**操作系统课程设计**

**一、课程设计要求**

1. **按时参加上机实习，不得旷课、迟到、早退。**
2. **每个学生需在Linux下用C或C++完成六道上机实习题。**
3. **最后两次上机实习组织验收，要求每个学生演示六个实验的运行情况，并回答老师的提问；**
4. **每个学生需独立完成上机实习，不得相互抄袭。如发现，抄袭者和被抄袭者成绩均不及格。**
5. **每个学生需在实习结束一周内将实习报告和6个实验的源程序电子档打包后（文件命名为191171张三.rar或.zip）发给班级学习委员，再由学习委员打包发到老师的邮箱，用于检查是否存在抄袭现象。(老师邮箱:51782281@qq.com）**
6. **每个学生需在实习结束一周内提交纸质实习报告一份。****应包括如下内容:**

**1）课程设计题目**

**2）程序功能及设计思路**

**3）数据结构及算法设计**

**4）****程序运行情况**

**5）编程中遇到的困难及解决方法****、实习心得或良好建议**

**二、 评分**

**1、考勤占20%，上机验收占40%，实习报告占40%。**

**2、抄袭程序者和被抄袭程序者，课程设计成绩不及格。**

**3、抄袭报告者和被抄袭报告者，课程设计成绩不及格。**

1. **上机时间地点**

**8周周二晚6:30-9:30**

**9-13周周五晚6:30-9:30**

**地点：未来城校区公教二 五楼机房**

**1-2班 502机房 3-4班503机房**

**502机房（Fedora） 503机房（Ubuntu）已预安装Linux;**

**开机时进入Linux系统，默认使用用户名：suer，密码：123456。若默认密码登录不了，可能是被同学修改过了，换一台电脑使用。请同学们千万不要修改系统管理员登录密码。若有同学自己的笔记本电脑已经装了Linux，欢迎自带电脑上机实习。**

1. **每次上机实习前先自学预备知识，并在作业本上编写初步的源程序。**
2. **课程设计题目**

**课程设计分六个实验，具体如下：**

**实验1：作业调度**

**1.1实验目的**

1、 对作业调度的相关内容作进一步的理解。

2、 明白作业调度的主要任务。

3、 通过编程掌握作业调度的主要算法。

**1.2实验内容**

1、假设系统中可同时运行两道作业，给出每道作业的到达时间和运行时间，如下表所示：

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 作业名 | A | B | C | D | E | F | G | H | I | J |
| 到达时间 | 0 | 2 | 5 | 7 | 12 | 15 | 4 | 6 | 8 | 10 |
| 运行时间 | 7 | 10 | 20 | 30 | 40 | 8 | 8 | 20 | 10 | 12 |

1. 分别用先来先服务算法、短作业优先和响应比高者优先三种算法给出作业的调度顺序。
2. 计算每一种算法的平均周转时间及平均带权周转时间并比较不同算法的优劣。

**预备知识：** 响应比=等待时间/运行时间+1

周转时间=完成时间-到达时间

带权周转时间=周转时间/运行时间

**实验2：磁盘调度**

**2.1实验目的**

1、对磁盘调度的相关知识作进一步的了解，明确磁盘调度的原理。

2、加深理解磁盘调度的主要任务。

3、通过编程，掌握磁盘调度的主要算法。

**2.2实验内容**

1、对于如下给定的一组磁盘访问进行调度：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 请求服务到达 | A | B | C | D | E | F | G | H | I | J | K | L | M | N |
| 访问的磁道号 | 30 | 50 | 100 | 180 | 20 | 90 | 150 | 70 | 80 | 10 | 160 | 120 | 40 | 110 |

2、要求分别采用先来先服务、最短寻道优先以及电梯调度算法进行调度。

3、要求给出每种算法中磁盘访问的顺序，计算出平均移动道数。

4、假定当前读写头在90号，电梯调度算法向磁道号增加的方向移动。

**实验3：熟悉linux文件系统调用**

**3.1实验目的**

1.掌握 linux 提供的文件系统调用的使用方法；

2.熟悉文件系统的系统调用用户接口；

3.了解操作系统文件系统的工作原理和工作方式。

# 3.2实验内容

使用文件系统调用编写一个文件工具 filetools，使其具有以下功能：

1.创建新文件

2.写文件

3.读文件

4.修改文件权限

5.查看当前文件权限

0.退出

提示用户输入功能号，并根据用户输入的功能选择相应的功能。

文件按可变记录文件组织，具体记录内容自行设计。

**3.3预备知识：**

用户在针对文件进行操作之前时一定要先打开它，这是由于系统需要根据用户提供的参数来查找文件的目录项，并由目录项找到文件的磁盘 i 结点，再将它调到内存 i 结点，才能建立用户进程与该文件之间的联系。

同样，在文件操作完毕后要关闭文件，以切断用户进程与文件的联系，释放相关资源。

Open 系统调用

int open(const char \*path, int flags);

