****



**SHANGHAI UNIVERSITY**

**操作系统（一）实验报告**

|  |  |
| --- | --- |
| **组 号** | **第 7 组** |
| **学号姓名** | **20121034 胡才郁** |
| **实验序号** | **3** |
| **日期** | **2022年10月7日** |

实验三 进程管理及进程通信

# **1 实验目的与要求**

1. 利用Linux提供的系统调用设计程序，加深对进程概念的理解。

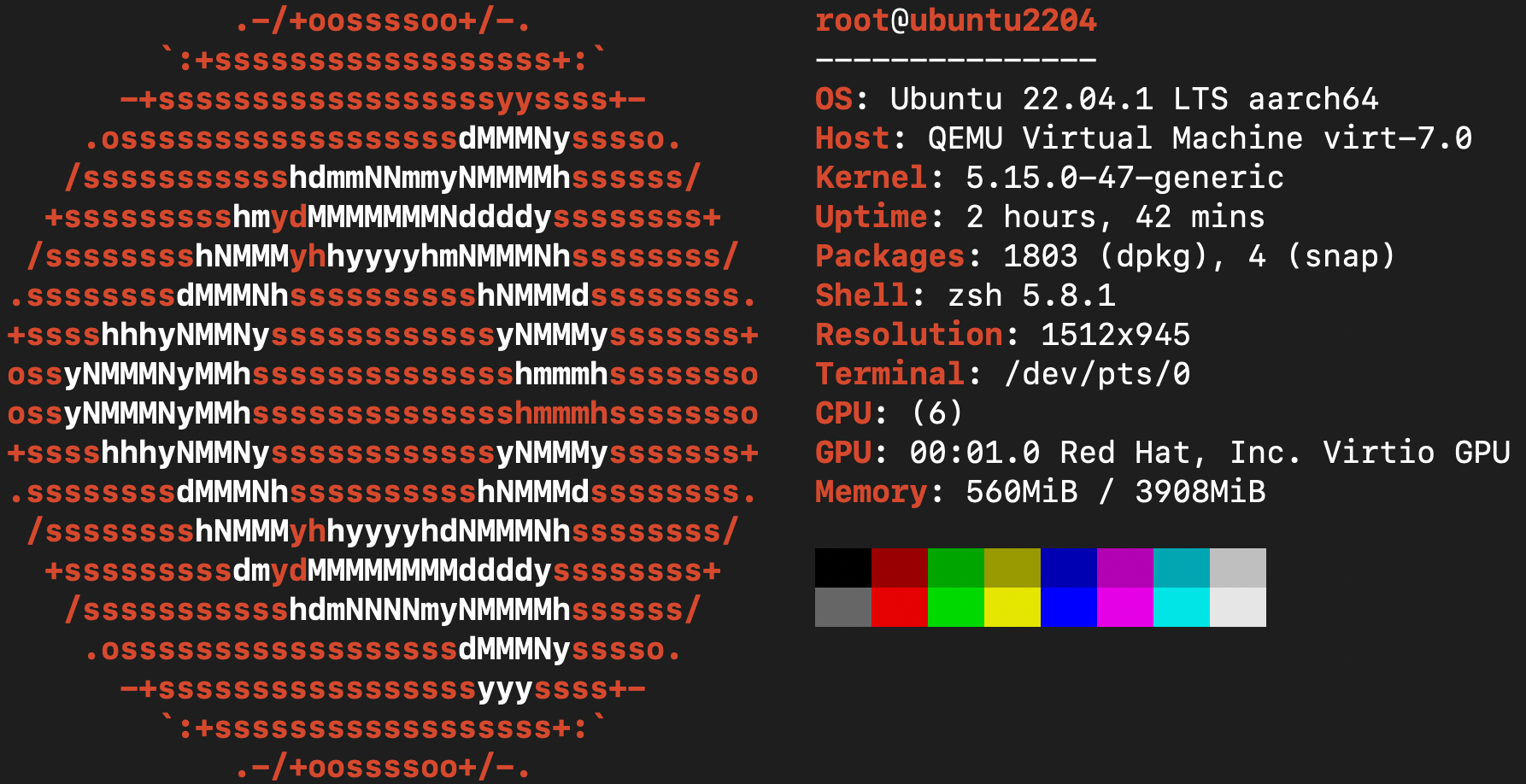
2. 体会系统进程调度的方法和效果。

3. 了解进程之间的通信方式以及各种通信方式的使用。

# **2 实验环境**

* 操作系统: Ubuntu 22.04
* 宿主机:QEMU Virtual Machine

其余主要环境如下图所示：



**图1. 实验环境**

# **3 实验内容及其设计与实现**

**问题一**：**编写程序。显示进程的有关标识（进程标识、组标识、用户标识等）。经过5秒钟后，执行另一个程序，最后按用户指示（如：Y/N）结束操作。**

**答：**问题一主要是帮助熟悉如何在C语言中使用系统调用接口。对于fork函数而言，当一个进程调用fork函数后，通过系统调用的方式创建子进程，系统先为子进程分配资源，例如存储数据和代码的空间，然后把父进程的所有值都复制到新的子进程中。

在fork函数执行完毕后，如果创建新进程成功，则出现两个进程，一个是子进程，一个是父进程。在子进程中，fork函数返回0，在父进程中，fork函数返回新创建子进程的进程ID。对于下图代码的第14行，当创建了一个子进程后，原先的父进程与子进程继续执行剩下的代码，由于各自fork得到的返回值不同，执行不同的逻辑。在子进程中利用execlp函数请求系统调用执行pwd程序，而在父进程中sleep并等待用户输入。

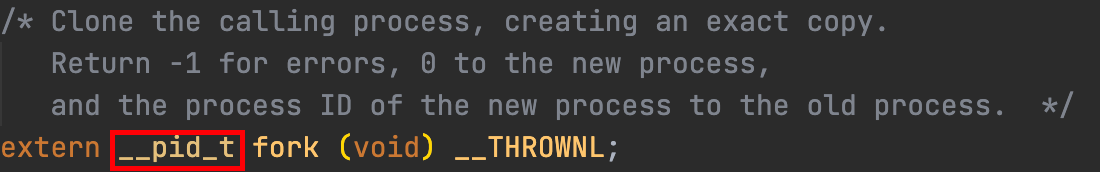
详细代码如下图所示：



