# 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

## 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 实验思想

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。

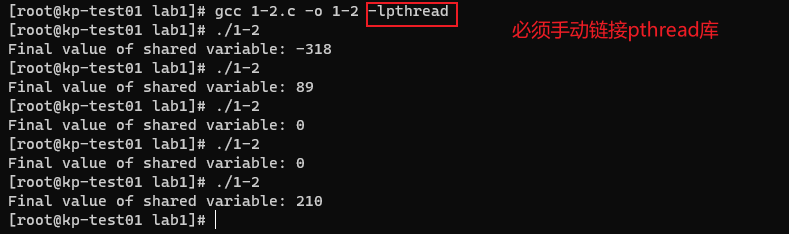
（3）互斥与同步： 为了解决竞态条件带来的问题，可以使用互斥锁（Mutex）来保护共享变量的访问。在每个线程对变量进行操作之前，先获取互斥锁，操作完成后再释放锁。这样一来，每次只有一个线程能够访问变量，从而避免了并发访问带来的不稳定性。

（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。



可以发现，此时输出的结果并不fixed。

#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。

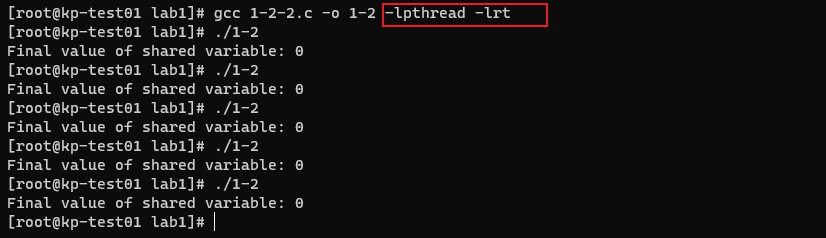
int sem\_destroy(sem\_t \*sem);

* **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。

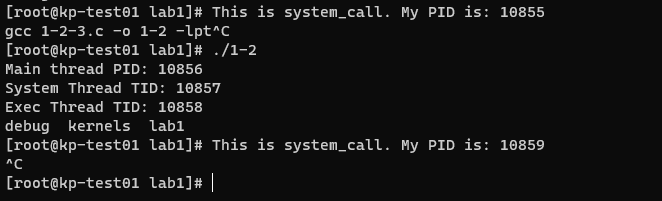
整体的代码为：

#include <stdio.h>  
#include <stdlib.h>  
#include <pthread.h>  
#include <semaphore.h>  
  
int shared\_variable = 0;  
  
sem\_t semaphore;  
  
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;  
}



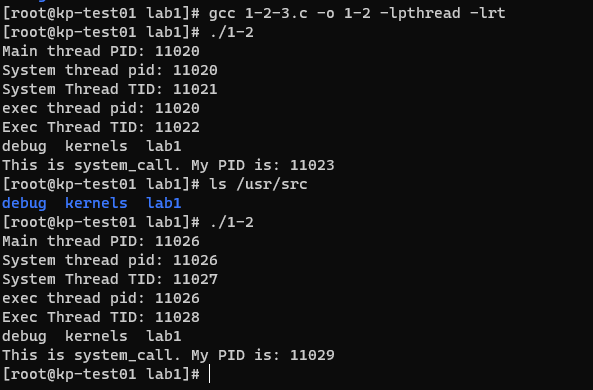
步骤三： 在第一部分实验了解了 system()与 exec 族函数的基础上，将这两个函数的调用改为在线程中实现，输出进程 PID 和线程的 TID 进行分析。

#include <pthread.h>  
#include <stdio.h>  
#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;  
}



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;   
}



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

## 1.2.5 测试数据设计

无需数据测试。

## 1.2.6 程序运行初值及运行结果分析

运行结果已经分析。

## 1.2.7 实验总结

### 1.1.7.1 实验中的问题与解决过程

在本次实验中，我遇到了多个问题，这些问题涵盖了多线程编程、进程与线程的ID差异、以及程序中的同步和互斥等方面。

线程创建和编译问题

首先，我试图在Linux环境中使用C语言创建两个线程，并对一个共享变量进行加一和减一操作。编译的过程中遇到了**undefined reference to 'pthread\_create'**等错误。这个问题是因为在链接阶段没有加上**-lpthread**标志，这个标志是必需的，因为它告诉编译器需要链接到Pthreads库。

输出不一致性问题

接下来，我注意到多次运行同一个程序会产生不一致的输出。这是因为多个线程在访问和修改共享变量时没有进行同步，导致了竞态条件。解决这个问题的一个常见方法是使用互斥锁或信号量进行同步。

信号量与同步机制

如何使用信号量（semaphore）进行同步以及P（Proberen，测试）和V（Verhogen，增加）操作的具体用法。信号量是用于控制多个线程对共享资源访问的一个同步原语。通过PV操作，我们可以确保在任何时刻只有一个线程能够访问共享变量，从而避免竞态条件。

System与Exec调用

我分析了如何在多线程环境中使用**system()**与**exec**族函数，并观察其对进程ID和线程ID的影响。注意到使用**system()**调用的进程ID与主线程不同，这是因为**system()**内部会使用**fork()**创建一个新的子进程来执行命令。

程序终止问题

我遇到了一个问题，即程序无法自动终止。这个问题出现在使用**execvp()**执行**ls**命令的部分。解决方法是在**fork()**创建的子进程中调用**execvp()**，而父进程则通过**waitpid()**来等待子进程的完成。

综上所述，这些问题和解决方案都是多线程和多进程编程中常见的问题。每一步都需要仔细地考虑同步机制、系统调用和进程管理，以确保程序的正确性。我对这些细节有了深入的理解，这对我未来在并发编程方面将是非常有价值的经验。

### 1.1.7.2 实验收获

本次实验让我对多线程和多进程编程有了更深入的了解，特别是在Linux环境下使用C语言。掌握了如何创建和管理线程，以及如何使用信号量进行同步，学习了**system()**和**exec**族函数在多线程环境下的行为，这对于理解进程和线程的区别非常有帮助。

我研究了多线程对共享资源访问的问题，尤其是竞态条件和其对程序输出的影响，成功地应用了互斥锁和信号量来解决这些问题，增强了程序的健壮性。

除了技术细节，实验还锻炼了我的问题解决能力。遇到编译错误、不一致的输出和程序终止问题时，都能够分析问题的根本原因，并找到相应的解决方案。这些都是软件开发中非常重要的技能。

### 1.1.7.3 意见与建议

1. **更多的实例操作**：尽管本次实验已经涵盖了多个重要的概念，但实践是检验真理的唯一标准。在未来的实验中，更多的实例操作可能会更有助于理解这些复杂的概念。
2. **代码审查与反思**：完成代码和实验后，进行代码审查和反思总结也是非常有价值的。这不仅能帮助你找出可能的错误或不足，还能促使思考如何优化和改进。
3. **深入理解同步机制**：本实验主要集中在使用信号量进行同步，但还有其他同步机制，如条件变量、读写锁等，也值得进一步研究。
4. **探索更多系统调用**：除了**system()**和**exec**族函数，还有很多其他有用的系统调用，如**pipe()**、**dup()**等，这些都是进程和线程间通信的重要手段。

## 1.2.8 附件

### 1.2.8.1 附件1 程序

[lab1\1-2-1.c](lab1/1-2-1.c)

[lab1\1-2-2.c](lab1/1-2-2.c)

[lab1\1-2-3.c](lab1/1-2-3.c)

### 1.2.8.2 附件2 Readme

<lab1-2Readme.md>