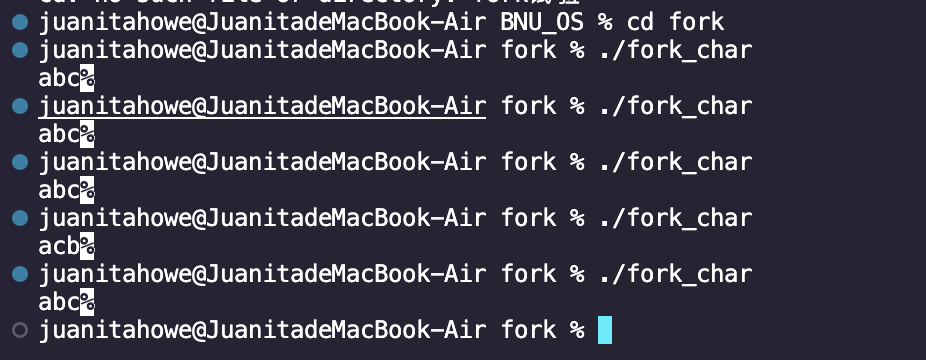
# 实验1 fork

# q1

实验实现代码如下

#include <stdio.h>  
#include <unistd.h>  
#include <sys/types.h>  
  
int main() {  
 pid\_t pid1, pid2;  
 // 创建第一个子进程  
 pid1 = fork();  
 if (pid1 == 0) {  
 // 子进程1的代码  
 printf("b");  
 } else {  
 // 在父进程中创建第二个子进程  
 pid2 = fork();  
 if (pid2 == 0) {  
 // 子进程2的代码  
 printf("c");  
 } else {  
 // 父进程的代码  
 printf("a");  
 }  
 }  
  
 return 0;  
}

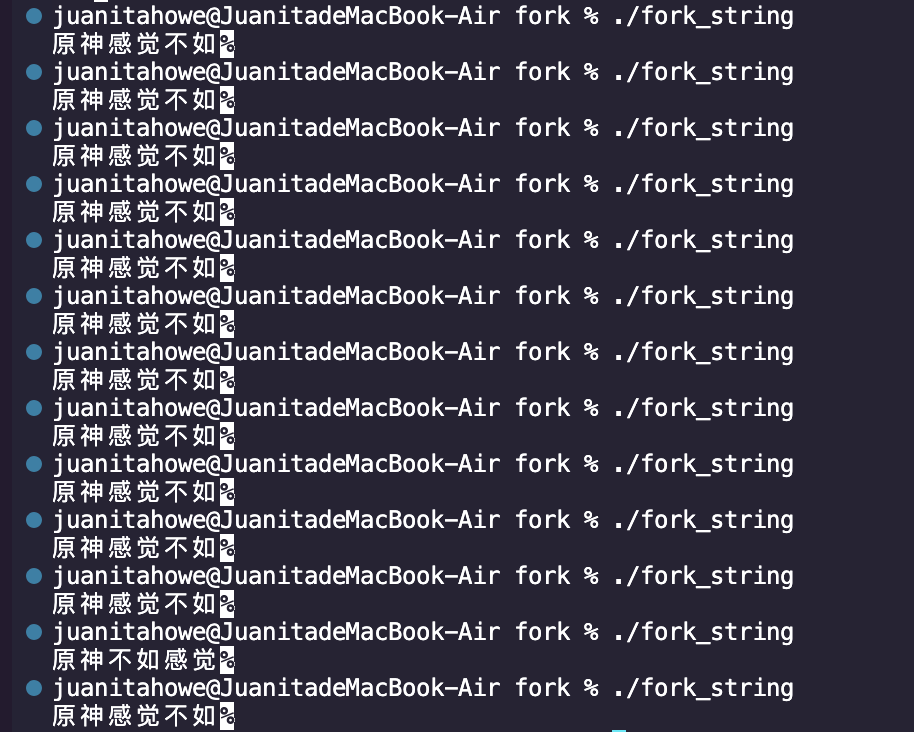
我通过fork创建子进程，并且依进程id不同，输出不同的字母，每次运行输出结果不同。这是因为进程调度是由操作系统控制的，而操作系统调度进程需要考虑很多综合因素和随机性，难以复现，所以每个运行结果不同

