README

实验1：openEuler系统环境实验

Process

(1)

Problems:

1. 使用gcc命令编译时，忘记进入test文件目录下，导致error的发生；

```
[root@kp-test01 ~]# gcc -o lab1 lab1.c
gcc: error: lab1.c: No such file or directory
gcc: fatal error: no input files
compilation terminated.
```

2. 使用cd /usr/local/src/test 命令成功进入test文件的目录下，尝试运行lab1.c文件发生error；

```
[root@kp-test01 ~]# cd /usr/local/src/test
[root@kp-test01 test]# ls
hello  hello.c  lab1.c
[root@kp-test01 test]# gcc -o lab1 lab1.c
lab1.c:2:10: fatal error: sys/type.h: No such file or directory
 #include <sys/type.h>
          ^~~~~~~~~~~~

compilation terminated.
```

发现头文件拼写错误，修改后发现问题并未解决；

3. 仅在windows下修改C源文件并保存，不会在Euler中修改，必须在winscp中修改才能成功对Euler中的内容进行修改；

4. 发现报错wait函数未声明，增加头文件<wait.h>；

```
[root@kp-test01 test]# gcc -o lab1 lab1.c
lab1.c: In function ‘main’:
lab1.c:22:3: warning: implicit declaration of function ‘wait’; did you mean ‘main’? [-Wimplicit-function-declaratio
n]
   wait(NULL);
   ^~~~
   main
```

以上4个问题解决后，运行成功。以下是多次运行程序后的结果：

```
[root@kp-test01 test]# ./lab1
parent: pid = 2720
child: pid = 0
parent: pid1 = 2719
child: pid1 = 2720
[root@kp-test01 test]# ./lab1
parent: pid = 2723
child: pid = 0
parent: pid1 = 2722
child: pid1 = 2723
[root@kp-test01 test]# ./lab1
parent: pid = 2725
child: pid = 0
parent: pid1 = 2724
child: pid1 = 2725
```

将wait函数去除后的运行结果：

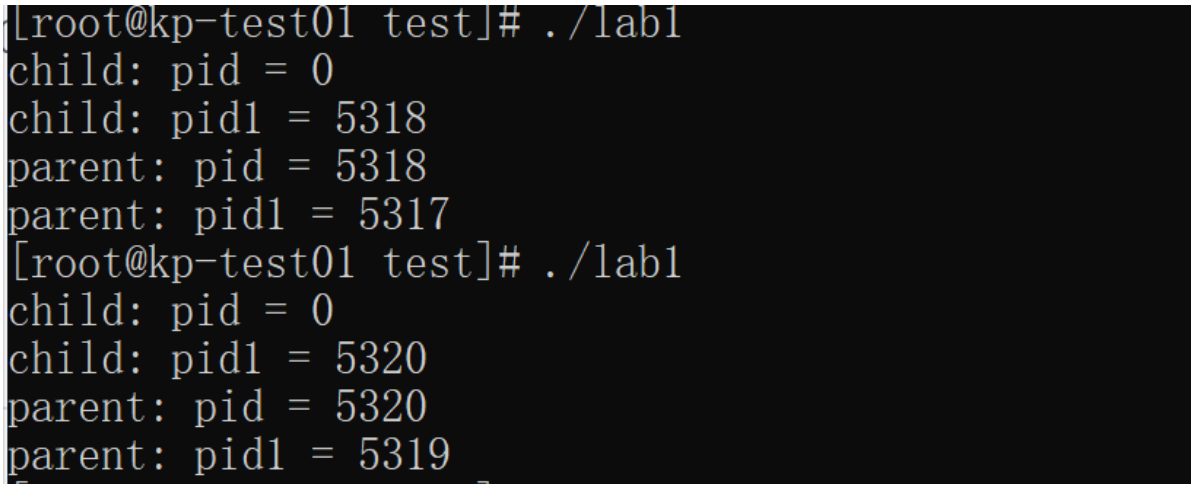
```
[root@kp-test01 test]# ./lab1
parent: pid = 2705
child: pid = 0
parent: pid1 = 2704
child: pid1 = 2705
[root@kp-test01 test]# ./lab1
parent: pid = 2707
child: pid = 0
parent: pid1 = 2706
child: pid1 = 2707
[root@kp-test01 test]# ./lab1
parent: pid = 2709
child: pid = 0
parent: pid1 = 2708
child: pid1 = 2709
```

