目录

[操作系统实验报告 3](#_Toc148532304)

[实验一 进程、线程相关编程经验 3](#_Toc148532305)

[1.1 进程相关编程实验 3](#_Toc148532306)

[1.1.1 实验目的 3](#_Toc148532307)

[1.1.2 实验内容 3](#_Toc148532308)

[1.1.3 实验思想 5](#_Toc148532309)

[1.1.4 实验步骤 5](#_Toc148532310)

[1.1.5 测试数据设计 8](#_Toc148532311)

[1.1.6 程序运行初值及运行结果分析 8](#_Toc148532312)

[1.1.7 实验总结 8](#_Toc148532313)

[1.1.7.1 实验中的问题与解决过程 8](#_Toc148532314)

[1.1.7.2 实验收获 9](#_Toc148532315)

[1.1.7.3 意见与建议 9](#_Toc148532316)

[1.1.8 附件 10](#_Toc148532317)

[1.1.8.1 附件1 程序 10](#_Toc148532318)

[1.1.8.2 附件2 Readme 10](#_Toc148532319)

[1.2 线程相关编程实验 10](#_Toc148532320)

[1.2.1 实验目的 10](#_Toc148532321)

[1.2.2 实验内容 10](#_Toc148532322)

[1.2.3 实验思想 10](#_Toc148532323)

[1.2.4 实验步骤 11](#_Toc148532324)

[1.2.5 测试数据设计 18](#_Toc148532325)

[1.2.6 程序运行初值及运行结果分析 18](#_Toc148532326)

[1.2.7 实验总结 18](#_Toc148532327)

[1.2.7.1 实验中的问题与解决过程 18](#_Toc148532328)

[1.2.7.2 实验收获 19](#_Toc148532329)

[1.2.7.3 意见与建议 19](#_Toc148532330)

[1.2.8 附件 19](#_Toc148532331)

[1.2.8.1 附件1 程序 19](#_Toc148532332)

[1.2.8.2 附件2 Readme 20](#_Toc148532333)

[1.3 自旋锁实验 20](#_Toc148532334)

[1.3.1 实验目的 20](#_Toc148532335)

[1.3.2 实验内容 20](#_Toc148532336)

[1.3.3 实验思想 20](#_Toc148532337)

[1.3.4 实验步骤 21](#_Toc148532338)

[1.3.5 测试数据设计 23](#_Toc148532339)

[1.3.6 程序运行初值及运行结果分析 23](#_Toc148532340)

[1.3.7 实验总结 23](#_Toc148532341)

[1.3.7.1 实验中的问题与解决过程 23](#_Toc148532342)

[1.3.7.2 实验收获 23](#_Toc148532343)

[1.3.7.3 意见与建议 23](#_Toc148532344)

[1.3.8 附件 24](#_Toc148532345)

[1.3.8.1 附件1 程序 24](#_Toc148532346)

[1.3.8.2 附件2 Readme 24](#_Toc148532347)