​​

# q2

#include <stdio.h>  
#include <unistd.h>  
#include <sys/types.h>  
  
int main() {  
 pid\_t pid1, pid2;  
 // 创建第一个子进程  
 pid1 = fork();  
 if (pid1 == 0) {  
 // 子进程1的代码  
 printf("感觉");  
 } else {  
 // 在父进程中创建第二个子进程  
 pid2 = fork();  
 if (pid2 == 0) {  
 // 子进程2的代码  
 printf("不如");  
 } else {  
 // 父进程的代码  
 printf("原神");  
 }  
 }  
  
 return 0;  
}

将上述问题中char换做string，每次运行输出结果也不同，愿意同上

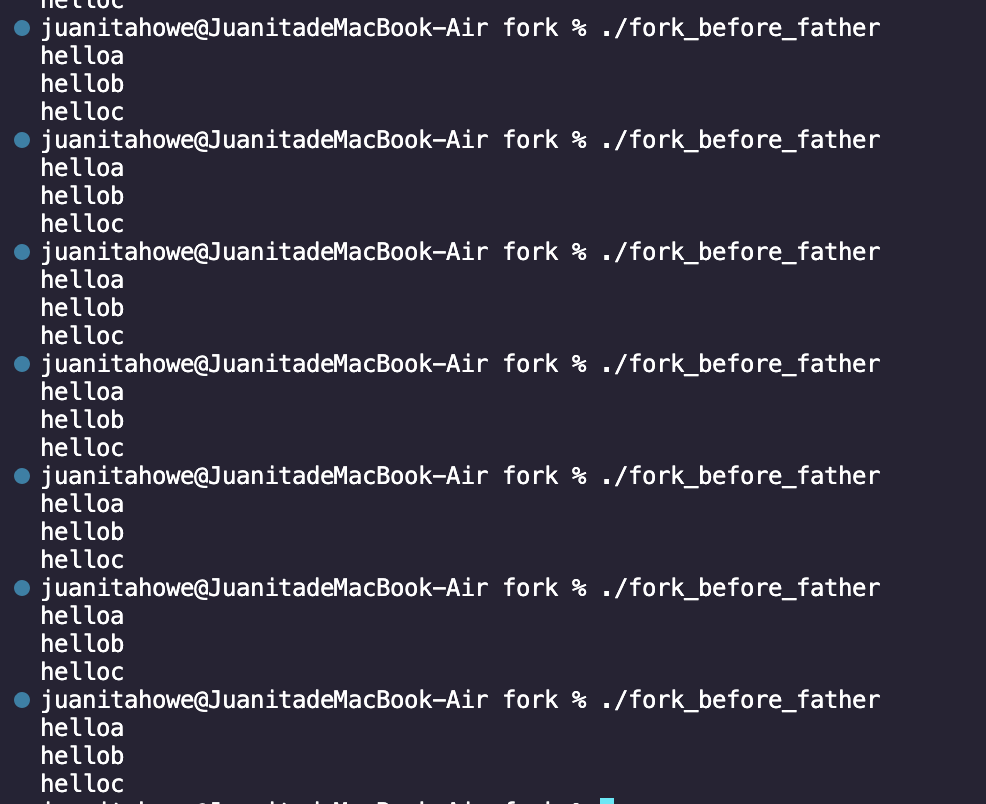
​​

# q3

代码如下

#include <stdio.h>  
#include <unistd.h>  
#include <sys/types.h>  
  
int main() {  
 pid\_t pid1, pid2;  
 printf("hello");  
 // 创建第一个子进程  
 pid1 = fork();  
 if (pid1 == 0) {  
 // 子进程1的代码  
   
