os第一次实验

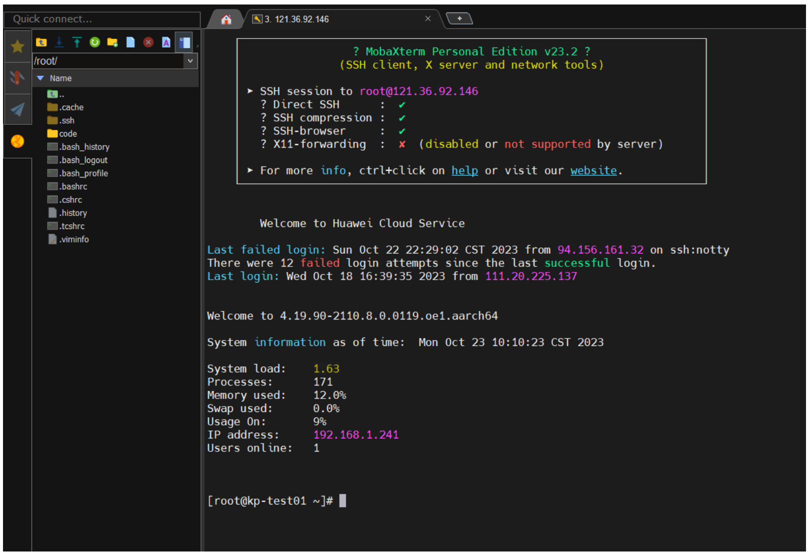
实验前置 华为云环境搭建

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

2. **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： ()



3. **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

```
[root@kp-test01 ~]# gcc --version
gcc (GCC) 7.3.0
Copyright (C) 2017 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

   - 查看内存信息

```
[root@kp-test01 ~]# free
              total        used        free      shared  buff/cache   available
Mem:        3047872      312128     2224832       13440      510912     2396608
Swap:             0           0           0
```

   - 查看CPU信息

```
[root@kp-test01 ~]# lscpu
Architecture:                    aarch64
CPU op-mode(s):                  64-bit
Byte Order:                      Little Endian
CPU(s):                          2
On-line CPU(s) list:             0,1
Thread(s) per core:              1
Core(s) per socket:              2
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       HiSilicon
Model:                           0
Model name:                      Kunpeng-920
Stepping:                        0x1
CPU max MHz:                     2400.0000
CPU min MHz:                     2400.0000
BogoMIPS:                        200.00
L1d cache:                       128 KiB
L1i cache:                       128 KiB
L2 cache:                        1 MiB
L3 cache:                        32 MiB
NUMA node0 CPU(s):               0,1
Vulnerability Itlb multihit:     Not affected
Vulnerability L1tf:              Not affected
Vulnerability Mds:               Not affected
Vulnerability Meltdown:          Not affected
Vulnerability Spec store bypass: Vulnerable
Vulnerability Spectre v1:        Mitigation; __user pointer sanitization
Vulnerability Spectre v2:        Not affected
Vulnerability Srbds:             Not affected
Vulnerability Tsx async abort:   Not affected
Flags:                           fp asimd evtstrm aes pmull sha1 sha2 crc32 atom
                                 ics fphp asimdhp cpuid asimdrdm jscvt fcma dcpo
                                 p asimddp asimdfhm
```

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**

