

北 京 邮 电 大 学
计 算 机 学 院

《操作系统》课程实验指导书

2022 年 10 月

目 录

1. 实验目的.....	4
2. 实验环境.....	5
3. 实验内容.....	6
3.1 进程创建与管理.....	6
3.2 线程创建管理与通信.....	6
3.3 进程通信.....	7
3.4 进程/线程同步互斥.....	10
3.5 多核多线程编程及性能分析.....	15
4. 实验步骤.....	16
5. 实验要求.....	20
6. 实验指导.....	21
6.1 阿里云注册登录-实验步骤说明.....	21
6.1.1 步骤一 注册&登录.....	21
6.1.2 步骤二 进入课程实验.....	22
6.1.3 步骤三 申请实验环境.....	23
6.1.4 步骤四 进行实验.....	24
6.1.5 步骤五 结束实验.....	25
6.2 进程创建管理程序示例.....	26
6.2.1 进程管理系统调用/API 函数.....	26
6.2.2 父子进程创建与控制.....	36
6.3 线程创建管理程序示例.....	38
6.3.1 Pthread 线程库 API.....	38
6.3.2 示例 2-2-1.....	43
6.3.3 示例 2-2-2.....	45
6.3.4 示例 2-2-3.....	46
6.3.5 示例 2-2-5.....	48
6.3.6 示例 2-2-6.....	50
6.3.7 Linux-c/c++线程间参数传递.....	52
6.4 进程通信示例.....	59
6.4.1 消息队列.....	60
6.4.2 共享内存通信.....	65
6.4.3 管道通信.....	70
6.4.4 信号 signal 通信.....	77
6.5 Linux 线程通信示例.....	87
6.6 进程同步互斥示例.....	88
6.6.1 System V 信号量操作原语.....	88
6.6.2 基于 System V 信号量的生产者-消费者问题 1.....	90
6.6.3 基于信号量的生产者-消费者问题 2.....	95
6.7 线程同步与互斥示例.....	100
6.7.1 Pthread 线程同步互斥机制及 API.....	100
6.7.2 基于线程的生产者-消费者问题.....	102
6.7.3 线程间同步互斥.....	106

6.7.4	抢票程序.....	108
6.7.5	PThread 多线程打印	112
6.8	基于 C++ 中互斥锁和条件变量的同步互斥.....	114
6.8.1	互斥锁/互斥量 mutex 及程序示例.....	114
6.8.2	条件变量.....	121
6.8.3	基于 condition_variable 和 mutex 的信号量实现	123
6.8.4	基于条件变量的生产者-消费者问题.....	124
6.9	多核多线程编程及性能分析	129
6.9.1	实验 6.1. 观察实验平台物理 cpu、CPU 核和逻辑 cpu 的数目	129
6.9.2	实验 6.2 单线程/进程串行 vs2 线程并行 vs3 线程加锁并行程序	130
6.9.3	实验 6.3 线程加锁 vs 3 线程不加锁 对比	133
6.9.4	实验 6.4. 针对 Cache 的优化	133
6.9.5	实验 6.5 CPU 亲和力对并行程序影响	134
6.9.6	8 种实现方案运行时间对比总结.....	135
6.9.7	K-Best 测量方法	136

1. 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。
5. 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 Linux 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。
3. 掌握 `ps`、`top`、`ps tree -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 $m_1=2$ 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 $n_1=4$ 个 A 和 $n_2=3$ 个 B。因此, 如果当前工作台空位小于 m_1 , 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 m_1 个工作台空位作为一个整体一次性地申请, worker C 所需的工作台空位中的 n_1 个 A 和 n_2 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 $P_1, P_2, P_3, P_4, P_5, P_6$, 每个进程执行时需要同时使用 $m_1=1$ page 的内存资源和 $m_2=1$ 个 I/O 设备资源。系统内存资源总量为 $N_1=3$ 个 pages, I/O 设备数目为 $N_2=3$ 。

对进程 P_i , $1 \leq i \leq 6$ 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 P_i 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 $i \in \{\text{memory}, \text{I/O_device}\}$ 所要求的 m_1 或 m_2 个资源实例; (ii) 申请资源 $j \in \{\text{memory}, \text{I/O_device}\} - \{i\}$ 所要求的 m_2 或 m_1 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 P_1, P_2, P_3 ;然后,将 3 个 I/O 设备依次分配给 P_4, P_5, P_6 。之后, P_1, P_2, P_3 中任一进程再申请 I/O 设备,或 P_4, P_5, P_6 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。

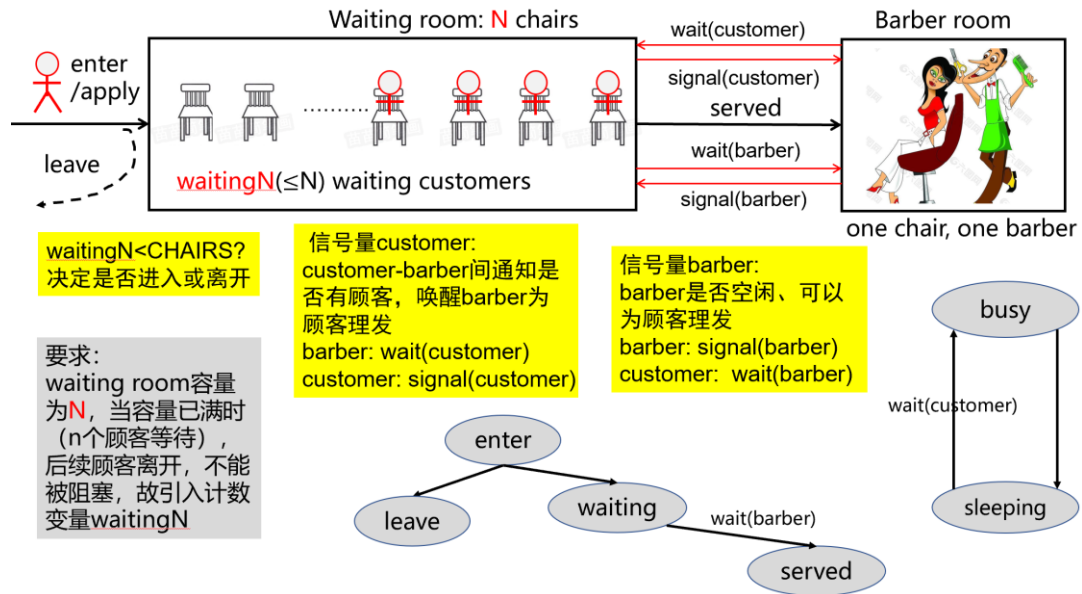
要求:

- (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 semaphore，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. 实验步骤

分步骤完成以下实验内容。

1. 注册登录阿里云弹性计算服务 ECS 实验环境【Linux 操作系统安装】

参照指导书后续内容，注册登录阿里云，以使用阿里云弹性计算服务 ECS 资源，根据部署在 ECS 上的实验内容，依次完成各项实验。

【也可以采用方式 2：不使用阿里云 ECS 资源，自己安装配置实验环境，完成实验。

选择 CentOS、Ubuntu、AliOS 等 Linux 发行版本，观察所采用的内核版本，采用硬盘分区模式安装 Linux 系统；也可以先在本机操作系统上安装 VirtualBox、VMware、Virtual PC 等虚拟机软件，在虚拟机之上安装 Linux 系统。

注意安装操作系统所需的软硬件环境和硬件配置要求。

✧ 虚拟机安装 Linux 操作系统注意事项：

采用 VirtualBox、VMware 安装操作系统时，需要配置虚拟机的 CPU 核数目，建议：虚拟机 CPU 的核数等同于实验所用计算机的物理核数。

✧ Pthread 线程库安装

观察确认所安装的 Linux 发行版本带有 Pthread 线程库，注意：

- (1) 某些版本 Linux（如 Ubuntu）默认不带 Pthread 线程库，即使在编译的时候加上 `-lpthread` 也不行，`man` 不到相关 Pthread 函数。此时，需要在 `/usr/lib/...` 下导入动态库 `libpthread.a`，具体方法可以查阅网上相关资料。
- (2) 后续编程时导入头文件：`#include <pthread.h>`

】

2. 第一组 进程的创建与管理

实验 1-1. 进程观察

查阅相关资料，阅读 Linux 内核源码，分析 Linux 进程的组成，了解进程状态，观察进程 `PCB/task_struct` 等进程管理数据结构；

实验 1-2. 进程创建与管理

参照“【实验指导】1.进程创建及管理示例”中的程序，结合所查阅的参考资料，利用

Linux 内核提供的 `fork()`、`exec()`、`wait()`、`exit()`、`kill()`等系统调用，创建管理多个进程，观察父子进程的结构和并发行为，掌握睡眠、撤销等进程控制方法；

要求：本组实验至少用到 `fork()`、`exec()`、`wait()`、`exit()`、`kill()`、`getpid` 五个系统调用。

exec()实验特别要求：

- 设计实验方案，使得父进程创建子进程后，子进程通过 `exec()`调入自身的执行代码
- 自己查阅 `exec` 系统调用相关资料和样例，完成实验

实验 1-3. 进程管理命令

了解 `ps`、`top`、`pstree -h`、`vmstat`、`strace`、`ltrace`、`sleep x`、`kill`、`jobs` 等命令的功能，使用这些命令观察进程结构和行为；

3. 第二组 线程的创建与管理

实验 2-1. Pthread 线程库背景知识

了解 POSIX 线程标准库（Pthread 线程库）相关知识，分析 Pthread 线程结构，掌握所提供的线程管理 API，如 `pthread_create`、`pthread_join`、`pthread_t`、`pthread_self`、`pthread_detach`、`pthread_exit`；

实验 2-2. 线程创建与管理

参照“【实验指导】 2. 线程创建及管理程序示例”，查阅参考资料，利用 Pthread API，创建和管理线程，观察线程的结构和并发执行行为；

要求：本组实验至少用到 `pthread_create`、`pthread_exit`、`pthread_join`、`pthread_self` 等四个 API。

4. 第三组 进程间通信

了解 Linux 提供的消息队列（消息传递）共享内存、管道/命名管道、信号(signal)/软中断四种进程间通信机制的实现原理和方法；

参照“【实验指导】 6.3 进程间通信示例”，查阅参考资料，**选择上述四种通信方式中的一种**，编程实现进程间通信，观察进程间通信过程。

5. 第四组 线程间通信

了解 Linux 中的线程概念、线程通信机制、线程间同步互斥模式，以及多线程编程方式；

参照“【实验指导】6.2.7 Linux c/c++线程间参数传递”中的【示例 2-3-3】~【示例 2-3-5】，编程实现线程间参数传递。

6. 第五组 进程/线程间同步与互斥（二选一）

- (1) 针对“3.4.1 实验题目”中的三个典型问题，从中选取一个；
- (2) 参照“【实验指导】6.6 线程间同步互斥示例”，根据第一步的设计方案，利用 Pthread 提供的 `pthread_mutex_init()`、`pthread_mutex_lock()`、`pthread_mutex_unlock()` 等线程同步互斥 API，以线程方式编程实现该设计方案，观察分析程序运行结果。重点分析信号量设计方案是否合理、程序运行是否符合预期，应避免设计方案导致死锁或不符合题目要求。
- (3) 参照“【实验指导】6.5 进程间同步互斥示例”，根据上一步的设计方案，利用 `semget`、`semctl`、`semop` 等信号量原语，以进程方式编程实现该设计方案；
观察分析程序运行结果，重点分析信号量设计方案是否合理、程序运行是否符合预期，应避免设计方案导致死锁或不符合题目要求。

要求：

- ✧ 基于进程和基于线程的同步互斥实现方案二选一，完成其中一个即可；
- ✧ 不论采用进程方式、还是线程方式，模拟 A、B、C 三个 worker 的三个进程/线程 A、B、C 应当多次循环反复执行，便于观察同步互斥效应；
- ✧ 根据设计方案，给出程序代码，程序运行结果，结果分析。
- ✧ 也可以使用 C++11 的条件变量

7. 第六组 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，依次完成下述实验：

- 实验 6.1 观察实验平台物理 cpu、CPU 核和逻辑 cpu 的数目
- 实验 6.2. 单线程/进程串行 vs 2 线程并行 vs 3 线程加锁并行程序对比
- 实验 6.3. 3 线程加锁 vs 3 线程不加锁 对比
- 实验 6.4. 针对 Cache 的优化

- 实验 6.5. CPU 亲和力对并行程序影响



利用多核多线程进行程序优化.pdf

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

5. 实验要求

【具体实验要求，如分组、实验内容取舍等，听从任课教师安排】

- ✧ 上述实验内容分为**六组**，每组包括多个实验。按照要求，从每组中选择合适的实验，分别涵盖：进程创建及管理、线程创建及管理、进程间通信、线程间通信、基于进程或线程的进程间同步互斥（面向期末试题）、多核多线程编程，共 6 个实验；
- ✧ **实验分组进行，每组 3-6 位同学**，相互分工协作，共同完成实验。每组提交一份完整的实验报告；
- ✧ 实验完成后提交课程实验报告，报告内容应包括：题目，实验目的、实验内容、实验设计原理、实验步骤、实验结果及分析、程序代码等内容；
要求：实验报告文档应格式规范，内容完整。包括封面、目录、正文内容、案例图片、附录等，应附有程序代码和实验运行截图。

实验报告为Word文档，文档命名方式：

2020211314-姓名1-操作系统实验报告-

跨班组合的实验报告文档命名方式：

2020211314-姓名1-315-姓名2-322-姓名3-操作系统实验报告

- ✧ 课程实验完成后，提交实验报告文档、程序，通过上机演示进行实验验收。

要求：

- 按时完成实验，提交课程实验报告文档，并进行验收，不得缺席验收；
- 验收前，实验报告文档和程序发至：kcsjbupt@126.com
- 验收时间、地点：**待定**

6. 实验指导

下面给出一些关于阿里云、Linux 进程、Pthread 线程的基础知识，以及实现进程/线程创建、通信、同步互斥和多核多线程编程系统调用、API 函数接收，以及源程序代码示例，作为实验参考。参加实验的同学可以参照这些源码程序，结合各个实验的内容和具体要求，以及机器配置和实验环境，设计完成各自的实验内容。

注意：示例程序的是否能正确运行与具体的 Linux 环境密切相关，如内核版本。

以下部分内容参考了 CSDN 等网站、知乎公开出版的操作系统教科书和实验指导书上的内容。

6.1 阿里云注册登录-实验步骤说明

阿里云官网：

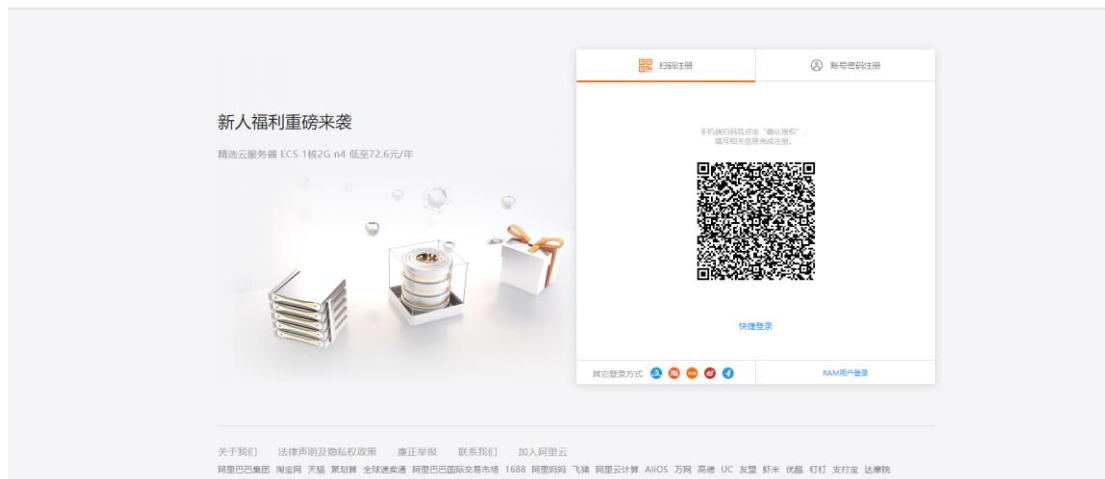
<https://cn.aliyun.com/>

6.1.1 步骤一 注册&登录

(1) 浏览器搜索或者直接点击进入阿里云官网（<https://cn.aliyun.com/>）



(如果没有阿里云账号，点击右上角“立即注册”。)



(选择扫码注册或者账号密码注册，也支持下方支付宝、淘宝账号等直接注册
账号密码注册时需要使用手机号、接收验证码)

