操作系统实验报告

实验一 进程、线程相关编程实验

1.1 进程相关编程实验

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

```
[root@kp-test01 lab1]# gcc 1.c -o 1-1
[root@kp-test01 lab1]# ./1-1
parent: pid = 6350
child: pid = 0
parent: pid_new = 6349
child: pid_new = 6350
[root@kp-test01 lab1]# ./1-1
parent: pid = 6352
child: pid = 0
parent: pid_new = 6351
child: pid_new = 6352
[root@kp-test01 lab1]# ./1-1
parent: pid = 6354
child: pid = 0
parent: pid_new = 6353
child: pid_new = 6354
[root@kp-test01 lab1]#
```

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@kp-test01 lab1]# ./1-1
parent: pid = 6362
child: pid = 0
parent: pid_new = 6361
child: pid_new = 6362
[root@kp-test01 lab1]# ./1-1
child: pid = 0
parent: pid = 6364
child: pid_new = 6364
parent: pid_new = 6363
[root@kp-test01 lab1]# ./1-1
parent: pid = 6366
child: pid = 0
parent: pid_new = 6365
child: pid_new = 6366
[root@kp-test01 lab1]# ./1-1
parent: pid = 6368
child: pid = 0
parent: pid_new = 6367
child: pid_new = 6368
[root@kp-test01 lab1]#
```

可以发现当前情况下存在child有可能先于parent进程运行。

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，观察并解释所做操作和输出结果。

```
[root@kp-test01 lab1]# ./1-1
parent: pid = 8835
child: pid = 0
parent: pid_new = 8834
child: pid_new = 8835
parent: flag = 2000
child: flag = 1000
[root@kp-test01 lab1]# ./1-1
parent: pid = 8837
child: pid = 0
parent: pid_new = 8836
child: pid_new = 8837
parent: flag = 2000
child: flag = 1000
[root@kp-test01 lab1]# ./1-1
parent: pid = 8839
parent: pid_new = 8838
child: pid = 0
parent: flag = 2000
child: pid_new = 8839
child: flag = 1000
[root@kp-test01 lab1]#
```

我定义了一个全局变量flag，在parent进程中修改其为2000，child进程中修改为1000并分别打印结果。

父进程与子进程先后交替运行，合理的。

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@kp-test01 lab1]# ./1-1
parent: pid = 6378
child: pid = 0
parent: pid_new = 6377
child: pid_new = 6378
parent: flag = 2000
child: flag = 1000
pid = 0 program before end: flag = 1000
pid = 6378 program before end: flag = 2000
[root@kp-test01 lab1]# ./1-1
parent: pid = 6380
child: pid = 0
parent: pid_new = 6379
child: pid_new = 6380
parent: flag = 2000
child: flag = 1000
pid = 0 program before end: flag = 1000
pid = 6380 program before end: flag = 2000
[root@kp-test01 lab1]#
```

