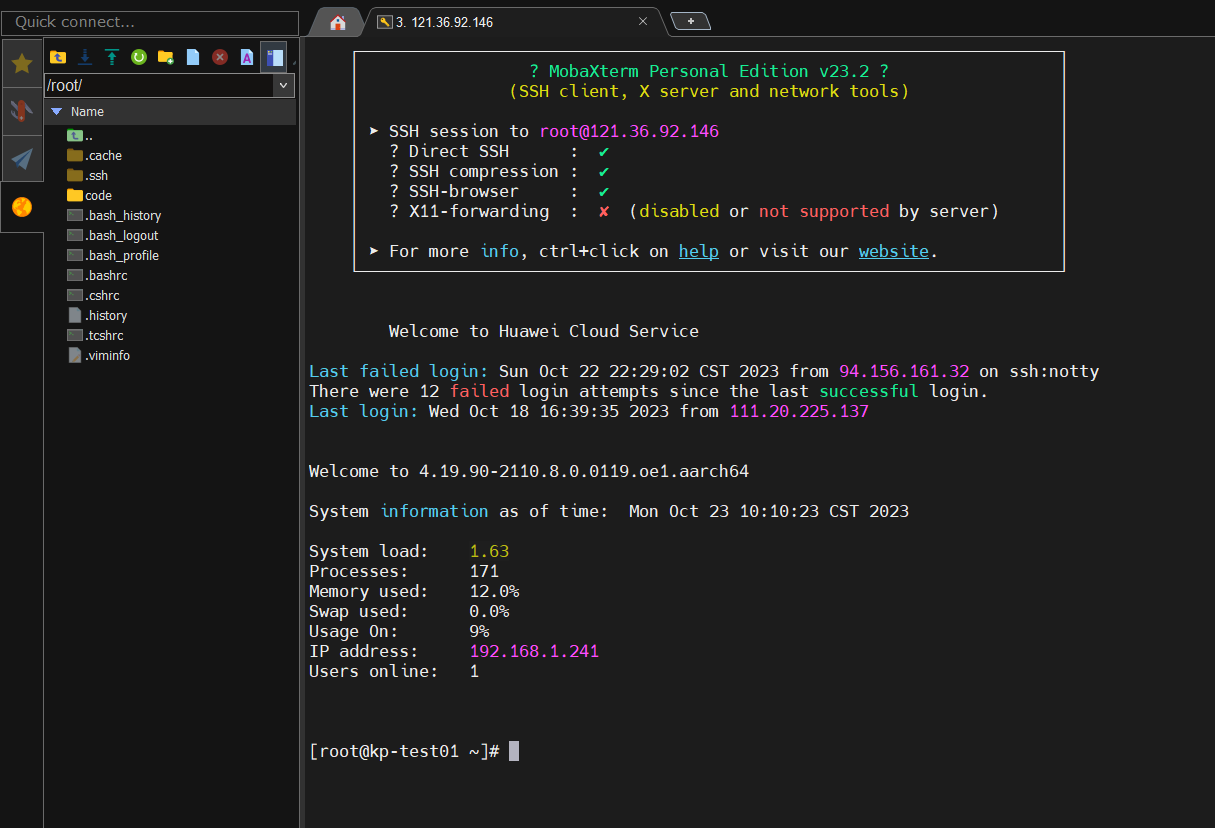
# os第一次实验

#### 实验前置 华为云环境搭建

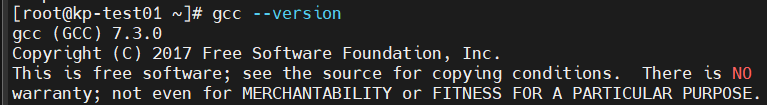
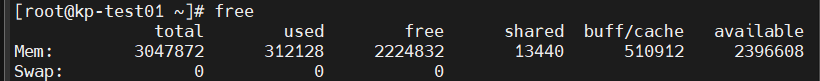
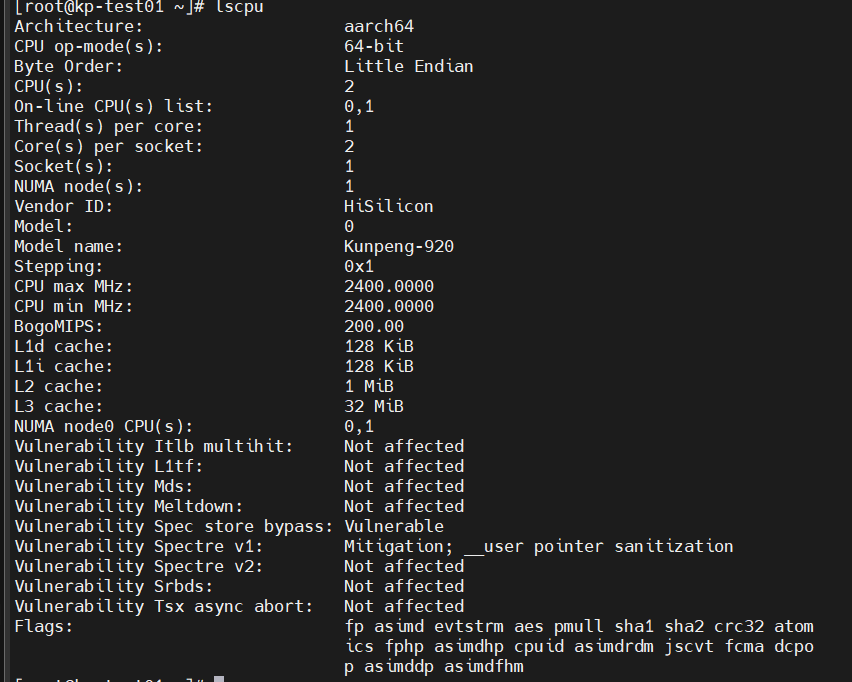
1. **在云端布置服务器**