理论分析：

wait函数的作用是在父进程中等待子进程的退出，回收子进程的资源，防止无父情况的发生（形成僵尸进程或孤儿进程）；而本题程序中的wait函数位于父进程的最后，所以父进程运行到wait时会阻塞，等待子进程先退出。所以有wait函数的情况下，我们可以保证子进程先退出，并且回收子进程的资源；没有wait函数的话，如果父进程先退出那子进程就变成孤儿进程，如果子进程先退出就会变成僵尸进程，因为子进程资源未被回收，会造成内存泄漏。因为wait函数在父进程最后调用，所以有无wait函数不会影响程序执行的顺序，都以CADB（源码中标注）的顺序执行。

如果wait函数在父进程中间或者前面出现，就会影响进程之间的执行顺序。

将wait函数移动至父进程的前面运行结果如下：先运行了子进程，然后运行父进程。



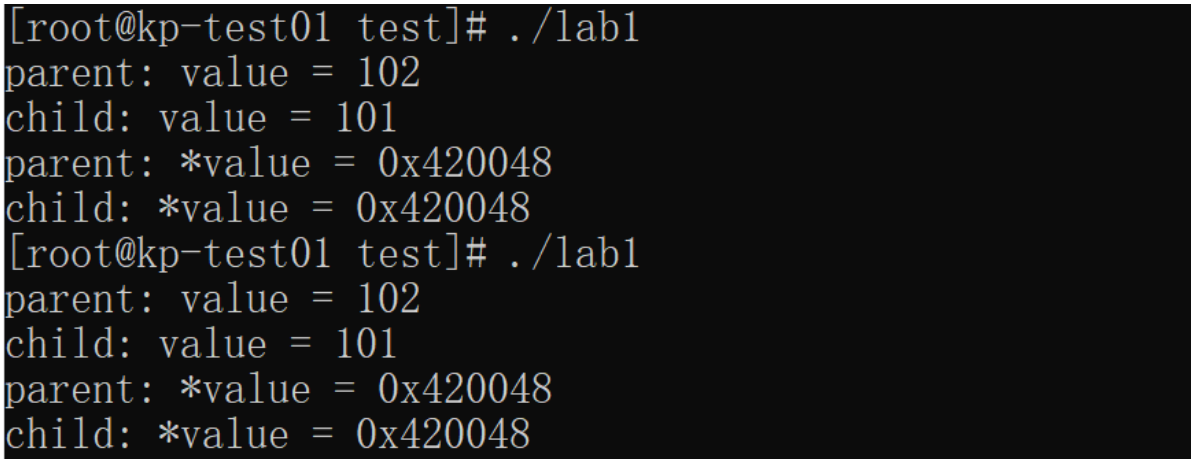
1.1(1)源码如下：

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

int main(){
    pid_t pid,pid1;
    pid=fork();

