实验一 进程、线程相关编程实验

1.1进程相关编程实验

1.1.1 实验目的

（1）熟悉 Linux 操作系统的基本环境和操作方法，通过运行系统命令查看系统基本信息以了解系统；

（2）编写并运行简单的进程调度相关程序，体会进程调度、进程间变量的管理等机制在操作系统实际运行中的作用。

1.1.2 实验内容

（1）熟悉操作命令、编辑、编译、运行程序。完成图 1-1 程序的运行验证，多运行几次程序观察结果；去除 wait 后再观察结果并进行理论分析。

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

int main()
{
pid_t pid, pid1;

   /* fork a child process */
   pid = fork();

   if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
   }
   else if (pid == 0) { /* child process */
      pid1 = getpid();
      printf("child: pid = %d",pid); /* A */
      printf("child: pid1 = %d",pid1); /* B */
   }
   else { /* parent process */
      pid1 = getpid();
      printf("parent: pid = %d",pid); /* C */
      printf("parent: pid1 = %d",pid1); /* D */
      wait(NULL);
   }

   return 0;
}
```

图 1-1 教材中所给代码（p103 作业 3.7）

（2）扩展图 1-1 的程序：

a） 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，输出操作结果并解释；

b） 在 return 前增加对全局变量的操作并输出结果，观察并解释；

c） 修改程序体会在子进程中调用 system 函数和在子进程中调用 exec 族函数；

1.1.3 实验原理

（1）进程：进程是计算机科学中的一个重要概念，它是操作系统中的基本执行单位。进程代表着一个正在执行的程序实例，它包括了程序的代码、数据和执行状态等信息。操作系统通过进程管理来实现对计算机资源的有效分配和控制；

（2）PID：PID 是进程标识符（Process Identifier）的缩写，它是用来唯一标识一个操作系统中的进程的数值。每个正在运行或已经终止的进程都会被分配一个唯一的 PID，这个标识符可以用来在操作系统内部识别和管理进程；

（3）fork()函数：fork() 是一个在类 Unix 操作系统中常见的系统调用，用于创建一个新的进程，新进程是原进程（父进程）的副本。新进程被称为子进程，它与父进程共享很多资源，但也有一些独立的属性。fork() 被用于实现多进程编程，常见于操作系统和并发编程中。函数返回一个整数，如果返回值为负数，则表示创建进程失败。如果返回值为 0，表示当前正在执行的代码是在子进程中。如果返回值大于 0，表示当前正在执行的代码是在父进程中，返回值是子进程的 PID。调用 fork() 函数时，操作系统会创建一个新的进程，该进程是调用进程的一个副本，称为子进程。子进程几乎与父进程相同，包括代码、数据、文件描述符等。但是子进程拥有自己的独立的内存空间和资源。

1.1.4 实验步骤

本实验通过在程序中输出父、子进程的 pid，分析父子进程 pid 之间的关系，进一步加入 wait()函数分析其作用。

步骤一：编写并多次运行图 1-1 中代码（运行次数不限，我们默认大家已经安装较新版本的 Linux 系统）。

```
[root@localhost ~]# gcc 1-1.c -o 1-1
[root@localhost ~]# ./1-1
parent:pid =20097parent:pid1 =20096child:pid =0child:pid1 =20097[root@localhost ~]#
[root@localhost ~]# ./1-1
parent:pid =20111parent:pid1 =20110child:pid =0child:pid1 =20111[root@localhost ~]#
[root@localhost ~]# ./1-1
parent:pid =20125parent:pid1 =20124child:pid =0child:pid1 =20125[root@localhost ~]#
[root@localhost ~]#
```

图 1-2 步骤一运行实例

步骤二：删去图 1-1 代码中的 wait()函数并多次运行程序，分析运行结果。

```
[root@localhost ~]# gcc 1-1.c -o 1-1
[root@localhost ~]# ./1-1
parent:pid =23397parent:pid1 =23396child:pid =0child:pid1 =23397[root@localhost ~]#
[root@localhost ~]# ./1-1
parent:pid =23423parent:pid1 =23422child:pid =0child:pid1 =23423[root@localhost ~]#
[root@localhost ~]# ./1-1
parent:pid =23437parent:pid1 =23436child:pid =0child:pid1 =23437[root@localhost ~]#
[root@localhost ~]# ./1-1
parent:pid =23451parent:pid1 =23450child:pid =0child:pid1 =23451[root@localhost ~]#
[root@localhost ~]#
```

图 1-3 步骤二运行实例

步骤三：修改图 1-1 中代码，增加一个全局变量并在父子进程中对其进行不同的操作（自行设计），观察并解释所做操作和输出结果。

