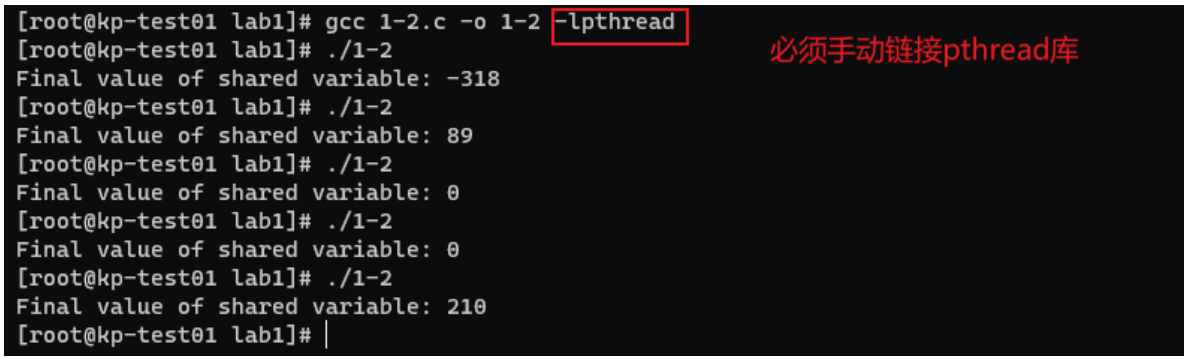
操作系统实验报告

实验一 进程、线程相关编程经验

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。

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

```c
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;

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;
}
```

```
[root@kp-test01 lab1]# gcc 1-2-2.c -o 1-2 -lpthread -lrt
[root@kp-test01 lab1]# ./1-2
Final value of shared variable: 0
[root@kp-test01 lab1]# ./1-2
Final value of shared variable: 0
[root@kp-test01 lab1]# ./1-2
Final value of shared variable: 0
[root@kp-test01 lab1]# ./1-2
Final value of shared variable: 0
[root@kp-test01 lab1]# ./1-2
Final value of shared variable: 0
[root@kp-test01 lab1]#
```

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>
#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