清晰明了。

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@kp-test01 lab1]# ./1-1
parent: pid = 8854
Using system to call system_call:
parent: pid_new = 8853
parent: flag = 2000
This is system_call. My PID is: 8855
Using exec to call system_call:
This is system_call. My PID is: 8854
pid = 8854 program before end: flag = 2000
[root@kp-test01 lab1]#
```

使用 `system()` 调用

1. **父进程PID：** 父进程的 PID（Process ID）是 8853。这是该进程的唯一标识符。
2. **子进程PID：** 子进程通过 `system()` 调用 `system_call` 执行文件时，生成了一个新的进程，其 PID 是 8855。
3. **函数执行顺序：** 父进程的代码先执行了，之后子进程通过 `system()` 执行了 `system_call`。

使用 `exec()` 调用

1. **子进程替换：** `exec()` 函数替换了子进程（PID 8854）的内容。因此，这个PID与父进程中显示的子进程PID一致。
2. **程序流：** 由于 `exec()` 替换了子进程的内容，`exec()` 之后的任何代码都不会被执行。

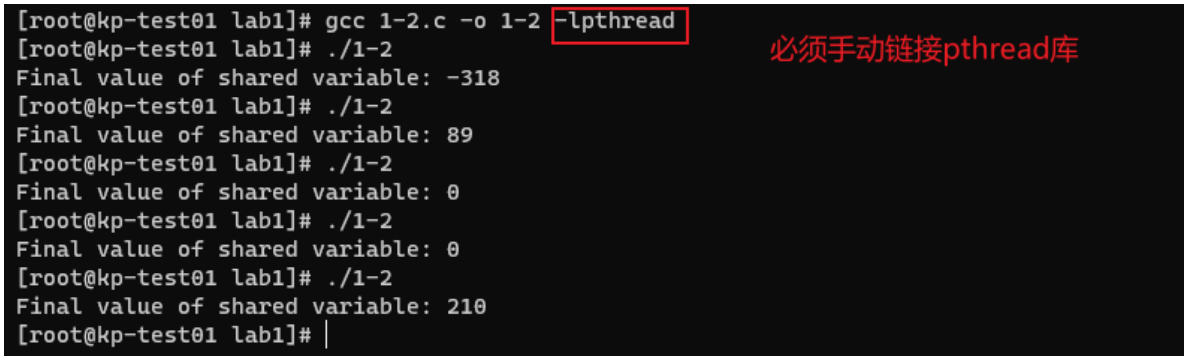
总结

1. **进程独立性：** 使用 `system()` 创建了一个全新的进程（PID 8855）来执行 `system_call`，而父进程（PID 8853）和子进程（PID 8854）都继续执行了剩下的代码。
2. **进程替换：** 使用 `exec()` 替换了子进程的内容，所以新的 `system_call` 运行在原子进程（PID 8854）的上下文中，而没有创建新的进程。
3. **控制流：** 两种方法都在子进程中成功调用了 `system_call`，但 `system()` 允许子进程继续执行其他代码，而 `exec()` 则完全替换了子进程，使得 `exec()` 之后的代码不会被执行。

1.2 线程相关编程实验

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。



可以发现，此时输出的结果并不fixed。

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int shared_variable = 0;

void *increment() {
    for (int i = 0; i < 5000; i++) {
        shared_variable++;
    }
    return NULL;
}

void *decrement() {
    for (int i = 0; i < 5000; i++) {
        shared_variable--;
    }
    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    if (pthread_create(&thread1, NULL,
increment, NULL) != 0) {
        perror("Failed to create thread1");
        exit(1);
    }
```

```
    if (pthread_create(&thread2, NULL,
decrement, NULL) != 0) {
        perror("Failed to create thread2");
        exit(1);
    }

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

    printf("Final value of shared variable:
%d\n", shared_variable);

    return 0;
}
```

结果不一致是因为程序没有进行线程同步。当多个线程访问和修改同一个共享变量（`shared_variable`）而没有进行任何形式的同步时，程序的行为就变得不可预测。

一个线程正在尝试增加变量的值，而另一个线程正在尝试减少它。这两个操作可能几乎同时发生，因为操作系统的调度器可以随时中断一个线程并转而执行另一个线程。这样的话，两个线程可能会“看到”共享变量几乎同时的不同值，或者同时对它进行修改，从而导致不一致的结果。

考虑一个简单的情境：

1. `shared_variable` 当前值为 0。

2. 线程1 读取其值（0），准备加1。
3. 线程2 被调度，读取其值（还是0），准备减1。
4. 线程1 继续执行，将值加1，结果为1。
5. 线程2 继续执行，减1（认为之前的值是0），结果为-1。

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。

同步（Synchronization）在多线程编程中指的是协调多个线程的执行，以确保它们能够正常、可预测地访问共享资源或完成某些任务。互斥（Mutual Exclusion）是同步的一种特殊形式，确保一次只有一个线程能访问某个特定的资源或代码段。互斥通常通过互斥锁（Mutex）来实现，但它也可以通过其他机制来实现，比如信号量（Semaphore）。信号量和互斥锁类似，但更为通用。信号量可以用来解决除了互斥之外的其他同步问题。

PV操作是信号量（Semaphores）操作的传统术语，源自荷兰语的Proberen（尝试）和Verhogen（增加）。在信号量的上下文中，P操作通常用于申请或等待资源，而V操作用于释放或发出信号。

