实　　验　　报　　告

学　　号	2302001 1046	姓　　名	翟一航	专业班级	软件工程 23
课程名称	操作系统			学期	2025 秋
任课教师	李晓慧		完成日期		12.22

一、实验目的及要求

（1）了解进程与程序的区别，加深对进程概念的理解；
（2）进一步认识进程并发执行的原理，理解进程并发执行的特点，区别进程并发执行与顺序执行；
（3）分析进程争用临界资源的现象，学习解决进程互斥的方法。
（4）了解 fork( )系统调用的返回值，掌握用 fork()创建进程的方法；
（5）熟悉 wait、exit 等系统调用。

二、实验内容与源码

实验内容 1：

- 编写 C 语言程序（程序名为 fork1.c），使用系统调用 fork( )创建两个子进程。当程序运行时，系统中有一个父进程和两个子进程在并发执行。父亲进程执行时屏幕显示“I am father”，儿子进程执行时屏幕显示“I am son”，女儿进程执行时屏幕显示“I am daughter”。
- 多次连续反复运行这个程序，观察屏幕显示结果的顺序，直至出现不一样的情况为止。记下这种情况，试简单分析其原因。

源码：

```
Documents/OS/Lab1 via C v17.0.0-clang
❯ vim fork1.c

Documents/OS/Lab1 via C v17.0.0-clang took 22s
❯ cat fork1.c
# include <stdio.h>
# include <sys/types.h>
# include <unistd.h>

int main()
{
    int pid1,pid2;
    printf("I am father!\n");
    if ((pid1 = fork())<0)
    {
        printf("Child1 fail create!\n");
        return;
    }
    else if (pid1 == 0)
    {
        printf("I am son!\n");
        return;
    }
    if ((pid2 = fork())<0)
    {
        printf("Child2 fail create!\n");
        return;
    }
    else if (pid2 == 0)
    {
        printf("I am daughter!\n");
        return;
    }
}
```

实验内容 2：

- 编写 C 语言程序（程序名为 fork2.c），使用系统调用 fork( )创建一个子进程，然后在子进程中再创建子子进程。当程序运行时，系统中有一个父进程、一个子进程和一个子子进程在并发执行。父亲进程执行时屏幕显示“I am father”，儿子进程执行时屏幕显示“I am son”，孙子进程执行时屏幕显示“I am grandson”。

- 多次连续反复运行这个程序，观察屏幕显示结果的顺序，直至出现不一样的情况为止。记下这种情况，试简单分析其原因。

源码：

```
Documents/OS/Lab1 via C v17.0.0-clang took 13s
❯ cat fork2.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    int pid1, pid2;

    if ((pid1 = fork()) < 0)
    {
        printf("Child fail create!\n");
        return 0;
    }
    else if (pid1 == 0)
    {
        // 子进程
        if ((pid2 = fork()) < 0)
        {
            printf("Grandchild fail create!\n")
            return 0;
        }
        else if (pid2 == 0)
        {
            // 孙子进程
            printf("I am grandson!\n");
            return 0;
        }
        else
        {
            // 子进程
            printf("I am son!\n");
            return 0;
        }
    }
    else
    {
        // 父进程
        printf("I am father!\n");
        return 0;
    }
}
```

三、实验结果与分析

实验内容 1：

正常预期运行结果：这是大多数情况下的结果

```
Documents/OS/Lab1 via C v17.0.0-clang
❯ gcc -o fork2 fork2.c

Documents/OS/Lab1 via C v17.0.0-clang
❯ ./fork2
I am father!
I am son!
I am grandson!
```

不一样的结果：在多次运行时，出现输出顺序改变，son 与 daughter 的输出顺序反过来

```
Documents/OS/Lab1 via C v17.0.0-clang
❯ ./fork1
I am father!
I am daughter!
I am son!

Documents/OS/Lab1 via C v17.0.0-clang
❯
```

分析：

观察源码可以发现，父进程总是先被创建，输出“I am father!”后，父进程先后创建两个子进程，分别用于输出“I am son!”与“I am daughter!”。

因此，父进程的输出先于两个子进程的创建，在输出中一定是排在第一位的，但是两个子进程虽然是线

后创建的，但是它们的执行是并发的，由于操作系统的进程调度，它们执行完的先后顺序可能会有所不同。
在大多数情况之下，由于子进程 2 的创建晚于子进程 1，并且 fork()语句的执行也需要时间，所以子进程 1 的执行多数情况下应该先于子进程 2。最终，会输出：
I am father!
I am son!
I am daughter!

但是在有些情况下，操作系统对两个子进程的调度出现不确定性，这可能导致输出顺序的变化。两个子进程创建完成后，它们的执行是并发的，子进程 1 创建后可能被操作系统挂起，子进程 2 虽然创建的晚，但是却可能先获得 CPU 时间片，最终输出：
I am father!
I am daughter!
I am son!