```
[root@localhost ~]# gcc 1-1.c -o 1-1
[root@localhost ~]# ./1-1
parent:value =-1parent:*value =4210828child:value =1child:*value =4210828[root@localhost ~]#
[root@localhost ~]# ./1-1
parent:value =-1parent:*value =4210828child:value =1child:*value =4210828[root@localhost ~]#
[root@localhost ~]# ./1-1
parent:value =-1parent:*value =4210828child:value =1child:*value =4210828[root@localhost ~]#
[root@localhost ~]# ./1-1
parent:value =-1parent:*value =4210828child:value =1child:*value =4210828[root@localhost ~]#
[root@localhost ~]# ./1-1
parent:value =-1parent:*value =4210828child:value =1child:*value =4210828[root@localhost ~]#
[root@localhost ~]#
```

图 1-4 步骤三运行实例

步骤四：在步骤三基础上，在 return 前增加对全局变量的操作（自行设计）并输出结果，观察并解释所做操作和输出结果。

```
[root@localhost ~]# gcc 1-1.c -o 1-1
[root@localhost ~]# ./1-1
parent:value =-1parent:*value =4210828child:value =1child:*value =4210828
 brefore return value =6, *value =4210828
 brefore return value =4, *value =4210828[root@localhost ~]#
[root@localhost ~]# ./1-1
parent:value =-1parent:*value =4210828child:value =1child:*value =4210828
 brefore return value =6, *value =4210828
 brefore return value =4, *value =4210828[root@localhost ~]#
[root@localhost ~]# ./1-1
parent:value =-1parent:*value =4210828child:value =1child:*value =4210828
 brefore return value =6, *value =4210828
 brefore return value =4, *value =4210828[root@localhost ~]#
```

图 1-5 步骤四运行实例

步骤五：修改图 1-1 程序，在子进程中调用 system()与 exec 族函数。编写 system_call.c 文件输出进程号 PID，编译后生成 system_call 可执行文

件。在子进程中调用 system_call,观察输出结果并分析总结。

```
[root@localhost ~]# ./1-1
parent process PID: 249749
child process1 PID: 249750
system_call PID: 249751
child process PID: 249750
[root@localhost ~]#
```

图 1-6 步骤五 system 函数运行实例