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

一般情况下使用两个参数的格式，只有当打开的文件不存在时才使用 3 个参数的格式。参数：

Path 指向所要打开的文件的路径名指针。

Flag 标志参数，用来规定打开方式，必须包含以下 3 个之一：

O\_RDONLY 只读方式

O\_WRONLY 只写方式

O\_RDWR 读写方式

利用按位逻辑或“|”对下列标志进行任意组合：

O\_CREAT 如果文件不存在则创建该文件，若存在则忽略。

O\_TRUNC 如果文件存在则将文件长度截为 0，属性和所有者不变。

C\_EXECL 如果文件存在且 O\_CREAT 被设置则强制 open 调用失败。

O\_APPEND 每次写入时都从文件尾部开始。

Mode 是文件的访问权限，分为文件所有者、文件用户组和其他用户。

Close 系统调用

对于一个进程来说可以同时打开的文件是有限的，为了使文件标识符能够及时释放，系统必须提供关闭文件操作。

Int close(int fd)

Fd 为打开文件时系统返回的文件标识符

系统执行该系统调用时，根据 fd 值在该进程的进程打开文件表中找到 fd 标识，根据指针找到系统打开文件表，再找到内存 i 结点表中相应的 i 结点，对其 i\_count 进行减 1 操作，

然后释放系统打开文件表中的表项和进程打开文件表的表项。返回结果：调用成功返回 0

实习3说明：1、必须用系统调用实现，不能使用C库函数；

2、文件按可变记录文件组织，具体记录内容自行设计。

3、文件系统调用详见：

[**http://www.linuxidc.com/Linux/2016-11/137612.htm**](http://www.linuxidc.com/Linux/2016-11/137612.htm)

4、改变文件权限的系统调用为chmod，详见：

[**www.cnblogs.com/nufangrensheng/p/3502457.html**](https://www.cnblogs.com/nufangrensheng/p/3502457.html)

[**www.cnblogs.com/chengJAVA/p/4319420.html**](https://www.cnblogs.com/chengJAVA/p/4319420.html)

char\* pargv[4]={"ls","-l","file1",NULL};

  显示文件权限 execv("/bin/ls",pargv);

备注：系统调用提供的函数需包含头文件unistd.h

**实验4：进程管理**

**4.1实验目的**

1. 理解进程的概念，明确进程和程序的区别。
2. 理解并发执行的实质。
3. 掌握进程的同步、撤销等进程控制方法。

**4.2实验内容**

父进程使用系统调用pipe()建立一个管道，然后使用系统调用fork()创建两个子进程：子进程1和子进程2

子进程1每隔1秒通过管道向子进程2发送数据：I send message x times.（x初值为1，以后发送一次后做加一操作）子进程2从管道读出信息，并显示在屏幕上

父进程用系统调用signal()来捕捉来自键盘的中断信号SIGINT（即按Ctrl+C键,）；当捕捉到中断信号后，父进程用系统调用kill()向两个子进程发出信号，子进程捕捉到信号后分别输出如下信息后终止： Child Process 1 is killed by Parent!

Child Process 2 is killed by Parent!

父进程等待两个子进程终止后，释放管道并输出如下的信息后终止   
 Parent Process is Killed!

**预备知识：**

1.管道

管道是 Linux 支持的最初 Unix IPC 形式之一，具有以下特点：

①管道是半双工的，数据只能向一个方向流动（用lockf 或关闭文件）；需要双向通信时，需要建立起两个管道；

②只能用于父子进程或者兄弟进程之间（具有亲缘关系的进程）；

③单独构成一种独立的文件系统：管道对于管道两端的进程而言，就是一个文件，但它不是普通的文件，它不属于某种文件系统，而是自立门户，单独构成一种文件系统，并且只存在于内存中。

④数据的读出和写入：一个进程向管道中写的内容被管道另一端的进程读出。写入的内容每次都添加在管道缓冲区的末尾，并且每次都是从缓冲区的头部读出数据。

管道通过系统调用 pipe()来实现，函数的功能和实现过程如下。原型：int pipe( int fd[2] )

返回值：如果系统调用成功，返回 0。如果系统调用失败返回-1： errno = EMFILE (没有空闲的文件描述符)

ENFILE (系统文件表已满) EFAULT (fd 数组无效)

注意：fd[0] 用于读取管道，fd[1] 用于写入管道。

2.fork()

创建一个新进程。系统调用格式：

Pid=fork() 参数定义：

Int fork()

fork()返回值意义如下：

0：在子进程中，pid 变量保存的 fork()返回值为 0，表示当前进程是子进程。

>0：在父进程中，pid 变量保存的 fork()返回值为子进程的 id 值（进程唯一标识符）。

-1：创建失败。

如果 fork()调用成功，它向父进程返回子进程的 pid，并向子进程返回 0，即 fork() 被调用了一次，但返回了两次。此时 OS 在内存中建立一个新进程，所建的新进程是调用fork() 父进程（parent process）的副本，称为子进程（child process）。子进程继承了父进程的许多特性，并具有父进程完全相同的用户级上下文，父进程与子进程并发执行。

内核为 fork()完成以下操作：

1）为新进程分配一进程表项和进程标识符

进入 fork()后，内核检查系统是否有足够的资源来建立一个新进程。若资源不足，则

fork()系统调用失败；否则，核心为新进程分配一进程表项和唯一的进程标识符。

2）检查同时运行的进程数目

超过预先规定的最大数目时，fork()系统调用失败。

3）拷贝进程表项中的数据

将父进程的当前目录和所有已打开的数据拷贝到子进程表项中，并置进程的状态为“创建”状态。

4）子进程继承父进程的所有文件

对父进程当前目录和所有已打开的文件表项中的引用计数加 1.