 printf("b\n");  
 } else {  
 // 在父进程中创建第二个子进程  
 pid2 = fork();  
 if (pid2 == 0) {  
 // 子进程2的代码  
 printf("c\n");  
 } else {  
 // 父进程的代码  
 printf("a\n");  
 }  
 }  
  
 return 0;  
}

在父进程之前输出一句话，结果如下

​​

将上述hello后加上换行符后，仅输出一个hello，而且后面不同进程输出变为乱序

​​

这一不同非常有趣，我参考了[这篇资料](https://www.cnblogs.com/fenglongyu/p/7624440.html)，其大致含义是fork在创建子进程时：

* 在前一个实验时，由于没有换行符，系统将printf输出的内容（stdout？）写入缓冲区，所以三个子进程都会打印这个hello

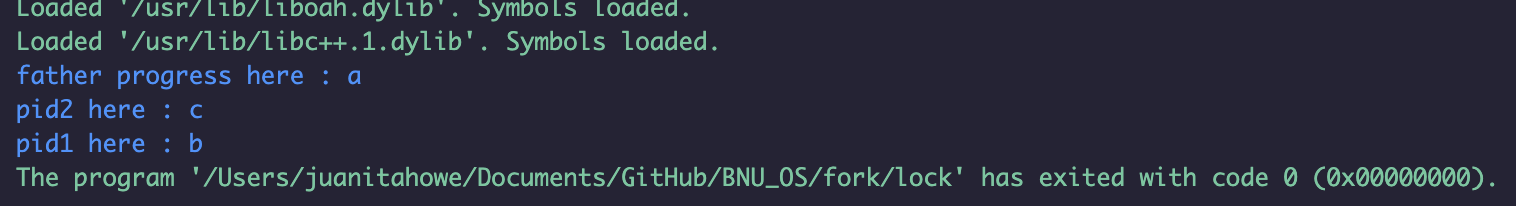
* 在后一个实验时，因为换行符存在，系统并不会将printf输出的内容写入缓冲区，所以只有一个父进程打印hello