```
[root@localhost ~]# ./1-1
parent process PID: 251569
child process1 PID: 251570
system_call PID: 251570
[root@localhost ~]#
```

图 1-7 步骤五 exec 族函数运行实例

1.2线程相关编程实验

1.2.1 实验目的

探究多线程编程中的线程共享进程信息。在计算机编程中，多线程是一种常见的并发编程方式，允许程序在同一进程内创建多个线程，从而实现并发执行。由于这些线程共享同一进程的资源，包括内存空间和全局变量，因此可能会出现线程共享进程信息的现象。本实验旨在通过创建多个线程并使其共享进程信息，以便深入了解线程共享资源时可能出现的问题。

1.2.2 实验内容

（1）在进程中给一变量赋初值并成功创建两个线程；

（2）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（自行设计）并输出结果；

（3）多运行几遍程序观察运行结果，如果发现每次运行结果不同，请解释原因并修改程序解决，考虑如何控制互斥和同步；

（4）将任务一中第一个实验调用 system 函数和调用 exec 族函数改成在线程中实现，观察运行结果输出进程 PID 与线程 TID 进行比较并说明原因。

1.2.3 实验原理

本实验旨在通过创建两个线程，它们分别对一个共享的变量进行多次循环操

作，并观察在多次运行实验时可能出现的不同结果。在观察到结果不稳定的情况下，引入互斥和同步机制来确保线程间的正确协同操作。

（1）线程创建与变量操作： 首先，在一个进程内创建两个线程，并在进程内部初始化一个共享的变量。这两个线程将并发地对这个共享变量进行循环操作，执行不同的操作。

（2）竞态条件和不稳定结果： 由于线程并发执行，存在竞态条件，即两个线程可能同时读取和修改共享变量。在没有适当的同步措施的情况下，不同线程的操作可能会交叉执行，导致结果不稳定，每次运行可能都会得到不同的结果。

（3）互斥与同步：为了解决竞态条件带来的问题，可以使用互斥锁（Mutex）来保护共享变量的访问。在每个线程对变量进行操作之前，先获取互斥锁，操作完成后再释放锁。这样一来，每次只有一个线程能够访问变量，从而避免了并发访问带来的不稳定性。

（4）观察结果与比较： 运行多次实验，观察使用互斥锁后的运行结果。应该可以发现，通过互斥锁的保护，不再出现不稳定的结果，每次运行得到的结果都是一致的。

（5）调用系统函数和线程函数的比较： 在任务一中，如果将调用系统函数和调用 exec 族函数改成在线程中实现，观察运行结果。可以发现，调用系统函数和 exec 族函数时，会输出进程的 PID（Process ID），而在线程中运行时，会输出线程的 TID（Thread ID）。这是因为线程是进程的子任务，它们共享进程的资源，但有自己的执行流程。

1.2.4 实验步骤

步骤一：设计程序，创建两个子线程，两线程分别对同一个共享变量多次操作，观察输出结果。

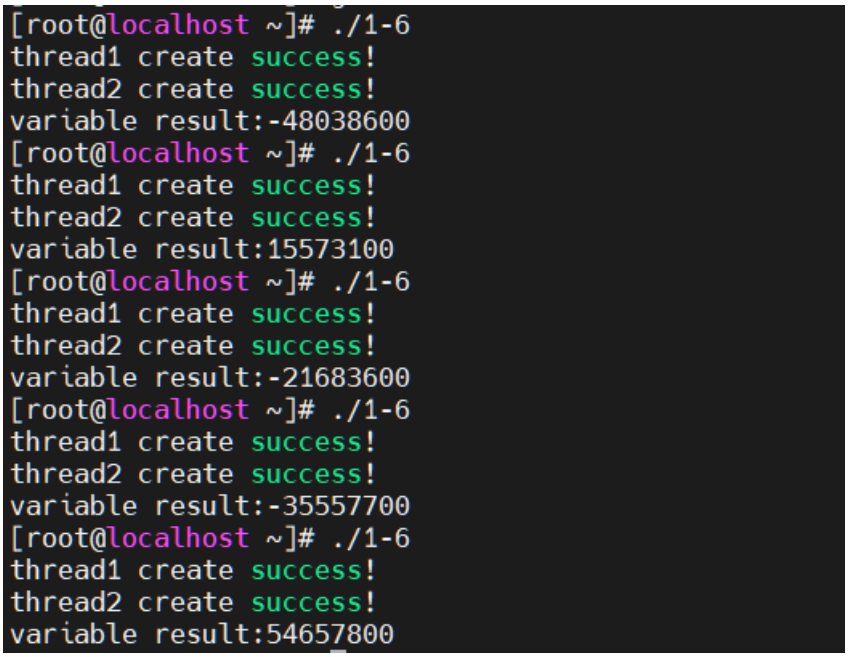

图 1-8 步骤一运行结果

上图中程序所做操作：定义共享变量初始值为 0，两个线程分别对其进行 100000 次+/- 100 操作，最终在主进程中输出处理后的变量值，可以观察到出现图中结果，尝试分析原因。