5）为子进程创建进程上、下文

进程创建结束，设子进程状态为“内存中就绪”并返回子进程的标识符。

6）子进程执行

虽然父进程与子进程程序完全相同，但每个进程都有自己的程序计数器 PC，然后根据

Pid 变量保存的 fork()返回值的不同，执行了不同的分支语句。

3.exec()系列

系统调用 exec()系列，也可用于新程序的运行。fork()只是将父进程的用户级上下文拷贝到新进程中，而 exec()系列可以将一个可执行的二进制文件覆盖在新进程的用户级上下文的存储空间上，以更改新进程的用户级上下文。exec()系列中的系统调用都完成相同的功能，它们把一个新程序装入内存，来改变调用进程的执行代码，从而形成新进程。如果

exec()调用成功后，没有任何数据返回，这与 fork()不同。exec()系列调用在 LINUX 系统库 unistd.h 中，共有 execl、execlp、execv、execvp 五个，其基本功能相同，只是以不同的方式来给出参数。

一种是直接给出参数的指针，如：

Int execl(path,arg0[,arg1,...argn],0); Char \*path,\*argv[];

exec()和 fork()联合使用

系统调用 exec()和 fork()联合使用能为程序开发提供有力支持。用 fork()建立子进程，

然后再在子进程中使用 exec()，这样就实现了父进程与一个和它完全不同子进程的并发执行。

一般，wait、exec 联合使用的模型为： Int status；

......

If(fork()==0)

{

......;

execl(...);

......;

}

Wait(&status);

4.wait()

等待子进程运行结束。如果子进程没有完成，父进程一直等待。wait()将调用进程挂起， 直至其子进程因暂停或终止而发来软中断信号为止。如果在 wait()前已有子进程暂停或终止，则调用进程做适当处理后便返回。

系统调用格式：

Int wait(status) Int \*status;

其中，status 是用户空间的地址。它的低 8 位反应子进程状态，为 0 表示子进程正常

结束，非 0 则表示出现了各种各样的问题；高 8 位则带回了 exit()的返回值。exit()返回值由系统给出。

核心对 wait()作以下处理：

① 首先查找调用进程是否有子进程，若无，则返回出错码；

② 若找到一处于“僵死状态”的子进程，则将子进程的执行时间加到父进程的执行时间上，并释放子进程的进程表项；

③ 若未找到处于“僵死状态”的子进程，则调用进程便在可被中断的优先级上睡眠， 等待其子进程发来软中断信号时被唤醒。

5.exit()

终止进程的执行。系统调用格式：

Void exit(status) Int status;

其中，status 是返回给父进程的一个整数，以备查考。

为了及时回收进程所占用的资源并减少父进程的干预，LINUX/LINUX 利用 exit()来实现进程的自我终止，通常父进程在创建子进程时，应在进程的末尾安排一条 exit()，使子进程自我终止。exit(0)表示进程正常终止，exit(1)表示进程运行有错，异常终止。

如果调用进程在执行 exit()时，其父进程正在等待它的终止，则父进程可立即得到其返回的整数。核心须为 exit()完成以下操作：

1）关闭软中断

2）回收资源

3）写记账信息

4）置进程为“僵死状态”

# Signal ()函数详细介绍:

http://blog.csdn.net/ta893115871/article/details/7475095

**实验5：请求分页系统中的置换算法**

**5.1实验目的**

1. 了解虚拟存储技术的特点；
2. 掌握请求分页系统的页面置换算法。

# 5.2实验内容

1.通过如下方法产生一指令序列，共 320 条指令。

* 1. 在[1，32k-2]的指令地址之间随机选取一起点,访问 M；
  2. 顺序访问M+1；
  3. 在[0，M-1]中随机选取M1，访问 M1；
  4. 顺序访问M1+1；
  5. 在[M1+2，32k-2]中随机选取M2，访问 M2；
  6. 顺序访问M2+1；
  7. 重复 A—F，直到执行 320 次指令。

1. 指令序列变换成页地址流设：（1）页面大小为 1K；

（2） 分配给用户的内存页块个数为 4 页到 32 页,步长为1页；

（3）用户虚存容量为 32K。

1. 计算并输出下述各种算法在不同内存页块下的命中率。
   1. 先进先出（FIFO）页面置换算法
   2. 最近最久未使用（LRU）页面置换算法
   3. 最佳（Optimal）页面置换算法

**预备知识：**

1. 先进先出（FIFO）页面置换算法

该算法总是淘汰最新进入内存的页面，即选择在内存中驻留时间最久的页面予以淘汰。该算法实现简单，只需把一个进程已调入内存的页面，按先后次序链接成一个队列，并设置一个指针，称为替换指针，使它总是指向最老的页面。

1. 最近最久未使用（LRU）页面置换算法

最近最久未使用（LRU）页面置换算法，是根据页面调入内存后的使用情况进行决策的。由于无法预测各页面将来的使用情况，只能利用“最近的过去”作为“最近的将来”的近似， 因此，LRU 置换算法是选择最近最久未使用的页面予以淘汰。该算法赋予每个页面一个访问字段，用来记录一个页面自上次被访问以来所经历的时间 t，当需淘汰一个页面时，选择现有页面中其 t 值最大的，即最近最久未使用的页面予以淘汰。