```
[root@kp-test01 1]# ./1-1
parent: pid = 2838
child: pid = 0
parent: pid1 = 2837
child: pid1 = 2838
[root@kp-test01 1]# ./1-1
parent: pid = 2840
child: pid = 0
parent: pid1 = 2839
child: pid1 = 2840
[root@kp-test01 1]# ./1-1
parent: pid = 2842
child: pid = 0
parent: pid1 = 2841
child: pid1 = 2842
[root@kp-test01 1]# ./1-1
parent: pid = 2844
child: pid = 0
parent: pid1 = 2843
child: pid1 = 2844
[root@kp-test01 1]# ./1-1
parent: pid = 2846
parent: pid1 = 2845
child: pid = 0
child: pid1 = 2846
[root@kp-test01 1]# ./1-1
parent: pid = 2848
parent: pid1 = 2847
child: pid = 0
child: pid1 = 2848
[root@kp-test01 1]# ./1-1
parent: pid = 2850
child: pid = 0
parent: pid1 = 2849
child: pid1 = 2850
[root@kp-test01 1]# ./1-1
parent: pid = 2852
child: pid = 0
parent: pid1 = 2851
child: pid1 = 2852
[root@kp-test01 1]# ./1-1
parent: pid = 2854
child: pid = 0
parent: pid1 = 2853
child: pid1 = 2854
[root@kp-test01 1]# ./1-1
parent: pid = 2856
parent: pid1 = 2855
child: pid = 0
child: pid1 = 2856
[root@kp-test01 1]# ./1-1
```

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

```
[root@kp-test01 1]# ./1-1
child: pid = 0
parent: pid = 2873
child: pid1 = 2873
parent: pid1 = 2872
[root@kp-test01 1]# ./1-1
parent: pid = 2875
child: pid = 0
parent: pid1 = 2874
child: pid1 = 2875
[root@kp-test01 1]# ./1-1
parent: pid = 2877
child: pid = 0
parent: pid1 = 2876
child: pid1 = 2877
```

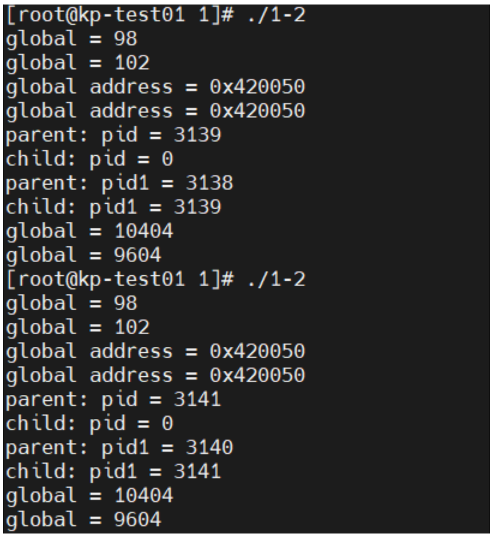
在去掉wait()后，同样也是可能parent 先执行，又可能child 先执行。

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。

2. 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:



```
[root@kp-test01 1]# ./1-2
global = 98
global = 102
global address = 0x420050
global address = 0x420050
parent: pid = 3139
child: pid = 0
parent: pid1 = 3138
child: pid1 = 3139
global = 10404
global = 9604
[root@kp-test01 1]# ./1-2
global = 98
global = 102
global address = 0x420050
global address = 0x420050
parent: pid = 3141
child: pid = 0
parent: pid1 = 3140
child: pid1 = 3141
global = 10404
global = 9604
```

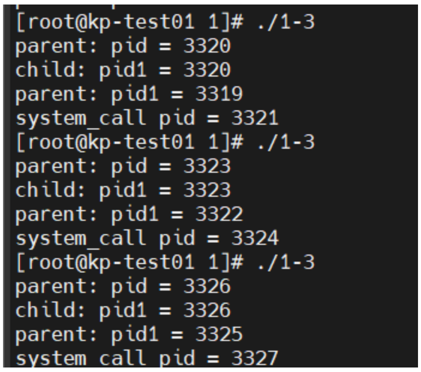
定义全局变量global，初值100.在子进程加2，父进程减2。并返回全局变量地址，在return前做global平方操作。**发现二者global地址一样，但二者global改变是独立进行的。**

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。



```
[root@kp-test01 1]# ./1-3
parent: pid = 3320
child: pid1 = 3320
parent: pid1 = 3319
system_call pid = 3321
[root@kp-test01 1]# ./1-3
parent: pid = 3323
child: pid1 = 3323
parent: pid1 = 3322
system_call pid = 3324
[root@kp-test01 1]# ./1-3
parent: pid = 3326
child: pid1 = 3326
parent: pid1 = 3325
system_call pid = 3327
```

exec 族函数:

发现调用systemcall后pid未改变，**与child的pid一样。**