    if(pid<0){
        fprintf(stderr,"Fork failed");
        return 1;
    }
    else if(pid==0){
        pid1=getpid();
        printf("child: pid = %d\n",pid);   //A
        printf("child: pid1 = %d\n",pid1); //B
    }
    else{
        pid1=getpid();
//      wait(NULL);
        printf("parent: pid = %d\n",pid);   //C
        printf("parent: pid1 = %d\n",pid1); //D
//      wait(NULL);
    }
    return 0;
}
```

(2)

a. 加入一个全局变量value，在子进程中对其进行加1操作，在父进程中对其进行加2操作；

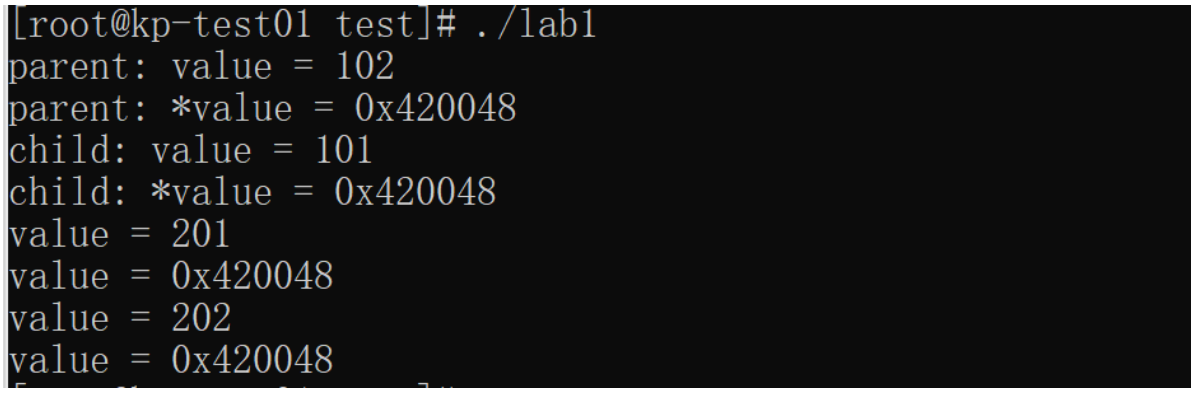
运行结果如下图：



可以看到父子进程操作的value值并不共享，value的值在父进程操作后是102，但是在子进程操作中还是只有101。虽然值不同，但是输出的value的地址两个进程相同，因为输出的逻辑地址，所以尽管两个进程中的value的值都不同但是输出的地址却相同。而value变量的物理地址在两个进程其实是不同的，所以它们的值在两个进程不相同。父子进程中对全局变量的修改不会对其他进程产生影响，因此不能通过这种方式实现数据共享。

b. 在return 0; 前对value加100的操作，并输出了value的值和其地址；

运行结果如下图：



可以看到在return 0 前的操作输出了两遍，这是因为父子进程都运行了新增的三行程序，所以分别输出了父子进程下执行value+100，输出value的值和地址的结果。显然和上一题的结果相对应，父进程中的value值为102，加100后输出了202，而子进程同理为201，输出的地址也仍相同。

1.1（2）ab源码：

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

int value =100;

int main(){
    pid_t pid;
    pid=fork();
```

```
    if(pid<0){
        fprintf(stderr,"Fork failed");
        return 1;
    }
    else if(pid==0){
        value++;
        printf("child: value = %d\n",value);
        printf("child: *value = %p\n",&value);
    }
    else{
        value+=2;
        printf("parent: value = %d\n",value);
        printf("parent: *value = %p\n",&value);
    }
    value+=100;
    printf("value = %d\n",value);
    printf("*value = %p\n",&value);
    return 0;
}
```

c.

problem:

```
    else if(pid==0){
            system("./system_call");
            printf("child process PID: %d\n",getpid());
            
1.  }
    else{
            printf("parent process PID: %d\n",getpid());

    }
```

调用system函数后未考虑周全，在子进程中调用 system() 函数后，子进程会创建一个新的 shell 进程来执行 system_call 可执行文件，而不会等待该命令执行完成。因此，在输出子进程的 PID 之前，system_call 可能还没有执行完毕。添加wait函数来解决此问题。

```
[root@kp-test01 test]# ./lab1
parent process PID: 2401
[root@kp-test01 test]# System call PID = 2403
child process PID: 2402
```

2. exec函数族的使用格式与规范，语法生疏，导致错误；

```
[root@kp-test01 test]# gcc -o lab1 lab1.c
lab1.c: In function ‘main’:
lab1.c:17:25: warning: passing argument 2 of ‘execv’ from incompatible pointer type [-Wincompatible-pointer-types]
   execv("./system_call","system_call",NULL);

In file included from lab1.c:3:0:
/usr/include/unistd.h:563:12: note: expected ‘char * const*’ but argument is of type ‘char *’
 extern int execv (const char *__path, char *const __argv[])

lab1.c:17:3: error: too many arguments to function ‘execv’
   execv("./system_call","system_call",NULL);

In file included from lab1.c:3:0:
/usr/include/unistd.h:563:12: note: declared here
 extern int execv (const char *__path, char *const __argv[])
```

exec函数族的使用：

```c
#include <unistd.h>

extern char **environ;

int execl(const char *path, const char *arg, .../* (char  *) NULL */);
int execlp(const char *file, const char *arg, .../* (char  *) NULL */);
int execle(const char *path, const char *arg, ...
                /*, (char *) NULL, char * const envp[] */);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],char *const envp[]);
```