6.1.2 步骤二 进入课程实验

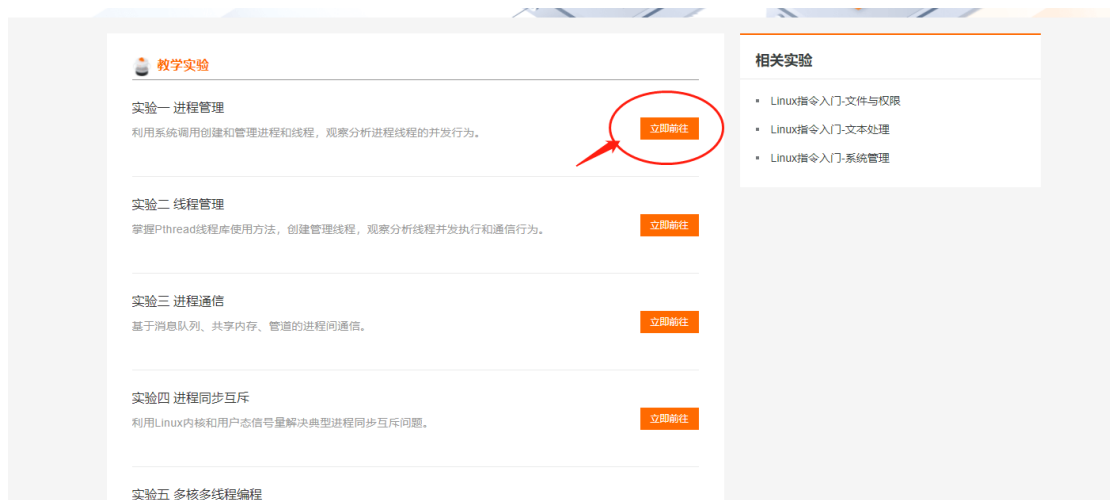
(1) 浏览器输入或者直接点击链接进入北京邮电大学-操作系统实验课程页面
(<https://developer.aliyun.com/adc/series/university-beijingyoudian>)



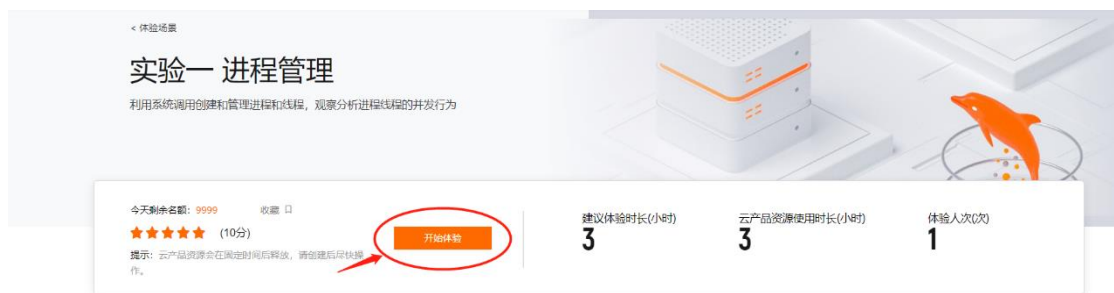
实验相关问题可咨询任课教师：yewen@bupt.edu.cn

6.1.3 步骤三 申请实验环境

(1) 选择一个要进行的实验，点击“立即前往”进入实验



(2) 在实验介绍界面中，点击开始体验。



(开始体验之前可在下方可以提前获悉实验目的和课程提醒。)

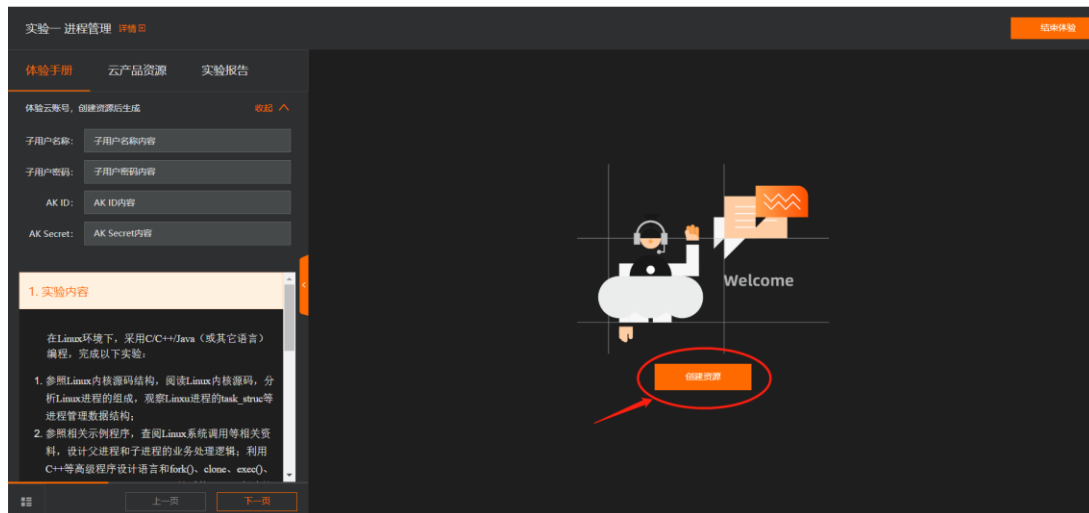
实验简介



体验简介

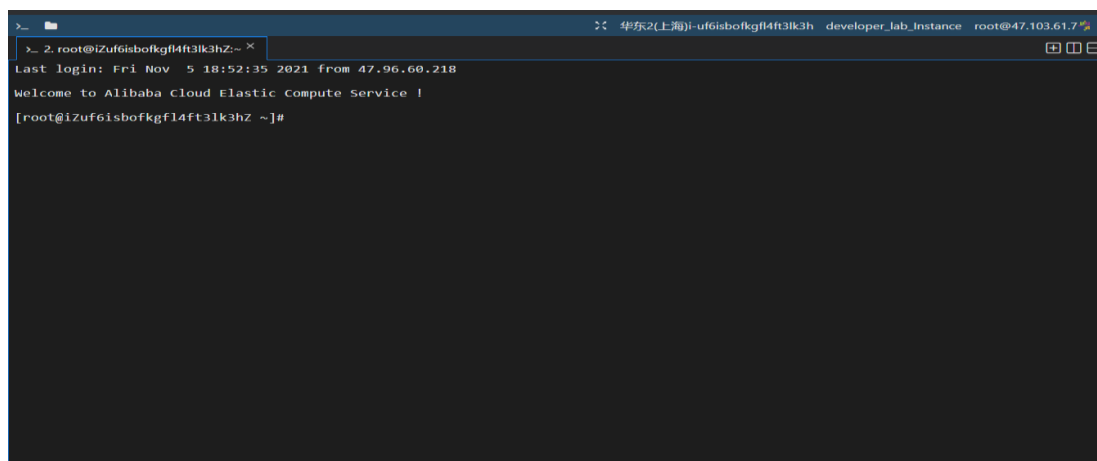
以Linux操作系统为实验对象，理解Linux系统中任务（task）和进程、线程等概念，掌握利用Linux API函数/系统调用创建和管理进程的方法，认识进程生命周期、状态转换和并发执行的实质。

(3) 在体验界面，点击“创建资源”，申请实验资源，耐心等待资源创建。

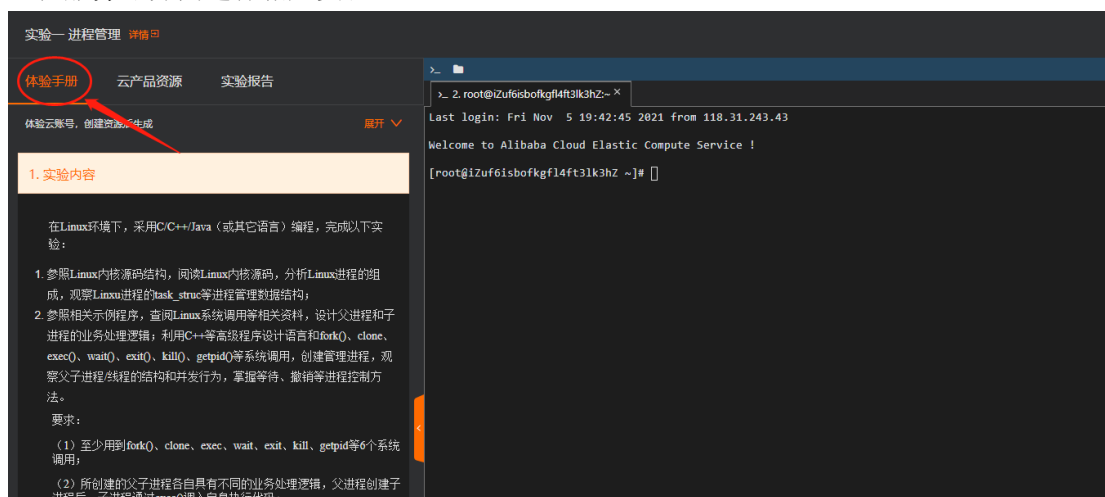


6.1.4 步骤四 进行实验

实验资源创建完成后，进入阿里云 ECS 实验环境，完成相应实验。

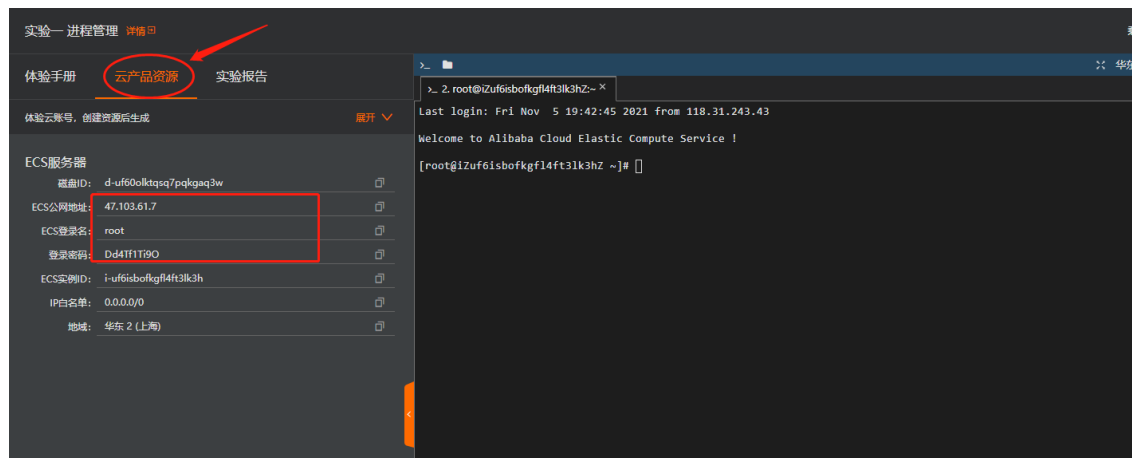


（在服务器界面进行相应实验）



（点击体验手册可以查看部署的实验指导书）

也可以使用远程连接进行实验
使用 ssh 连接到 ecs。

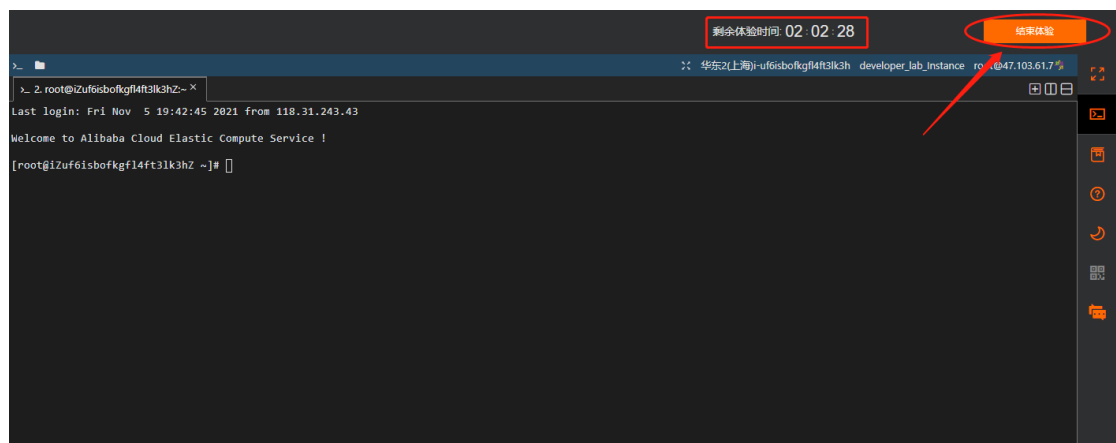


(点击“云产品资源”，可以看到资源的 ip 地址，用户名和密码。)

6.1.5 步骤五 结束实验

实验完成时，点击右上角“结束体验”，释放资源，方可进行其他实验。

注：实验结束后必须点击结束体验，否则无法进入其他实验。



6.2 进程创建管理程序示例

6.2.1 进程管理系统调用/API 函数

Linux 系统提供了以下进程管理系统调用/API 函数。

1. fork()

fork()函数创建一个新进程。

调用格式:

```
#include<sys/types.h>
```

```
#include<unistd.h>
```

```
int fork();
```

返回值:

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。

用法举例:

```
#include<stdio.h>
```

```
#include<sys/types.h>
```

```
#include<unistd.h>
```

```
int main(int argc, char** argv)
```

```
{
```

```
    if((pid=fork())>0)
```

```
    {
```

```
        printf("I am the parent process.\n");
```

```
        /*父进程处理过程*/
```

```
    }
```

```
    else if(pid==0)
```

```
    {
```

```
        printf("I am the child process.\n");
```

```
        /*子进程处理过程*/
```

```
        exit(0)
```

```
    }
```

```
    else
```

```

    {
        printf("fork error\n");
        exit(0);
    }
}

```

`fork()`用来创建一个新的进程，该进程几乎是当前进程的一个完全复制；`exec()`用来启动另外的进程以取代当前运行的进程。

2. clone()

`clone()`用于创建 Linux 线程。创建时可以通过多个参数，指定新的命名空间(namespace)，有选择地继承父进程的资源(如虚拟内存空间)，将创建的子进程指定为父进程的兄弟进程。当创建出的子进程与父进程共享同一虚拟内存空间时，子进程被认为是线程。

调用格式:

```

#include

int clone(int (*fn)(void *), void *child_stack, int flags, void *arg, ...
        /* pid_t *pid, struct user_desc *tls ", pid_t *" ctid " */);

int __clone2(int (*fn)(void *), void *child_stack_base, size_t stack_size, int flags, void *arg, ...
        /* pid_t *pid, struct user_desc *tls ", pid_t *" ctid " */);"

#include <sched.h>

#include<sched.h>

int clone(int (*fn)(void *), void *child_stack, int flags, void *arg)

```

各参数含义如下:

fn: 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: 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: 传给子进程的参数一般为 (0)；

flags: 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，`flags` 的主要取值

如下：

标志	含义
CLONE_PARENT	创建的子进程的父进程是调用者的父进程，新进程与创建它的进程成了“兄弟”而不是“父子”
CLONE_FS	<p>子进程与父进程共享相同的文件系统,包括 root、当前目录、umask。</p> <p>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。</p>
CLONE_FILES	<p>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。</p> <p>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。</p>
CLONE_NEWNS	在新的 namespace 启动子进程，namespace 描述了进程的文件 hierarchy
CLONE_SIGHAND	子进程与父进程共享相同的信号处理（signal handler）表
CLONE_PTRACE	若父进程被 trace，子进程也被 trace
CLONE_VFORK	<p>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。</p> <p>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。</p>
CLONE_VM	<p>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。</p> <p>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。</p>
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。

CLONE_THREAD	支持 POSIX 线程标准,子进程与父进程共享相同的线程群。
--------------	--------------------------------

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。

用法举例1:

`#define _GNU_SOURCE` //注意,宏设置在程序开头部分,否则编译报错

```
#include <sched.h>
```

```
#include <unistd.h>
```

```
#include <stdio.h>
```

```
#include <sys/types.h>
```

```
#include <sys/select.h>
```

```
#include <stdlib.h>
```

```
int value = 0;
```

```
int child_progress(void *arg)
```

```
{
    while(1)
    {
        printf("child,value = %d\r\n",value);
        value ++;
        sleep(1);
        if(value == 5)break;
    }
    return 0;
}
```

```
void main(int argc,char *argv[])
```

```
{
    int ret = -1;
    char *stack = NULL;
    pid_t tid = 0;
```

```

    stack = malloc(4096);
    if(NULL == stack)
    {
        printf("malloc fail]\r\n");
        return;
    }
//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
    int mask = CLONE_VM|CLONE_VFORK|CLONE_CHILD_SETTID;

    ret = clone(child_progress, stack+4096, mask, NULL, NULL, NULL, &tid);
//栈地址向下增长，因此起始地址为stack+4096
    if(ret < 0)
    {
        printf("clone error\r\n");
        return;
    }
    printf("clone child success, pid:%d %d\r\n", ret, tid);
    while(1)
    {
        printf("father, value = %d\r\n",value);
        value ++;
        sleep(1);
    }
}

```

运行结果：

```

child, value =0
child, value =1
child, value =2
child, value =3
child, value =4
clone child success, pid: 2800, 2800
father, value =5
father, value =6
father, value =7

```

```
father, value =8
father, value =9
father, value =10
father, value =11
```

用法举例 2:

父进程创建一个子进程/线程，子进程共享父进程的虚存空间，没有自己独立的虚存空间。父进程被挂起，当子线程释放虚存资源后再继续执行。

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <sched.h>

#define FIBER_STACK 8192

int a;

void * stack;

int do_something(){

    a=10;

    printf("child task, the pid is:%d, the a is: %d\n", getpid(), a);

    free(stack);

    exit(1);

}

int main() {

    void * stack;

    a = 1;

    stack = malloc(FIBER_STACK); //为子进程申请系统堆栈

    if(!stack) {

        printf("The stack failed\n");

        exit(0);

    }

    printf("creating child task/thread!!\n");
```

```

clone(&do_something, (char *)stack + FIBER_STACK,
      CLONE_VM|CLONE_VFORK, 0); //创建子任务

printf("This is parent, my pid is: %d, the a is: %d\n", getpid(), a);

exit(1);