**图2. 问题一代码**

execlp函数的作用是运行可执行文件，优先在环境变量中搜索，所以此处不需要把pwd的完整路径写出，此函数也可以添加执行文件时附加的参数。pwd是一个shell built-in command，本质是运行了/bin/pwd，即bin目录下的可执行文件，在shell中的命令都是可执行文件的映射，这也体现了Linux中“万物皆文件”的设计哲学。

一个有趣的点在于fork的函数原型返回值首先为标准库文件unistd.h中宏定义的\_\_pid\_t数据类型，\_\_pid\_t数据类型分别经过多次链接的宏定义，最终为int，这也解释了为什么实验指导书使用int类型接收fork的返回值，也体现出底层库函数开发的规范与复杂。通过多重宏定义的方式，更好的组织代码结构。此处不再展开层层宏定义的过程，详细各层宏定义展开在附录中补充介绍。

****

**图3. 标准库文件unistd.h中fork函数的原型**

\_\_pid\_t 🡪 \_\_PID\_T\_TYPE 🡪 \_\_S32\_TYPE 🡪 int

**图4. 返回值类型宏定义过程**

**问题二：参考例程1，编写程序。实现父进程创建一个子进程。体会子进程与父进程分别获得不同返回值，进而执行不同的程序段的方法。**

**答：**问题一中的程序(图2)就很好的展示了如何使父进程与子进程执行不同的代码逻辑。

**思考题：子进程是如何产生的？又是如何结束的？子进程被创建后它的运行环境是怎样建立的？**

**答：**子进程是由父进程fork创建形成的，开始时就是父进程的副本，通过exit函数自我结束。子进程被创建后被分配给pid，检查同时运行的进程数目，并且拷贝进程表项的数据，由子进程继承父进程所有文件。

通常，在系统调用fork之后，子进程使用系统调用 exec，以用新程序来取代进程的内存空间。系统调用 exec加载二进制文件到内存中（破坏了包含系统调用 exec的原来程序的内存内容），并开始执行。采用这种方式，这两个进程能相互通信，并能按各自方法运行。父进程能够创建更多子进程，或者如果在子进程运行时没有什么可做，那么它采用系统调用 wait把自己移出就绪队列，直到子进程终止。因为调用 exec用新程序覆盖了进程的地址空间，所以调用 exec除非出现错误，不会返回控制。