最终调用execl函数实现；

解决此问题后，程序正确运行，结果如下：

system：

```
[root@kp-test01 test]# ./lab1
parent process PID: 2437
child process1 PID: 2438
System call PID = 2439
child process PID: 2438
```

exec：

```
[root@kp-test01 test]# gcc -o lab1 lab1.c
[root@kp-test01 test]# ./lab1
parent process PID: 2660
child process1 PID: 2661
System call PID = 2661
```

理论分析：

观察1：system函数和exec函数族的输出结果并不相同，可以看到调用system函数的程序输出有四行而调用exec函数族的输出只有三行。因为system函数和exec函数执行后的返回不同。

reason：system函数执行成功会回到原点继续执行，也就是会执行接下来的printf函数，所以调用system函数时会有四行输出。而exec函数族执行失败才会回到原点继续执行，执行成功则会直接退出当前进程，也就是并不会执行exec函数后的包括printf在内的一系列程序，因此只有三行输出。

观察2：system函数调用的system_call文件输出的pid和子进程的pid并不相同，而exec函数调用的system_call文件后输出的pid与子进程的pid相同。

reason：system函数和exec函数运行的原理并不相同；system函数的功能是执行命令并等待命令执行完成后返回。它会创建一个子进程来执行命令，而父进程会阻塞等待子进程的结束。因此system_call输出的其实是system命令新创建的子进程的pid值，是child pid的值加1；而exec函数的功能是用指定的命令替换当前进程的映像，并开始执行新的命令。它不会创建新的进程，而是直接替换当前进程的代码和数据，因此运行system_call时输出的pid就是child的pid值，与之相等。

1.1 （2）c源码

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

int main(){
    pid_t pid;
    pid=fork();

    if(pid<0){
        fprintf(stderr,"Fork failed");
        return 1;
    }
    else if(pid==0){
        printf("child process1 PID: %d\n",getpid());
        execl("/usr/local/src/test/system_call","system_call",NULL);
        //system("./system_call");            调用system函数的情况；
        printf("child process PID: %d\n",getpid());
        perror("execl");
        exit(1);
        //exit(0);
    }
    else{
        printf("parent process PID: %d\n",getpid());
        wait(NULL);
    }

    return 0;
}
```

Thread

(1)

problem:

1. 采用gcc -o lab1.2 lab1.2.c 命令编译文件发生错误：

```
[root@kp-test01 test]# gcc -o lab1.2 lab1.2.c
/usr/bin/ld: /tmp/ccKq9QQ4.o: in function `main':
lab1.2.c:(.text+0xe8): undefined reference to `pthread_create'
/usr/bin/ld: lab1.2.c:(.text+0x104): undefined reference to `pthread_create'
/usr/bin/ld: lab1.2.c:(.text+0x110): undefined reference to `pthread_join'
/usr/bin/ld: lab1.2.c:(.text+0x11c): undefined reference to `pthread_join'
collect2: error: ld returned 1 exit status
```

发现是因为pthread不是Linux下的默认库，所以即使加了<pthread.h> 头文件也无法链接到thread库中，在 gcc编译时，加入-lpthread参数即可成功编译；*gcc -o lab1.2 lab1.2.c -lpthread*

2. 当循环的次数设置为10000时，运行的结果一直都是0；

```
[root@kp-test01 test]# gcc -o lab1.2 lab1.2.c -lpthread
[root@kp-test01 test]# ./lab1.2
thread1 create successfully!
thread2 create successfully!
variable = 0
[root@kp-test01 test]# ./lab1.2
thread1 create successfully!
thread2 create successfully!
variable = 0
```

可能循环次数不多时，CPU调度使线程按某种顺序执行，并不会出现并发执行的情况。将循环次序设置成100000次出现理论的结果；

运行结果如下：