}

```

运行的结果【假设】:

child 的 PID: 10692;

parent 的 PID: 10691;

parent 和 child 中的 a 都为 10，说明两个进程公用了一份变量 a，是指针的复制，而不是值的复制

clone 和 fork 的区别:

- (1) clone 需要通过参数传入一个函数，作为子进程的执行体，在创建的子进程中执行；
- (2) clone 不复制父进程的栈空间，而是通过第 2 个参数 (void *child_stack) 为创建的子进程分配一个新的栈空间，需要分配栈指针的空间大小。因此，clone 不再是简单地继承或者复制父进程；
- (3) fork 函数创建的子进程完全拷贝父进程，clone 则是有选择性地拷贝。

3. exec()

exec()函数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，新的程序可以是可执行文件或Shell批命令。

调用格式:

```

#include<unistd.h>

int execl(path, arg0, ..., argn, (char *)0)
char *path, *arg0, ..., *argn;

int execv(path, argv)
char *path, *argv[];

int execl(path, arg0, ..., argn, (char *)0, envp)
char *path, *arg0, ..., *argn, *envp[];

int execve(path, argv, envp)

```



```
char *path,*argv[],*envp[];
```

```
int execvp(file,argv)
```

```
char *file,*argv[];
```

说明: exec()是一个系统调用族, 包括 execl、execv、execle、execve、execvp 等。

用法举例:

```
/*exec.c*/
```

```
#include<unistd.h>
```

```
int main(int argc,char*argv[])
```

```
{
```

```
    char *envp[]={ "PATH=/tmp", "USER=lei", "STATUS = testing", NULL};
```

```
    char *argv_execv[]={ "echo", "executed by execv", NULL};
```

```
    char *argv_execvp[]={ "echo", "executed by execvp", NULL};
```

```
    char *argv_execve[]={ "env", NULL};
```

```
    if(fork()==0)
```

```
    {        if(execl("/bin/echo","echo","executed by execl",NULL)<0)
```

```
                perror("Err on execl");
```

```
    }
```

```
    if(fork()==0)
```

```
    {        if(execlp("echo","echo","executed by execlp",NULL)<0)
```

```
                perror("Err on execlp");
```

```
    }
```

```
    if(fork()==0)
```

```
    {        if(execle("/usr/bin/env","env",NULL,envp)<0)
```

```
                perror("Err on execle");
```

```
    }
```

```
    if(fork()==0)
```

```
    {        if(execv("/bin/echo",argv_execv)<0)
```

```
                perror("Err on execv");
```

```
    }
```

```
    if(fork()==0)
```

```
    {        if(execvp("echo",argv_execvp)<0)
```

```
                perror("Err on execvp");
```

```
    }
```

```

if(fork()==0)
{
    if(execve("/usr/bin/env",argv_execve, envp)<0)
        perror("Err on execve");
}

```

程序中调用了echo和env两个系统命令，echo打印后面的命令行参数，env用于列出所有环境变量。

由于各个子进程执行的顺序无法控制，因此有可能出现一个比较混乱的输出---各子进程打印的结果交杂在一起，而不是严格按照程序中列出的次序。

4. wait()

wait()用于控制父进程与子进程的同步。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。

调用格式：

```

#include <sys/types.h>

#include<sys/wait.h>

(pid_t)wait(int *statloc);

```

返回值：

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。

5. exit()

exit()用于结束进程自身。在进程正常结束时，exit()函数返回进程结束状态。在主程序main()函数中调用return，实际是调用exit()函数。

调用格式：

```

#include<stdio.h>

void exit(int status);

```

其中，status 为进程结束状态。

6. kill()

kill()函数用于删除执行中的程序或者任务。

调用格式:

```
kill(int PID, int IID);
```

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。

7. signal()

signal()函数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。

调用格式:

```
#include<signal.h>
int sig;
void(*func)();
signal(sig, function);
```

● sig取值如下:

SIGHUP 挂起 1
SIGINT 按Delete键或Break键 2
SIGQUIT 按Ctrl+Q组合键 3
SIGILL非法指令 4
SIGTRAP 跟踪中断 5
SIGIOT IOT指令 6
SIGBUS 总线错 7
SIGFPE 浮点运算溢出 8
SIGKILL 要求终止进程 9
SIGUSR1 用户定义信号#1 10
SIGSEGV 段违法 11
SIGUSR2 用户定义信号#2 12
SIGPIPE 向无读者管道上写 13
SIGALRM 定时器告警, 时间到 14
SIGTERM kill发出的软件结束信号 15
SIGCHLD 子进程被kill 17

SIGPWR 电源故障 30

- **function的解释如下**

在该进程中的一个函数地址，在内核态返回用户态时，它以软件中断信号的序号作为参数调用该函数，对除了信号SIGILL、SIGTRAP和SIGPWR以外的信号，内核自动地重新设置软中断信号处理程序的值为SIG_DFL，一个进程不能捕获 SIGKILL信号

SIG.DFL：默认操作。对除SIGPWR和SIGCHLD外所有信号的默认操作是进程终结。对信号SIGQUIT、SIGTRAP、SIGILL、SIGIOT、SIGTEMT、SIGFPE，SIGBUS SIGSEGV和SIGSYS，它产生一内存映像文件。

SIG_IGN：忽视该信号的出现。

8. getpid()

获取目前进程的进程标识 pid

调用格式：int getpid()

示例： # include<unistd.h>

```
main() {  
    printf("pid=%d\n", getpid());  
}
```

6.2.2 父子进程创建与控制

1. **程序 1.** 父进程生成子进程，父进程等待子进程 wait()，子进程执行完成后自我终止 exit()，并唤醒父进程，父子进程执行时打印相关信息。

```
main(){  
    int i,j,k;  
    if(i=fork()){                //非零值  
        j=wait();  
        printf("Parent process! \n");  
        printf("i=%d j=%d\n",i,j);
```

```

    }else{
        k=getpid();
        printf("Child process! \n");
        printf("i=%d k=%d\n",i,k);
    }
}

```

2. **程序 2.** 父进程生成子进程，父进程发送信号并等待，子进程接收信号并完成某种功能。

```

int func();
main(){
    int i,j;
    signal(17,func);
    if(i=fork()){
        printf("Parent:Signal 17 will be send to child!\n");
        kill(1,17);
        wait(0);
        printf("Parent:finished!\n");
    }else{
        sleep(10);
        printf("Child:A signal from my Parent is received!\n");
        exit();
    }
}
func(){
    printf("It is signal 17 processing function!\n");
}

```

执行结果为:

```

Parent:signal 17 will be send to Child!
It is signal 17 processing function!
Child:A signal from my Parent is received!
Parent:finished!

```

6.3 线程创建管理程序示例

6.3.1 Pthread 线程库 API

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多个线程。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

1. 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

2. 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

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用 pthread_join 为止。

3. 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

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。

4. 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

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。

5. 线程清除 `pthread_cleanup_push()/pthread_cleanup_pop()`

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

功能说明：

将清除函数压入清除栈

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

功能说明:

将清除函数弹出清除栈

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。

6. 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

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。

7. 线程标识获取 `pthread_self()`

调用格式：

```
#include<pthread.h>
```

```
Pthread_t pthread_self(void)
```

参数：

返回值：

返回调用该函数的线程的标识 ID

6.3.2 示例 2-2-1

功能：使用 pthread_create()函数创建线程的实例

代码：thread_create.c 文件

```
1  /* thread_create.c */
2  #include<stdio.h>
3  #include<stdlib.h>
4  #include<pthread.h>
5
6  /*线程函数1*/
7  void *mythread1(void)
8  {
9      int i;
10     for(i=0;i<5;i++)
11     {
12         printf("I am the 1st pthread,created by mybelief321\n");
13         sleep(2);
14     }
15 }
16 /*线程函数2*/
17 void *mythread2(void)
18 {
19     int i;
20     for(i=0;i<5;i++)
21     {
22         printf("I am the 2st pthread,created by mybelief321\n");
23         sleep(2);
24     }
25 }
26
```

```

27 int main()
28 {
29     pthread_t id1,id2; /*线程ID*/
30     int res;
31     /*创建一个线程，并使得该线程执行mythread1函数*/
32     res=pthread_create(&id1,NULL,(void *)mythread1,NULL);
33     if(res)
34     {
35         printf("Create pthread error!\n");
36         return 1;
37     }
38     /*创建一个线程，并使得该线程执行mythread2函数*/
39     res=pthread_create(&id2,NULL,(void *)mythread2,NULL);
40     if(res)
41     {
42         printf("Create pthread error!\n");
43         return 1;
44     }
45     /*等待两个线程均推出后，main()函数再退出*/
46     pthread_join(id1,NULL);
47     pthread_join(id2,NULL);
48
49     return 1;
50 }

```

编译: 使用命令: gcc thread_create.c -o thread_create -lpthread 编译, 注意不要忘了加 -lpthread, 否则会出现如下的错误

```

/tmp/ccjFZIN3.o: In function `main':
thread_create.c:(.text+0x8b): undefined reference to `pthread_create'
thread_create.c:(.text+0xc0): undefined reference to `pthread_create'
thread_create.c:(.text+0xeb): undefined reference to `pthread_join'
thread_create.c:(.text+0xfc): undefined reference to `pthread_join'
collect2: ld returned 1 exit status

```

执行:

```

song@ubuntu:~/lianxi$ vim thread_create.c
song@ubuntu:~/lianxi$ gcc thread_create.c -o thread_create -lpthread
song@ubuntu:~/lianxi$ ./thread_create
I am the 2st pthread,created by mybelief321
I am the 1st pthread,created by mybeilef321
I am the 2st pthread,created by mybelief321
I am the 1st pthread,created by mybeilef321
I am the 2st pthread,created by mybelief321
I am the 1st pthread,created by mybeilef321
I am the 2st pthread,created by mybelief321
I am the 1st pthread,created by mybeilef321
I am the 2st pthread,created by mybelief321
I am the 1st pthread,created by mybeilef321

```

6.3.3 示例 2-2-2

功能：使用 pthread_exit()函数退出线程的举例

代码：thread_exit.c 文件

```
1 /* thread_exit.c文件 */
2 #include<stdio.h>
3 #include<pthread.h>
4 /*进程函数*/
5 void *create(void *arg)
6 {
7     printf("New thread is created...\n");
8     pthread_exit((void *)6); /*这里的6也可以设置成其它数值*/
9 }
10
11 int main()
12 {
13     pthread_t tid;
14     int res;
15     void *temp;
16     res=pthread_create(&tid,NULL,create,NULL);
17     printf("I am the main thread!\n");
18     if(res)
19     {
20         printf("thread is not created...\n");
21         return -1;
22     }
23
24     res=pthread_join(tid,&temp);
25     if(res)
26     {
27         printf("Thread is not exit...\n");
28         return -2;
29     }
30     printf("Thread is exit code %d \n",(int)temp);
31     return 0;
32 }
```

编译：gcc thread_exit.c -o thread_exit -lpthread

执行：./thread_exit

```
song@ubuntu:~/lianxi$ ./thread_exit
I am the main thread!
New thread is created...
Thread is exit code 6
```

6.3.4 示例 2-2-3

功能：用 pthread_join()实现线程等待。

代码：thread_join.c 文件

```
1 /*thread_join.c*/
2 #include<pthread.h>
3 #include<stdio.h>
4 /*线程函数*/
5 void *thread(void *str)
6 {
7     int i;
8     for(i=0;i<4;++i)
9     {
10         sleep(2);
11         printf("This is the thread:%d\n",i);
12     }
13     return NULL;
14 }
15
16 int main()
17 {
18     pthread_t pth; /*线程ID*/
19     int i;
20     int ret=pthread_create(&pth,NULL,thread,(void *)(&i));
21     pthread_join(pth,NULL);
22     printf("123\n");
23     for(i=0;i<3;++i)
24     {
25         sleep(1);
26         printf("This is the main:%d\n",i);
27     }
28     return 0;
29 }
```

编译：gcc thread_join.c -o thread_join -lpthread

执行：./thread_join

```
song@ubuntu:~/lianxi$ ./thread_join
This is the thread:0
This is the thread:1
This is the thread:2
This is the thread:3
123
This is the main:0
This is the main:1
This is the main:2
```

可以看出，pthread_join()等到线程结束后，程序才继续执行。

【示例 2-2-4】

功能：使用 pthread_self()获取线程 ID

代码：thread_id.c 文件

```
1 /*thread.c*/
2 #include<stdio.h>
3 #include<pthread.h>
4 #include<unistd.h>
5 void *create(void *arg)
6 {
7     printf("New thread.....\n");
8     printf("This thread's id is %u \n",(unsigned int)pthread_self());
9     printf("This thread process pid is %d \n",getpid());
10    return NULL;
11 }
12
13 int main()
14 {
15     pthread_t tid;
16     int res;
17     printf("Main thread is starting...\n");
18     res=pthread_create(&tid,NULL,create,NULL);
19     if(res)
20     {
21         printf("thread is not created...\n");
22         return -1;
23     }
24     printf("The main process pid is %d \n",getpid());
25     sleep(1);
26     return 0;
27 }
```

编译：gcc thread_id.c -o thread_id -lpthread

执行：./thread_id

```
song@ubuntu:~/lianxi$ ./thread_id
Main thread is starting...
The main process pid is 2944
New thread.....
This thread's id is 873625344
This thread process pid is 2944
song@ubuntu:~/lianxi$
```

6.3.5 示例 2-2-5

功能：线程清理函数的使用

代码：thread_clean.c

```
1  /* thread_clean.c */
2  #include<stdio.h>
3  #include<pthread.h>
4  /*清除函数*/
5  void *clean(void *arg)
6  {
7      printf("cleanup:%s  \n",(char *)arg);
8      return (void*)0;
9  }
10
11 /*线程函数1*/
12 void *thr_fn1(void *arg)
13 {
14     printf("Thread1 start \n");
15     /*注意我这里是将清除函数压入了两次清除栈，两次的清除函数的
16     参数不同*/
17     pthread_cleanup_push((void *)clean,"thread1 first  handler");
18     pthread_cleanup_push((void *)clean,"thread1 second handler");
19     printf("thread1 push complete \n");
20     if(arg) /*main()函数中如果传递的参数arg非0,则退出此线程*/
21     {
22         return ((void *)1);
23     }
24     pthread_cleanup_pop(1);/*弹出清理函数的时候执行清理函数，
25     一定要注意，弹出栈的顺序，是先将最后
26     压入栈的函数先弹出来*/
27     pthread_cleanup_pop(1);/*弹出清理函数的时候执行清理函数*/
28     return (void *)2;
29 }
30
31 void *thr_fn2(void *arg)
32 {
33     printf("Thread2 start \n");
34     pthread_cleanup_push((void *)clean,"thread2 first  handler");
35     pthread_cleanup_push((void *)clean,"thread2 second handler");
36     printf("thread2 push complete \n");
37     if(arg)
38     {
39         return ((void *)3);
40     }
41     pthread_cleanup_pop(1);/*弹出清理函数的时候执行清理函数*/
42     pthread_cleanup_pop(0);/*弹出清理函数的时候不执行清理函数*/
43     return (void *)4;
44 }
45
```



```

46 int main()
47 {
48     int res;
49     pthread_t tid1,tid2;
50     void *tret;
51     /*创建一个线程，注意传递的参数是1*/
52     res=pthread_create(&tid1,NULL,thr_fn1,(void *)1);
53     if(res!=0)
54     {
55         printf("Create thread error...\n");
56         return -1;
57     }
58     /*创建一个线程，注意传递的参数是0*/
59     res=pthread_create(&tid2,NULL,thr_fn2,(void *)0);
60     if(res!=0)
61     {
62         printf("Create thread error...\n");
63         return -1;
64     }
65     /*等待线程1结束*/
66     res=pthread_join(tid1,&tret);
67     if(res!=0)
68     {
69         printf("Thread_join error...\n");
70         return -1;
71     }
72     printf("Thread1 exit code: %d\n",(int)tret);
73     /*等待线程2结束*/
74     res=pthread_join(tid2,&tret);
75     if(res!=0)
76     {
77         printf("Thread_join error...\n");
78         return -1;
79     }
80     printf("Thread2 exit code: %d\n",(int)tret);
81     return 0;
82 }
83 }

```

注意，在编写的代码的时候，可以修改一下传递的参数和 clean_pop 函数的参数。

编译：gcc thread_clean.c -o thread_clean -lpthread

执行：./thread_clean

```

song@ubuntu:~/lianxi$ ./thread_clean
Thread2 start
thread2 push complete
Thread1 start
cleanup:thread2 second handler
thread1 push complete
Thread1 exit code: 1
Thread2 exit code: 4
song@ubuntu:~/lianxi$

```

6.3.6 示例 2-2-6

功能：本实验创建了 3 个进程，为了更好的描述线程之间的并行执行，让 3 个线程共用同一个执行函数。每个线程都有 5 次循环(可以看成 5 个小任务)，每次循环之间会随机等待 1~10s 的时间，意义在于模拟每个任务的到达时间是随机的，并没有任何特定的规律。

代码：thread.c 文件，

```

1  /* thread.c */
2  Dash Home <stdio.h>
3  #include<stdlib.h>
4  #include<pthread.h>
5  #define THREAD_NUMBER 3 /*线程数*/
6  #define REPEAT_NUMBER 5 /*每个线程中的小任务数*/
7  #define DELAY_TIME_LEVELS 10.0 /*小任务之间的最大时间间隔*/
8
9  /*线程函数例程*/
10 void *thrd_func(void *arg)
11 {
12     int thrd_num=(int)arg;
13     int delay_time=0;
14     int count=0;
15
16     printf("Thread %d is starting\n",thrd_num);
17     for(count=0;count<REPEAT_NUMBER;count++)
18     {
19         delay_time=(int)(rand()*DELAY_TIME_LEVELS/(RAND_MAX))+1;
20         sleep(delay_time);
21         printf("\tThread %d:job %d delay=%d\n",thrd_num,count,delay_time);
22     }
23     printf("Thread %d finished\n",thrd_num);
24     pthread_exit(NULL); /*退出线程*/
25 }
26
27 int main(void)
28 {
29     pthread_t thread[THREAD_NUMBER]; /*存放线程ID*/
30     int no,res;
31     void *thrd_ret;
32

```