# q4

直接对stdout加锁，发现和之前的输出没有区别，因为进行互斥约束后仍然可以访问stdout中的资源

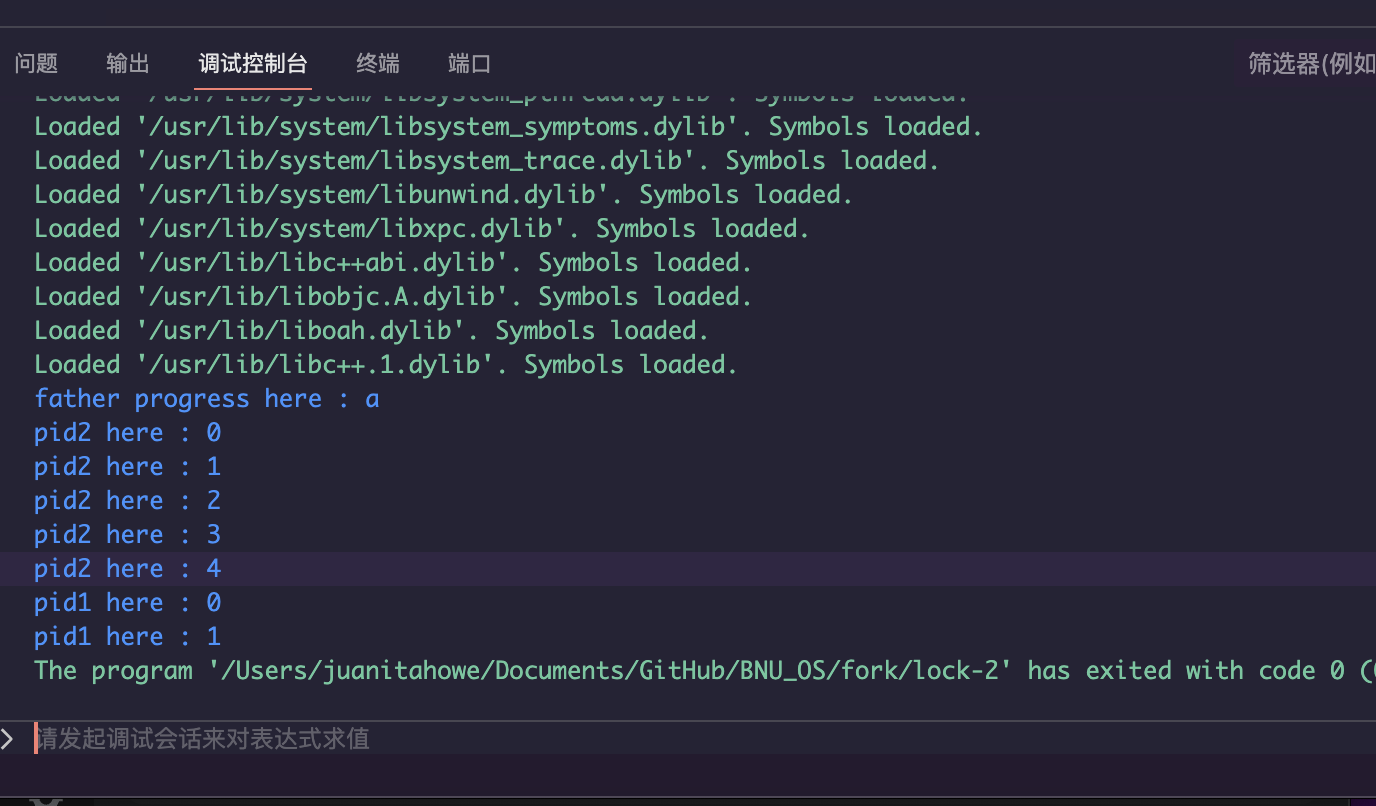
#include <stdio.h>  
#include <unistd.h>  
#include <stdlib.h>  
  
int main()  
{  
 pid\_t pid1, pid2;  
 while ((pid1 = fork()) == -1);  
 if (pid1 == 0)  
 {  
 lockf(1, 1, 0);  
 printf("pid1 here : b\n");  
 lockf(1, 0, 0);  
 }  
 else  
 {  
 while ((pid2 = fork()) == -1);  
 if (pid2 == 0)  
 {  
 lockf(1, 1, 0);  
 printf("pid2 here : c\n");  
 lockf(1, 0, 0);  
 }  
 else  
 {  
 lockf(1, 1, 0);  
 printf("father progress here : a\n");  
 lockf(1, 0, 0);  
 }  
 }  
 return 0;  
}