P操作（也叫`wait`或`down`或`sem_wait`）

- 当一个线程执行P操作时，它会检查信号量的值。

- 如果信号量的值大于0，那么它将减少信号量的值（通常是减1）并继续执行。
- 如果信号量的值为0，线程将被阻塞，直到信号量的值变为大于0。

V操作（也叫`signal`或`up`或`sem_post`）

- V操作增加信号量的值（通常是加1）。
- 如果有线程因执行P操作而阻塞在这个信号量上，一个或多个线程将被解除阻塞，并被允许减少信号量的值。

在C语言中，使用POSIX信号量

- `sem_wait(&semaphore);`：执行P操作。
- `sem_post(&semaphore);`：执行V操作。

其中，`semaphore`是一个`sem_t`类型的变量，代表信号量。

`sem_init`和`sem_destroy`是POSIX信号量（Semaphores）的初始化和销毁函数，它们用于设置和清理信号量。

`sem_init`

这个函数用于初始化一个未命名的信号量。

```
int sem_init(sem_t *sem, int pshared,
unsigned int value);
```

- **sem**: 一个指向信号量对象的指针。

- **pshared**: 如果这个参数是0，信号量就是当前进程的局部信号量。如果这个参数非0，则该信号量在多个进程间共享。
- **value**: 信号量的初始值。

这个函数成功时返回0，失败时返回-1。

`sem_destroy`

这个函数用于销毁一个未命名的信号量，释放其占用的资源。

```
int sem_destroy(sem_t *sem);
```

- **sem**: 一个指向信号量对象的指针。

这个函数成功时返回0，失败时返回-1。在使用`sem_destroy`之前，确保没有线程被阻塞在该信号量上。

整体的代码为：

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

int shared_variable = 0;

sem_t semaphore;
```

```c
void *increment() {
    for (int i = 0; i < 5000; i++) {
        sem_wait(&semaphore);
        shared_variable++;
        sem_post(&semaphore);
    }
    return NULL;
}

void *decrement() {
    for (int i = 0; i < 5000; i++) {
        sem_wait(&semaphore);
        shared_variable--;
        sem_post(&semaphore);
    }
    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    if (sem_init(&semaphore, 0, 1) == -1) {
        perror("Failed to initialize
semaphore");
        exit(1);
    }

    if (pthread_create(&thread1, NULL,
increment, NULL) != 0) {
        perror("Failed to create thread1");
```

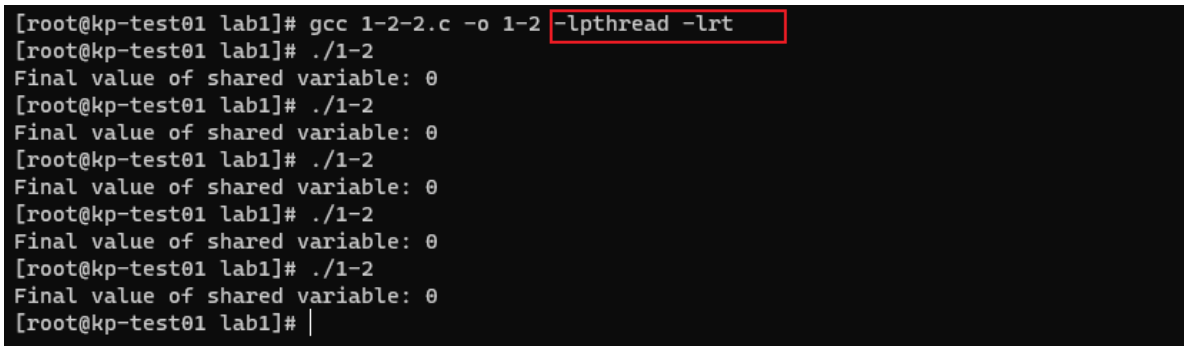
```
        exit(1);
    }

    if (pthread_create(&thread2, NULL,
decrement, NULL) != 0) {
        perror("Failed to create thread2");
        exit(1);
    }

