**进程控制与进程通信实验**

1. **实验目的**

1、理解Linux下进程的结构；

2、理解Linux下产生新进程的方法（系统调用—fork函数）；

3、掌握如何启动另一程序的执行；

4、理解Linux下线程的结构；

5、理解Linux下产生新线程的方法；

6、理解Linux系统下多进程与多线程的区别

7、了解什么是管道

8、熟悉UNIX/LINUX支持的管道通信方式

9、理解内核模块的编写和装载方法

1. **实验环境**

1. MacOSX10.14.sdk/usr/include/c++/4.2.1

Apple LLVM version 10.0.1 (clang-1001.0.46.4)

Target: x86\_64-apple-darwin18.2.0

Thread model: posix

2. /usr/lib/gcc/x86\_64-linux-gnu/4.8/lto-wrapper

Target: x86\_64-linux-gnu

gcc version 4.8.2 (Ubuntu 4.8.2-19ubuntu1)

1. **实验内容**

1、利用fork函数创建新进程，并根据fork函数的返回值，判断自己是处于父进程还是子进程中；

2、在新创建的子进程中，使用exec类的函数启动另一程序的执行；分析多进程时系统的运行状态和输出结果；

3、利用最常用的三个函数pthread\_create，pthread\_join和pthread\_exit编写了一个最简单的多线程程序。理解多线程的运行和输出情况；

4、利用信号量机制控制多线程的运行顺序，并实现多线程中数据的共享；

5、分析Linux系统下多进程与多线程中的区别。

6、编写程序实现进程的管道通信。用系统调用pipe( )建立一管道，二个子进程P1和P2分别向管道各写一句话：

Child 1 is sending a message!

Child 2 is sending a message!

父进程从管道中读出二个来自子进程的信息并显示（要求先接收P1，后P2）。

1. 编写一个HelloWorld内核模块，并进行装载和卸载操作
2. **实验原理 实验中用到的系统调用函数（包括实验原理中介绍的和自己采用的），实验步骤，**

实验原理

1、理解Linux下进程的结构

Linux下一个进程在内存里有三部份的数据，就是“数据段”，“堆栈段”和“代码段”，一般的CPU, 如I386，都有上述三种段寄存器，以方便操作系统的运行。“代码段”，顾名思义，就是存放了程序代码的数据，假如机器中有数个进程运行相同的一个程序，那么它们就可以使用同一个代码段。

 　　堆栈段存放的就是子程序的返回地址、子程序的参数以及程序的局部变量。而数据段则存放程序的全局变量，常数以及动态数据分配的数据空间（比如用malloc之类的函数取得的空间）。系统如果同时运行数个相同的程序，它们之间就不能使用同一个堆栈段和数据段。

2、如何使用fork函数

在Linux下产生新进程的系统调用就是fork函数，这个函数名是英文中“分叉”的意思。一个进程在运行中，如果使用了fork，就产生了另一个进程，于是进程就“分叉”了，所以这个名字取得很形象。

那么调用这个fork函数时发生了什么呢？一个程序调用fork函数，系统就为一个新的进程准备了前述三个段，首先，系统让新的进程与旧的进程使用同一个代码段，因为它们的程序还是相同的，对于数据段和堆栈段，系统则复制一份给新的进程，这样，父进程的所有数据都可以留给子进程，但是，子进程一旦开始运行，虽然它继承了父进程的一切数据，但实际上数据却已经分开，相互之间不再有影响了，也就是说，它们之间不再共享任何数据了。而如果两个进程要共享什么数据的话，就要使用另一套函数（shmget，shmat，shmdt等）来操作。现在，已经是两个进程了，对于父进程，fork函数返回了子程序的进程号，而对于子程序，fork函数则返回零，这样，对于程序，只要判断fork函数的返回值，就知道自己是处于父进程还是子进程中。