运行结果如下：

​​

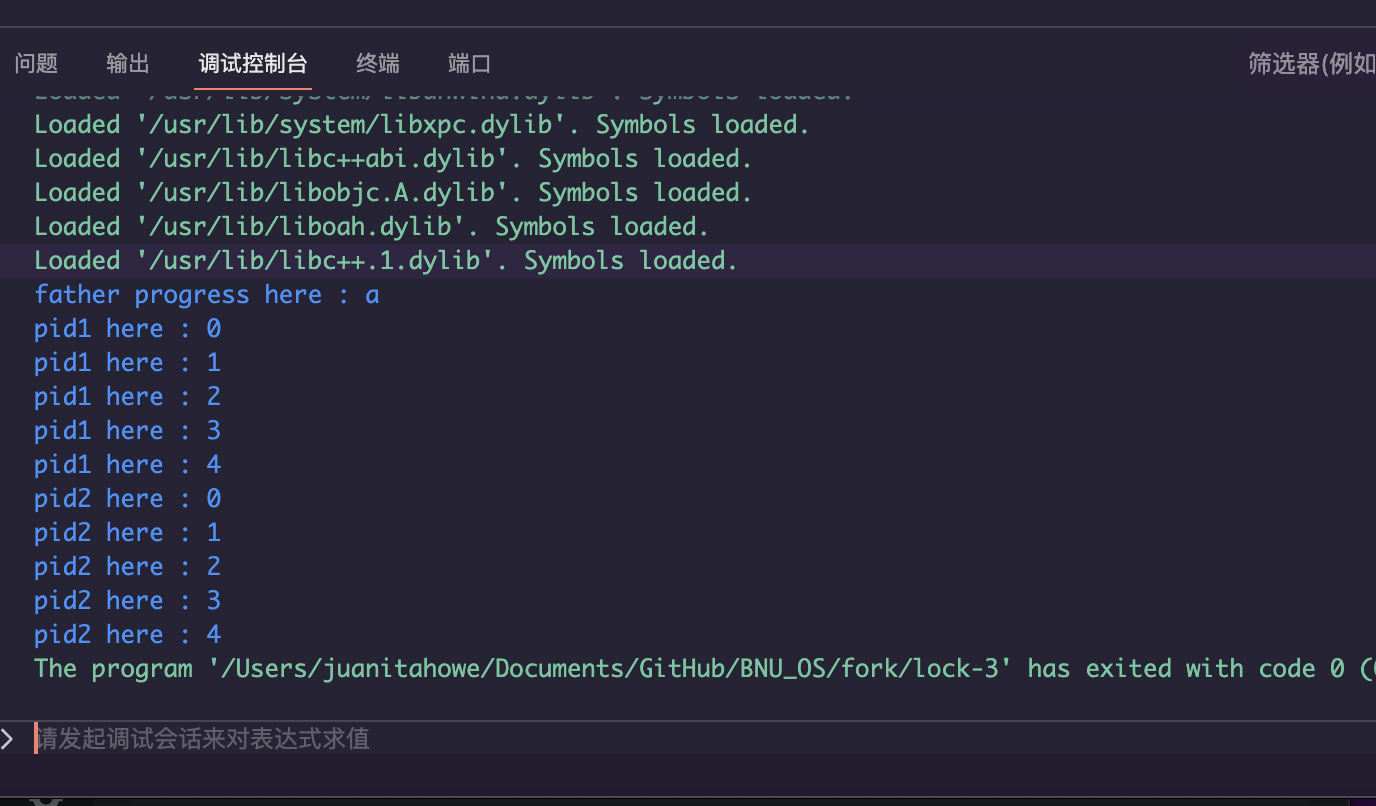
进行循环访问并在内部加锁，发现子进程pid1先执行5次输出，然后轮到pid2，最后是parent输出

#include <stdio.h>  
#include <unistd.h>  
#include <stdlib.h>  
  
int main()  
{  
 pid\_t pid1, pid2;  
 while ((pid1 = fork()) == -1);  
 if (pid1 == 0)  
 {  
 for (int i = 0; i < 5; ++i)  
 {  
 lockf(1, 1, 0);  
 printf("pid1 here : %d\n", i);  
 lockf(1, 0, 0);  
 }  
 }  
 else  
 {  
 while ((pid2 = fork()) == -1);  
 if (pid2 == 0)  
 {  
 for (int i = 0; i < 5; ++i)  
 {  
 lockf(1, 1, 0);  
 printf("pid2 here : %d\n", i);  
 lockf(1, 0, 0);  
 }  
 }  
 else printf("father progress here : a\n");  
 }  
 return 0;  
}

​​

进行循环访问并在外部加锁，发现输出情况一样：

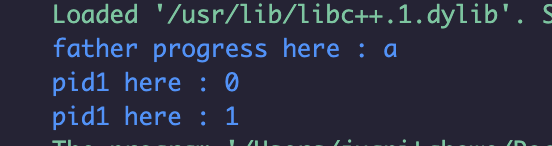
#include <stdio.h>  
#include <unistd.h>  
#include <stdlib.h>  
  
int main()  
{  
 pid\_t pid1, pid2;  
 while ((pid1 = fork()) == -1);  
 if (pid1 == 0)  
 {  
 lockf(1, 1, 0);  
 for (int i = 0; i < 5; ++i)  
 printf("pid1 here : %d\n", i);  
 lockf(1, 0, 0);  
  