# 操作系统实验报告

# 实验一 进程、线程相关编程经验

# 1.1 进程相关编程实验

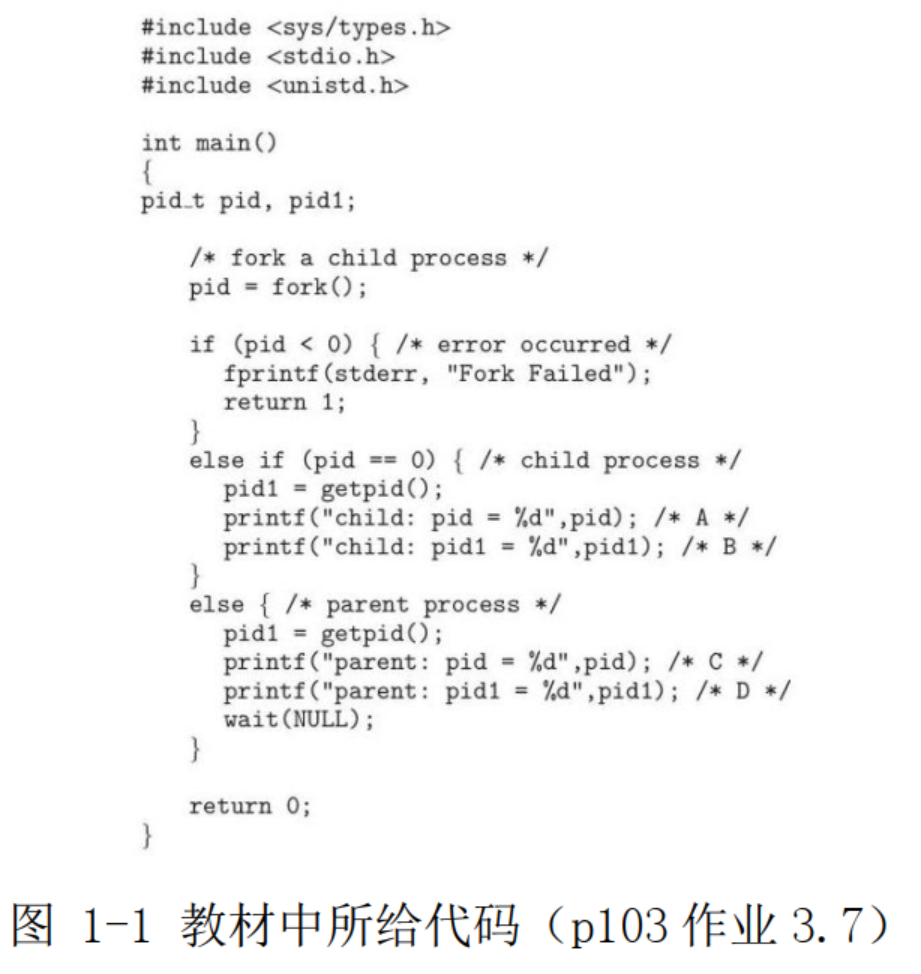
## 1.1.1 实验目的

（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。



（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 实验思想

（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；

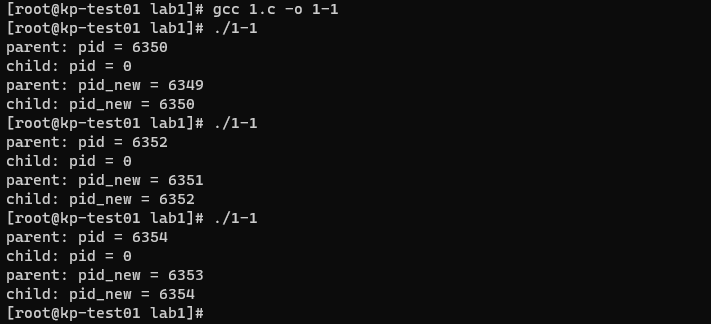
（2） PID： PID 是进程标识符（Process Identifier）的缩写，它是用来唯一标识一个操作系统中的进程的数值。每个正在运行或已经终止的进程都会被分配一个唯一的 PID，这个标识符可以用来在操作系统内部识别和管理进程；