图示

描述已自动生成

**图5. 父子进程间执行不同程序段**

**问题三：参考例程2，编写程序。父进程通过循环语句创建若干子进程。探讨进程的家族树以及子进程继承父进程的资源的关系。**

**思考题：**

**① 画出进程的家族树。子进程的运行环境是怎样建立的？反复运行此程序看会有什么情况？解释一下。**

文本

描述已自动生成

**图6. 父进程在循环中i值对应状态**

图示

描述已自动生成

**图7. 进程家族树**

子进程的运行环境是从父进程中fork一份资源，拷贝了相同的环境，但由系统创建PCB时给予一个新的pid。在这个程序中，子进程对于父进程的资源，变量i进行了fork，这个拷贝可以理解为编程语言中说的“深拷贝”。把相关变量在子进程的内存中创建，它们的地址值不同于父进程的原始值地址。所以在子进程中修改i不会影响父进程中的i。

pid为10511的进程是执行的SHELL程序，他是运行二进制文件程序的父进程，它的pid不会随着多次运行代码而改变。pid为13814进程是负责运行二进制文件的进程，此二进制文件由c语言编译后产生获得。反复运行此程序时，虽然pid每次各不相同，但是执行结果的顺序与逻辑关系不变。此程序的执行方式也类似于DFS，对于进程家族树进行前序遍历，也体现出进程家族树“树”的数据结构特点。

**② 修改程序，使运行结果呈单分支结构，即每个父进程只产生一个子进程。画出进程树，解释该程序。**

通过wait函数的特性来控制，wait函数的作用之一是监控子进程运行状态，只要有一个子进程运行结束，wait函数就会返回该子进程的进程号。

因此修改父进程部分的逻辑，当只要wait函数监控到有一个子进程运行结束，就退出循环，这样就不会产生新的子进程，完成单分支进程树的要求。核心代码如下图所示：

图形用户界面, 文本, 应用程序, 聊天或短信

描述已自动生成

**图8. 单分支核心代码**

按照上述代码修改，进程的创建与消亡又类似于栈的方式，父进程先创建后后终止，子进程后创建先终止。即栈的“先进后出”。具体顺序如下图左红色箭头部分。

图示

描述已自动生成文本, 聊天或短信

描述已自动生成

**图9. 单分支结果**

值得注意的是，第一行的my father’s pid is所输出的进程号10511仍与第一小问中相同，原因是这两问虽然执行的可执行文件不同的，但两问的父进程都是shell，此处没有创建新的shell，自然pid也是不变的

**问题四：参考例程3编程，使用fork( )和exec( )等系统调用创建三个子进程。子进程分别启动不同程序，并结束。反复执行该程序，观察运行结果，结束的先后，看是否有不同次序。**

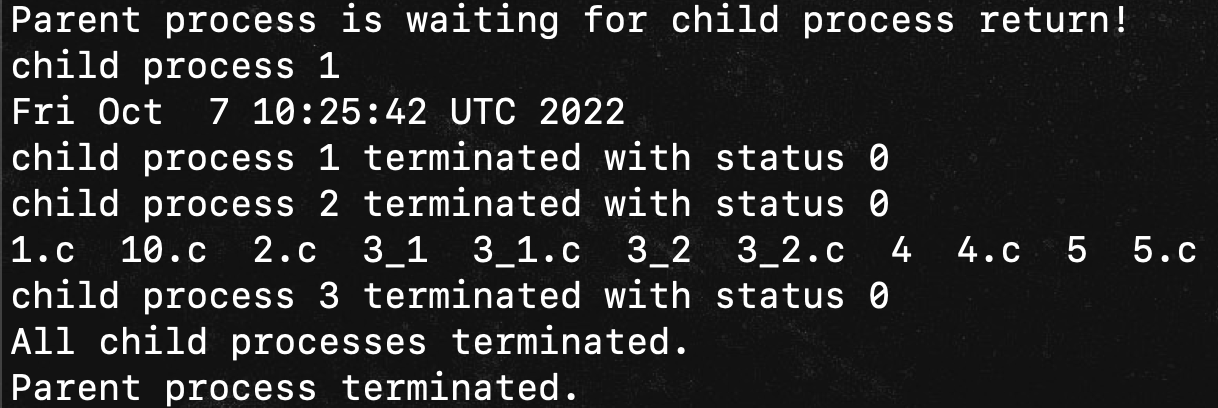
**思考题：子进程运行其它程序后，进程运行环境怎样变化的？反复运行此程序看会有什么情况？解释一下。**