1. 最佳（Optimal）页面置换算法

该算法选择的被淘汰页面，将是以后永远不使用的，或许是在最长（未来）时间内不再被访问的页面。采用该算法，通常可保证获得最低的缺页率。但由于人们目前还无法预知一个进程在内存的若干个页面中，哪一个页面是未来最长时间内不再被访问的，因而该算法是无法实现的，但可以利用该算法去评价其他算法。

提示：A.命中率=1-页面失效次数/页地址流长度

B.本实验中，页地址流长度为 320，页面失效次数为每次访问相应指令时，该指令所对应的页不在内存的次数。

4.关于随机数产生方法，采用系统提供函数 srand(）设定种子和 rand()来产生。

**实验6：进程通信**

**6.1实验目的**

1. 理解管道机制、消息缓冲队列、共享存储区机制进行进程间的通信；
2. 理解通信机制。

# 6.2实验内容

编写一主程序可以由用户选择如下三种进程通信方式：

1. 使用管道来实现父子进程之间的进程通信

子进程向父进程发送自己的进程标识符，以及字符串“is sending a message to

parent”。父进程则通过管道读出子进程发来的消息，将消息显示在屏幕上，然后终止。

1. 使用消息缓冲队列来实现 client 进程和 server 进程之间的通信

server 进程先建立一个关键字为 SVKEY（如 75）的消息队列，然后等待接收类型为 REQ

（例如 1）的消息；在收到请求消息后，它便显示字符串“serving for client”和接收到的 client 进程的进程标识数，表示正在为 client 进程服务；然后再向 client 进程发送应答消息，该消息的类型是 client 进程的进程标识数，而正文则是 server 进程自己的标识

ID。client 进程则向消息队列发送类型为 REQ 的消息（消息的正文为自己的进程标识 ID） 以取得 sever 进程的服务，并等待 server 进程发来的应答；然后显示字符串“receive reply

from”和接收到的 server 进程的标识 ID。

1. 使用共享存储区来实现两个进程之间的进程通信

进程 A 创建一个长度为 512 字节的共享内存，并显示写入该共享内存的数据；进程 B 将共享内存附加到自己的地址空间，并向共享内存中写入数据。

**预备知识：**

1. 管道

管道是 Linux 支持的最初 Unix IPC 形式之一，具有以下特点：

①管道是半双工的，数据只能向一个方向流动；需要双方通信时，需要建立起两个管道；

②只能用于父子进程或者兄弟进程之间（具有亲缘关系的进程）；

③单独构成一种独立的文件系统：管道对于管道两端的进程而言，就是一个文件，但它不是普通的文件，它不属于某种文件系统，而是自立门户，单独构成一种文件系统，并且只存在于内存中。

④数据的读出和写入：一个进程向管道中写的内容被管道另一端的进程读出。写入的内容每次都添加在管道缓冲区的末尾，并且每次都是从缓冲区的头部读出数据。

管道通过系统调用 pipe()来实现，函数的功能和实现过程如下。原型：int pipe( int fd[2] )

返回值：如果系统调用成功，返回 0。如果系统调用失败返回-1： errno = EMFILE (没有空闲的文件描述符)

ENFILE (系统文件表已满) EFAULT (fd 数组无效)

注意：fd[0] 用于读取管道，fd[1] 用于写入管道。

1. 消息队列

消息队列是消息的链接表，包括 Posix 消息队列 systemV 消息队列。它克服了另外两种通信方式中信息量有限的缺点，对消息队列具有写权限的进程可以向消息队列中按照一定的规则添加新消息；对消息队列有读权限的进程则可以从消息队列中读取消息。

消息队列通过系统调用 msgget()、msgsnd()、msgrcv()来实现，函数的功能和实现过程分别如下。原型：int msgget（key\_t key，int msgflg）

参数：

key：消息队列关联的键。

msgflg：消息队列的建立标志和存取权限。

返回值：成功执行时，返回消息队列标识值。失败返回-1，errno 被设为以下的某个值：

EACCES ： 指定的消息队列已存在， 但调用进程没有权限访问它， 而且不拥有

CAP\_IPC\_OWNER 权能。

EEXIST：key 指定的消息队列已存在，而 msgflg 中同时指定 IPC\_CREAT 和 IPC\_EXCL 标志。

ENOENT：key 指定的消息队列不存在同时 msgflg 中不指定 IPC\_CREAT 标志。

ENOMEM：需要建立消息队列，但内存不足。

ENOSPC：需要建立消息队列，但已达到系统的限制。

原型：ssize\_t msgrcv(int msqid, void \*msgp, size\_t msgsz, long msgtyp, int msgflg);

参数：

msqid：消息队列的识别码。

msgp：指向消息缓冲区的指针，此位置用来暂时存储发送和接收的消息，是一个用户可定义的通用结构。

msgsz：消息的大小。

msgtyp：从消息队列内读取的消息形态。如果值为零，则表示消息队列中的所有消息都会被读取。

msgflg：用来指明核心程序在队列没有数据的情况下所应采取的行动。如果 msgflg 和常数 IPC\_NOWAIT 合用，则在消息队列呈空时，不做等待马上返回-1，并设定错误码为