（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。

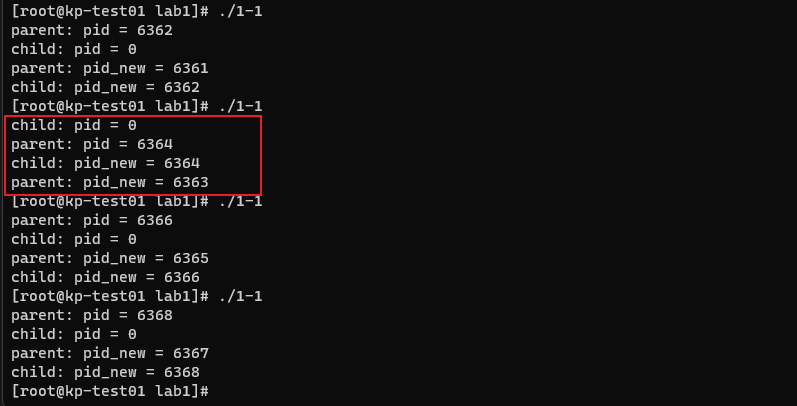
## 1.1.4 实验步骤

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

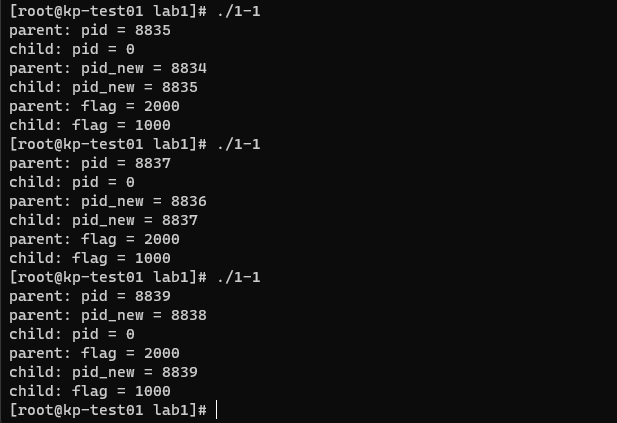


步骤二： 删去图 1-1 代码中的 wait()函数并多次运行程序，分析运行结果。



可以发现当前情况下存在child有可能先于parent进程运行。

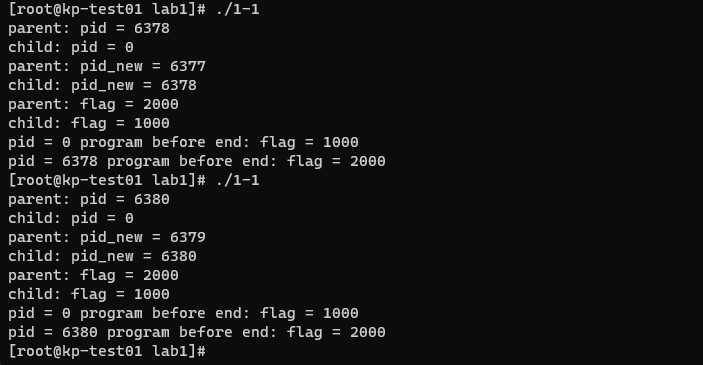
步骤三： 修改图 1-1 中代码，增加一个全局变量并在父子进程中对其进行不同的操作，观察并解释所做操作和输出结果。



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。

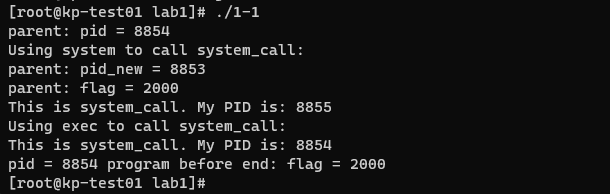
父进程与子进程先后交替运行，合理的。

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



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。



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。

## 1.1.5 测试数据设计

无需数据测试。

## 1.1.6 程序运行初值及运行结果分析

运行结果已经分析。

## 1.1.7 实验总结

### 1.1.7.1 实验中的问题与解决过程

1. **问题：隐式函数声明警告**
   * **描述：** 在最初的版本中，使用了 **wait(NULL)** 函数，但没有包含 **<sys/wait.h>** 头文件，导致编译器发出“implicit declaration of function”警告。
   * **解决：** 在代码中加入 **#include <sys/wait.h>** 来解决这个问题。
2. **问题：全局变量的影响**
   * **描述：** 当添加了全局变量后，发现父子进程中全局变量的变化是独立的。
   * **解决：** 经研究，明确了 fork() 在复制进程时会复制数据段，因此全局变量在父子进程中是独立的。
3. **问题：system() 和 exec() 的用法**
   * **描述：** 在尝试在子进程中调用 **system()** 和 **exec()** 函数时，初次遇到一些困惑和不熟悉的用法。
   * **解决：** 通过查阅文档和测试，理解了这两个函数的基本用法和作用，并成功地在代码中应用了它们。