* 
* 按实验指导书要求配置，服务器参数截图

1. **远程登陆服务器**

* 使用软件MobaXterm远程ssh登陆服务器 ip :121.36.92.146 登录用户：root 密码：（）
* 

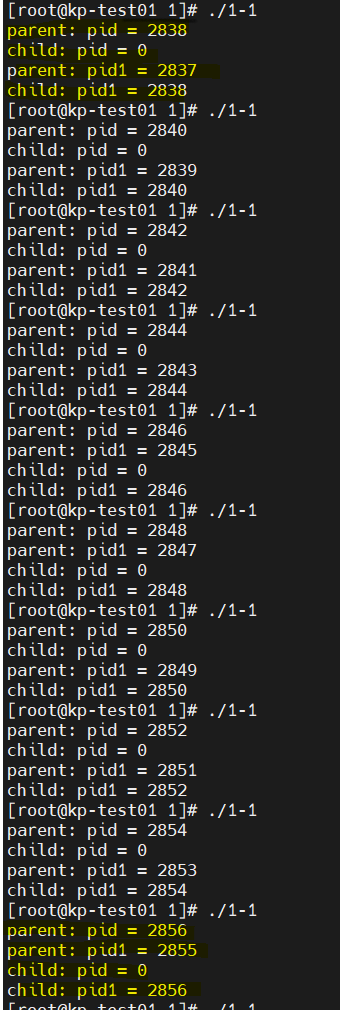
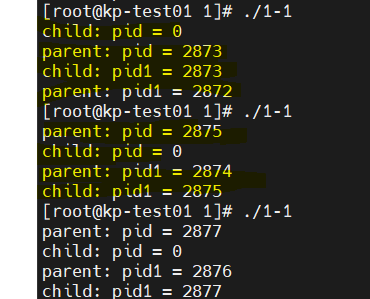
1. **查看服务器信息**