```
[root@kp-test01 test]# ./lab1.2
thread1 create successfully!
thread2 create successfully!
variable = -2759
[root@kp-test01 test]# ./lab1.2
thread1 create successfully!
thread2 create successfully!
variable = -2408
[root@kp-test01 test]# ./lab1.2
thread1 create successfully!
thread2 create successfully!
variable = 1594
```

理论分析：

两个线程分别对变量variable进行自增和自减操作，循环100000次。由于线程的执行顺序不确定，可能会出现以下情况：

- 线程1执行自增操作，然后线程2执行自减操作，最终结果为0。
- 线程2执行自减操作，然后线程1执行自增操作，最终结果为0。
- 线程1和线程2交替执行自增和自减操作，最终结果不确定。

由于没有使用互斥锁或原子操作来保护共享变量的访问，多线程对variable进行读写操作是不安全的，可能会出现竞态条件。因此，最终输出的结果是不确定的，可能是0，也可能是其他值。**所以多次输出的结果variable的值一直在变化。**

1.2（1）源码

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

```c
int variable = 0;

void* func1(void* arg) {
    printf("thread1 create successfully!\n");
    for(int i=0;i<10000;i++){
        variable++;
    }
    return NULL;
}

void* func2(void* arg) {
    printf("thread2 create successfully!\n");
    for(int j=0;j<10000;j++){
        variable--;
    }
    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    pthread_create(&thread1, NULL, func1, NULL);
    pthread_create(&thread2, NULL, func2, NULL);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    printf("variable = %d\n", variable);

    return 0;
}
```

(2)

运行结果如下:

```
[root@kp-test01 test]# gcc -o lab1.2 lab1.2.c -lpthread
[root@kp-test01 test]# ./lab1.2
thread1 create successfully!
thread2 create successfully!
variable = 0
[root@kp-test01 test]# ./lab1.2
thread1 create successfully!
thread2 create successfully!
variable = 0
[root@kp-test01 test]# ./lab1.2
thread1 create successfully!
thread2 create successfully!
variable = 0
```

理论分析:

通过sem_init函数初始化了一个初始值为1的信号量mutex。在每个线程的循环中，首先使用sem_wait函数进行P操作，申请资源。然后对共享变量进行操作。最后使用sem_post函数进行V操作，释放资源。

使用信号量的PV操作可以确保每个线程在访问共享变量时都能够互斥地进行操作，避免竞态条件的发生。保证了同一时刻只有一个线程才能对variable进行操作，确保操作的正确性。实现了对两个线程的互斥访问。

1.2（2）源码

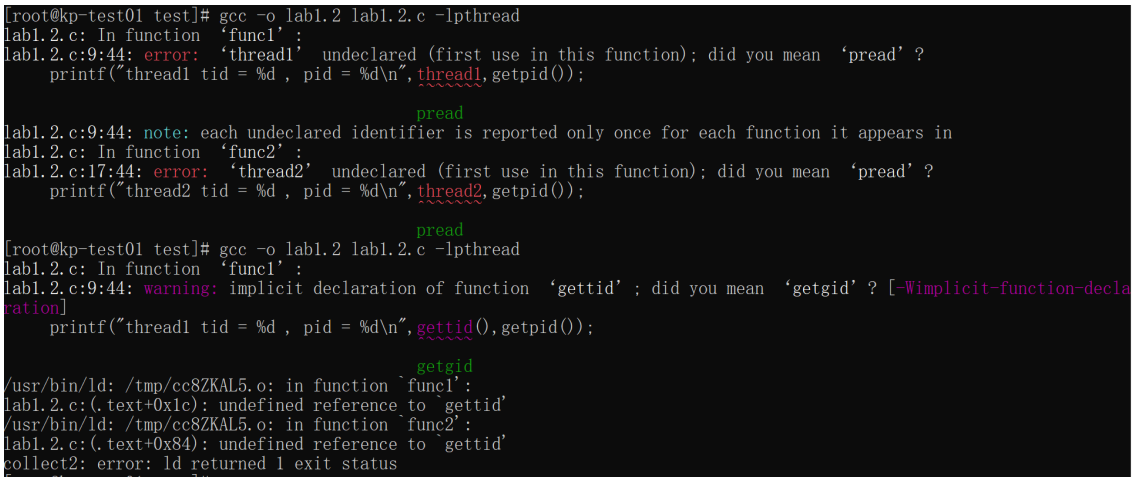
```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