ENOMSG。当 msgflg 为 0 时， msgrcv()在队列呈满或呈空的情形时，采取阻塞等待的处理模式。

返回值：当成功执行时， msgrcv()返回拷贝到 mtext 数组的实际字节数。失败返回-1，

errno 被设为以下的某个值：

E2BIG：消息文本长度大于 msgsz，并且 msgflg 中没有指定 MSG\_NOERROR。

EACCES：调用进程没有读权能，同时没具有 CAP\_IPC\_OWNER 权能。

EAGAIN：消息队列为空，并且 msgflg 中没有指定 IPC\_NOWAIT。

EFAULT：msgp 指向的空间不可访问。

EIDRM：当进程睡眠等待接收消息时，消息已被删除。

EINTR：当进程睡眠等待接收消息时，被信号中断。

EINVAL：参数无效。

ENOMSG：msgflg 中指定了 IPC\_NOWAIT，同时所请求类型的消息不存在。

1. 共享[内存](http://hw.rdxx.com/Memory/)

共享[内存是操作系统](http://hw.rdxx.com/Memory/)常采用的进程间通信方式。它使得多个进程可以访问同一块内存空间，不同进程可以及时看到对方进程中对共享内存中数据的更新。这种通信方式需要依靠某种同步机制，如互斥锁和信号量等。

共享内存通过系统调用 shmget()、shmat()、shmdt()、shmctl()来实现，函数的功能和实现过程分别如下。

系统调用：shmget()

原型：int shmget(key\_t key, int size, int shmflg)参数：

Key:共享存储区的名字；

Size：共享存储区的大小（以字节计）；

Shmflag：用户设置的标志，如 IPC\_CREAT，IPC\_CREAT 表示若系统中尚无指名的共享存储区，则由核心建立一个共享存储区；如系统中已有共享存储区，便忽略 IPC\_CREAT。

返回值：

成功返回共享内存的标识符；不成功返回-1，errno 储存错误原因。

EINVAL 参数 size 小于 SHMMIN 或大于 SHMMAX。

EXIST 预建立 key 所致的共享内存，但已经存在。

EIDRM 参数 key 所致的共享内存已经删除。

ENOSPC 超过了系统允许建立的共享内存的最大值(SHMALL )。

ENOENT 参数 key 所指的共享内存不存在，参数 shmflg 也未设 IPC\_CREAT 位。

EACCES 没有权限。

ENOMEM 核心内存不足。系统调用：shmat()

原型：void\* shmat(int shmid, const void \*shmaddr, int shmflg);参数：

Shmid：共享存储区的标识符；

Shmaddr：用户给定的，将共享存储区附接到进程的虚拟地址空间；

Shmflg：规定共享存储区的读、写权限，以及系统是否应对用户规定的地址做舍入操作。其值为 SHM\_RDONLY 时，表示只能读，其值为 0 时，表示可读、可写，其值为 SHM\_RND

（取整）时，表示操作系统在必要时舍去这个地址。返回值：

该系统调用的返回值为共享存储区所附接到的进程虚地址。系统调用：shmdt()

原型：int shmdt(void \*shmaddr)参数：

Shmaddr：要断开连接的虚地址，亦即以前由连接的系统调用 shmat（）所返回的虚地址。

返回值：

调用成功时，返回 0 值，调用不成功，返回-1 值。系统调用：shmctl()

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

Shmid：由 shmget 所返回的标识符；

Cmd：是操作命令，可以分多种类型：

（1）用于查询有关共享存储区的情况。如其长度、当前连接的进程数、共享区的创建者标识符。

（2）用于设置或改变共享存储区的属性，如共享存储区的许可权、当前连接的进程计数等。

（3）对共享存储区的加锁和解锁命令。

（4）删除共享存储区标识符等。

Buf：用户缓冲区地址。返回值：

调用成功则返回 0，如果失败则返回-1。

# 消息队列函数及其范例:http://blog.csdn.net/guoping16/article/details/6584024

# 共享内存函数及其范例http://blog.csdn.net/guoping16/article/details/6584058

# 附录 A Linux 中 Vi 和 GCC 的使用

**A1 实验目的**

1. 复习 C 语言程序基本知识；
2. 练习并掌握 LINUX 提供的 vi 编辑器来编译 C 程序；
3. 学会利用 gcc(cc)编译、运行 C 程序。

# A2 实验内容

1. 用 vi 编写一个简单的、显示"Hello,World!"的 C 程序，用 gcc 编译并观察编译后的结果；
2. 运行生成的可执行文件。

# A3 实验准备

1.C 语言使用简介

LINUX 中包含了很多软件开发工具。它们中的很多是用于 C 和 C++应用程序开发的。C 是一种能在 LINUX 的早期就被广泛使用的通用编程语言。它最早是由 Bell 实验室的 Dennis

Ritchie 为了 LINUX 的辅助开发而写的，从此 C 就成为世界上使用最广泛的计算机语言。C 能在编程领域里得到如此广泛支持的原因有：

① 它是一种非常通用的语言，并且它的语法和函数库在不同的平台上都是统一的，对开发者非常有吸引力；

② 用 C 写的程序执行速度很快；

③ C 是所有版本 LINUX 上的系统语言；

2.文件编辑器 vi

vi 是在 LINUX 上被广泛使用的中英文编辑软件。vi 是 visual editor 的缩写，是 LINUX 提供给用户的一个窗口化编辑环境。进入 vi，直接执行 vi 编辑程序即可。