```
[root@kp-test01 1]# ./1-4
parent: pid = 3368
child: pid1 = 3368
parent: pid1 = 3367
system_call pid = 3368
[root@kp-test01 1]# ./1-4
parent: pid = 3370
child: pid1 = 3370
parent: pid1 = 3369
system_call pid = 3370
[root@kp-test01 1]# ./1-4
parent: pid = 3372
child: pid1 = 3372
parent: pid1 = 3371
system_call pid = 3372
```

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]

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。

   ```
   [root@kp-test01 1]# gcc -o thread thread.c -lpthread
   [root@kp-test01 1]# ./thread
   thread1 success create
   thread2 success create
   thread1 global = -2530200
   thread2 global = -264800
   gloabl = -264800
   [root@kp-test01 1]# ./thread
   thread1 success create
   thread2 success create
   thread1 global = -2375600
   thread2 global = -101700
   gloabl = -101700
   ```

   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]

   编译问题：[Linux下undefined reference to ‘pthread_create’问题解决-CSDN博客]

2. 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 success create
thread1 global = -100
thread1 global = -200
thread1 global = -300
thread1 global = -400
thread1 global = -500
thread1 global = -600
thread2 success create
thread1 global = -700
thread1 global = -800
thread1 global = -900
thread1 global = -1000
thread1 global = -1100
thread1 global = -1200
thread1 global = -1300
thread1 global = -1400
thread1 global = -1500
thread1 global = -1600
thread1 global = -1700
thread1 global = -1800
thread1 global = -1900
thread1 global = -2000
thread1 global = -2100
thread1 global = -2200
thread1 global = -2300
thread1 global = -2400
thread1 global = -2500
thread1 global = -2600
```

thread1减法操作完thread2进行操作。

```
thread1 global = -18600
thread1 global = -18700
thread1 global = -18800
thread1 global = -18900
thread1 global = -19000
thread1 global = -19100
thread1 global = -19200
thread1 global = -19300
thread1 global = -19400
thread1 global = -19500
thread1 global = -19600
thread1 global = -19700
thread1 global = -19800
thread1 global = -19900
thread1 global = -20000
thread2 global = -19900
thread2 global = -19800
thread2 global = -19700
thread2 global = -19600
thread2 global = -19500
thread2 global = -19400
thread2 global = -19300
thread2 global = -19200
thread2 global = -19100
thread2 global = -19000
thread2 global = -18900
thread2 global = -18800
thread2 global = -18700
```

最后结果为0

```
thread2 global = -1400
thread2 global = -1300
thread2 global = -1200
thread2 global = -1100
thread2 global = -1000
thread2 global = -900
thread2 global = -800
thread2 global = -700
thread2 global = -600
thread2 global = -500
thread2 global = -400
thread2 global = -300
thread2 global = -200
thread2 global = -100
thread2 global = 0
[root@kp-test01 1]#
```

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]()

3. 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。

```
[root@kp-test01 1]# ./pth_sys
thread1 success create
thread2 success create
thread1 global = -1000000
thread1 getpid: 2928 , the tid=281458690224608
thread1 getpid: 2931 , the tid=281459530084096,syscall_pid=2931
thread1 return
thread2 global = -500000
thread2 getpid: 2928 , the tid=281458681770464
thread2 getpid: 2932 , the tid=281464635338496,syscall_pid=2932
thread2 return
global = -500000 [root@kp-test01 1]#
```

     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

```
[root@kp-test01 1]# ./pth_exec
thread1 success create
thread2 success create
thread1 getpid: 2864 , the tid=281461631742432global = 5000
thread1 getpid: 2864 , the tid=281460498050816,syscall_pid=2864
thread1 return
```

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]()

[【编程基础の基础】syscall(SYS_gettid)_sys_getpid-CSDN博客]()

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);
    }
```

```c
    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。

```
[root@kp-test01 1]# ./spinlock
initial: 0
thread1 success create
thread2 success create
final: 10000
[root@kp-test01 1]#
```