步骤二：修改程序，定义信号量 signal，使用 PV 操作实现共享变量的访问与互斥。运行程序，观察最终共享变量的值。

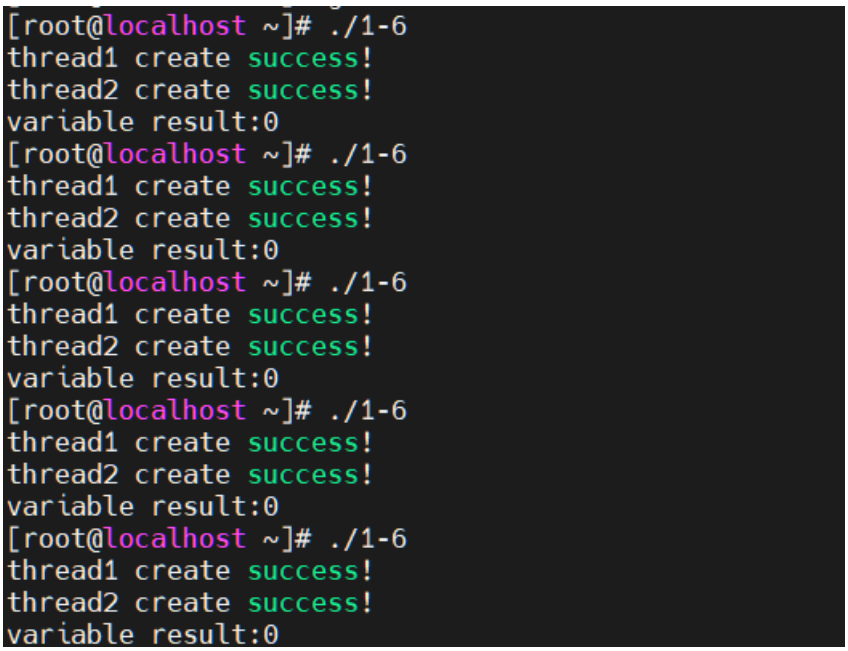

图 1-9 步骤二运行结果

引入 PV 操作对变量进行加锁后，多次运行观察到图 1-9 所示结果，分析出

现与步骤一不同结果的原因，并尝试灵活运用信号量和 PV 操作实现线程间的同步互斥。

步骤三：在第一部分实验了解了 system()与 exec 族函数的基础上，将这两个函数的调用改为在线程中实现，输出进程 PID 和线程的 TID 进行分析。

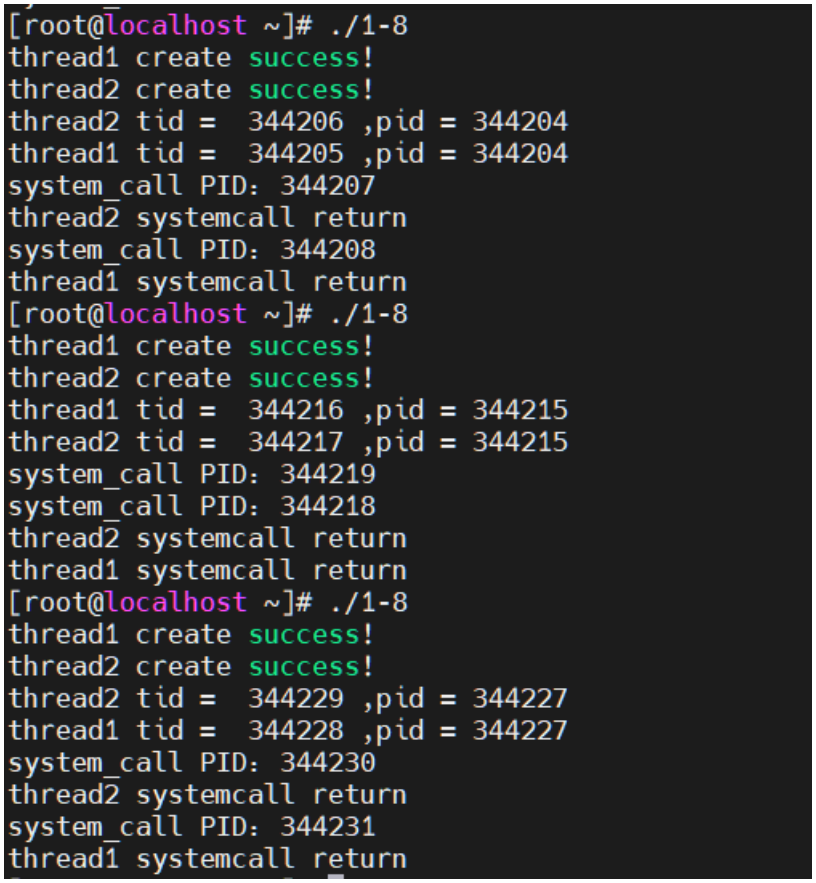

图 1-10 步骤三运行结果

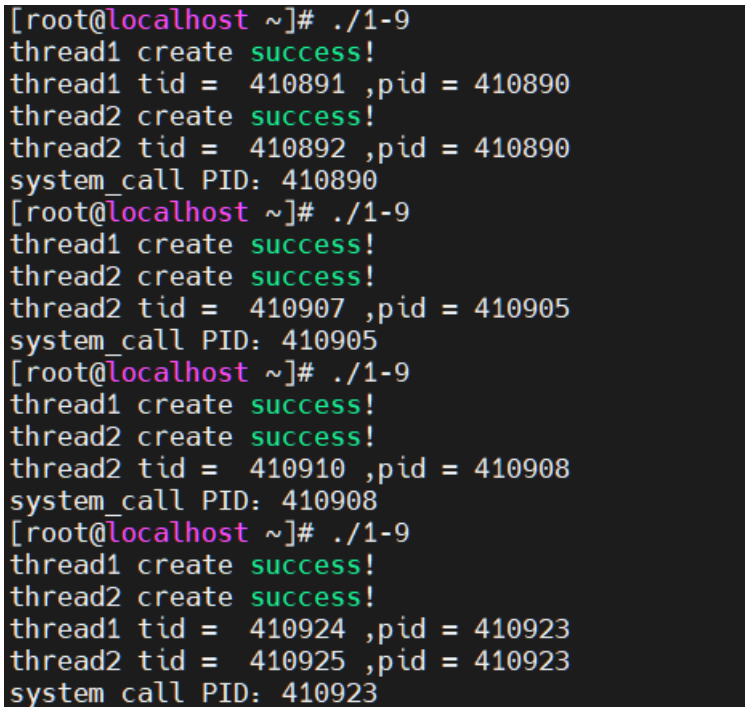

图 1-11 步骤三运行结果

图 1-10 展示了在线程中调用了 system()函数，多次运行的结果。图 1-11 展示了在线程中调用 exec 族函数，多次运行的结果。对所得结果进行比较并分析原因。

1.3自旋锁实验

1.3.1 实验目的

自旋锁作为一种并发控制机制，可以在特定情况下提高多线程程序的性能。本实验旨在通过设计一个多线程的实验环境，以及使用自旋锁来实现线程间的同步，从而实现以下目标：

（1）了解自旋锁的基本概念： 通过研究自旋锁的工作原理和特点，深入理解自旋锁相对于其他锁机制的优势和局限性；

（2）实验自旋锁的应用： 在一个多线程的实验环境中，设计一个竞争资源的场景，让多个线程同时竞争对该资源的访问；

（3）实现自旋锁的同步： 使用自旋锁来保护竞争资源的访问，确保同一时间只有一个线程可以访问该资源，避免数据不一致和竞态条件；

1.3.2 实验内容

（1）在进程中给一变量赋初值并成功创建两个线程；

（2）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（自行设计）并输出结果；

（3）使用自旋锁实现互斥和同步；

1.3.3 实验原理

自旋锁是一种基于忙等待（busy-waiting）的同步机制，用于在线程竞争共享资源时，不断尝试获取锁，而不是阻塞等待。它的工作原理可以简单地概括为以下几个步骤：

（1）初始化锁： 自旋锁的开始是一个共享的标志变量（flag），最初为未锁定状态（0）。这个标志变量用于表示资源是否已被其他线程占用。

（2）获取锁： 当一个线程尝试获取锁时，它会循环检查标志变量的状态。如果发现标志变量是未锁定状态（0），那么该线程将通过原子操作将标志变量设置为锁定状态（1），从而成功获取锁。如果标志变量已经是锁定状态，线程会一直在循环中等待，直到标志变量变为未锁定状态为止。

（3）释放锁： 当持有锁的线程完成对共享资源的操作后，它会通过原子操作将标志变量设置回未锁定状态（0），从而释放锁，允许其他等待的线程尝试获取锁。

自旋锁的工作原理中关键的部分在于“自旋”这一概念，即等待获取锁的线程会循环忙等待，不断检查标志变量的状态，直到能够成功获取锁。这种方式在锁的占用时间很短的情况下可以减少线程切换的开销，提高程序性能。

1.3.4 实验步骤

步骤一：根据实验内容要求，编写模拟自旋锁程序代码 spinlock.c，待补充主函数的示例代码如下：