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

    sem_destroy(&semaphore);
    printf("Final value of shared variable:
%d\n", shared_variable);

    return 0;
}
```



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。

```
#include <pthread.h>
#include <stdio.h>
```

```c
#include <unistd.h>
#include <stdlib.h>
#include <syscall.h>

void *system_thread_function() {
    printf("System Thread TID: %ld\n",
syscall(SYS_gettid));
    system("./system_call");
    return NULL;
}

void *exec_thread_function() {
    printf("Exec Thread TID: %ld\n",
syscall(SYS_gettid));
    char *args[] = {"ls", "/usr/src",
NULL};
    execvp(args[0], args);
    return NULL;
}

int main() {
    pthread_t system_thread, exec_thread;

    printf("Main thread PID: %d\n",
getpid());

    if (pthread_create(&system_thread,
NULL, system_thread_function, NULL)) {
        fprintf(stderr, "Error creating
system thread\n");
```

```
        return 1;
    }

    if (pthread_create(&exec_thread, NULL,
exec_thread_function, NULL)) {
        fprintf(stderr, "Error creating
exec thread\n");
        return 1;
    }

    pthread_join(system_thread, NULL);
    pthread_join(exec_thread, NULL); // 注
意，如果exec_thread成功运行了，这里实际上不会被执
行

    return 0;
}
```

```
[root@kp-test01 lab1]# This is system_call. My PID is: 10855
gcc 1-2-3.c -o 1-2 -lpt^C
[root@kp-test01 lab1]# ./1-2
Main thread PID: 10856
System Thread TID: 10857
Exec Thread TID: 10858
debug  kernels  lab1
[root@kp-test01 lab1]# This is system_call. My PID is: 10859
^C
[root@kp-test01 lab1]#
```

我们会发现程序不会自动终止，因为exec运行后使得该线程无法返回NULL终止。解决方案是在exec之前先fork一个进程，在子进程中调用exec

```
void *system_thread_function() {

```

```
    printf("System thread pid: %d\n",
getpid());
    printf("System Thread TID: %ld\n",
syscall(SYS_gettid));
    system("./system_call");
    return NULL;
}

void *exec_thread_function() {
    printf("exec thread pid: %d\n",
getpid());
    printf("Exec Thread TID: %ld\n",
syscall(SYS_gettid));
    pid_t pid = fork();
    if (pid == 0) {
        char *args[] = {"ls", "/usr/src",
NULL};
        execvp(args[0], args);
        exit(0);
    } else if (pid > 0) {
        int status;
        waitpid(pid, &status, 0);
    } else {
        perror("fork");
    }
    return NULL;
}
```

```
[root@kp-test01 lab1]# gcc 1-2-3.c -o 1-2 -lpthread -lrt
[root@kp-test01 lab1]# ./1-2
Main thread PID: 11020
System thread pid: 11020
System Thread TID: 11021
exec thread pid: 11020
Exec Thread TID: 11022
debug  kernels  lab1
This is system_call. My PID is: 11023
[root@kp-test01 lab1]# ls /usr/src
debug  kernels  lab1
[root@kp-test01 lab1]# ./1-2
Main thread PID: 11026
System thread pid: 11026
System Thread TID: 11027
exec thread pid: 11026
Exec Thread TID: 11028
debug  kernels  lab1
This is system_call. My PID is: 11029
[root@kp-test01 lab1]#
```

可以看到线程共享进程的PID，但会有自身独立的线程TID，同时system在实际过程中会调用fork但是这个行为被封装起来了

1.3 自旋锁实验

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;
}
```

完整代码为：

```
/***spinlock.c*in xjtu*2023.8
 */
#include <pthread.h> // 定义自旋锁结构体
#include <stdio.h>
#include <stdlib.h>

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
  spinlock_init(&lock);

  if (pthread_create(&thread1, NULL,
thread_function, &lock) != 0) {
    perror("Failed to create thread1");
    exit(1);
  }
  if (pthread_create(&thread2, NULL,
thread_function, &lock) != 0) {
    perror("Failed to create thread2");
    exit(1);
  }

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

  printf("shared value is %d\n",
shared_value);

  return 0;
}
```

补充完成代码后，编译并运行程序，分析运行结果

```
[root@kp-test01 lab1]# gcc 1-3-1.c -o 1-3 -lpthread
[root@kp-test01 lab1]# ./1-3
shared value is 10000
[root@kp-test01 lab1]# ./1-3
shared value is 10000
[root@kp-test01 lab1]# ./1-3
shared value is 10000
[root@kp-test01 lab1]#
```

自旋锁设定成功，每个线程都正确地对shared_value进行了修改