```

33  srand(time(NULL)); /*随机数发生器的初始化函数*/
34
35  for(no=0;no<THREAD_NUMBER;no++)
36  {
37      /*创建多线程*/
38      res=pthread_create(&thread[no],NULL,(void *)thrd_func,(void *)no);
39      if(res!=0)
40      {
41          printf("Create thread %d failed\n",no);
42          exit(res);
43      }
44  }
45
46  printf("Creating threads success\nWaiting for threads to finish...\n");
47  for(no=0;no<THREAD_NUMBER;no++)
48  {
49      /*等待线程结束*/
50      res=pthread_join(thread[no],&thrd_ret);
51      if(!res)
52      {
53          printf("Thread %d joined\n",no);
54      }
55      else
56      {
57          printf("Thread %d join failed\n",no);
58      }
59  }
60
61  return 0;
62 }

```

编译: gcc thread.c -o thread -lpthread

执行: ./thread

```

song@ubuntu:~/lianxi$ ./thread
Thread 1 is starting
Thread 0 is starting
Creating threads success
Waiting for threads to finish...
Thread 2 is starting
    Thread 2:job 0 delay=3
    Thread 1:job 0 delay=5
    Thread 2:job 1 delay=3
    Thread 0:job 0 delay=7
    Thread 1:job 1 delay=5
    Thread 0:job 1 delay=3
    Thread 1:job 2 delay=2
    Thread 0:job 2 delay=2
    Thread 1:job 3 delay=1
    Thread 2:job 2 delay=9
    Thread 0:job 3 delay=5
    Thread 0:job 4 delay=1
Thread 0 finished
Thread 0 joined
    Thread 1:job 4 delay=10
Thread 1 finished
Thread 1 joined
    Thread 2:job 3 delay=9
    Thread 2:job 4 delay=3
Thread 2 finished
Thread 2 joined
song@ubuntu:~/lianxi$

```

从实验结果可以看出，线程执行的顺序杂乱无章的，可以利用线程之间的同步与互斥机制规范线程的执行。

6.3.7 Linux-c/c++线程间参数传递

【 https://blog.csdn.net/hxlawf/article/details/96478722?utm_medium=distribute.pc_relevant.none-task-blog-OPENSEARCH-3&depth_1-utm_source=distribute.pc_relevant.none-task-blog-OPENSEARCH-3】

线程创建函数原型：

```
include <pthread.h>
```

```
int pthread_create(pthread_t *restrict tidp,const pthread_attr_t *restrict attr, void
*(*start_rtn)(void),void *restrict arg);
```

返回值：若是成功建立线程返回 0,否则返回错误的编号

形式参数：

pthread_t *restrict tidp 要创建的线程的线程 id 指针
const pthread_attr_t *restrict attr 创建线程时的线程属性
void* (start_rtn)(void) 返回值是 void 类型的指针函数
void *restrict arg start_rtn 的行参

【示例 2-3-1 线程创建】

创建一个简单的线程，代码如下：

```
1.  #include <stdio.h>
2.  #include <pthread.h>
3.
4.  void *mythread1(void)
5.  {
6.      int i;
7.      for(i = 0; i < 10; i++)
8.      {
9.          printf("This is the 1st pthread,created by Robben!\n");
10.         sleep(1);
11.     }
12. }
13.
14. void *mythread2(void)
15. {
16.     int i;
17.     for(i = 0; i < 10; i++)
18.     {
19.         printf("This is the 2st pthread,created by Robben!\n");
20.         sleep(1);
21.     }
22. }
23.
24. int main(int argc, const char *argv[])
25. {
26.     int i = 0;
27.     int ret = 0;
28.     pthread_t id1,id2;
```

```

29.
30.     ret = pthread_create(&id1, NULL, (void *)mythread1,NULL);
31.     if(ret)
32.     {
33.         printf("Create pthread error!\n");
34.         return 1;
35.     }
36.
37.     ret = pthread_create(&id2, NULL, (void *)mythread2,NULL);
38.     if(ret)
39.     {
40.         printf("Create pthread error!\n");
41.         return 1;
42.     }
43.
44.     pthread_join(id1,NULL);
45.     pthread_join(id2,NULL);
46.
47.     return 0;
48. }

```

执行结果如下:

```
robben@ubuntu:~/work/project/test$ g++ cpp.cpp -o cpp -lpthread
```

```
robben@ubuntu:~/work/project/test$ ./cpp
```

```
This is the 2st pthread,created by Robben!
```

```
This is the 1st pthread,created by Robben!
```

```
This is the 2st pthread,created by Robben!
```

```
This is the 1st pthread,created by Robben!
```

```
This is the 2st pthread,created by Robben!
```

```
This is the 1st pthread,created by Robben!
```

```
This is the 2st pthread,created by Robben!
```

```
This is the 1st pthread,created by Robben!
```

两个线程交替执行

【示例 2-3-2: 向新的线程传递整形值】

代码如下:

1. `#include <stdio.h>`
2. `#include <pthread.h>`

```

3.
4. void *create(void *arg)
5. {
6.     int *num;
7.     num = (int *)arg;
8.     printf("Create parameter is %d\n",*num);
9.     return (void *)0;
10. }
11.
12. int main(int argc, const char *argv[])
13. {
14.     pthread_t id1;
15.     int error;
16.
17.     int test = 4;
18.     int *attr = &test;
19.
20.     error = pthread_create(&id1,NULL,create,(void *)attr);
21.
22.     if(error)
23.     {
24.         printf("Pthread_create is not created!\n");
25.         return -1;
26.     }
27.     sleep(1);
28.
29.     printf("Pthread_create is created..\n");
30.     return 0;
31. }

```

执行结果如下：

```

robben@ubuntu:~/work/project/test$ g++ cpp.cpp -o cpp -lpthread
robben@ubuntu:~/work/project/test$ ./cpp
create parameter is 4
Pthread_create is created..

```

从上可看出，在 main 函数中传递的 int 指针，传递到新建的线程函数中。

【示例 2-3-3：向新建的线程传递字符串】

代码如下：

```

1. #include <stdio.h>
2. #include <pthread.h>
3.
4. void *create(char *arg)

```

```

5. {
6.     char *str;
7.     str = arg;
8.     printf("The parameter passed from main is %s\n",str);
9.
10.    return (void *)0;
11. }
12.
13. int main()
14. {
15.     int error;
16.     pthread_t id1;
17.     char *str1 = "Hello!";
18.     char *attr = str1;
19.     error = pthread_create(&id1, NULL, create, (void *)attr);
20.
21.     if(error != 0)
22.     {
23.         printf("This pthread is not created!\n");
24.         return -1;
25.     }
26.     sleep(1);
27.
28.     printf("pthread is created..\n");
29.     return 0;
30. }

```

执行结果如下：

```
robben@ubuntu:~/work/project/test$ ./thread3
```

```
The parameter passed from main is Hello!
```

```
pthread is created..
```

可看出，main 函数中的字符串传入到新建的线程中。

【示例 2-3-4：向新建的线程传递字符串】

程序名称：pthread_struct.c，代码如下：

```

1.  #include <stdio.h>
2.  #include <pthread.h>
3.  #include <stdlib.h>
4.
5.  struct member
6.  {

```



```

7.     int a;
8.     char *s;
9. };
10.
11. void *create(void *arg)
12. {
13.     struct member *temp;
14.     temp = (struct member *)arg;
15.     printf("member->a = %d\n",temp->a);
16.     printf("member->s = %s\n",temp->s);
17.
18.     return (void *)0;
19. }
20.
21. int main()
22. {
23.     int error;
24.     pthread_t id1;
25.     struct member *p;
26.     p = (struct member *)malloc(sizeof(struct member));
27.     p->a = 1;
28.     p->s = "Robben!";
29.
30.     error = pthread_create(&id1,NULL,create,(void *)p);
31.
32.     if(error)
33.     {
34.         printf("pthread is not created!\n");
35.         return -1;
36.     }
37.     sleep(1);
38.     printf("pthread is created!\n");
39.
40.     free(p);
41.     p = NULL;
42.     return 0;
43. }

```

执行结果如下：

```

robben@ubuntu:~/work/project/test$ ./thread4
member->a = 1
member->s = Robben!
pthread is created!

```

可以看出，main 函数中的一个结构体传入了新建的线程中。线程包含了标识进程内执行环境必须的信息，集成了进程中的所有信息，对线程进行共享的，包括文本程序、程序的全局内存和堆内存、栈以及文件描述符。

【示例 2-3-5：验证新建立的线程可以共享进程中的数据】

程序名称：pthread_share.c，代码如下：

```
1.  #include <stdio.h>
2.  #include <pthread.h>
3.
4.  static int a = 5;
5.
6.  void *create(void *arg)
7.  {
8.      printf("New pthread...\n");
9.      printf("a = %d\n",a);
10.
11.     return (void *)0;
12. }
13.
14. int main(int argc, const char *argv[])
15. {
16.     int error;
17.     pthread_t id1;
18.
19.     error = pthread_create(&id1, NULL, create, NULL);
20.     if(error != 0)
21.     {
22.         printf("new thread is not created!\n");
23.         return -1;
24.     }
25.     sleep(1);
26.     printf("New thread is created...\n");
27.
28.     return 0;
29. }
```

结果如下：

```
robben@ubuntu:~/work/project/test$ ./thread5
New pthread...
a = 5
New thread is created...
```

可以看出，在主线程更改全局变量 a 的值的时候，新建立的线程打印出改变后的值，可以看出可以访问线程所在进程中的数据信息。

6.4 进程通信示例

Linux 内核提供以下几种进程间通信（IPC）机制。

1. **消息队列：**消息队列是消息的链接表，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；
2. **共享内存：**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；
3. **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；
4. **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；
5. **信号量（semaphore）：**提供进程间以及同一进程内不同线程之间的同步与互斥。
6. **套接口（Socket）：**用于不同机器之间的进程间通信。起初是由 Unix 系统的 BSD 分支开发出来的，但现在一般可以移植到其它类 Unix 系统上：Linux 和 System V 的变种都支持套接字。

本次实验要求完成消息队列、共享内存、管道、信号四种进程通信实验。

6.4.1 消息队列

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 msgsnd(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，消息数据的一份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，就把消息队列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。

以下 2 段程序利用 `msgreceive` 和 `msgsned` 接收和发送信息。允许两个程序都可以创建消息，但只有接收者在接收完最后一个消息之后，才删除消息。

（1）消息接收者源文件为 `msgreceive.c`，源代码为：

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <sys/msg.h>
```

```
struct msg_st
{
```

```

        long int msg_type;
        char text[BUFSIZ];
    };

int main()
{
    int running = 1;
    int msgid = -1;
    struct msg_st data;
    long int msgtype = 0; //注意 1

    //建立消息队列

    msgid = msgget((key_t)1234, 0666 | IPC_CREAT);

    if(msgid == -1)
    {
        fprintf(stderr, "msgget failed with error: %d\n", errno);
        exit(EXIT_FAILURE);
    }

    //从队列中获取消息，直到遇到 end 消息为止

    while(running)
    {
        if(msgrcv(msgid, (void*)&data, BUFSIZ, msgtype, 0) == -1)
        {
            fprintf(stderr, "msgrcv failed with errno: %d\n", errno);
            exit(EXIT_FAILURE);
        }

        printf("You wrote: %s\n", data.text);

        //遇到 end 结束

        if(strncmp(data.text, "end", 3) == 0)
            running = 0;    }
    }

    //删除消息队列

    if(msgctl(msgid, IPC_RMID, 0) == -1)

```

```

    {
        fprintf(stderr, "msgctl(IPC_RMID) failed\n");
        exit(EXIT_FAILURE);
    }
    exit(EXIT_SUCCESS);
}

```

(2) 消息发送者程序的源文件 msgsend.c, 源代码为:

```

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <sys/msg.h>

#include <errno.h>

#define MAX_TEXT 512

struct msg_st
{
    long int msg_type;
    char text[MAX_TEXT];
};

int main()
{
    int running = 1;

    struct msg_st data;

    char buffer[BUFSIZ];

    int msgid = -1;

    //建立消息队列

    msgid = msgget((key_t)1234, 0666 | IPC_CREAT);

    if(msgid == -1)
    {
        fprintf(stderr, "msgget failed with error: %d\n", errno);
    }
}

```

```

        exit(EXIT_FAILURE);
    }

    //向消息队列中写消息，直到写入 end
    while(running)
    {
        //输入数据
        printf("Enter some text: ");
        fgets(buffer, BUFSIZ, stdin);
        data.msg_type = 1;    //注意 2
        strcpy(data.text, buffer);
        //向队列发送数据
        If (msgsnd(msgid, (void*)&data, MAX_TEXT, 0) == -1)
        {
            fprintf(stderr, "msgsnd failed\n");
            exit(EXIT_FAILURE);
        }
        //输入 end 结束输入
        if(strncmp(buffer, "end", 3) == 0)
            running = 0;
        sleep(1);
    }
    exit(EXIT_SUCCESS);
}

```

运行结果如下：


```

[ljianhui@localhost CppCode]$ gcc -o msgreceive.exe msgreceive.c -lm
[ljianhui@localhost CppCode]$ gcc -o msgsend.exe msgsend.c -lm
[ljianhui@localhost CppCode]$ ./msgreceive.exe &
[1] 5539
[ljianhui@localhost CppCode]$ ./msgsend.exe
Enter some text: Linux
You wrote: Linux

Enter some text: Programming
You wrote: Programming

Enter some text: end
You wrote: end

[1]+  Done                  ./msgreceive.exe
[ljianhui@localhost CppCode]$

```

6.4.2 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，以及系统 V 共享内存。下面针对 Ubuntu2014 环境，介绍 System V 共享内存实现方法。基于内存映射 `mmap()` 的共享内存通信实验安排在“实验六 内存管理”实验中。

本示例创建 2 个程序，`write.c` 向共享内存写入数据，随后 `read.c` 从共享内存读取数据。

1. 原理及系统调用 API

进程间需要共享的数据被放在一个称为 IPC 共享内存区域的地方，需要访问该共享区域的进程需要将该共享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 `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。

注意：所创建的每个共享内存区都有一个重要的控制结构 `struct shmid_kernel`，该结构是联系内存管理和文件系统的桥梁，定义如下：

```

struct shmid_kernel /*"private to the kernel"*/
{
    struct kern_ipc_perm    shm_perm;
    struct file*           shm_file;
    int    id;
    unsigned long    shm_nattch;
    unsigned long    shm_segsz;
    time_t           shm_atim;

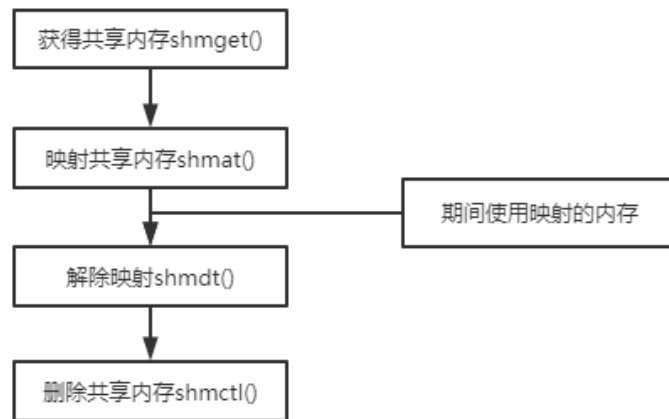
```

```

time_t      shm_dtim;
time_t      shm_ctim;
pid_t       shm_cprid;
pid_t       shm_lprid;
};

```

其中，最重要的域是 `shm_file`，它存储了将被映射文件的地址。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，一般情况下，特殊文件系统 `shm` 中的文件是不能用 `read()`、`write()`等方法访问的。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，可直接采用访问内存的方式访问该文件，如下图所示。



共享内存通信采用的系统调用有：

（1）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。

（2）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。

（3）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。

(4) 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。

2. 实验步骤及结果

(1) 开启一个终端，创建两个程序：read.c及write.c;

(2) 编译这两个程序输入命令：gcc -g write.c -o write及gcc -g read.c -o read，然后执行输入命令：./write及./read，可执行文件为write及read。

可得到读写进程间通信后的程序结果如下：

```
shikaijing@ubuntu:~/Desktops$ gcc -read.c -o read
shikaijing@ubuntu:~/Desktops$ ./read
key=50397187
shm_id=1507338
shikaijing@ubuntu:~/Desktops$ gcc -g write.c -o write
key=50397187
shm_id=1507338
name:test1
age0
name:test2
age
name:test3
age 2
shikaijing@ubuntu:~/Destops$
```

3. 源程序

write.c

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <unistd.h>
```

```

typedef struct{
    char name[8];
    int age;}
people;

int main(int argc,char** argv)

int shm_id.i;
key_t key;
char temp[8];
people* p_map;
char pathname[30];
strcpy(pathname,"/tmp");
key=ftok(pathname,0x03);//利用ftok()函数创建key。
if(key=-1)
{
    perror("ftok error");
    return -1;
}
Printf("key=%d\n",key);
shm_id=shmget(key,4096,IPC_CREAT|IPC_EXCL|0600);
//shm_id是共享内存标识符。Shmget()创建共享内存
if(shm_id== -1)
{
    perror("shmget error");
    return -1;
}
printf("shm_id=%d\n",shm_id);
p_map=(people*)shmat(shm_id,NULL,0); //映射共享内存
memset(temp,0x00,sizeof(temp));
strcpy(temp,"test");
temp[4]='0';
for(i=0;i<3;i++)
{
    temp[4]+=1;
    strncpy((p_map+i)->name,temp,5);
    (p_map+i)->age=0+i;
}

```