 　　但是，如果一个大程序在运行中，它的数据段和堆栈都很大，调用一次fork就要复制一次，那么fork的系统开销不是很大吗？其实，一般CPU都是以“页”为单位分配空间的，像INTEL的CPU，其一页在通常情况下是4K字节大小，而无论是数据段还是堆栈段都是由许多“页”构成的，fork函数复制这两个段，只是“逻辑”上的，并非“物理”上的，也就是说，实际执行fork时，物理空间上两个进程的数据段和堆栈段都还是共享着的，当有一个进程写了某个数据时，利用写时复制技术，该进程的数据被写入另一数据段，这时两个进程之间的数据才有了区别，系统就将有区别的“页”从物理上也分开。系统在空间上的开销就可以达到最小。

3、如何启动另一程序的执行

在Linux中要使用exec类的函数来启动另一程序的执行，exec类的函数不止一个，但大致相同，在Linux中，它们分别是：execl，execlp，execle，execv，execve和execvp。一个进程一旦调用exec类函数，它本身就“死亡”了，系统把代码段替换成新的程序的代码，废弃原有的数据段和堆栈段，并为新程序分配新的数据段与堆栈段，唯一留下的，就是进程号，也就是说，对系统而言，还是同一个进程，不过已经是另一个程序了。（不过exec类函数中有的还允许继承环境变量之类的信息）。那么如果我的程序想启动另一程序的执行但自己仍想继续运行的话，怎么办呢？那就是结合fork与exec的使用。

程序在调用fork( )建立一个子进程后，马上调用wait( )，使父进程在子进程结束之前，一直处于睡眠状态。子进程用exec( )装入命令ls ，exec( )后，子进程的代码被ls的代码取代，这时子进程的PC指向ls的第1条语句，开始执行ls的命令代码。

（二）Linux系统下多线程编程：

1、为什么有了进程的概念后，还要再引入线程呢？使用多线程到底有哪些好处？什么的系统应该选用多线程？

线程（thread）技术早在60年代就被提出，但真正应用多线程到操作系统中去，是在80年代中期，solaris是这方面的佼佼者。传统的Unix也支持线程的概念，但是在一个进程（process）中只允许有一个线程，这样多线程就意味着多进程。现在，多线程技术已经被许多操作系统所支持，包括Windows/NT，当然，也包括Linux。  
 使用多线程的理由之一是和进程相比，它是一种非常"节俭"的多任务操作方式。在Linux系统下，启动一个新的进程必须分配给它独立的地址空间，建立众多的数据表来维护它的代码段、堆栈段和数据段，这是一种"昂贵"的多任务工作方式。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的时间也远远小于进程间切换所需要的时间。据统计，总的说来，一个进程的开销大约是一个线程开销的30倍左右，当然，在具体的系统上，这个数据可能会有较大的区别。  
　　使用多线程的理由之二是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，这种方式不仅费时，而且很不方便。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它线程所用，这不仅快捷，而且方便。当然，数据的共享也带来其他一些问题，有的变量不能同时被两个线程所修改，有的子程序中声明为static的数据更有可能给多线程程序带来灾难性的打击，这些正是编写多线程程序时最需要注意的地方。  
　　除了以上所说的优点外，多线程程序作为一种多任务、并发的工作方式，还有以下的优点：  
　　1) 提高应用程序响应。这对图形界面的程序尤其有意义，当一个操作耗时很长时，整个系统都会等待这个操作，此时程序不会响应键盘、鼠标、菜单的操作，而使用多线程技术，将耗时长的操作（time consuming）置于一个新的线程，可以避免这种尴尬的情况。  
　　2) 使多CPU系统更加有效。操作系统会保证当线程数不大于CPU数目时，不同的线程运行于不同的CPU上。  
　　3) 改善程序结构。一个既长又复杂的进程可以考虑分为多个线程，成为几个独立或半独立的运行部分，这样的程序会利于理解和修改。  
2、**简单的多线程编程**  
　　Linux系统下的多线程遵循POSIX线程接口，称为pthread。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。顺便说一下，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，它的使用方式类似fork。函数pthread\_create用来创建一个线程，它的原型为：

extern int pthread\_create \_\_P ((pthread\_t \*\_\_thread, \_\_const pthread\_attr\_t \*\_\_attr,void \*(\*\_\_start\_routine) (void \*), void \*\_\_arg));