```
/**
*spinlock.c
*in xjtu
*2023.8
*/
#include <stdio.h>
#include <pthread.h>
```

```
// 定义自旋锁结构体
typedef struct {
    int flag;
} spinlock_t;

// 初始化自旋锁
void spinlock_init(spinlock_t *lock) {
    lock->flag = 0;
}

// 获取自旋锁
void spinlock_lock(spinlock_t *lock) {
    while (__sync_lock_test_and_set(&lock->flag, 1)) {
        // 自旋等待
    }
}

// 释放自旋锁
void spinlock_unlock(spinlock_t *lock) {
    __sync_lock_release(&lock->flag);
}

// 共享变量
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;
    // 输出共享变量的值

    // 初始化自旋锁

    // 创建两个线程

    // 等待线程结束

    // 输出共享变量的值
        return 0;
}
```

步骤二：补充完成代码后，编译并运行程序，分析运行结果

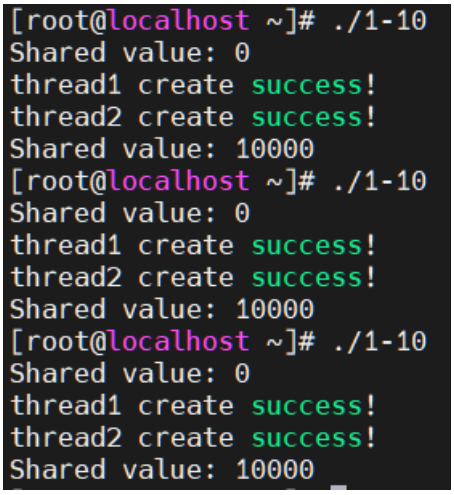

图 1-12 自旋锁运行结果