(**int**) \* MAX, MS\_SYNC);
59. close(fd);
60. gettimeofday(&tv2, NULL);
61. printf("Time of mmap: %ldms\n", tv2.tv\_usec - tv1.tv\_usec);
62. **return** 0;
63. }

解决：在比较思考后，将prepare的代码单独执行，mmap和read的时间出现明显变化。实验者认为可能是读取一次time\_test.txt后，该文件被系统缓存，并在下次read时快速被读入缓存区，导致时间明显变快。