* ssh界面键入命令查看服务器的相关信息。
  + 查看gcc版本
  + 
  + 查看内存信息
  + 
  + 查看CPU信息
  + 

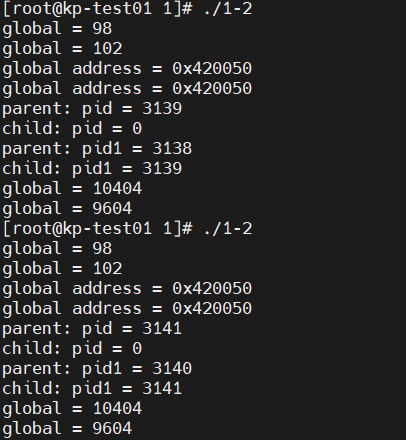
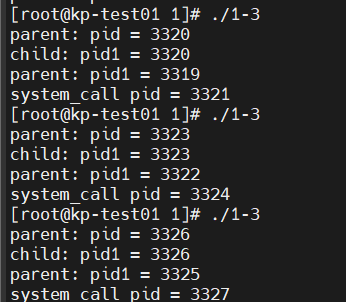
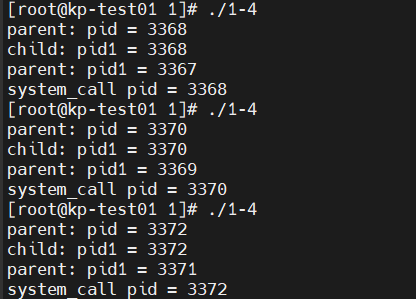
## os实验1：进程、线程相关编程实验

#### 1.1进程相关编程实验

1. **完成图1 .1 程序的运行**

* 
* 有实验截图可知父子进程**执行顺序并不固定**。
* 去除wait 后再观察结果
* 
* 在去掉wait()后，同样也是可能parent 先执行，又可能child 先执行。
* **理论分析：**
  + fork创建子进程后，父子进程并行执行，两者执行顺序由cpu调度决定，所以二者执行顺序不固定。
  + 对于子进程来说，fork(）后返回的pid为0，getpid返回当前进程（调用这一函数的进程，子进程的pid）所以父进程pid与子进程的pid1一样。
  + wait（）的作用是让父进程在子进程结束后继续执行,等待挂起，防止僵尸进程的出现.仍会出现parent 先执行，或child 先执行。

1. 扩展图1 1 的程序：

* 添加一个全局变量并在父进程和子进程中对这个变量做不同操作||在return 前增加对全局变量的操作并输出结果：
* 
* 定义全局变量global，初值100.在子进程加2，父进程减2。并返回全局变量地址，在return前做global平方操作。**发现二者global地址一样，但二者global改变是独立进行的。**
* **理论分析：**子进程“继承”父进程的变量，其地址总是一样的，因为在fork时整个虚拟地址空间被复制，但是虚拟地址空间所对应的物理内存却没有复制。所以对变量的操作是独立的。
* 调用system 函数和在子进程中调用exec 族函数：
* system 函数：
* 发现调用systemcall后**pid改变**，说明调用该函数创建了一个进程。
* 
* exec 族函数：
* 发现调用systemcall后pid未改变，**与child的pid一样**。
* 

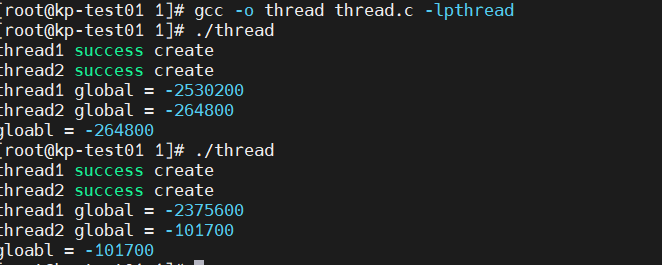
**理论分析：**

* 当进程调用exec函数时，该进程被完全替换为新程序。 因为**调用exec函数并不创建新进程**，所以前后进程的ID并没有改变
* system函数会执行参数要求的命令**创建新的进程**所以pid改变。