```

}
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。
return 0;
}

```

read.c

```

typedef struct{
char name[8];
int age;
}people;
int main(int argc,char**argv)
{
    int shm_id,i;
    key_t key;
    people* p_map;
    char pathname[30];
    strcpy(pathname,"/tmp");
    key=ftok(pathname,0x03);
    if(key==-1)
    {
        perror("ftok error");
        return -1;
    }
    printf("key=%d\n",key);
    shm_id=shmget(key,0,0);
    if(shm_id==-1)
    {
        perror("shmget error");
        return -1;
    }
    printf("shm_id=%d\n", shm_id);
    p_map=(people*)shmat(shm_id,NULL,0);
    for(i=0;i<3;i++)
    {

```

```

        printf("name:%s\n",(*(p_map+i)).name);
        printf("age:%d\n",(*(p_map+i)).age);
    }
    if(shmctl(p_map)==-1)
    {
        perror("detach error");
        return -1;
    }
    return 0;
}

```

6.4.3 管道通信

实验环境：Ubuntu 14.0.

1. 原理

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(**pipe**)。

(1) 无名管道（用于具有父子关系的进程间通信）

一个利用**pipe()**建立起来的无名文件（无路径名）、临时文件。使用该系统调用所返回的文件描述符来标识该管道文件，因此只有调用**pipe()**的进程及其子孙进程才能识别此文件描述符，才能利用该文件（管道）进行通信。

(2) 命名管道（用于任意两个进程间通信）

一个可以在文件系统中长期存在的、具有路径名的一种特殊类型的文件，也称为**FIFO**文件。它克服了无名管道使用上的局限性，允许更多的进程利用管道进行通信。其他进程可以通过文件系统知道管道的存在，并能利用路径名来访问该文件。对命名管道的访问方式与访问其他文件一样，先使用**open()**打开。

pipe文件建立过程包括：分配磁盘和内存索引结点、为读进程分配文件表项，为写进程分配文件表项，分配用户文件描述符。

无名管道和命名管道的差异在于：

使用无名管道通信的进程之间需要一个父子关系，通信的两个进程一定是由一个共同的祖先进程启动；但无名管道没有数据交叉的问题，命名管道**FIFO**不同于无名管道之处在于它提供了一个路径名与之关联，以**FIFO**的文件形式存在于文件系统中。这样，即使与**FIFO**的创建进程不存在亲缘关系的进程，只要可以访问该路径，就能够彼此通过**FIFO**相互通信。

因此，通过FIFO不相关的进程也能交换数据。且FIFO严格遵循先进先出（first in first out），对管道及FIFO的读总是从文件开始处返回数据，对管道的写则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。

管道是Linux的一种通信方式，一种两个进程间进行单向通信的机制，它提供了简单的流控制机制。管道是单向的、半双工的，数据只能向一个方向流动，双方通信时需要建立两个管道。无名管道具有很多限制，即只能是在父子进程中，只能在一台机器上，同一时间只能读或者写。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。

2. 管道相关系统调用

(1) 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。

(2) 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;
```

(3) write()

系统调用格式：write(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。

(4) 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。

(5)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。

(6) 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

3. 管道示例程序及结果

```
#include <unistd.h>
#include<signal.h>
#include <stdio.h>
#include <stdlib.h>
int pid;          /*定义进程变量*/
main(){
int fd[2];
char outpipe[100], inpipe[100]; /*定义两个字符数组*/
pipe(fd); /*创建一个管道*/
while(pid=fork())!=-1; /*如果进程创建不成功，则空循环*/
if(pid==0){      /*如果子进程创建成功，pid为进程号*/
lockf(fd[1],1,0); /*锁定管道*/
sprintf(outpipe,"child process is sending message! "); /*把串放入数组outpipe中*/
write(fd[1],outpipe,50); /*向管道写长为50字节的串*/
sleep(5); /*自我阻塞5秒*/
lockf(fd[1],0,0); /*解除管道的锁定*/
exit(0);
}
```



```

else{
    wait(0);    /*等待子进程结束*/
    read(fd[0],inpipe,50); /*从管道中读长为50字节的串*/
    printf("%s\n",inpipe); /*显示读出的数据*/
    exit(0);    /*父进程结束*/
}
}

```

实验结果:

Ubuntu 14.04.0 LT3 ubuntu ttyl

Llnt: Num Lock on

Ubuntu login: jenny

Password:

Last loginL Sat Dec 20 04:27:14 pst 2015 in ttyl

Welcome to Ubuntu 14.04.3 LTD(CNU/Linux 3.19.0 25 gneric x86_64)

* Documentation: <https://help.ubuntu.com>.

jenny@ buntu:~\$ ls

Desktop Downloads Music public test Unitted Document~

Docmets examples.destop pictures Teplates test~ vides

jcnny@ubuntu:~\$ cp test pipe1.c

jenny@ubuntur:~\$ gcc pipe1.c o pipe1

jenny@ubuntu:~\$./pipe1

child process is sending message!

jenny@ubuntu:~\$ _

4. 命名管道示例程序

使用mknod 创建命名管道:

```

umask(0);
if (mknod("/tmp/fifo",S_IFIFO | 0666) == -1)
{

```

```
    perror("mkfifo error");
    exit(1);
}
```

使用mkfifo 创建命名管道:

```
umask(0);
if (mkfifo("/tmp/fifo",S_IFIFO|0666) == -1)
{
    perror("mkfifo error!");
    exit(1);
}
```

Reader 端:

```
#include<stdlib.h>
#include<stdio.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<errno.h>
#define PATH "/fifo"
#define SIZE 128
int main()
{
    umask(0);
    if (mkfifo (PATH,0666|S_IFIFO) == -1)
    {
        perror ("mkefifo error");
        exit(0);
    }
    int fd = open (PATH,O_RDONLY);
    if (fd<0)
    {
        printf("open fd is error\n");
        return 0;
    }

    char Buf[SIZE];
    while(1){
        ssize_t s = read(fd,Buf,sizeof(Buf));
        if (s<0)
        {
            perror("read error");
        }
    }
}
```

```

    exit(1);
}
else if (s == 0)
{
    printf("client quit! i should quit!\n");
    break;
}
else
{
    Buf[s] = '\0';
    printf("client# %s ",Buf);
    fflush(stdout);
}
}
close (fd);
return 3;
}

```

Writer 端:

```

#include<stdlib.h>
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<string.h>
#include<errno.h>
#include<fcntl.h>

#define PATH "./fifo"
#define SIZE 128
int main()
{
    int fd = open(PATH,O_WRONLY);
    if (fd < 0)
    {
        perror("open error");
        exit(0);
    }

    char Buf[SIZE];
    while(1)
    {
        printf("please Enter#.");
    }
}

```

```

fflush(stdout);
ssize_t s = read(0,Buf,sizeof(Buf));
if (s<0)
{
    perror("read is failed");
    exit(1);
}
else if(s==0)
{
    printf("read is closed!");
    return 1;
}
else{
    Buf[s]= '\0';
    write(fd,Buf,strlen(Buf));
}
}
return 0;
}

```

```

File Edit View Search Terminal Help
[weiiixng@localhost Fifo]$ ./writefifo
please Enter#:ibaddda
please Enter#:fsaffsaf
please Enter#:faaf
please Enter#:fadfasf
please Enter#:faadf
please Enter#:fvfd
please Enter#:rgreg
please Enter#:vvvrrrr
please Enter#:

```

```
File Edit View Search Terminal Help
[weixng@localhost Fifo]$ ls
makefile readfifo.c writefifo.c
[weixng@localhost Fifo]$ make all
gcc -o readfifo readfifo.c
gcc -o writefifo writefifo.c
[weixng@localhost Fifo]$ ls
makefile readfifo readfifo.c writefifo writefifo.c
[weixng@localhost Fifo]$ ./readfifo
client# ibadda
client# fsaffsaf http://blog.csdn.net/qq_35116353
client# faaf
client# fadfasf
client# faadf
client# fvfd
client# rgreg
client# vvvvrv
█
```

6.4.4 信号 signal 通信

1. Linux signal/软中断机制及 API

软中断信号（signal，又简称为信号）用来通知进程发生了异步事件。进程之间可以互相通过系统调用 `kill` 发送软中断信号。内核也可以因为内部事件而给进程发送信号，通知进程发生了某个事件。注意，信号只是用来通知某进程发生了什么事情，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。

2. 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。

(1) 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`。

(2) 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，即该信号发送组，则该错误表示组中有成员进程不能接收该信号。

(3) `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）。

下面给出一个程序示例。

```
#include <unistd.h>
```

```
#include <stdio.h>
```

```
#include <signal.h>
```

```
void sigroutine(int unused) {
```

```
    printf("Catch a signal SIGINT ");
```

```
}
```

```
int main() {
```

```
    signal(SIGINT, sigroutine);
```

```
    pause();
```

```
    printf("receive a signal ");
```

```
}
```

在这个例子中，程序开始执行，就象进入了死循环一样，这是因为进程正在等待信号，

当我们按下 Ctrl-C 时，信号被捕捉，并且使得 pause 退出等待状态。

(4) 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，如果要多发多次，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_VIRTUAL 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。

3. 实验内容

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。

4. 相关函数

- `signal` 函数

`signal(sig,function)`: 捕捉中断信号 `sig` 后执行 `function` 规定的操作。

头文件为: `#include <signal.h>`

参数定义: `signal(sig,function)`

`int sig;`

`void (*func) ();`

其中 `sig` 共有 19 个值

值	名字	说明
01	SIGHUP	挂起
02	SIGINT	中断, 当用户从键盘键入 “del”键时
03	SIGQUIT	退出, 当用户从键盘键入 “quit”键时
04	SIGILL	非法指令
05	SIGTRAP	断点或跟踪指令
06	SIGIOT	IOT指令
07	SIGEMT	EMT指令
08	SIGFPE	浮点运算溢出
09	SIGKILL	要求终止进程
10	SIGBUS	总线错误
11	SIGSEGV	段违例, 即进程试图去访问其地址空间以外的地址
12	SIGSYS	系统调用错
13	SIGPIPE	向无读者的管道中写数据
14	SIGALARM	闹钟
15	SIGTERM	软件终止
16	SIGUSR1	用户自定义信号
17	SIGUSR2	用户自定义信号
18	SIGCLD	子进程死
19	SIGPWR	电源故障

这里用到了 3 号, 即相应键盘的停止信号, `ctrl+\`(也可能是 `ctrl + c`), 和 16,17 号自定义信号

- `wait()`函数

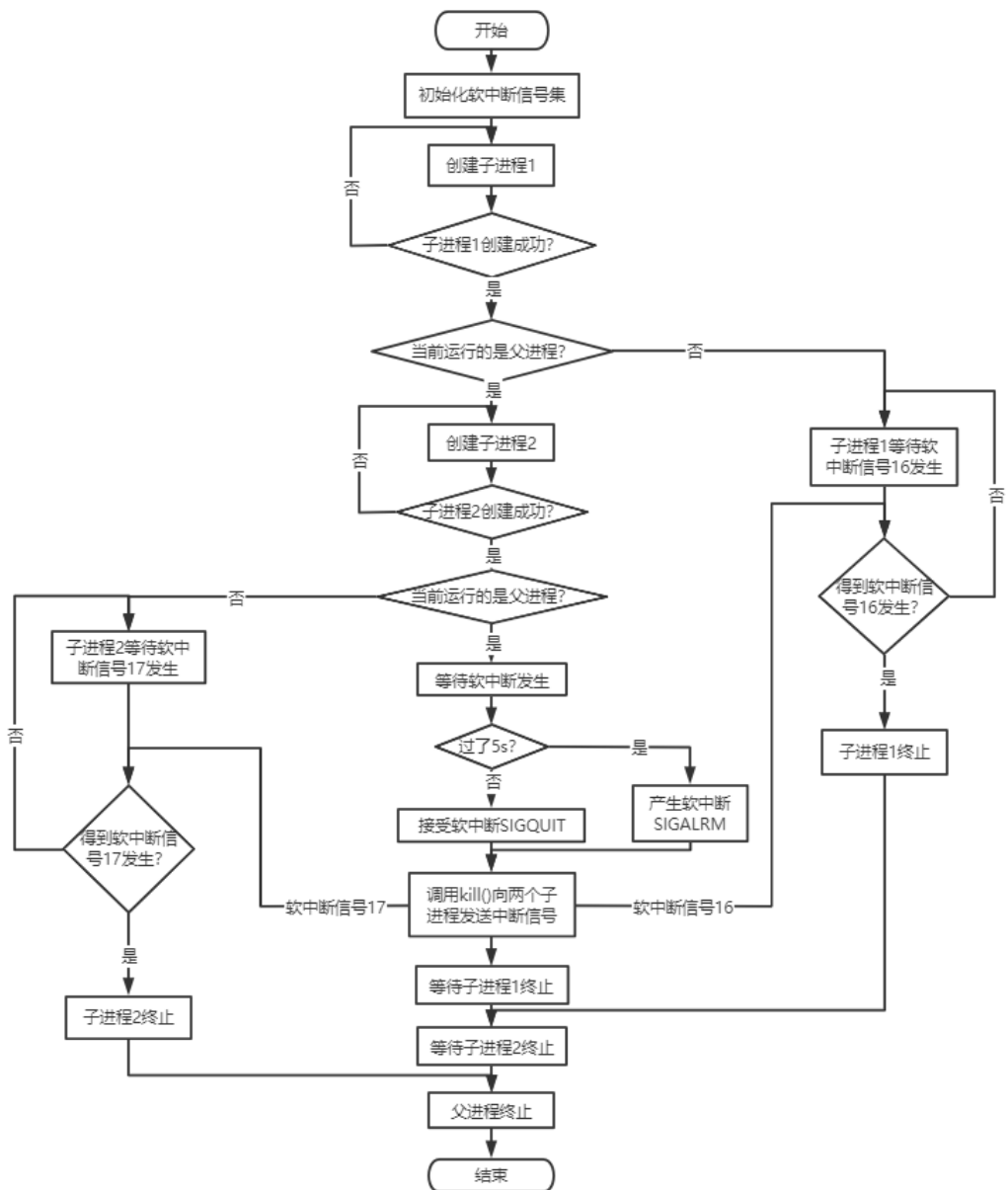
父进程处于阻塞状态,等待子进程终止,其返回值为所等待子进程的进程号

- `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

5. 示例程序通信流程

基于 `signal` 的进程间通信流程图如下所示:



6. 示例程序

示例程序 1:

```

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/wait.h>
#include <unistd.h>
#include <sys/types.h>

```

```

int flag_wait=1;

```

```

void stop2()
{

```

```

        flag_wait=0;
        printf("\nson interruption\n");
    }
void stop()
{
    printf("\ninterruption\n");
}

int main()
{
    pid_t pid1,pid2;
    signal(3,stop);

    while((pid1=fork())!=-1);
    if(pid1>0)
    {
        while((pid2=fork())!=-1);
        if(pid2>0)
        {
            sleep(5);
            kill(pid1,16);
            wait(0);
            kill(pid2,17);
            wait(0);
            printf("\nParent process is killed\n");
            exit(0);
        }
        else
        {
            signal(17,stop2);
            while(flag_wait);
            printf("\nchild process 2 is killed\n");
            exit(0);
        }
    }
    else
    {
        signal(16,stop2);
        while(flag_wait);
        printf("\nchild process 1 is killed\n");
        exit(0);
    }
    return 0;
}

```

示例程序 2:

```
# include <stdio.h>
# include <signal>
# include <unistd.h>
# include <sys/types.h>
int wait_flag;
void stop();
main() {
    int pid1, pid2;
    signal (3, stop);// 或者 signal(14, stop);
    while( (pid1 = fork())!=-1);
    if(pid1 > 0) {
        while( (pid2 = fork())!=-1);
        if(pid2 > 0) {
            wait_flag = 1;
            sleep(5);
            kill(pid1, 16);
            kill(pid2, 17);
            wait(0);
            wait(0);
            printf("\n Parent process is killed! !\n");
            exit(0);
        }
        else {
            wait_flag = 1;
            signal (17, stop);
            printf("\n Child process 2 is killed by parent!!\n");
            exit(0);
        }
    }
    else {
        wait_flag = 1;
        signal (16, stop);
        printf("\n Child process 1 is killed by parent!!\n");
        exit(0);
    }
}
```

7. 运行结果及分析

运行结果:

Child process 1 is killed by parent! !

Child process 2 is killed by parent!!

Parent process is killed!!!

或者(运行多次后会出现如下结果)

Child process 2 is killed by parent!!

Child process 1 is killed by parent!!

Parent process is killed! !