int variable = 0;
sem_t mutex;

void* func1(void* arg) {
    printf("thread1 create successfully!\n");
    for(int i=0;i<10000;i++){
        sem_wait(&mutex);
        variable++;
        sem_post(&mutex);
    }
    return NULL;
}

void* func2(void* arg) {
    printf("thread2 create successfully!\n");
    for(int j=0;j<10000;j++){
        sem_wait(&mutex);
        variable--;
        sem_post(&mutex);
    }
    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    sem_init(&mutex,0,1);

    pthread_create(&thread1, NULL, func1, NULL);
    pthread_create(&thread2, NULL, func2, NULL);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    printf("variable = %d\n", variable);

    sem_destroy(&mutex);
```

```
    return 0;
}
```

(3)

problem:

1. 输出线程的id时，并不能直接通过返回pthread_t类型或者gettid（）来得到tid的值；



修改获取线程id的方式，使用`syscall(SYS_gettid)`的方式获取当前线程的id；

2. 使用exec函数的时候出现多线程未并发的情况；

线程2还未创建exec函数就执行完成了，退出进程了。采用sleep函数来模拟实现并发程序；

运行结果如下：

system:

```
[root@kp-test01 test]# ./lab1.2
thread1 create successfully!
thread1 tid = 3709 , pid = 3708
thread2 create successfully!
thread2 tid = 3710 , pid = 3708
System call PID = 3711
System call PID = 3712
thread2 systemcall return
thread1 systemcall return
[root@kp-test01 test]# ./lab1.2
thread1 create successfully!
thread1 tid = 3714 , pid = 3713
thread2 create successfully!
thread2 tid = 3715 , pid = 3713
System call PID = 3717
System call PID = 3716
thread1 systemcall return
thread2 systemcall return
[root@kp-test01 test]# ./lab1.2
thread1 create successfully!
thread1 tid = 3719 , pid = 3718
thread2 create successfully!
thread2 tid = 3720 , pid = 3718
System call PID = 3722
System call PID = 3721
thread1 systemcall return
thread2 systemcall return
```

exec:

