1.1 进程相关编程实验

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

```
[root@kp-test01 lab1]# gcc 1.c -o 1-1
[root@kp-test01 lab1]# ./1-1
parent: pid = 6350
child: pid = 0
parent: pid_new = 6349
child: pid_new = 6350
[root@kp-test01 lab1]# ./1-1
parent: pid = 6352
child: pid = 0
parent: pid_new = 6351
child: pid_new = 6352
[root@kp-test01 lab1]# ./1-1
parent: pid = 6354
child: pid = 0
parent: pid_new = 6353
child: pid_new = 6354
[root@kp-test01 lab1]#
```

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。

```
[root@kp-test01 lab1]# ./1-1
parent: pid = 6362
child: pid = 0
parent: pid_new = 6361
child: pid_new = 6362
[root@kp-test01 lab1]# ./1-1
child: pid = 0
parent: pid = 6364
child: pid_new = 6364
parent: pid_new = 6363
[root@kp-test01 lab1]# ./1-1
parent: pid = 6366
child: pid = 0
parent: pid_new = 6365
child: pid_new = 6366
[root@kp-test01 lab1]# ./1-1
parent: pid = 6368
child: pid = 0
parent: pid_new = 6367
child: pid_new = 6368
[root@kp-test01 lab1]#
```

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。

```
[root@kp-test01 lab1]# ./1-1
parent: pid = 8835
child: pid = 0
parent: pid_new = 8834
child: pid_new = 8835
parent: flag = 2000
child: flag = 1000
[root@kp-test01 lab1]# ./1-1
parent: pid = 8837
child: pid = 0
parent: pid_new = 8836
child: pid_new = 8837
parent: flag = 2000
child: flag = 1000
[root@kp-test01 lab1]# ./1-1
parent: pid = 8839
parent: pid_new = 8838
child: pid = 0
parent: flag = 2000
child: pid_new = 8839
child: flag = 1000
[root@kp-test01 lab1]#
```

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

```
[root@kp-test01 lab1]# ./1-1
parent: pid = 6378
child: pid = 0
parent: pid_new = 6377
child: pid_new = 6378
parent: flag = 2000
child: flag = 1000
pid = 0 program before end: flag = 1000
pid = 6378 program before end: flag = 2000
[root@kp-test01 lab1]# ./1-1
parent: pid = 6380
child: pid = 0
parent: pid_new = 6379
child: pid_new = 6380
parent: flag = 2000
child: flag = 1000
pid = 0 program before end: flag = 1000
pid = 6380 program before end: flag = 2000
[root@kp-test01 lab1]#
```

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。

```
[root@kp-test01 lab1]# ./1-1
parent: pid = 8854
Using system to call system_call:
parent: pid_new = 8853
parent: flag = 2000
This is system_call. My PID is: 8855
Using exec to call system_call:
This is system_call. My PID is: 8854
pid = 8854 program before end: flag = 2000
[root@kp-test01 lab1]#
```

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。

1.1.2实验总结

1.1.2.1 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。

1.1.2.2 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。

1.1.2.3 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。