简要分析:

(1) 上述程序中, 调用函数 `signal()`都放在一段程序的前面部分, 而不是在其他接收信号处, 这是因为 `signal()`的执行只是为进程指定信号量 16 和 17 的作用, 以及分配相应的与 `stop()`过程链接的指针。因而 `signal()`函数必须在程序前面部分执行。

(2) 该程序段前面部分用了两个 `wait(0)`, 这是因为父进程必须等待两个子进程终止后才终止。`wait()`函数常用来控制父进程与子进程的同步。在父进程中调用 `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, 系统会向父进程发出 `SIGCHLD` 信号。当接收到信号后, 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。

(3) 该程序中每个进程退出时都用了语句 `exit(0)`, 这是进程的正常终止。在正常终止时, `exit()`函数返回进程结束状态。异常终止时, 则由系统内核产生一个代表异常终止原因的终止状态, 该进程的父进程都能用 `wait()`函数得到其终止状态。在子进程调用 `exit ()`函效后, 子进程的结束状态会返回给系统内核, 由内核根据状态字生成终止状态, 供父进程在 `wait()`函数中读取数据。若子进程结束后, 父进程还没有读取子进程的终止状态, 则系统将于进程的终止状态置为“ZOMBIE”并保留子进程的进程控制块等信息, 等父进程读取信息后, 系统才彻底释放子进程的进程控制块。若父进程在子进程结束之前就结束, 则子进程变孤儿进程”, 系统进程 `init` 会自动“收养”该子进程, 成为该子进程的父进程, 即父进程标识号变为 1, 当子进程结束时, `init` 会自动调用 `wait ()`函数读取子进程的遗留数据, 从而避免系统中留下大量的垃圾。

(4) 上述结果中"Child process 1 is killed by parent!" 和"Child process 2 is killed by parent!" 的出现, 当运行几次后, 谁在前谁在后是随机的, 这是因为从进程调度的角度看, 子建任被创建后处于就绪态, 此时, 父讲程和子进程作为两个独立的进程, 共享同一个代码段, 参加调度、执行直至进程结束, 但是谁会先得到调度, 与系统的调度策略和系统当前的资源状态

有关，是不确定的，因此，谁先从 `fork()` 函数中返回继续执行后面的语句也是不确定的。

6.5 Linux 线程通信示例

线程间通信分为两种情况：

- (1) 分属于不同进程的线程间的通信。可以采用进程间通信机制，如消息队列；
- (2) 位于同一进程内的不同线程间的通信。此时，线程可以通过共享的进程全局数据结构进行快速通信，但需要考虑采用合适的同步互斥机制，以保证访问结果的正确，这类似于进程间的同步与互斥。

一般认为，Linux 系统中，线程间通信方式包括：

✧ 锁机制：包括互斥锁、条件变量、读写锁和自旋锁。

互斥锁确保同一时间只能有一个线程访问共享资源。当锁被占用时试图对其加锁的线程都进入阻塞状态(释放 CPU 资源使其由运行状态进入等待状态)。当锁释放时哪个等待线程能获得该锁取决于内核的调度。

读写锁当以写模式加锁而处于写状态时任何试图加锁的线程(不论是读或写)都阻塞，当以读状态模式加锁而处于读状态时“读”线程不阻塞，“写”线程阻塞。读模式共享，写模式互斥。

条件变量以原子的方式阻塞进程，直到某个特定条件为真为止。对条件的测试是在互斥锁的保护下进行的。条件变量始终与互斥锁一起使用。

自旋锁上锁受阻时线程不阻塞而是在循环中轮询查看能否获得该锁，没有线程的切换因而没有切换开销，不过对 CPU 的霸占会导致 CPU 资源的浪费。所以自旋锁适用于并行结构(多个处理器)或者适用于锁被持有时间短而不希望在线程切换产生开销的情况。

✧ 信号量机制(Semaphore)：包括无名线程信号量和命名线程信号量

✧ 信号机制(Signal)：类似进程间的信号处理

在本次课程设计中，对线程间的通信，考虑一种特殊情况，即线程间参数传递，具体见“6.2.7 Linux-c/c++线程间参数传递示例”中的【示例 2-3-3】~【示例 2-3-5】。

6.6 进程同步互斥示例

6.6.1 System V 信号量操作原语

✧ 创建一个新信号量或取得一个已有信号量 **semget**

```
#include <sys/types.h>
```

```
#include <sys/ipc.h>
```

```
#include <sys/sem.h>
```

```
int semget(key_t key, int num_sems, int sem_flags);
```

semget 函数成功返回一个相应信号标识符（非零），失败返回-1。

第一个参数 **key** 是整数值（唯一非零），不相关的进程可以通过它访问一个信号量，它代表程序可能要使用的某个资源，程序对所有信号量的访问都是间接的，程序先通过调用 **semget** 函数并提供一个键，再由系统生成一个相应的信号标识符（**semget** 函数的返回值），只有 **semget** 函数才直接使用信号量键，所有其他的信号量函数使用由 **semget** 函数返回的信号量标识符。如果多个程序使用相同的 **key** 值，**key** 将负责协调工作。

第二个参数 **num_sems** 指定需要的信号量数目，它的值几乎总是 1。

第三个参数 **sem_flags** 是一组标志，当想要当信号量不存在时创建一个新的信号量，可以和值 **IPC_CREAT** 做按位或操作。设置了 **IPC_CREAT** 标志后，即使给出的键是一个已有信号量的键，也不会产生错误。而 **IPC_CREAT | IPC_EXCL** 则可以创建一个新的，唯一的信号量，如果信号量已存在，返回一个错误。

✧ 信号量初始化 **semctl**

```
int semctl (int semid, int semnum, int cmd,...);
```

//semid 操作句柄 semnum 指定要操作几个信号量，

//cmd 具体的操作（SETVAL 设置单个信号量初值，SETALL 设置所有信号量初值，semnum 的值会被忽略）

//... 不定参数，这里是一个结构体获取信号量信息，一个联合获取信号量的值。

```
union semun {
```



```

int            val;    /* Value for SETVAL */

struct semid_ds *buf;   /* Buffer for IPC_STAT, IPC_SET */

unsigned short *array; /* Array for GETALL, SETALL */

struct seminfo  *__buf; /* Buffer for IPC_INFO
                        (Linux-specific) */

};

```

✧ 信号量 wait()/signal()操作 semop

(1)访问共享资源前，获取信号量-1 操作

```

//semid 句柄，

//struct sembuf, containing the following members:

// unsigned short sem_num; /* 信号量集合中的信号量编号，0 代表第 1 个信号量 */

//  short            sem_op;

/*若 op>0 进行 V 操作信号量值加 op，表示进程释放控制的资源*/

/*若 op<0 进行 P 操作信号量值减 op*/

//  short            sem_flg;

/* 设置信号量的默认操作，IPC_NOWAIT 设置信号量操作不等待*/

/*SEM_UNDO 选项会让内核记录一个与调用进程相关的 UNDO 记录，如果该进程崩溃，则根据这个进程的 UNDO 记录自动恢复相应信号量的计数值*/

struct sembuf buf;

buf.sem_num = 0;

buf.sem_op = -1;

buf.sem_flg = SEM_UNDO;

semop(id, &buf, 1);

```

(2)访问共享资源结束，释放资源，信号量+1 操作

```

//semid 句柄，

//struct sembuf, containing the following members:

//unsigned short sem_num; /* semaphore number */

//  short            sem_op; /* semaphore operation */

//  short            sem_flg; /* operation flags */

```

```
struct sembuf buf;

buf.sem_num = 0;

buf.sem_op = 1;

buf.sem_flg = SEM_UNDO;

semop(id, &buf, 1);
```

✧ 信号量删除/释放操作 **semctl**

```
int semctl (int semid,int semnum,int cmd,...);

//释放也使用 semctl, 但 cmd 的具体操作是 IPC_RMID
```

6.6.2 基于 System V 信号量的生产者-消费者问题 1

```
include<unistd.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<fcntl.h>

#include<stdlib.h>

#include<stdio.h>

#include<string.h>

#include<sys/sem.h>

#define KEY (key_t)14010322

union semun{

    int val;

    struct semid_ds *buf;

    unsigned short *array;

};

static int sem_id = 0;
```

```

static int set_semvalue();

static void del_semvalue();

static void semaphore_p();

static void semaphore_v();


int main(int argc, char *argv[])
{
    //创建信号量,利用 semget 函数
    //初始化信号量, 利用 semctl 函数

    int semid;

    int product = 1;

    int i;

    if((semid = semget(KEY,3,IPC_CREAT|0660))==-1)
    {
        printf("ERROR\n");

        return -1;
    }

    union semun arg[3];

    arg[0].val = 1;//mutex = 1;

    arg[1].val = 5;//empty = 5;

    arg[2].val = 0;//full = 0;


    for(i=0;i<3;i++)
    {
        semctl(semid,i,SETVAL,arg[i]);
    }


    for(i=0;i<3;i++)
    {
        printf("The semval(%d) = %d\n",i,semctl(semid,i,GETVAL,NULL));
    }
}

```

```
}
```

```
pid_t p1,p2;

if((p1=fork())==0){
    while(1){
        //生产者.....
        semaphore_p(semid,1); //P(empty)
        printf("1\n");
        semaphore_p(semid,0); //P(mutex)
        printf("2\n");
        product++;
        printf("Producer %d: %d things",getpid(),product);
        semaphore_v(semid,0); //V(mutex);
        semaphore_v(semid,2); //V(full);

        sleep(2);
    }
}else{
    if((p2=fork())==0){
        while(1){
            sleep(2);

            semaphore_p(semid,2);//p(full)
            printf("3\n");
            semaphore_p(semid,0);//p(mutex)
            printf("4\n");
            product--;
            printf("Consumer1 %d: %d things",getpid(),product);
            semaphore_v(semid,0);//v(mutex)
            semaphore_v(semid,1);//v(empty)

            sleep(5);
```

```

        }
    }

    else{
        while(1){
            sleep(2);

            semaphore_p(semid,2);//p(full)
            printf("5\n");

            semaphore_p(semid,0);//p(mutex)
            printf("6\n");

            product--;

            printf("Consumer2 %d: %d things",getpid(),product);

            semaphore_v(semid,0);//v(mutex)

            semaphore_v(semid,1);//v(empty)

            sleep(5);

        }
    }
}

```

```

static void del_semvalue()
{
    //删除信号量

    union semun sem_union;

    if(semctl(sem_id, 0, IPC_RMID, sem_union) == -1)
        fprintf(stderr, "Failed to delete semaphore\n");
}

```

```

static int set_semvalue()
{

```

```

//用于初始化信号量，在使用信号量前必须这样做

union semun sem_union;

sem_union.val = 1;

if(semctl(sem_id, 0, SETVAL, sem_union) == -1)

    return 0;

return 1;

}

void semaphore_p(int sem_id,int semNum)

{

    //对信号量做减 1 操作，即等待 P（sv）

    struct sembuf sem_b;

    sem_b.sem_num = semNum;

    sem_b.sem_op = -1;//P()

    sem_b.sem_flg = SEM_UNDO;

    // semop(semid,&sem_b,1);

    if(semop(sem_id, &sem_b, 1) == -1)

    {

        fprintf(stderr, "semaphore_p failed\n");

        return;

    }

    return;

}

void semaphore_v(int sem_id,int semNum)

{

    //这是一个释放操作，它使信号量变为可用，即发送信号 V（sv）

    struct sembuf sem_b;

    sem_b.sem_num = semNum;

    sem_b.sem_op = 1;//V()

```

```

sem_b.sem_flg = SEM_UNDO;
// semop(semid,&sem_b,1);
if(semop(sem_id, &sem_b, 1) == -1)
{
    fprintf(stderr, "semaphore_p failed\n");
    return;
}
return ;
}

```

上述程序实现了缓冲区为 5，一个生产者和两个消费者的同步。定义 3 个信号量，full，empty，mutex，分别表示产品个数，缓冲区空位个数，对缓冲区进行操作的互斥信号量，对应的初始化值分别为 0，5，1。

生产者业务逻辑：P（empty）---->P（mutex）----->V（mutex）----->V（full）

消费者业务逻辑：P（full）----->P（mutex）----->V（mutex）----->V（empty）

6.6.3 基于信号量的生产者-消费者问题 2

以生产者进程不断向数组添加数据(写入 100 次)，消费者从数组读取数据并求和为例，给出基于信号量解决生产者-消费者问题的程序框架。该程序假设有一个生产者进程和两个消费者进程，创建了 fullid、emptyid 和 mutexid 共 3 个信号量，供进程间同步访问临界区。同时，还建立 4 个共享主存区，其中 array 用于维护生产者、消费者进程之间的共享数据，sum 保存当前求和结果，而 set 和 get 分别记录当前生产者进程和消费者进程的读写次数。

```

#include <sys/mman.h>
#include <sys/types.h>
#include <linux/sem.h>
#include <fcntl.h>
#include <unistd.h>

```

```

#include <stdio.h>

#include <stdlib.h>

#include <errno.h>

#include <time.h>

#define MAXSEM 5

//声明三个信号量 ID

int fullid;//

int emptyid;

int mutexid;


int main()

{

//fullid、 emptyid 和 mutexid 共 3 个信号量，供进程间同步访问临界区。

    struct sembuf P, V;

    union semun arg;

//声明共享主存，实际就是一个数组

    int *array;//用于维护生产者、消费者进程之间的共享数据

    int *sum;//保存当前求和结果

    int *set;//生产者进程

    int *get;//消费者进程的读写次数

//映射共享主存

    array = (int *)mmap(NULL, sizeof( int ) * MAXSEM, PROT_READ | PROT_WRITE,
MAP_SHARED | MAP_ANONYMOUS, -1, 0);

    sum = (int *)mmap(NULL, sizeof( int), PROT_READ | PROT_WRITE, MAP_SHARED |
MAP_ANONYMOUS, -1, 0);

    get = (int *)mmap(NULL, sizeof( int), PROT_READ | PROT_WRITE, MAP_SHARED |
MAP_ANONYMOUS, -1, 0);

    set = (int *)mmap(NULL, sizeof( int), PROT_READ | PROT_WRITE, MAP_SHARED |
MAP_ANONYMOUS, -1, 0);

```



```

*sum = 0;

*get = 0;

*set = 0;

//生成信号量

fullid = semget(IPC_PRIVATE, 1, IPC_CREAT | 00666);

emptyid = semget(IPC_PRIVATE, 1, IPC_CREAT | 00666);

mutexid = semget(IPC_PRIVATE, 1, IPC_CREAT | 00666);

//为信号量赋值

arg.val = 0;//信号量的初值

if(semctl(fullid, 0, SETVAL, arg) == -1) perror("semctl setval error");

arg.val = MAXSEM;

if(semctl(emptyid, 0, SETVAL, arg) == -1) perror("semctl setval error");

arg.val = 1;

if(semctl(mutexid, 0, SETVAL, arg) == -1) perror("setctl setval error");

//初始化 P,V 操作

V.sem_num = 0;

V.sem_op = 1;

V.sem_flg = SEM_UNDO;

P.sem_num = 0;

P.sem_op = -1;

P.sem_flg = SEM_UNDO;

printf("fullid:%d ,emptyid:%d ,mutexid:%d .\n",fullid,emptyid,mutexid);

//生产者进程

if(fork() == 0 )

{

    //child 进程

    int i = 0;

    while( i < 100)

    {

        semop(emptyid, &P, 1 );

```

```

        semop(mutxid, &P, 1);

        array[*set % MAXSEM] = i + 1;

        printf("Producer %d\n", array[*set % MAXSEM]);

        (*set)++;

        semop(mutxid, &V, 1);

        semop(fullid, &V, 1);

        i++;

    }

    sleep(10);

    printf("Producer is over");

    exit(0);

}

else

{

//parent 进程

//ConsumerA 进程

    if(fork() == 0)

    {

        while(1)

        {

            if(*get == 100)

                break;

            semop(fullid, &P, 1);

            semop(mutxid, &P, 1);

            *sum += array[*get % MAXSEM];

            printf("The ConsumerA Get Number %d\n", array[*get % MAXSEM] );

            (*get)++;

            if( *get == 100)

                printf("The sum is %d \n ", *sum);

```

```

        semop(mutxid, &V, 1);

        semop(emptyid, &V, 1 );

        sleep(1);
    }

    printf("ConsumerA is over\n");

    exit(0);
}

else
{
//Consumer B 进程

while(1)

{

    if(*get == 100)

        break;

    semop(fullid, &P, 1);

    semop(mutxid, &P, 1);

    *sum += array[( *get) % MAXSEM];

    printf("The ConsumerB Get Number %d\n", array[( *get) % MAXSEM] );

    (*get)++;

    if( *get == 100)

        printf("The sum is %d \n ", *sum);

    semop(mutxid, &V, 1);

    semop(emptyid, &V, 1 );

    sleep(1);

}

    printf("ConsumerB is over\n");

    exit(0);

}

}

```

```
//printf("fullid:%d ,emptyid:%d ,mutexid:%d .\n",fullid,emptyid,mutexid);  
  
return 0;  
  
}
```

6.7 线程同步与互斥示例

6.7.1 Pthread 线程同步互斥机制及 API

Pthread/POSIX 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 semaphore，取值为 0、1；信号量则是教科书中的 counting semaphore，取值为整数。

✧ 互斥锁

互斥锁控制对共享资源的原子操作，互斥锁有两种状态，即上锁和解锁。在同一个时刻只能有一个线程掌握共享资源对应的互斥锁，拥有上锁状态的线程能够对共享资源进行操作。若其它线程希望上锁一个已经被上锁的互斥锁，则该线程就会被挂起，直到上锁的线程释放掉互斥锁。

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()

有三种类型的互斥锁：快速互斥锁、递归互斥锁和检错互斥锁，三种锁的区别在于：当其它未占有互斥锁的线程希望得到互斥锁时是否需要阻塞等待。快速互斥锁是指调用线程会阻塞直至拥有互斥锁的线程解锁为止；递归互斥锁能够成功地返回，并且增加调用线程在互斥上加锁的次数而检错互斥锁则为快速互斥锁的非阻塞版本，它会立即返回并返回一个错误信息。默认属性为快速互斥锁。