### 1.1.7.2 实验收获

1. **进程管理理解深化**：通过这个实验，更加深入地了解了 Linux 系统中进程的创建、管理和调度。特别是通过观察 **wait()** 函数的行为，理解了父子进程间同步的重要性。
2. **编程技巧提升**：这个实验让我更熟悉了 C 语言的编程模式，尤其是涉及到系统级调用和进程管理的函数。对 **fork()**, **wait()**, **system()**, 和 **exec()** 等函数有了更深入的了解。
3. **系统调用与命令行工具**：实验中涉及到 **system()** 和 **exec()** 系列函数，使我了解了如何在程序中执行系统命令，以及如何用 **exec()** 替换当前进程的执行内容。
4. **多进程编程模型**：通过在一个程序中创建多个进程，以及管理这些进程的行为和状态，我对多进程编程有了更实际的认识和理解。

### 1.1.7.3 意见与建议

1. **增加更多的进程管理实验**：当前实验内容虽然涵盖了基础的进程创建和管理，但在实际应用中还有更多高级的用法，比如多进程并发处理，进程通信等，建议加入这部分内容。
2. **提供更详细的函数文档和示例代码**：尽管实验手册给出了基础框架，但更多具体函数的使用例子和文档将会更有助于理解。
3. **加强对错误处理的教学**：在实际编程中，错误处理是非常重要的一环。本次实验虽然有简单的错误处理，但没有详细介绍这方面的最佳实践。

## 1.1.8 附件

### 1.1.8.1 附件1 程序

[1-1.c](E:\\OSLAB\\XJTU_OSLab\\lab_1\\lab1-1\\1-1.c)

[system\_call.c](file:///E:\OSLAB\XJTU_OSLab\lab_1\lab1-1\system_call.c)

### 1.1.8.2 附件2 Readme

<lab1-1Readme.md>

# 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。

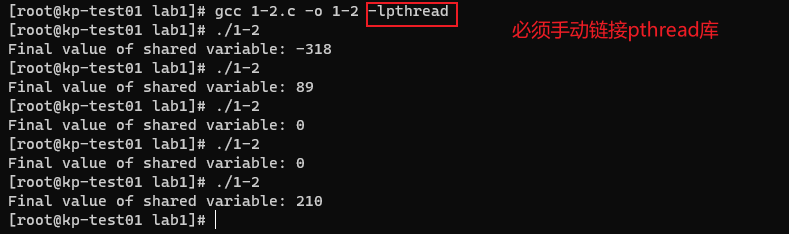
（3）互斥与同步： 为了解决竞态条件带来的问题，可以使用互斥锁（Mutex）来保护共享变量的访问。在每个线程对变量进行操作之前，先获取互斥锁，操作完成后再释放锁。这样一来，每次只有一个线程能够访问变量，从而避免了并发访问带来的不稳定性。

（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。

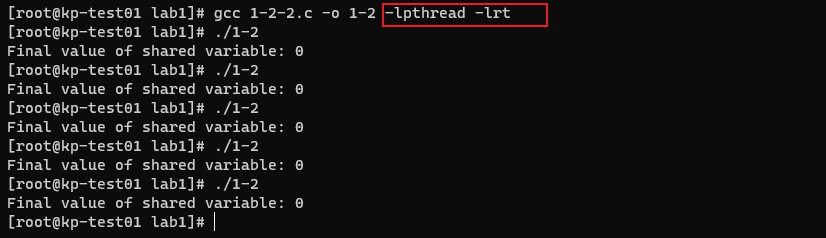
int sem\_destroy(sem\_t \*sem);