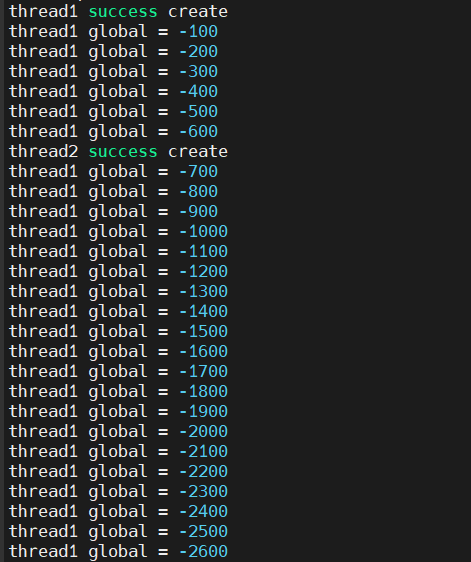
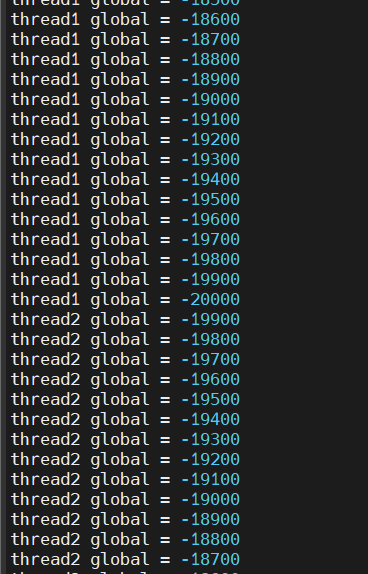
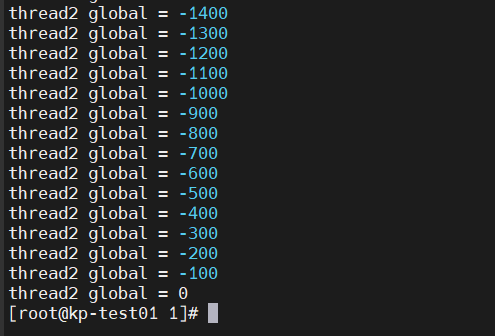
函数参考教程：[Linux系统学习——exec族函数、system函数、popen函数学习\_exec 跟system popen 区别-CSDN博客](https://blog.csdn.net/chinesekobe/article/details/107281995)

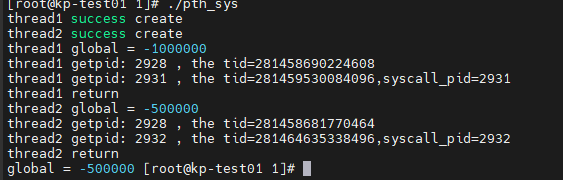
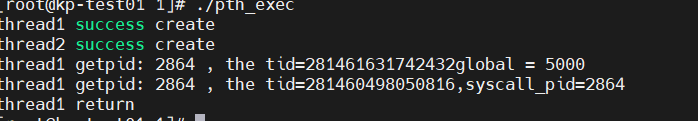
#### 1.2线程相关编程实验

1. 在进程中给一变量赋初值并成功创建两个线程||在两个线程中分别对此变量循环五千次以上做不同的操作

* 创建变量global（初值为0）两个线程分别执行加100和减100的操作。
* 
* 可以看出二者是并发执行。每次值都一样因为线程的执行并发，不能保证执行了相同的加和减的操作。
* 函数教程：[Linux——线程的创建\_linux 创建线程-CSDN博客](https://blog.csdn.net/qq_44824574/article/details/110672780)
* 编译问题：[Linux下undefined reference to ‘pthread\_create’问题解决-CSDN博客](https://blog.csdn.net/jiangxinyu/article/details/7778864)

1. 控制互斥和同步

* **使用pthread\_mutex\_函数对global变量进行互斥访问**。使线程1先执行，线程2后执行。
* 有图1可知thread2创建后仍在执行thread1的操作
* 
* thread1减法操作完thread2进行操作。
* 
* 最后结果为0
* 
* 函数教程：[Linux | 什么是互斥锁以及如何用代码实现互斥锁*linux实现互斥锁*瘦弱的皮卡丘的博客-CSDN博客](https://blog.csdn.net/ThinPikachu/article/details/113250072)