例：$vi test.c

显示器出现 vi 的编辑窗口，同时 vi 会将文件复制一份至缓冲区（buffer）。vi 先对缓冲区的文件进行编辑，保留在磁盘中的文件则不变。编辑完成后，使用者可决定是否要取代原来旧有的文件。

vi 提供二种工作模式：输入模式（insert mode）和命令模式（command mode）。使用者进入 vi 后，即处在命令模式下，此刻键入的任何字符皆被视为命令，可进行删除、修改、存盘等操作。要输入信息，应转换到输入模式。

1. 命令模式

在输入模式下，按 ESC 可切换到命令模式。命令模式下，可选用下列指令离开 vi：

：q! 离开 vi，并放弃刚在缓冲区内编辑的内容

：wq 将缓冲区内的资料写入磁盘中，并离开 vi

：ZZ 同 wq

：x 同 wq

：w 将缓冲区内的资料写入磁盘中，但并不离开 vi

：q 离开 vi，若文件被修改过，则要被要求确认是否放弃修改的内容，此指令可与：w 配合使用

1. 命令模式下光标的移动

H 左移一个字符

1. 下移一个字符
2. 上移一个字符
3. 右移一个字符

0 移至该行的首

$ 移至该行的末

^ 移至该行的第一个字符处

H 移至窗口的第一列

M 移至窗口中间那一列

L 移至窗口的最后一列

G 移至该文件的最后一列

W, W 下一个单词 (W 忽略标点) B, B 上一个单词 (B 忽略标点)

+ 移至下一列的第一个字符处

- 移至上一列的第一个字符处

( 移至该句首

) 移至该句末

{ 移至该段首

} 移至该段末

NG 移至该文件的第 n 列

N+ 移至光标所在位置之后第 n 列

n 移至光标所在位置之前第 n 列

Ctrld 向下半页

Ctrlf 向下一页

Ctrlu 向上半页

Ctrlb 向上一页

1. 输入模式

输入以下命令即可进入 vi 输入模式：

a(append) 在光标之后加入资料

A 在该行之末加入资料

i(insert) 在光标之前加入资料

I 在该行之首加入资料

o(open) 新增一行于该行之下，供输入资料用

O 新增一行于该行之上，供输入资料用

Dd 删除当前光标所在行

X 删除当前光标字符

X 删除当前光标之前字符

U 撤消

F 查找

s 替换,例如:将文件中的所有"FOX"换成"duck",用":%s/FOX/duck/g"

ESC 离开输入模式

更多用法见 info vi.

1. GNU C 编译器

LINUX 上可用的 C 编译器是 GNU(GNU's Not Unix) C 编译器，它建立在自由软件基金会编程许可证的基础上，因此可以自由发布。LINUX 上的 GNU C 编译器（GCC）是一个全功能的 ANCI C 兼容编译器。通过 GCC，由 C 源代码文件生成可执行文件的过程要经历 4 个阶段， 分别是预处理、编译、汇编和链接。不同的阶段分别调用不同的工具来实现，如图 B-1 所示。

源文件

预处理

编译

汇编

(\*.c)

可执行文件

(\*.out)

链接

预处理器 编译器 汇编器 链接器

图 B-1 GCC 的执行过程

* 1. 使用 GCC

通常后跟一些选项和文件名来使用 GCC 编译器。GCC 命令的基本用法如下： gcc [选项] 源文件 [目标文件]

命令行选项指定的编译过程中的具体操作。

* 1. GCC 常用选项

GCC 有超过 100 个的编译选项可用，这些选项中的许多可能永远都不会用到，但一些主要的选项将会频繁使用。很多的 GCC 选项包括一个以上的字符，因此必须为每个选项指定各自的连字符，并且就像大多数 LINUX 命令一样不能在一个单独的连字符后跟一组选项。

GCC 常用的选项说明如下：

-o file：编译产生的文件以指定文件名保存。如果 file 没有指定，默认文件名为 a.out。

-I：在 GCC 的头文件搜索路径中添加新的目录。

-L：在 GCC 的库文件搜索路径中添加新的目录。

-c：GCC 仅把源代码编译为目标代码，而不进行函数库链接。完成后输出一个与源文件名相同的，但扩展名为.o 的目标文件。

-O、-O1：GCC 对源代码进行基本优化，编译产生尽可能短、执行尽可能快的代码，但是在编译的过程中，会花费更多的时间和内存空间。

-O2：较-O 选项执行更进一步的优化，但编译过程开销更大。

-g：在编译产生的可执行文件中附加上 gdb 使用的调试信息。

-w：禁止所有的警告。不建议使用此选项。

-Wall：使 GCC 产生尽可能多的警告信息，对找出常见的隐式编程错误有帮助。

-v：显示编译器路径、版本及执行编译的过程。

当不用任何选项编译一个程序时，GCC 将建立（假定编译成功）一个名为 a.out 的可执行文件。

例如：gcc test.c

例如：gcc –o foo test.c

将 test.c 文件编译后生成 foo 的可执行文件。

GCC 也可以指定编译器处理步骤多少。c 选项告诉 GCC 仅把源代码编译为目标代码而跳过汇编和连接步骤。这个选项使用得非常频繁因为它编译多个 C 程序时速度更快且更易于管理。默认时 GCC 建立的目标代码文件有一个.out 的扩展名。