第一个参数为指向线程标识符的指针，第二个参数用来设置线程属性，第三个参数是线程运行函数的起始地址，最后一个参数是运行函数的参数。这里，我们的函数thread不需要参数，所以最后一个参数设为空指针。第二个参数我们也设为空指针，这样将生成默认属性的线程。当创建线程成功时，函数返回0，若不为0则说明创建线程失败，常见的错误返回代码为EAGAIN和EINVAL。前者表示系统限制创建新的线程，例如线程数目过多了；后者表示第二个参数代表的线程属性值非法。创建线程成功后，新创建的线程则运行参数三和参数四确定的函数，原来的线程则继续运行下一行代码。   
　　函数pthread\_join用来等待一个线程的结束。函数原型为：

extern int pthread\_join \_\_P ((pthread\_t \_\_th, void \*\*\_\_thread\_return));

第一个参数为被等待的线程标识符，第二个参数为一个用户定义的指针，它可以用来存储被等待线程的返回值。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。一个线程的结束有两种途径，一种是象我们上面的例子一样，函数结束了，调用它的线程也就结束了；另一种方式是通过函数pthread\_exit来实现。它的函数原型为：  
　　extern void pthread\_exit \_\_P ((void \*\_\_retval)) \_\_attribute\_\_ ((\_\_noreturn\_\_));  
　　唯一的参数是函数的返回代码，只要pthread\_join中的第二个参数thread\_return不是NULL，这个值将被传递给thread\_return。最后要说明的是，一个线程不能被多个线程等待，否则第一个接收到信号的线程成功返回，其余调用pthread\_join的线程则返回错误代码ESRCH。  
3、**线程的数据处理**  
　　和进程相比，线程的最大优点之一是数据的共享性，各个进程共享父进程处沿袭的数据段，可以方便的获得、修改数据。但这也给多线程编程带来了许多问题。我们必须当心有多个不同的进程访问相同的变量。许多函数是不可重入的，即同时不能运行一个函数的多个拷贝（除非使用不同的数据段）。在函数中声明的静态变量常常带来问题，函数的返回值也会有问题。因为如果返回的是函数内部静态声明的空间的地址，则在一个线程调用该函数得到地址后使用该地址指向的数据时，别的线程可能调用此函数并修改了这一段数据。在进程中共享的变量必须用关键字volatile来定义，这是为了防止编译器在优化时（如gcc中使用-OX参数）改变它们的使用方式。为了保护变量，我们必须使用信号量、互斥等方法来保证我们对变量的正确使用。

1. 信号量  
   　　信号量本质上是一个非负的整数计数器，它被用来控制对公共资源的访问。当公共资源增加时，调用函数sem\_post（）增加信号量。只有当信号量值大于０时，才能使用公共资源，使用后，函数sem\_wait（）减少信号量。函数sem\_trywait（）和函数pthread\_ mutex\_trylock（）起同样的作用，它是函数sem\_wait（）的非阻塞版本。下面我们逐个介绍和信号量有关的一些函数，它们都在头文件/usr/include/semaphore.h中定义。  
   　　信号量的数据类型为结构sem\_t，它本质上是一个长整型的数。函数sem\_init（）用来初始化一个信号量。它的原型为：  
   　　extern int sem\_init \_\_P ((sem\_t \*\_\_sem, int \_\_pshared, unsigned int \_\_value));  
   　　sem为指向信号量结构的一个指针；pshared不为０时此信号量在进程间共享，否则只能为当前进程的所有线程共享；value给出了信号量的初始值。  
   　　函数sem\_post( sem\_t \*sem )用来增加信号量的值。当有线程阻塞在这个信号量上时，调用这个函数会使其中的一个线程不在阻塞，选择机制同样是由线程的调度策略决定的。  
   　　函数sem\_wait( sem\_t \*sem )被用来阻塞当前线程直到信号量sem的值大于0，解除阻塞后将sem的值减一，表明公共资源经使用后减少。函数sem\_trywait ( sem\_t \*sem )是函数sem\_wait（）的非阻塞版本，它直接将信号量sem的值减一。  
   　　函数sem\_destroy(sem\_t \*sem)用来释放信号量sem。  
   　　多线程编程是一个很有意思也很有用的技术，使用多线程技术的网络蚂蚁是目前最常用的下载工具之一，使用多线程技术的grep比单线程的grep要快上几倍，类似的例子还有很多。希望大家能用多线程技术写出高效实用的好程序来。