实验内容 2:
正常预期运行结果：大多数情况下的结果

```
Documents/OS/Lab1 via C v17.0.0-clang
❯ gcc -o fork2 fork2.c
❯
Documents/OS/Lab1 via C v17.0.0-clang
❯ ./fork2
I am father!
I am son!
I am grandson!
```

不一样的结果：多次运行进程调度顺序不一定相同，输出顺序也不同

```
documents/OS/Lab1 via C v17.0.0-clang
❯ ./fork2
I am father!
I am son!
I am grandson!
❯
documents/OS/Lab1 via C v17.0.0-clang
❯ ./fork2
I am grandson!
I am son!
I am father!
❯
documents/OS/Lab1 via C v17.0.0-clang
❯ ./fork2
I am son!
I am father!
I am grandson!
❯
documents/OS/Lab1 via C v17.0.0-clang
❯ ./fork2
I am grandson!
I am son!
I am father!
❯
documents/OS/Lab1 via C v17.0.0-clang
❯ ./fork2
I am father!
I am son!
I am grandson!
❯
documents/OS/Lab1 via C v17.0.0-clang
❯ ./fork2
I am father!
I am grandson!
I am son!
```

分析：
观察源码可以发现，父进程、子进程、孙子进程先后被创建，在大多数情况下，这些进程也按顺序被调度，输出结果为：
I am father!
I am son!
I am grandson!

但是这三个进程并发执行，操作系统对进程的调度先后顺序并非一成不变的，我们在程序中也没有使用信号量等机制对进程的调度进行控制，因而每次运行的输出顺序可能发生变化，想要保证每次都按正常的顺序输出，需要使用 wait()使得子进程等待父进程或添加 sleep 观察。理论上来说，可能出现六种不同的情况（图中展示了其中的 4 种）

四、心得总结（写出在完成实验过程中遇到的问题、解决方法，以及体会、收获等）

实验过程中遇到的问题与解决方法

（1） fork1.c 在使用 gcc 编译时报错

```
Documents/OS/Lab1 via C v17.0.0-clang
❯ gcc -o fork1 fork1.c
fork1.c:12:9: error: non-void function 'main' should return a value [-Wreturn-mismatch]
   12 |         return;
      |         ^
fork1.c:17:9: error: non-void function 'main' should return a value [-Wreturn-mismatch]
   17 |         return;
      |         ^
fork1.c:22:9: error: non-void function 'main' should return a value [-Wreturn-mismatch]
   22 |         return;
      |         ^
fork1.c:27:9: error: non-void function 'main' should return a value [-Wreturn-mismatch]
   27 |         return;
      |         ^
4 errors generated.
```

这是因为在给出的程序中，主函数定义为 int main()，因此不能直接返回空值，需要返回一个整型数值，将 return;修改为 return 0;即可。

（2） 多次运行总是按照进程的创建顺序进行执行

这是因为我运行程序使用的是 MacOS 操作系统，它使用 XNU 内核，大量使用 GCD 进行并发管理，即便是传统的 fork()进程也会收到影响，调度倾向于保持“顺序性”和“可预测性”。
想要实现多进程每次运行结果不一定相同，可以给每个进程添加重负载，放大并发执行的不确定性，将每个进程的执行时间从微秒级延长到毫秒甚至秒级，使得操作系统有足够的时间进行多次调度决策，修改后的源码如下：

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>

void heavy_work() {
    for (long i = 0; i < 50000000; i++);
}

int main()
{
    srand(time(NULL));

    int pid1, pid2;

    if ((pid1 = fork()) < 0)
    {
        printf("Child fail create!\n");
        return 0;
    }
    else if (pid1 == 0)
    {
        // 子进程
        if ((pid2 = fork()) < 0)
        {
            printf("Grandchild fail create!\n");
            return 0;
        }
        else if (pid2 == 0)
        {
            // 孙子进程
            heavy_work();

            printf("I am grandson!\n");
            return 0;
        }
        else
        {
            // 子进程
            heavy_work();

            printf("I am son!\n");
            return 0;
        }
    }
    else
    {
        // 父进程
        heavy_work();

        printf("I am father!\n");
        return 0;
    }
}
```

（3） fork2 在连续多次运行时，出现输出混乱的问题

使用 shell 脚本连续多次运行 fork2，出现如图混乱问题：

```
documents/OS/Lab1 via C v17.0.0-clang
❯ for i in {1..10}; do echo -n "Run $i: "; ./fork2; done
Run 1: I am father!
Run 2: I am son!
I am grandson!
I am father!
I am son!
Run 3: I am grandson!
I am father!
Run 4: I am son!
I am grandson!
I am father!
Run 5: I am son!
I am grandson!
I am father!
Run 6: I am son!
I am grandson!
I am father!
Run 7: I am son!
I am grandson!
I am father!
Run 8: I am son!
I am grandson!
I am father!
Run 9: I am son!
I am grandson!
I am father!
Run 10: I am son!
I am grandson!
I am father!
I am son!
I am grandson!
```

这是因为程序创建了僵尸进程，子进程结束时父进程没有调用 wait()回收，使得子进程仍然占用系统资源，它们的输出在后续的运行中泄露了出来。想要解决可以添加 wait()进行回收，或者添加 sleep 观察。

体会与收获

通过本次实验，我对操作系统进程管理机制有了更深刻的理解。随着实验的深入，特别是通过添加重负载和观察异常输出，我逐渐认识到并发执行的真实性和复杂性。实验让我直观地体会到，即使是最简单的程序，在并发环境下也可能表现出意想不到的行为，fork()系统调用看似简单，但其背后涉及的进程复制、资源分配、调度决策等机制却相当复杂。

通过此次实验我不仅掌握了使用 fork()创建多进程的基本方法，更重要的是学会了分析和解释并发执行中的现象。同时，不同操作系统在进程调度上的差异也拓宽了我的视野，认识到操作系统设计与实现的多样性。这次实验经历为后续学习进程同步、死锁等更复杂的并发控制问题打下了坚实基础。