```
[root@kp-test01 test]# ./lab1.2
thread1 create successfully!
thread1 tid = 3789 , pid = 3788
System call PID = 3788
[root@kp-test01 test]# gcc -o lab1.2 lab1.2.c -lpthread
[root@kp-test01 test]# ./lab1.2
thread1 create successfully!
thread1 tid = 3819 , pid = 3818
thread2 create successfully!
thread2 tid = 3820 , pid = 3818
System call PID = 3818
[root@kp-test01 test]# ./lab1.2
thread1 create successfully!
thread1 tid = 3830 , pid = 3829
thread2 create successfully!
thread2 tid = 3831 , pid = 3829
System call PID = 3829
```

理论分析：

观察1：exec函数执行后未输出system call return，而system函数执行后有此输出；

reason：此原因类似上一题，因为exec成功执行后直接退出进程，而输出系统调用成功返回的输出语句在exec函数之下，所以执行成功的话是不会输出的。而system函数执行成功后仍会继续执行下面的程序。

观察2：不论system还是exec调用，两个线程输出的pid都相同，而system调用时system_call文件输出的pid有变化，exec函数调用system_call文件输出的pid与线程中输出的相同。

reason：system它会创建一个子进程来执行命令。因此system_call输出的其实是system命令新创建的子进程的pid值，是外部进程pid的值加1；而exec函数的功能是用指定的命令替换当前进程的映像，并开始执行新的命令。它不会创建新的进程，而是直接替换当前进程的代码和数据，因此运行system_call时输出的pid是不变的。

观察3：线程的tid是所属进程的pid向上增加。

观察4：调用exec时两个线程只有一个成功返回了结果；

reason：执行到exec语句时整个进程就被替换成exec内执行的内容了，所以另一个线程中的exec根本没有机会被执行。

观察5：不加sleep（）函数，不会发生线程的并发执行，线程2甚至不能被创建。

reason：语句太少，CPU执行效率太高，直接执行到线程1的exec函数还未开始执行线程2，所以导致看似非并发的情况发生。

1.2（3）源码

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

void* func1(void* arg) {
    printf("thread1 create successfully!\n");
    printf("thread1 tid = %d , pid = %d\n",syscall(SYS_gettid),getpid());
```

```
    system("./system_call");
    printf("thread1 systemcall return\n");
    return NULL;
}

void* func2(void* arg) {
    printf("thread2 create successfully!\n");
    printf("thread2 tid = %d , pid = %d\n",syscall(SYS_gettid),getpid());
    system("./system_call");
    printf("thread2 systemcall return\n");
    return NULL;
}
/*exec函数执行源码:
void* func1(void* arg) {
    printf("thread1 create successfully!\n");
    printf("thread1 tid = %d , pid = %d\n",syscall(SYS_gettid),getpid());
    sleep(1);
    execl("/usr/local/src/test/system_call","system_call",NULL);
    printf("thread1 systemcall return\n");
    perror("execl");
    exit(1);
    return NULL;
}

void* func2(void* arg) {
    printf("thread2 create successfully!\n");
    printf("thread2 tid = %d , pid = %d\n",syscall(SYS_gettid),getpid());
    execl("/usr/local/src/test/system_call","system_call",NULL);
    printf("thread2 systemcall return\n");
    perror("execl");
    exit(1);
    return NULL;
}
*/
int main() {
    pthread_t thread1, thread2;

    pthread_create(&thread1, NULL, func1, NULL);
    pthread_create(&thread2, NULL, func2, NULL);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    return 0;
}
```

Spin Lock

运行结果:

```
[root@kp-test01 test]# gcc -o spinlock spinlock.c -lpthread
[root@kp-test01 test]# ./spinlock
Share value: 0
thread1 create success!
thread2 create success!
Share value: 10000
[root@kp-test01 test]# ./spinlock
Share value: 0
thread1 create success!
thread2 create success!
Share value: 10000
[root@kp-test01 test]# ./spinlock
Share value: 0
thread1 create success!
thread2 create success!
Share value: 10000
```

理论分析：自旋锁的实现使用了一个简单的标志位flag来表示锁的状态。当flag为0时表示锁是空闲的，可以获取；当flag为1时表示锁被占用，需要等待。

自旋锁的初始化函数spinlock_init()用来初始化自旋锁的标志位flag，将其置为0，表示锁是空闲的。

自旋锁的获取函数spinlock_lock()使用了一个while循环来实现自旋等待的效果。在获取锁的过程中，通过__sync_lock_test_and_set()函数来将flag从0设置为1，表示锁被占用。如果获取锁成功，则退出循环；如果获取锁失败，则继续自旋等待。

自旋锁的释放函数spinlock_unlock()使用了__sync_lock_release()函数将flag设置为0，表示锁被释放。

在线程函数thread_function()中，先调用spinlock_lock()获取锁，然后对共享变量shared_value进行自增操作，最后调用spinlock_unlock()释放锁。这样可以保证每个线程在修改共享变量时都是互斥的，避免了竞争条件。

在主函数中，先输出共享变量的初始值0，然后创建两个线程，并等待线程结束。最后再次输出共享变量的值，可以看到共享变量的最终值为10000。

1.3源码

```c
#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;
```

```c
// 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
    printf("Share value: %d\n",shared_value);
 // 初始化自旋锁
    spinlock_init(&lock);
 // 创建两个线程
    int res1,res2;
    res1=pthread_create(&thread1,NULL,thread_function,&lock);
    if(res1==0) {printf("thread1 create success!\n");}
    res2=pthread_create(&thread2,NULL,thread_function,&lock);
    if(res2==0) {printf("thread2 create success!\n");}
// 等待线程结束
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
 // 输出共享变量的值
    printf("Share value: %d\n",shared_value);
    return 0;
}
```