（二）内核模块的编写

1、认识内核模块

内核模块是linux内核向外部提供的一个插口，是内核的一部分，但是并没有被编译到内核里面去，其全称为动态可加载内核模块(Loadable Kernel Module,LKM)，简称模块。

linux内核之所以提供模块机制，是因为它本身是一个单内核。而单内核的最大优点就是效率高，因为所有的内容都集成在一起，但其缺点是可扩展性和可维护性相对较差，模块机制就是为了弥补这一缺陷。

2、模块的定义

模块是具有独立功能的程序，它可以被单独编译，但不能独立运行。它在运行时被链接到内核作为内核的一部分在内核空间运行，这与运行在用户空间的进程是不同的。模块通常由一组函数和数据结构组成，用来实现一种文件系统、一个驱动程序或者其他内核上层的功能。

3、编写简单的内核模块

模块和内核都在内核空间运行，模块编程在一定意义上说就是内核编程。一个内核模块应该至少有两个函数，第一个为module\_init()，是模块加载函数，当模块被插入到内核时调用它；第二个为module\_exit()，是模块卸载函数，当模块从内核移走时调用它。

（1）任何模块都要包含的三个头文件：

#include <linux/module.h>

#include <linux/kernel.h>

#incldue <linux/init.h>

说明：module.h头文件包含了对模块的版本控制；kernel.h包含了常用的内核函数；init.h包含了宏\_\_init和\_\_exit，宏\_\_init告诉编译程序相关的函数和变量仅用于初始化，编译程序将标有\_\_init的所有代码存储到特殊的内存段中，初始化结束就释放这段内存。