* 1. 执行文件

格式： ./可执行文件名例：./a.out

./count

# A4 程序示例

通过 vi 编辑器，创建 greet.c。

>vi greet.c

通过输入 i，进入编辑模式，然后输入如下代码。

main( )

{

printf("Hello,world!\n");

}

输入完成后，按 ESC 可切换到命令模式，保存退出。

>wq

# A5 实验结果

1.编译代码

> gcc –o greet greet.c 2.执行代码

>./greet

# 附录 B Linux 中 C 语言调试器 GDB 的使用

**B1 实验目的**

1. 掌握最常用的代码调试器 gdb 的使用；
2. 掌握调试代码的基本方法，设置断点，查看变量。

# B2 实验内容

1. 用 vi 编写一个的 C 程序；
2. 利用 gdb 调试该程序。

# B3 实验准备

1. gdb 简介

gdb 是 GNU(GNU's Not Unix)开源组织发布的一个强大的 UNIX 下的程序调试工具。或许， 大家比较喜欢那种图形界面方式的，像 Visual C++、c++ builder 等 IDE(Integrated Development Environment)集成开发环境的调试，但如果你是在 UNIX 平台下做软件，你会发现 gdb 这个调试工具有比 Visual C++、c++ builder 的图形化调试器更强大的功能。所谓

“寸有所长，尺有所短”就是这个道理。一般来说，gdb 主要帮忙你完成下面四个方面的功能：

① 启动你的程序，可以按照你的自定义的要求随心所欲的运行程序。

② 可让被调试的程序在你所指定的调置的断点处停住。（断点可以是条件表达式）

③ 当程序被停住时，可以检查此时你的程序中所发生的事。

④ 动态的改变你程序的执行环境。

1. gdb 使用

一般来说 gdb 主要调试的是 C/C++的程序。要调试 C/C++的程序，首先在编译时，我们必须要把调试信息加到可执行文件中。使用编译器（cc/gcc/g++）的-g 参数可以做到这一点。如：

> gcc -g hello.c -o hello

> g++ -g hello.cpp -o hello

如果没有-g，你将看不见程序的函数名、变量名，所代替的全是运行时的内存地址。

* 1. 启动 GDB

1. gdb program

program 也就是你的执行文件，一般在当前目录下。

1. gdb core

用 gdb 同时调试一个运行程序和 core 文件，core 是程序非法执行后 core dump 后产生的文件。

1. gdb PID

如果你的程序是一个服务程序，那么你可以指定这个服务程序运行时的进程 ID。

gdb 会自动 attach 上去，并调试它。program 应该在 PATH 环境变量中搜索得到。

gdb 启动时，可以加上一些 gdb 的启动开关，详细的开关可以用 gdb -help 查看

* 1. 暂停/恢复程序运行

调试程序中，暂停程序运行是必须的，gdb 可以方便地暂停程序的运行。你可以设置程序的在哪行停住，在什么条件下停住，在收到什么信号时停往等等。以便于你查看运行时的变量，以及运行时的流程。

当进程被 gdb 停住时，你可以使用 info program 来查看程序的是否在运行，进程号， 被暂停的原因。

在 gdb 中，我们可以有以下几种暂停方式：断点（BreakPoint）、观察点（WatchPoint）、捕捉点（CatchPoint）、信号（Signals）、线程停止（Thread Stops）。如果要恢复程序运行，

可以使用 c 或是 continue 命令。下面主要介绍断点和观察点的设置。

1. 设置断点（BreakPoint）

break 命令（可以简写为 b）可以用来在调试的程序中设置断点，该命令有如下四种形式：break line-number 使程序恰好在执行给定行之前停止。

break function-name 使程序恰好在进入指定的函数之前停止。

break line-or-function if condition 如果 condition（条件）是真，程序到达指定行或函数时停止。

break routine-name 在指定例程的入口处设置断点。

如果该程序是由很多原文件构成的，你可以在各个原文件中设置断点，而不是在当前的原文件中设置断点，其方法如下：

(gdb) break filename:line-number (gdb) break filename:function-name

要想设置一个条件断点，可以利用 break if 命令，如下所示： (gdb) break line-or-function if expr

例：

(gdb) break 46 if testsize==100 从断点继续运行：countinue 命令

1. 设置观察点（WatchPoint）

观察点一般来观察某个表达式（变量也是一种表达式）的值是否有变化了，如果有变化， 马上停住程序。我们有下面的几种方法来设置观察点：

watch

为表达式（变量）expr 设置一个观察点。一量表达式值有变化时，马上停住程序。rwatch

当表达式（变量）expr 被读时，停住程序。awatch

当表达式（变量）的值被读或被写时，停住程序。info watchpoints

列出当前所设置了的所有观察点。

1. 维护停止点

① 显示当前 gdb 的断点信息： (gdb) info break

执行后，会以如下的形式显示所有的断点信息： Num Type Disp Enb Address What

1. breakpoint keep y 0x000028bc in init\_random at qsort2.c:155
2. breakpoint keep y 0x0000291c in init\_organ at qsort2.c:168