所需头文件	#include<pthread.h>	
函数原型	int pthread_mutex_init(pthread_mutex_t *mutex,const pthread_mutexattr_t *mutexattr)	
函数参数	mutex:互斥锁标识符	
	mutexattr	PTHREAD_MUTEX_INITIALIZER: 创建快速互斥锁
		PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP: 创建递归互斥锁
		PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP: 创建检错互斥锁
函数返回值	成功: 0	
	出错: 返回错误码	

表 1 互斥锁初始化

所需头文件	#include<pthread.h>	
函数原型	int pthread_mutex_lock(pthread_mutex_t *mutex) int pthread_mutex_trylock(pthread_mutex_t *mutex) int pthread_mutex_unlock(pthread_mutex_t *mutex) int pthread_mutex_destroy(pthread_mutex_t *mutex)	
函数传入值	mutex:互斥锁标识符	
函数返回值	成功: 0	
	出错: -1	

表 2 pthread_mutex_lock()等函数

◇ 信号量

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。经典的同步互斥问题有：生产者-消费者问题，读写者问题，哲学家就餐问题。

Pthread 线程库提供的信号量访问操作有：

- sem_init(), 创建一个信号量，并初始化它的值；
- sem_wait()和 sem_trywait(), 相当于 wait/P 操作，在信号量>0 时，它们能将信号量的值减 1。两者的区别在于信号量<0 时，sem_wait(0 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，同时发出信号来唤醒等待的进程/线程。

- sem_getvalue(), 得到信号量的值。
- sem_destroy(), 删除信号量。

所需头文件	#include<semaphore.h>
函数原型	int sem_init(sem_t *sem,int pshared,unsigned int value)
函数传入值	sem:信号量指针
	pshared:决定信号量能否在几个进程间共享。由于目前 Linux 还没有实现进程间共享信号量，所以这个值只能取 0,表示这个信号量是当前进程的局部信号量
	value:信号量初始化值
函数返回值	成功: 0
	出错: -1

表 3 sem_init()函数

所需头文件	#include<semaphore.h>
函数原型	int sem_wait(sem_t *sem) int sem_trywait(sem_t *sem) int sem_post(sem_t *sem) int sem_getvalue(sem_t *sem) int sem_destroy(sem_t *sem)
函数传入值	sem:信号量指针
函数返回值	成功: 0
	出错: -1

表 4 sem_wait()等函数

6.7.2 基于线程的生产者-消费者问题

通过 Pthread API，创建多个线程，模拟生产者和消费者问题的求解，参考代码如下：

```
#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<pthread.h>

#include<semaphore.h>

#define PRODUCER_NUM 5 //生产者数目

#define CONSUMER_NUM 5 //消费者数目

#define POOL_SIZE      11 //缓冲池大小

int pool[POOL_SIZE];    //缓冲区
```

```

int head=0; //缓冲池读取指针

int rear=0; //缓冲池写入指针

sem_t   room_sem;           //同步信号信号量，表示缓冲区有可用空间
sem_t   product_sem;        //同步信号量，表示缓冲区有可用产品
pthread_mutex_t mutex;

void *producer_fun(void *arg)
{
    while (1)
    {
        sleep(1);

        sem_wait(&room_sem);

        pthread_mutex_lock(&mutex);

        //生产者往缓冲池中写入数据

        pool[rear] = 1;

        rear = (rear + 1) % POOL_SIZE;

        printf("producer %d write to pool\n", (int)arg);

        printf("pool size is %d\n", (rear-head+POOL_SIZE)%POOL_SIZE);

        pthread_mutex_unlock(&mutex);

        sem_post(&product_sem);

    }
}

```

```

void *consumer_fun(void *arg)
{
    while (1)
    {
        int data;

        sleep(10);

        sem_wait(&product_sem);
    }
}

```

```

        pthread_mutex_lock(&mutex);

        //消费者从缓冲池读取数据

        data = pool[head];

        head = (head + 1) % POOL_SIZE;

        printf("consumer %d read from pool\n", (int)arg);

        printf("pool size is %d\n", (rear-head+POOL_SIZE)%POOL_SIZE);

        pthread_mutex_unlock(&mutex);

        sem_post(&room_sem);

    }

}

int main()
{
    pthread_t producer_id[PRODUCER_NUM];

    pthread_t consumer_id[CONSUMER_NUM];

    pthread_mutex_init(&mutex, NULL);    //初始化互斥量

    int ret = sem_init(&room_sem, 0, POOL_SIZE-1); //初始化信号量 room_sem 为缓冲池大小
    if (ret != 0)
    {
        printf("sem_init error");

        exit(0);

    }

    ret = sem_init(&product_sem, 0, 0); //初始化信号量 product_sem 为 0，开始时缓冲池中没
    有数据
    if (ret != 0)
    {
        printf("sem_init error");

        exit(0);

    }
}

```



```
for (int i = 0; i < PRODUCER_NUM; i++)
{
    //创建生产者线程

    ret = pthread_create(&producer_id[i], NULL, producer_fun, (void*)i);
    if (ret != 0)
    {
        printf("producer_id error");
        exit(0);
    }
    //创建消费者线程

    ret = pthread_create(&consumer_id[i], NULL, consumer_fun, (void*)i);
    if (ret != 0)
    {
        printf("consumer_id error");
        exit(0);
    }
}
for(int i=0;i<PRODUCER_NUM;i++)
{
    pthread_join(producer_id[i],NULL);
    pthread_join(consumer_id[i],NULL);
}

exit(0);
}
```

6.7.3 线程间同步互斥

<https://blog.csdn.net/mybelief321/article/details/9390707>

在示例 2-2-6 中，创建了 3 个线程，为了更好的描述线程之间的并行执行，让 3 个线程共用同一个执行函数。每个线程都有 4 次循环(可以看成 4 个小任务)，每次循环之间会随机等待 1~10s 的时间，意义在于模拟每个任务的到达时间是随机的，并没有任何特定的规律。但是可以看到执行结果是无序的。本示例在原有代码的基础上增加互斥锁功能，实现原本独立与无序的多个线程按顺序执行。

代码如下：

```
/*thread_mutex.c*/
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>

#define THREAD_NUMBER 3
#define REPEAT_NUMBER 4
#define DELAY_TIME_LEVELS 10.0
pthread_mutex_t mutex; /*定义一个互斥锁的全局变量*/

/*线程函数*/
void *thrd_func(void *arg)
{
    int thrd_num=(int)arg;
    int delay_time=0,count=0;
    int res;
    /*互斥锁上锁*/
    res=pthread_mutex_lock(&mutex);
    if(res)
    {
        printf("Thread %d lock failed\n",thrd_num);
        pthread_exit(NULL);
    }
    printf("Thread %d is starting\n",thrd_num);
    for(count=0;count<REPEAT_NUMBER;count++)
    {
        delay_time=(int)(rand()*DELAY_TIME_LEVELS/(RAND_MAX))+1;
        sleep(delay_time);
        printf("\tThread %d: job %d delay=%d\n",thrd_num,count,delay_time);
    }
    printf("Thread %d finished\n",thrd_num);
}
```

```

    pthread_exit(NULL);/*线程退出*/
}

int main(void)
{
    pthread_t thread[THREAD_NUMBER];
    int no,res;
    void *thrd_ret;

    srand(time(NULL));
    /*互斥锁初始化*/
    pthread_mutex_init(&mutex,NULL);
    for(no=0;no<THREAD_NUMBER;no++)
    {
        res=pthread_create(&thread[no],NULL,thrd_func,(void*)(no));
        if(res!=0)
        {
            printf("Create thread %d failed\n",no);
            exit(res);
        }
    }
    printf("Create threads success\nWaiting for threads to finish...\n");
    for(no=0;no<THREAD_NUMBER;no++)
    {
        res=pthread_join(thread[no],&thrd_ret);
        if(!res)
        {
            printf("Thread %d joined\n",no);
        }
        else
        {
            printf("Thread %d join failed\n",no);
        }
    }

    /*互斥锁解锁*/
    pthread_mutex_unlock(&mutex);
}

```

6.7.4 抢票程序

抢票实现逻辑：票数作为全局共享数据，共有一百张；创建 4 个线程模拟黄牛来抢票倒卖；

程序 1. 带有 race condition，无法保证逻辑正确

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

#define THREADCOUNT 4 //四个线程
int g_tickes = 100; //一百张票
void* ThreadStart(void* arg)
{
    (void)arg;
    while(1)
    {
        if(g_tickes > 0)
        {
            g_tickes--; //减减操作代表被抢
            usleep(100000); //模拟阻塞抢
            printf("i am thread [%p], i have ticket num is [%d]\n", pthread_self(), g_tickes + 1);
        }
        else
        {
            break;
        }
    }
    return NULL;
}

int main()
{
    pthread_t tid[THREADCOUNT];
    int i = 0;
    for(; i < THREADCOUNT; i++)
    {
        int ret = pthread_create(&tid[i], NULL, ThreadStart, NULL);
        if(ret < 0)
        {
            perror("pthread_create");
            return 0;
        }
    }
}
```

```

for(i = 0; i < THREADCOUNT; i++)
{
    pthread_join(tid[i], NULL); //等待线程退出回收资源
}
return 0;
}

```

程序执行结果如下：

```

i am thread [0x7f9116a0e700], i have ticket num is [18]
i am thread [0x7f911720f700], i have ticket num is [17]
i am thread [0x7f9118211700], i have ticket num is [16]
i am thread [0x7f9117a10700], i have ticket num is [15]
i am thread [0x7f9116a0e700], i have ticket num is [14]
i am thread [0x7f9118211700], i have ticket num is [13]
i am thread [0x7f911720f700], i have ticket num is [12]
i am thread [0x7f9117a10700], i have ticket num is [12]
i am thread [0x7f9116a0e700], i have ticket num is [10]
i am thread [0x7f9117a10700], i have ticket num is [9]
i am thread [0x7f911720f700], i have ticket num is [8]
i am thread [0x7f9118211700], i have ticket num is [8]
i am thread [0x7f9116a0e700], i have ticket num is [6]
i am thread [0x7f911720f700], i have ticket num is [5]
i am thread [0x7f9117a10700], i have ticket num is [4]
i am thread [0x7f9118211700], i have ticket num is [4]
i am thread [0x7f9116a0e700], i have ticket num is [2]
i am thread [0x7f9118211700], i have ticket num is [1]
i am thread [0x7f9117a10700], i have ticket num is [1]
i am thread [0x7f9116a0e700], i have ticket num is [1]
i am thread [0x7f911720f700], i have ticket num is [1]

```

有的票都没被抢

四个线程都访问了同一张票号

https://blog.csdn.net/qq_44785014

存在问题及原因：

四个线程对临界资源的访问并非预期的按序访问，因为线程是抢占式执行的，而减减操作也是非原子操作，可能票已经被抢了，但是还没打印，时间片到了，或者刚一减减，还没来的急在内存中更新数据，就再次被其他执行流访问，造成多次访问操作，从而造成程序结果的二义性。具体地，从代码逻辑分析发现：

- if 语句判断条件为真以后，代码可以并发的切换到其他线程
- usleep 这个模拟等待的过程，在这个漫长的业务过程中，可能有很多个线程会进入该代码段
- 减减操作本身就不是一个原子操作，可以被打断

程序 2. 通过基于锁的互斥，实现对票数的正确访问

```
#include <stdio.h>
```

```

#include <unistd.h>

#include <pthread.h>

#define THREADCOUNT 4 //模拟四个线程执行流

int g_tickes = 100; //访问全局变量临界资源

pthread_mutex_t lock; //定义互斥量

void* ThreadStart(void* arg)
{
    (void)arg;

    while(1)
    {
        // 1 加锁

        pthread_mutex_lock(&lock);

        if(g_tickes > 0)
        {
            g_tickes--;

            usleep(100000); //模拟等待让线程睡眠一段时间，就是为了防止一个线程始终
            占据此函数。

            printf("i am thread [%p], i have ticket num is [%d]\n", pthread_self(), g_tickes + 1);
        }
        else
        {
            //假设有一个执行流判断了 g_tickets 之后发现，g_tickets 的值是小于等于 0 的
            //则会执行 else 逻辑，直接就被 break 跳出 while 循环

            //跳出 while 循环的执行流还加着互斥锁

            //所以在所有有可能退出线程的地方都需要进行解锁操作

            pthread_mutex_unlock(&lock);

            break;
        }

        pthread_mutex_unlock(&lock); //用完后解锁
    }
}

```

```

    }

    return NULL;
}

int main()
{
    pthread_mutex_init(&lock, NULL); //初始化互斥锁
    pthread_t tid[THREADCOUNT]; //声明线程
    int i = 0;
    for(; i < THREADCOUNT; i++)
    {
        int ret = pthread_create(&tid[i], NULL, ThreadStart, NULL); //创建线程
        if(ret < 0)
        {
            perror("pthread_create");
            return 0;
        }
    }
    for(i = 0; i < THREADCOUNT; i++)
    {
        pthread_join(tid[i], NULL); //等待线程退出回收资源
    }

    pthread_mutex_destroy(&lock); //销毁互斥锁
    return 0;
}

```

6.7.5 PThread 双线程打印

1. 实验目的及要求

- (1) 掌握互斥锁的概念。
- (2) 掌握编写线程程序的方法。
- (3) 了解线程的调度和执行过程。

2. 实验环境

Linux 或者 MacOs 10.9.5,用 C++语言编写。

3. 实验内容

- (1)在 C++中创建 2 个线程,实现依次打印 0~100 的任务。
- (2)在执行当中加入互斥锁,使 2 个进程互斥执行打印,同步完成任务操作。

4. 实验步骤

- (1) 首先创建一个线程, 它的目的是对操作数执行减 1 的操作。
- (2)再创建一个同样操作的进程 2。
- (3)为了使两个进程互斥执行, 需要利用互斥锁的概念,首先声明锁 `pthread_mutex_t mutex`, 然后利用 `pthread_mutex_lock(&mutex)`将进程锁住, 防止其他进程争夺资源,最后,利用 `pthread_mutex_unlock(&mutex)`来解锁,使得其他进程访问资源。
- (4)主函数部分。 CPU 会动态分配任务给两个线程,线程 `tprocess1` 和 `tprocess2` 交替打印。

5. 实验代码

```
int num=100;

pthread_mutex_t mutex;    声明锁函数

void* tprocess1(void*args){

while (num>0){

    pthread_mutex_lock(&mutex);    上锁

    int i=num;

    printf("tprocess 1--%d\n",i);
```



```

        i--;

        num=i;

        pthread_mutex_unlock(&mutex);    开锁
    }

    return NULL;

}

void* tprocess2(void* args){
while (num>0){

    pthread_mutex_lock(&mutex);    上锁

    int i=num;

    printf("tprocess2--%d\n",i);

    i--;

    num=i;

    pthread_mutex_unlock(&mutex);    开锁
}

return NULL;

}

int main(int argc, const char* argv[]) {
pthread_mutex_init(&mutex, NULL);

pthread_t t1;        创建进程 1
pthread_t t2;        创建进程 2

pthread_create(&t1, NULL, tprocess1, NULL);

pthread_create(&t2, NULL, tprocess2, NULL);

pthread_join(t1, NULL);    关联进程 1,只有当进程 1 结束后,所有进程才能全部结束

return 0;

}

```

6. 实验结果

.....

tprocess1--5
tprocess2--4
tprocess1--3
tprocess2--2
tprocess1--1
tprocess2--0

6.8 基于 C++ 中互斥锁和条件变量的同步互斥

6.8.1 互斥锁/互斥量 mutex 及程序示例

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()`。

成员函数

- 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。

程序示例 1：

```
#include <iostream>
```

```
#include <mutex>
```

```
#include <thread>
```

```
using namespace std;
```

```
volatile int counter(0); // non-atomic counter
```

//volatile 表示不进行优化 多次循环不把 i 保存在寄存器，每次从内存获取。

```
mutex mtx; // locks access to counter
```

```
void increase10Ktime()
```

```
{
```

```
    for (int i = 0; i < 10000; i++)
```

```
    {
```

```
        mtx.lock();
```

```
        counter++;
```

```

        mtx.unlock();
    }
}

int main()
{
    thread ths[10];
    for (int i = 0; i < 10; i++)
    {
        ths[i] = thread(increase10Ktime);
    }
    for (auto& th: ths)
        th.join();

    cout << "after successful increase : " << counter << endl;
    return 0;
}

```

运行结果:

```
after successful increase :100000
```

程序示例 2:

```

#include <iostream>
#include <mutex>
#include <thread>

```

```
using namespace std;
```

```
volatile int counter(0); // non-atomic counter
```

//volatile 表示不进行优化 多次循环不把 i 保存在寄存器，每次从内存获取。

```

mutex mtx; // locks access to counter
void increase10Ktime()
{
    for (int i = 0; i < 10000; i++)
    {

```

```

        counter++;
    }
}

int main()
{
    thread ths[10];
    for (int i = 0; i < 10; i++)
    {
        ths[i] = thread(increase10Ktime);
    }
    for (auto& th : ths)
        th.join();

    cout << "after successful increase : " << counter << endl;
    return 0;
}

```

运行结果：

```
after successful increase :93256
```

程序示例 3：try_lock/unclok

```

#include <iostream>

#include <mutex>

#include <thread>

using namespace std;

volatile int counter(0); // non-atomic counter

mutex mtx; // locks access to counter

void increase10Ktime()
{
    for (int i = 0; i < 10000; i++)
    {
        while (!mtx.try_lock()); //没有锁死等
    }
}

```

```

        counter++;
        mtx.unlock();
    }
}

int main()
{
    thread ths[10];
    for (int i = 0; i < 10; i++)
    {
        ths[i] = thread(increase10Ktime);
    }
    for (auto& th : ths)
        th.join();
    cout << "after successful increase : " << counter << endl;
    return 0;
}

```

运行结果：