1. 调用系统函数和线程函数的比较
   * 调用system 函数
   * 使用syscall(SYS\_gettid)和pthread\_self()输出真实tid和tid，使用getpid()输出pid。
   * 
   * 线程1、2的getpid相同，线程编号不同。调用system时创建全新的进程，编号均不同。
   * 每个进程有一个pid（进程ID），获取函数：getpid()，系统内唯一，除了和自己的主线程一样
   * **主线程的pid和所在进程的pid一致，可以通过这个来判断是否是主线程**
   * 每个线程有一个tid（线程ID），获取函数：pthread\_self()，所在进程内唯一，有可能两个进程中都有同样一个tid
   * **每个线程有一个pid（,获取函数：syscall(SYS\_gettid)**，系统内唯一，除了主线程和自己的进程一样，其他子线程都是唯一的。在linux下每一个进程都一个进程id，类型pid\_t，可以由 **getpid()获取。**  
     POSIX线程也有线程id，类型pthread\_t，可以由 pthread\_self()获取，线程id由线程库维护。  
     但是各个进程独立，所以**会有不同进程中线程号相同节的情况**。
   * 进程id不可以，线程id又可能重复，所以这里会有一个**真实的线程id唯一标识，tid。可以通过linux下的系统调用syscall(SYS\_gettid)来获得**。
   * 调用exec 族函数
   * 
   * 指行exec函数后，原来的进程被调用的内容取代thread2的systemcall不会再进行。
   * 所以调用的systemcall产生了输出，此时systemcall为主进程所以syscall(SYS\_gettid)与pid一样。

pid问题：[linux中线程的pid，线程的tid和线程pid以及\_\_thread-CSDN博客](https://blog.csdn.net/qq_34489443/article/details/100585685?spm=1001.2101.3001.6650.1&utm_medium=distribute.pc_relevant.none-task-blog-2~default~CTRLIST~Rate-1-100585685-blog-51248051.235^v38^pc_relevant_sort_base2&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2~default~CTRLIST~Rate-1-100585685-blog-51248051.235^v38^pc_relevant_sort_base2&utm_relevant_index=2)

[【编程基础の基础】syscall(SYS\_gettid)\_sys\_getpid-CSDN博客](https://blog.csdn.net/weixin_42157432/article/details/115832748)

#### 1.3自旋锁实验

#include <stdio.h>
  
#include <pthread.h>
  
  
typedef struct
  
{
  
 int flag;
  
} spinlock\_t;
  
  
// 初始化自旋锁
  
void spinlock\_init(spinlock\_t \*lock)
  
{
  
 lock->flag = 0;
  
}
  
  
void spinlock\_lock(spinlock\_t \*lock)
  
{
  
 while (\_\_sync\_lock\_test\_and\_set(&lock->flag, 1))
  
 {
  
 // 自旋等待
  
 }
  
}
  
  
void spinlock\_unlock(spinlock\_t \*lock)
  
{
  
 \_\_sync\_lock\_release(&lock->flag);
  
}
  
  
int shared\_value = 0;
  
  
// 线程函数
  
void \*thread\_function(void \*arg)
  
{
  
 spinlock\_t \*lock = (spinlock\_t \*)arg;
  
  
 for (int i = 0; i < 5000; ++i)
  
 {
  
 spinlock\_lock(lock);
  
 shared\_value++;
  
 spinlock\_unlock(lock);
  
 }
  
  
 return NULL;
  
}
  
  
int main()
  
{
  
 pthread\_t thread1, thread2;
  
 spinlock\_t lock;
  
 int status;
  
 spinlock\_init(&lock);
  
  
 // 输出共享变量的值
  
 printf("initial: %d\n", shared\_value);
  
  
 // thread 1
  
 status = pthread\_create(&thread1, NULL, thread\_function, &lock);
  
 if (status != 0)
  
 {
  
 printf("thread1 default = %d\n ", status);
  
 return 1;
  
 }
  
 printf("thread1 success create\n");
  
  
 // thread 2
  
 pthread\_create(&thread2, NULL, thread\_function, &lock);
  
 if (status != 0)
  
 {
  
 printf("threa2 default = %d\n ", status);
  
 return 1;
  
 }
  
 printf("thread2 success create\n");
  
 // 等待线程结束
  
 pthread\_join(thread1, NULL);
  
 pthread\_join(thread2, NULL);
  
  
 // 输出共享变量的值
  
 printf("final: %d\n", shared\_value);
  
  
 return 0;
  
}

定义了一个spinlock\_t结构体，用于表示自旋锁。spinlock\_init函数用于初始化自旋锁，spinlock\_lock函数用于获取自旋锁，spinlock\_unlock函数用于释放自旋锁。

在线程函数thread\_function中，通过调用spinlock\_lock和spinlock\_unlock函数来保护对共享变量shared\_value的访问。每个线程循环执行5000次，每次获取自旋锁后将共享变量加1，然后释放自旋锁。