② 删除指定的某个断点： (gdb) delete breakpoint 1

该命令将会删除编号为 1 的断点，如果不带编号参数，将删除所有的断点(gdb) delete breakpoint

③ 禁止使用某个断点

(gdb) disable breakpoint 1

该命令将禁止断点 1，同时断点信息的 (Enb)域将变为 n。

④ 允许使用某个断点 (gdb) enable breakpoint 1

该命令将允许断点 1，同时断点信息的 (Enb)域将变为 y。

⑤ 清除原文件中某一代码行上的所有断点(gdb)clean number

注：number 为原文件的某个代码行的行号。

1. 恢复程序运行

当程序被停住了，使用 continue 命令恢复程序的运行直到程序结束，或下一个断点到来。也可以使用 step 或 next 命令单步跟踪程序。

continue [ignore-count] c [ignore-count]

fg [ignore-count]

恢复程序运行，直到程序结束，或是下一个断点到来。ignore-count 表示忽略其后的断点次数。continue，c，fg 三个命令都是一样的意思。

1. 单步调试

使用 step 命令进行单步跟踪，如果有函数调用，它会进入该函数。进入函数的前提是， 此函数被编译有 debug 信息。很像 VC 等工具中的 step in。后面可以加 count 也可以不加， 不加表示一条条地执行，加表示执行后面的 count 条指令，然后再停住。

同样使用 next 命令进行单步跟踪，如果有函数调用，它不会进入该函数。很像 VC 等工具中的 step over。后面可以加 count 也可以不加，不加表示一条条地执行，加表示执行后面的 count 条指令，然后再停住。

使用 finish 命令运行程序，直到当前函数完成返回。并打印函数返回时的堆栈地址和返回值及参数值等信息。

使用 until 或 u 命令可以运行程序直到退出循环体。

1. 变量的检查和赋值

Whatis：识别数组或变量的类型

ptype：比 whatis 的功能更强，它可以提供一个结构的定义set variable：将值赋予变量

print：除了显示一个变量的值外，还可以用来赋值。

# B4 程序示例

源程序：test.c

#includes <stdio.h> int func(int n)

{

int sum=0,i; for(i=0; i

{

sum+=i;

}

return sum;

}

main()

{

int i;

long result = 0; for(i=1; i<=100; i++)

{

result += i;

}

printf("result[1-100] = %d \n", result ); printf("result[1-250] = %d \n", func(250) );

}

# B5 实验结果

首先编译生成执行文件 test：（Linux 下）。

> gcc -g test.c -o test

然后使用 GDB 调试对 test 程序进行调试。

> gdb test <---------- 启动 gdb GNU gdb 5.1.1

Copyright 2002 Free Software Foundation, Inc.

GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions. Type "show copying" to see the conditions.

There is absolutely no warranty for GDB. Type "show warranty" for details. This GDB was configured as "i386-suse-linux"...

(gdb) list <-------------------- list 命令，从第一行开始显示源码。

1 #include <stdio.h> 2

3 int func(int n) 4 {

1. int sum=0,i;
2. for(i=0; i 7 {

8 sum+=i;

9 }

10 return sum;

(gdb) <-------------------- 直接回车表示，重复上一次命令11 }

12

13

14 main() 15 {

1. int i;
2. long result = 0;

18 for(i=1; i<=100; i++)

19 {

20 result += i;

(gdb) break 16 <-------------------- 设置断点，在源程序第 16 行处。Breakpoint 1 at 0x8048496: file test.c, line 16.

(gdb) break func <-------------------- 设置断点，在函数 func()入口处。Breakpoint 2 at 0x8048456: file test.c, line 5.

(gdb) info break <-------------------- 查看断点信息。Num Type Disp Enb Address What

1. breakpoint keep y 0x08048496 in main at test.c:16
2. breakpoint keep y 0x08048456 in func at test.c:5 (gdb) run <--------------------- 运行程序 Starting program: /home/test

Breakpoint 1, main () at test.c:17 <---------- 在断点处停住。

17 long result = 0;

(gdb) next <--------------------- 单条语句执行18 for(i=1; i<=100; i++)

(gdb) next

20 result += i; (gdb) next

18 for(i=1; i<=100; i++)

(gdb) next

20 result += i;

(gdb) continue <--------------------- 继续运行程序。Continuing.

result[1-100] = 5050 <----------程序输出。

Breakpoint 2, func (n=250) at test.c:5

5 int sum=0,i; (gdb) n

6 for(i=1; i<=n; i++)

(gdb) print i <--------------------- 打印变量 i 的值。

$1 = 134513808

(gdb) next

8 sum+=i; (gdb) next

6 for(i=1; i<=n; i++) (gdb) print sum

$2 = 1

(gdb) next

8 sum+=i; (gdb) print i

$3 = 2

(gdb) next

6 for(i=1; i<=n; i++) (gdb) print sum

$4 = 3

(gdb) bt <--------------------- 查看函数堆栈。

#0 func (n=250) at test.c:5

#1 0x080484e4 in main () at test.c:24

#2 0x400409ed in libc\_start\_main () from /lib/libc.so.6

(gdb) finish <--------------------- 退出函数。Run till exit from #0 func (n=250) at test.c:5 0x080484e4 in main () at test.c:24

24 printf("result[1-250] = %d \n", func(250) );

Value returned is $6 = 31375

(gdb) continue <--------------------- 继续运行。Continuing.

result[1-250] = 31375 <----------程序输出。

Program exited with code 027. <--------程序退出，调试结束。(gdb) quit <--------------------- 退出 gdb。

附录 C 课程设计实习报告封面：

《操作系统课程设计》

实习报告

班级

学号

姓名

院系

专业

指导老师

完成日期