 }  
 else  
 {  
 while ((pid2 = fork()) == -1);  
 if (pid2 == 0)  
 {  
 lockf(1, 1, 0);  
 for (int i = 0; i < 5; ++i)  
 printf("pid2 here : %d\n", i);  
 lockf(1, 0, 0);  
 }  
 else printf("father progress here : a\n");  
 }  
 return 0;  
}

​​

‍

上锁后令进程进行短暂休眠，发现pid1和pid2交替输出，并且父进程不是最后输出：

#include <stdio.h>  
#include <unistd.h>  
#include <stdlib.h>  
  
int main()  
{  
 pid\_t pid1, pid2;  
 while ((pid1 = fork()) == -1);  
 if (pid1 == 0)  
 {  
 for (int i = 0; i < 5; ++i)  
 {  
 lockf(1, 1, 0);  
 printf("pid1 here : %d\n", i);  
 lockf(1, 0, 0);  
 sleep(1);  
 }  
 }  
 else  
 {  
 while ((pid2 = fork()) == -1);  
 if (pid2 == 0)  
 {  
 for (int i = 0; i < 5; ++i)  
 {  
 lockf(1, 1, 0);  
 printf("pid2 here : %d\n", i);  
 lockf(1, 0, 0);  
 sleep(1);  
 }  
 }  
 else printf("father progress here : a\n");  
 }  
 return 0;  
}

​​

# q5

以上各种情况都多运行几次，观察每次运行结果是否都一致？为什么？

不一致，因为os调度具有随机性和不可复现性

# 总结

在**操作系统**中，创建**进程**是通过调用操作系统提供的**系统调用**来实现的。进程是执行中的程序实例，它拥有自己的内存空间、代码、数据和执行状态。下面是一个通常的步骤来创建进程：

1. 操作系统提供一个用于创建新进程的系统调用，例如**fork**()函数（Unix/Linux）或**CreateProcess**()函数（Windows）。

1. 应用程序通过调用这个系统调用来创建新的进程。

1. 操作系统为新进程分配一块内存空间，这个内存空间将用于存储进程的代码、数据和堆栈。

1. 操作系统将父进程的代码、数据和执行状态复制到新进程的内存空间中。

1. 新进程获得一个唯一的进程标识符（PID），用于在操作系统中标识和管理进程。

1. 父进程和新进程开始并发执行，它们各自拥有自己的执行状态和执行流程。

1. 父进程和新进程可以通过进程间通信机制（例如管道、共享内存、消息队列等）进行数据交换和协作。

当父进程fork子进程后，父进程和子进程会从程序的不同位置开始执行。

对于父进程，它会从调用fork的位置继续执行。

对于子进程，它会从fork调用之后的位置开始执行，即它会从父进程的内存副本中的相同指令位置开始执行。

这个设计是为了实现进程的复制和并行执行。

创建子进程时，操作系统会为子进程分配一块与父进程相同大小的内存空间，也称为内存副本。然后，操作系统将父进程的内存副本复制给子进程，包括父进程的程序代码、数据和堆栈等。

由于子进程共享父进程的内存副本，它们从相同的程序代码位置开始执行，以确保它们能够执行相同的程序逻辑。然而，父进程和子进程并不是从完全相同的位置开始执行。父进程会从调用fork的位置继续执行，而子进程会从fork调用之后的位置开始执行，即它会从父进程的内存副本中的相同指令位置开始执行。

通过这种方式，父进程和子进程可以同时执行相同的程序代码，但它们是独立且并行执行的，从而实现了进程的复制和并发执行的效果。这对于实现并发程序和多任务处理非常重要。