* **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。

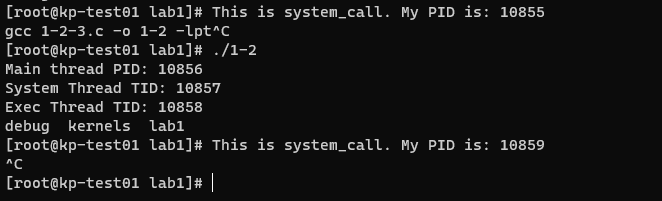
整体的代码为：

#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;  
}



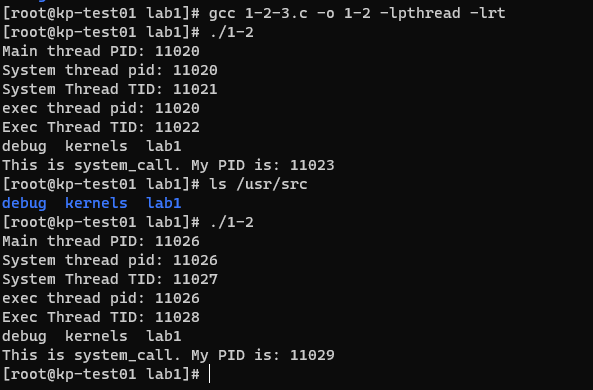
步骤三： 在第一部分实验了解了 system()与 exec 族函数的基础上，将这两个函数的调用改为在线程中实现，输出进程 PID 和线程的 TID 进行分析。

#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

### 1.2.7.1 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。

### 1.2.7.2 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。

### 1.2.7.3 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

[1-2-1.c](E:\\OSLAB\\XJTU_OSLab\\lab_1\\lab1\\1-2-1.c)

[1-2-2.c](file:///E:\OSLAB\XJTU_OSLab\lab_1\lab1\1-2-2.c)

[1-2-3.c](file:///E:\OSLAB\XJTU_OSLab\lab_1\lab1\1-2-3.c)

### 1.2.8.2 附件2 Readme

[lab1-2Readme.md](E:\\OSLAB\\XJTU_OSLab\\lab_1\\lab1-2Readme.md)

# 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 实验思想

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

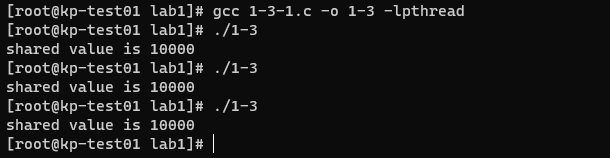
步骤一： 根据实验内容要求，编写模拟自旋锁程序代码 spinlock.c， 待补充主函数的示例代码如下：

/\*\*\*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;  
}

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



自旋锁设定成功，每个线程都正确地对shared\_value进行了修改

## 1.3.5 测试数据设计

无

## 1.3.6 程序运行初值及运行结果分析

shared\_value初始值为0

运行后获得了正确的结果：10000，这说明自旋锁设定有效

## 1.3.7 实验总结

### 1.3.7.1 实验中的问题与解决过程

在实验过程中，我遇到了多个问题。其中一个重要的问题是Segmentation fault（段错误）。这一问题是由于在创建线程时未将自旋锁作为参数传入线程函数，导致线程函数试图访问一个空指针，从而触发了段错误。经过仔细分析代码和逻辑，发现应将自旋锁的地址作为**pthread\_create**的第四个参数传入，解决了这一问题。

另一个问题是编译警告，关于**exit**函数未声明。这个问题的解决相对简单，仅需要包含相应的头文件**<stdlib.h>**即可。

### 1.3.7.2 实验收获

1. **提高了问题解决能力**: 遇到的问题和解决过程锻炼了分析问题和调试代码的能力，尤其是如何定位和修复内存访问相关的错误。
2. **加强了实际操作能力**: 不仅仅是理论知识，还有实际将理论应用到代码中的能力也得到了加强。

### 1.3.7.3 意见与建议

1. **注意代码的健壮性**: 在编写代码时，应考虑各种边界条件和异常情况，例如空指针、非法输入等。
2. **充分利用编译器**: 使用编译器的警告和错误信息作为调试的线索，而不仅仅是编译的障碍。
3. **注重代码可读性**: 代码是给人看的，其次才是给机器执行。应确保代码结构清晰，命名规范，同时也要充分注释。
4. **持续学习和实践**: 技术是日新月异的，通过不断的学习和实践，才能跟上技术的步伐，更好地应对各种问题。

## 1.3.8 附件

### 1.3.8.1 附件1 程序

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

### 1.3.8.2 附件2 Readme

<lab1-3Readme.md>