（2）printk()函数

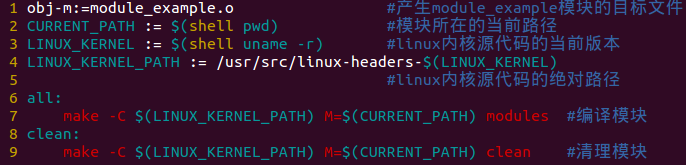
该函数是由内核定义的，功能和C库中的printf()类似，它把要打印的日志输

出到终端或系统日志。字符串中的<1>是输出的级别，表示立即在终端输出。

（3）内核模块的Makefile文件

内核模块不是独立的可执行文件，但在运行时其目标文件被链接到内核中。只有超级用户才能加载和卸载模块。

Makefile文件的基本内容如下：



**obj-m :=这个赋值语句的含义是说明要使用目标文件module\_example.o建立一个模块，最后生成的模块名为module\_example.ko。.o文件是经过编译和汇编，而没有经过链接的中间文件。**

**注：makefile文件中，若某一行是命令，则它必须以一个Tab键开头。**

（4）装载模块

当编译好模块，就可以用insmod命令将新的模块插入到内核中，如：

insmod module\_example.ko

然后，可以用lsmod命令查看模块是否正确地插入到了内核中。模块的输出由printk()产生。该函数默认打印系统文件/var/log/messages的内容。

（5）卸载模块

使用rmmod命令加上在insmod中看到的模块名，就可以从内核中移除该模块：

rmmod module\_example

实验步骤

1、利用fork函数编写一个简单的多进程程序，用ps命令查看系统中进程的运行状况，并分析输出结果。

2、在上面的多进程程序中利用exec函数，启动另一个程序的执行。用ps命令显示本机运行的所有进程的详细列表，并分析列表中不同进程的内存占用情况。

3、编写一个最简单的多线程程序。理解多线程的运行和输出情况；

4、利用信号量机制控制多线程的运行顺序，分析多线程中数据的共享情况；

5、分析Linux系统下多进程与多线程中的区别。

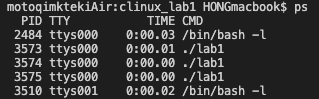
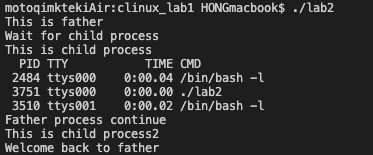
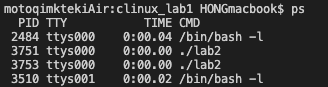
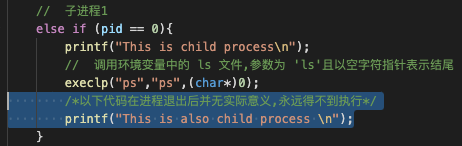
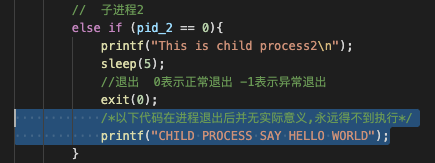
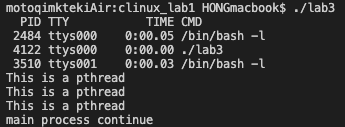
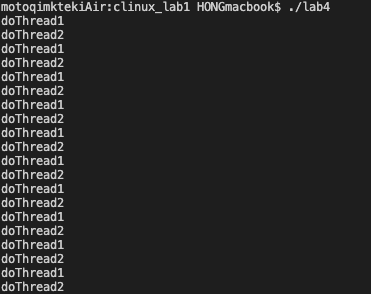
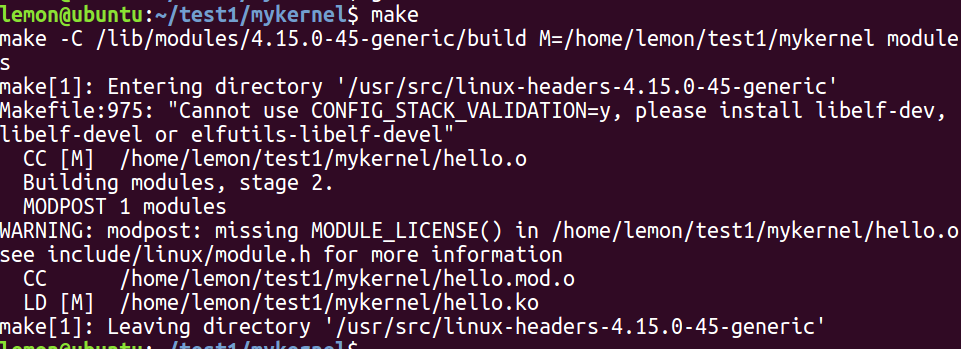
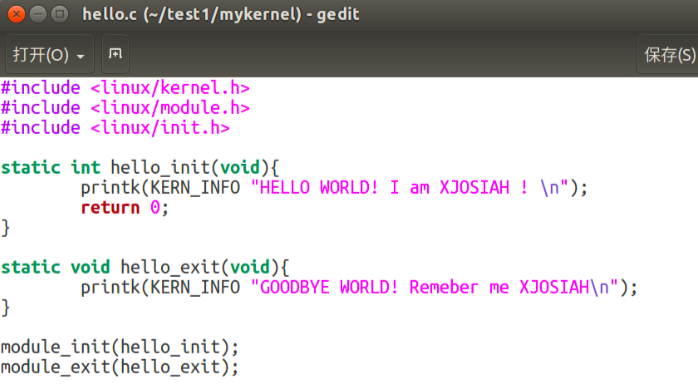
6、编写程序实现进程的管道通信。用系统调用pipe( )建立一管道，二个子进程P1和P2分别向管道各写一句话：

Child 1 is sending a message!

Child 2 is sending a message!