```
after successful increase :100000
```

程序示例 4: lock_guard

```

#include <iostream>
#include <thread>
#include <mutex>

using namespace std;

mutex mtx;

void printEven(int i)
{
    if (i % 2 == 0)
        cout << i << " is even" << endl;
    else

```

```

        throw logic_error("not even");    //抛出异常
    }

void printThreadId(int id)
{
    try {
        //RAII:   mtx 资源获取即初始化。
        lock_guard<mutex> lck(mtx); //栈自旋抛出异常时栈对象自我析构。
        printEven(id);
    }
    catch (logic_error&) {
        cout << "exception caught" << endl;
    }
    //离开当前作用于自动释放 mtx 锁。
}

int main()
{
    thread ths[10]; //spawn 10 threads
    for (int i = 0; i < 10; i++)
    {
        ths[i] = thread(printThreadId, i + 1);
    }
    for (auto& th : ths)
        th.join();
    return 0;
}

```

运行结果：

```
exception caught
4 is even
exception caught
6 is even
exception caught8 is even

10 is even
exception caught
2 is even
exception caught
```

CSDN @Code-Beginner

可以看到：当前线程抛出异常不影响其他线程。

程序示例 5:

与程序示例 4 对比，不使用 `lock_guard<mutex>`

```
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;
mutex mtx;
void printEven(int i)
{
    if (i % 2 == 0)
        cout << i << " is even" << endl;
    else
        throw logic_error("not even");    //抛出异常
}
void printThreadId(int id)
{
    try {
        mtx.lock();
        printEven(id);
        mtx.unlock();
    }
    catch (logic_error&) {
        cout << "exception caught" << endl;
    }
}
```

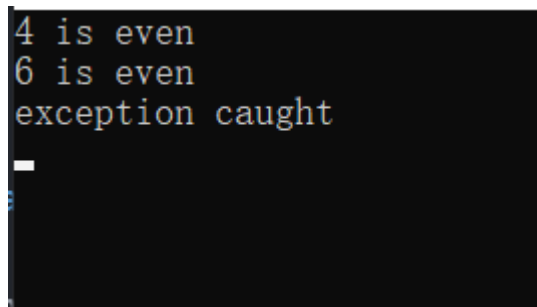


```

    }
}
int main()
{
    thread ths[10]; //spawn 10 threads
    for (int i = 0; i < 10; i++)
    {
        ths[i] = thread(printThreadId, i + 1);
    }
    for (auto& th : ths)
        th.join();
    return 0;
}

```

运行结果：



```

4 is even
6 is even
exception caught
_

```

抛出异常之后，抛出异常的线程只有加锁，没有释放锁的过程。只要有一个线程没有释放，和这个锁相关的所有线程都会阻塞。

6.8.2 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。

成员函数

(构造函数)	构造对象 (公开成员函数)
(析构函数)	析构对象 (公开成员函数)
<code>operator=</code> [被删除]	不可复制赋值 (公开成员函数)

通知

<code>notify_one</code>	通知一个等待的线程 (公开成员函数)
<code>notify_all</code>	通知所有等待的线程 (公开成员函数)

等待

<code>wait</code>	阻塞当前线程，直到条件变量被唤醒 (公开成员函数)
<code>wait_for</code>	阻塞当前线程，直到条件变量被唤醒，或到指定时限时长后 (公开成员函数)
<code>wait_until</code>	阻塞当前线程，直到条件变量被唤醒，或直到抵达指定时间点 (公开成员函数)

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。

6.8.3 基于 `condition_variable` 和 `mutex` 的信号量实现

使用条件变量和互斥锁，通过计数实现信号量：

```
class semaphore {
public:
    semaphore() : count_(0) {}

    void acquire() {
        std::lock_guard<std::mutex> lock(mutex_);
        while(count_ <= 0)
            cv_.wait(lock);
        --count_;
    }

    void release() {
        std::lock_guard<std::mutex> lock(mutex_);
        ++count_;
        cv_.notify_one();
    }
private:
    int count_;
    std::mutex mutex_;
    std::condition_variable cv_;
}
```

6.8.4 基于条件变量的生产者-消费者问题

示例程序 1:

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <thread>
#include <queue>
#include <mutex>

std::mutex g_cvMutex;    //二元互斥信号量

std::condition_variable g_cv; //条件变量

//缓冲区队列/buffer

std::deque<int> g_data_deque;

//缓冲区 buffer 最大数目

const int MAX_NUM = 30;

//缓冲区指针

int g_next_index = 0;

//生产者，消费者线程个数

const int PRODUCER_THREAD_NUM = 3;

const int CONSUMER_THREAD_NUM = 3;

void producer_thread(int thread_id)
{
    while (true)
    {
        std::this_thread::sleep_for(std::chrono::milliseconds(500)); //线程延时、睡眠

        //对缓冲区队列加锁

        std::unique_lock<std::mutex> lk(g_cvMutex);

        //当缓冲区队列未滿时，继续添加数据
```

```

    g_cv.wait(lk, [](){ return g_data_deque.size() <= MAX_NUM; }); //队列未满足
    g_next_index++;    //指针下移
    g_data_deque.push_back(g_next_index); //数据加入队列
    std::cout << "producer_thread: " << thread_id << " producer data: " << g_next_index;
    std::cout << " queue size: " << g_data_deque.size() << std::endl;
    //唤醒其他线程
    g_cv.notify_all();
    //自动释放锁
}
}

void consumer_thread(int thread_id)
{
    while (true)
    {
        std::this_thread::sleep_for(std::chrono::milliseconds(550));
        //对缓冲区队列加锁
        std::unique_lock<std::mutex> lk(g_cvMutex);
        //检测条件是否达成： 队列不为空， 有数据
        g_cv.wait(lk, []{return !g_data_deque.empty(); });
        //互斥操作， 从队列中取数据
        int data = g_data_deque.front();
        g_data_deque.pop_front();
        std::cout << "\tconsumer_thread: " << thread_id << " consumer data: ";
        std::cout << data << " deque size: " << g_data_deque.size() << std::endl;
        //唤醒其他线程
        g_cv.notify_all();
        //自动释放锁
    }
}
}

```

```

int main()
{
    std::thread arrRroducerThread[PRODUCER_THREAD_NUM];
    std::thread arrConsumerThread[CONSUMER_THREAD_NUM];

    for (int i = 0; i < PRODUCER_THREAD_NUM; i++)
    {
        arrRroducerThread[i] = std::thread(producer_thread, i);
    }

    for (int i = 0; i < CONSUMER_THREAD_NUM; i++)
    {
        arrConsumerThread[i] = std::thread(consumer_thread, i);
    }

    for (int i = 0; i < PRODUCER_THREAD_NUM; i++)
    {
        arrRroducerThread[i].join();
    }

    for (int i = 0; i < CONSUMER_THREAD_NUM; i++)
    {
        arrConsumerThread[i].join();
    }

    return 0;
}

```

运行结果：

```
C:\Windows\system32\cmd.exe

producer_thread: 0 producer data: 42 queue size: 5
consumer_thread: 0 consumer data: 38 deque size: 4
consumer_thread: 2 consumer data: 39 deque size: 3
producer_thread: 1 producer data: 43 queue size: 4
producer_thread: 2 producer data: 44 queue size: 5
producer_thread: 0 producer data: 45 queue size: 6
consumer_thread: 1 consumer data: 40 deque size: 5
consumer_thread: 0 consumer data: 41 deque size: 4
consumer_thread: 2 consumer data: 42 deque size: 3
producer_thread: 1 producer data: 46 queue size: 4
producer_thread: 2 producer data: 47 queue size: 5
producer_thread: 0 producer data: 48 queue size: 6
consumer_thread: 1 consumer data: 43 deque size: 5
consumer_thread: 0 consumer data: 44 deque size: 4
consumer_thread: 2 consumer data: 45 deque size: 3
producer_thread: 1 producer data: 49 queue size: 4
producer_thread: 2 producer data: 50 queue size: 5
producer_thread: 0 producer data: 51 queue size: 6
```

生产者

消费者

<https://blog.csdn.net/xiao3404>

示例程序 2:

```
// condition_variable::notify_one
#include <iostream>          // std::cout
#include <thread>             // std::thread
#include <mutex>              // std::mutex, std::unique_lock
#include <condition_variable> // std::condition_variable

std::mutex mtx;
std::condition_variable produce, consume;

int cargo = 0; // shared value by producers and consumers

void consumer () {
    std::unique_lock<std::mutex> lck(mtx);
    while (cargo==0) consume.wait(lck);
    std::cout << cargo << '\n';
    cargo=0;
    produce.notify_one();
}

void producer (int id) {
    std::unique_lock<std::mutex> lck(mtx);
    while (cargo!=0) produce.wait(lck);
    cargo = id;
    consume.notify_one();
}

int main ()
{
```

```
std::thread consumers[10],producers[10];  
// spawn 10 consumers and 10 producers:  
for (int i=0; i<10; ++i) {  
    consumers[i] = std::thread(consumer);  
    producers[i] = std::thread(producer,i+1);  
}  
  
// join them back:  
for (int i=0; i<10; ++i) {  
    producers[i].join();  
    consumers[i].join();  
}  
  
return 0;  
}
```


6.9 多核多线程编程及性能分析

6.9.1 实验 6.1. 观察实验平台物理 cpu、CPU 核和逻辑 cpu 的数目

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：CPU(e. g. 戴尔, i5 Dual-core 双核)，主频2.4G，内存4G

软件：Suse Linux Enterprise 10，内核版本：linux-2.6.16

- 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

方式 1： 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
```

方式 2: Windows 环境下，通过下述程序获取 cpu 逻辑核的数量

(<http://stackoverflow.com/questions/150355/programmatically-find-the-number-of-cores-on-a-machine>)

```
#ifdef _WIN32
#include <windows.h>
#elif MACOS
#include <sys/param.h>
#include <sys/sysctl.h>
#else #include <unistd.h>
```

```

#endif
int getNumCores() {
#ifdef WIN32
    SYSTEM_INFO sysinfo;
    GetSystemInfo(&sysinfo);
    return sysinfo.dwNumberOfProcessors;
#elif MACOS
    int nm[2];
    size_t len = 4;
    uint32_t count;

    nm[0] = CTL_HW;
    nm[1] = HW_AVAILCPU;
    sysctl(nm, 2, &count, &len, NULL, 0);
    if(count < 1) {
        nm[1] = HW_NCPU;
        sysctl(nm, 2, &count, &len, NULL, 0);
        if(count < 1) { count = 1; }
    }
    return count;
#else
    return sysconf(_SC_NPROCESSORS_ONLN);
#endif
}

```

6.9.2 实验 6.2 单线程/进程串行 vs2 线程并行 vs3 线程加锁并行程序

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/进程样例程序

```
#define ORANGE_MAX_VALUE    1000000

#define APPLE_MAX_VALUE     100000000

#define MSECOND             1000000


struct apple
{
    unsigned long long a;
    unsigned long long b;
};


struct orange
{
    int a[ORANGE_MAX_VALUE];
    int b[ORANGE_MAX_VALUE];
};


};


int main (int argc, const char * argv[]) {
    // insert code here...

    struct apple test;
    struct orange test1;

    for(sum=0;sum<APPLE_MAX_VALUE;sum++)
    {
        test.a += sum;
        test.b += sum;
    }
}
```

```
sum=0;

for(index=0;index<ORANGE_MAX_VALUE;index++)
{
    sum += test1.a[index]+test1.b[index];
}

return 0;
}
```

(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。

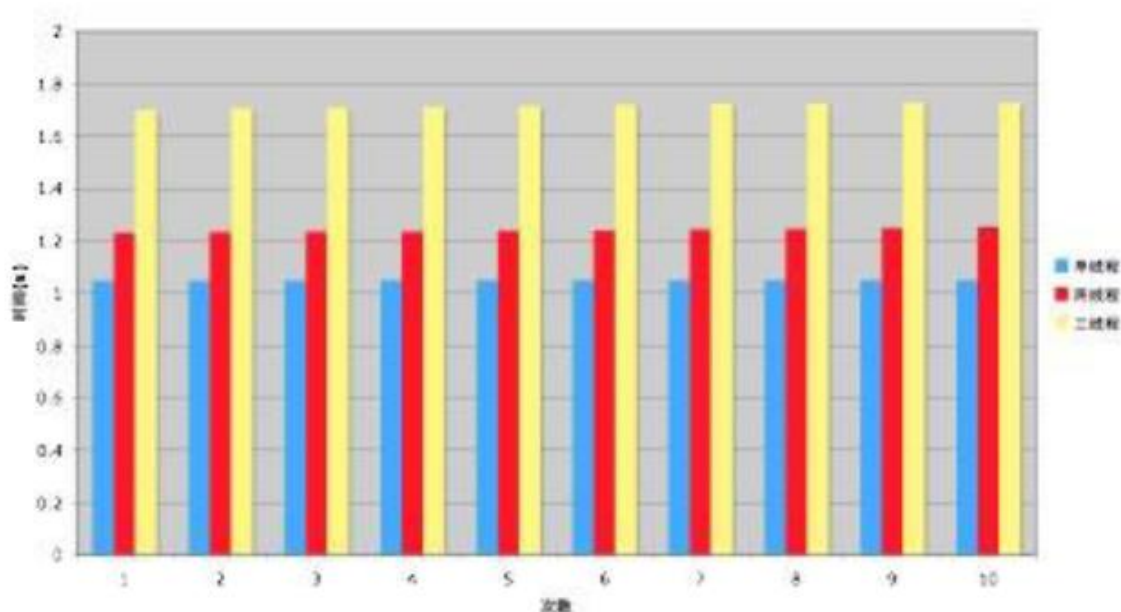
(3) 步骤3.3 线程加锁并行程序

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。

e. g.



6.9.3 实验 6.3 线程加锁 vs 3 线程不加锁 对比

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。。

6.9.4 实验 6.4. 针对 Cache 的优化

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。

实验内容:

- (1) 步骤1. 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,修改其部分代码如下:

清单 4. 针对**Cache**的优化

```
struct apple
{
    unsigned long long a;
    char c[128]; /*32,64,128*/
    unsigned long long b;
};
```

, 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。

- (2) 步骤2. 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。

6.9.5 实验 6.5 CPU 亲和力对并行程序影响

CPU 亲和力可分为两大类: 软亲和力和硬亲和力。



管理处理器的亲和性 (affinity).pd

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。

实验内容：

(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。

程序代码见参考文献“利用多核多线程进行程序优化”。

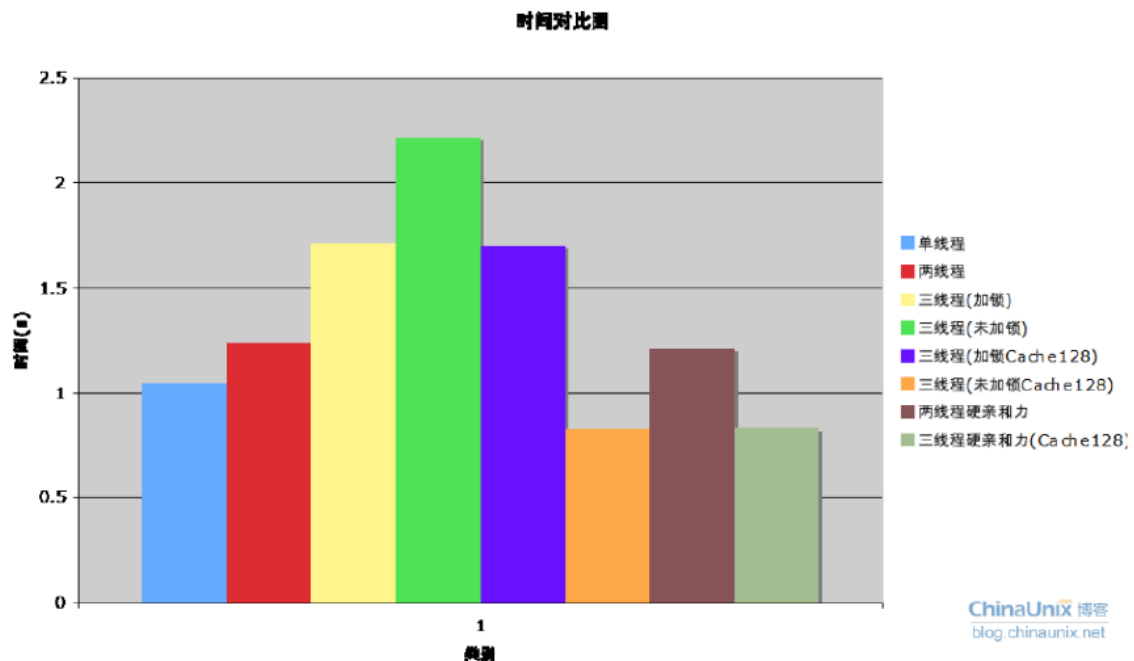
(2) 步骤2

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。

6.9.6 8种实现方案运行时间对比总结

根据前面实验内容，总结对比单线程/进程串程序和其它 7 种多线程并程序的运行时间，并以图表方式给出具体结果。

E.g.



6.9.7 K-Best 测量方法

引自 <http://www.ibm.com/developerworks/cn/linux/l-cn-optimization/index.html>

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 $V_1, V_2, V_3, \dots, V_i, \dots$ ，同时必须满足 $(1 + \epsilon)V_i \geq V_k$
- (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。

在多层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 $K=10$ ， $\varepsilon=2\%$ ， $M=200$ 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。