父进程从管道中读出二个来自子进程的信息并显示（要求先接收P1，后P2）。

7、编写一个HelloWorld内核模块，并进行装载和卸载操作

1. **实验结果分析（截屏的实验结果，与实验结果对应的实验分析）**
2. fork函数编写一个简单的多进程程序  
      
      
    程序创建了 两个子进程，子进程会输出进程的ID并且提醒当前处于哪个进程中，将两个子进程同时设置睡眠5秒钟并将父进程设置阻塞直到子进程完成，此时调用ps查看当前系统中运行的进程就可以看到创建出来的三个进程，当子进程完成后父进程也随之结束，此时程序才正式结束。
3. 分析列表中不同进程的内存占用情况  
      
   PS命令很难捕捉到子进程1调用命令的过程，因为这个进程执行的时间太短了  
      
    创建两个子进程，进程1调用exec执行ps命令，进程2睡眠5秒后直接退出不再进行任何操作。进程1调用exec后会变成另一个程序，不再按顺序执行原代码，故结合终端输出结果可看到在子进程1后续的输出并没有被执行：  
     
   同理，子进程2退出后，其后续输出操作也一样不会被执行：  
     
   父进程等待所有子进程结束后继续执行剩余代码，执行完毕后用ps命令可以看到所有进程全部结束：  
   
4. 一个最简单的多线程程序。理解多线程的运行和输出情况  
      
    pthread\_t可以用来创建一个线程标示符，结合pthread\_create函数就可以创建一个线程来执行特定的函数，函数至多可以有一个参数，调用线程时会自动执行函数，如果主线程没有拥塞等待，则多个线程会并发执行，利用pthread\_join就可以让主线程拥塞等待，于是可以看到子线程先执行、主线程后执行，且系统只有一个进程在执行。
5. 利用信号量机制控制多线程的运行顺序  
      
    这里采用了一个互斥信号量，程序执行时两个线程会分别抢占资源，图中线程1先占用了资源，由于是互斥资源，则在线程1输出信息后会释放资源，此时拥塞的线程2会就可以占用资源了，从而实现了两个线程交替执行，屏幕上交替输出线程1和2的信息。
6. 分析Linux系统下多进程与多线程中的区别  
    从实验1和3就可以看到最明显的差别，多进程会创建多个进程在执行，多线程实际 上还是同一个进程，只不过把执行单位再细化到了线程，多进程也可看成是多个单线程的程序在执行。多线程可以方便的使用同一进程中的资源，在多核处理器中可以将程序的性能进一步的释放出来，而多进程由于CPU处理的粒度只停留在以进程为单位的程度，只能看成的单线程执行，所以通常执行效果较差。
7. 编写程序实现进程的管道通信  
     
    实现先接受进程1再接受进程2时有两个方法，一种是可以先创建子进程1先执行，然后父进程执行等待子进程1执行完毕后再创建进程2，或者同时创建两个进程但在子进程1执行的时候如指导书中所使用的方法：给相关资源上锁并将该进程睡眠，用互斥操作让子进程2等待子进程1执行完毕方可占用相关资源。
8. 编写一个HelloWorld内核模块，并进行装载和卸载操作  
    先将内核模块进行编译  
    装入内核中并查看内核模块初始化的时候输出到内核中的信息  
     
    卸载内核模块并查看卸载时输出到内核中的信息  
     
    内核模块展示：  
   
9. **实验总结**
10. 实验4使用信号量进行进程间通信时，使用了实名信号量，需要通过sem\_open()函数实现信号量的初始化，而且在退出的时候需要sem\_close（）和sem\_unlink（）分别进行关闭信号量和卸载实名信号量
11. 写简单的内核模块的过程中，编写Makefile的时候必须注意缩进和空格的差别，本次实验过程中在使用make命令时一直出错，后来仔细查看实验指导书后才发现了这些细微的差别！这些小错误、小马虎反而往往会占用一大部分实验过程耗费的时间！

**思考题：**1、程序中的sleep(5)起什么作用？  
 使进程1和2有顺序执行，不至于偶然出现同时抢占资源导致程序错误，同时也方便我们观察实验的运行过程

1. 子进程1和2为什么也能对管道进行操作？

因为子进程1和2复制了父进程中的程序段，而管道是放在操作系统中的，利用write和read函数就可以进行操作了。

3、一个内核模块必须具备哪些内容？  
 用于初始化的函数和用于卸载的函数