**答：**子进程运行其他程序，相当于在子进程中又新建了一个进程。这个进程在本题中为执行echo、ls等命令的进程。

当进程调用 exec 族函数时，该进程的用户空间代码和数据完全被新程序替换，从新程序的起始处开始执行。调用 exec 族函数并不创建新进程，所以调用 exec 族函数前后该进程的 PID 并不改变。

exec族函数执行程序与直接在shell中执行命令是不一样的，在shell中直接执行命令会新建一个进程运行此程序。

反复运行此程序发现结束的先后次序是不可预知的，每次运行结果不一样。原因是父进程fork创建的三个子进程分别执行不同的逻辑，各自运行命令，他们的结束时间难以预测。如以下两张图，第一张图中就是一号子进程先于于二号子进程抢占了输出缓冲区，child process 1比date指令先输出，而第二张图中则相反。



**图10. 运行结果1**

文本

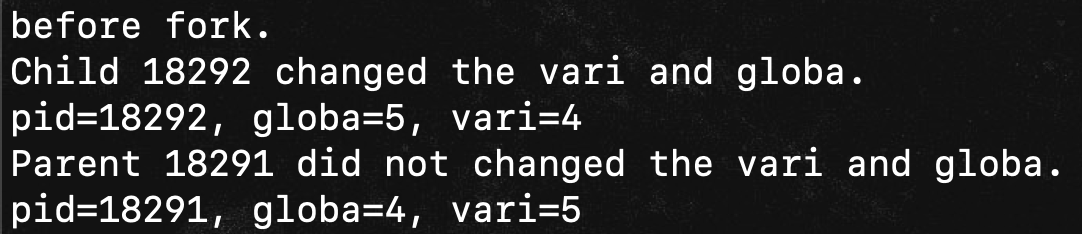
描述已自动生成

**图11. 运行结果2**

**问题五：参考例程4，编程验证子进程继承父进程的程序、数据等资源。如用父、子进程修改公共变量和私有变量的处理结果；父、子进程的程序区和数据区的位置。**

**思考题：子进程被创建后，对父进程的运行环境有影响吗？解释一下。**

**答：**运行结果如下图所示：



**图12. 父进程中，局部变量与全局变量都不变**

子进程被创建后，对父进程的运行环境没有影响。这个题可以类比于第三题，在循环中子进程继承i。子进程被创建后，子进程的运行环境是从父进程中fork一份资源，拷贝了相同的环境，但由系统创建PCB时给予一个新的pid。

在这个程序中，子进程对于父进程的资源，全局变量globa和局部变量vari进行了fork，这个拷贝可以理解为编程语言中说的“深拷贝”，把相关变量在子进程的内存中创建，它们的地址值不同于父进程的原始值地址。

除了进程标识符（PID） 等必要的信息外，子进程的 PCB结构中的绝大部分的信息都是从父进程中拷贝。父进程只复制了自己的PCB块，而代码段，数据段和用户堆栈内存空间并没有复制一份，而是与子进程共享。

而当子进程被撤销时，应归还从父进程那继承的资源；撤销父进程时，也必须同时撤销其所 有的子进程。

**问题六：参照《实验指导》第五部分中“管道操作的系统调用”。复习管道通信概念，参考例程5，编写一个程序。父进程创建两个子进程，父子进程之间利用管道进行通信。要求能显示父进程、子进程各自的信息，体现通信效果。**

**思考题：**

**①什么是管道？进程如何利用它进行通信的？解释一下实现方法。**

所谓“管道”，是指用于连接一个读进程和一个写进程以实现它们之间通信的一个共享文件，又名pipe文件。向管道(共享文件)提供输入的发送进程(即写进程) 以字符流形式将大量的数据送入管道；而接受管道输出的接收进程 (即读进程)则从管道中接收(读)数据。这样使得一系列进程 能通过标准输入输出链接起来，其中每一个进程的输出被直接作为下一个进程的输入。

管道类似于通信中[半双工](https://blog.csdn.net/Dontla/article/details/126603052)信道的进程通信机制，一个管道可以实现双向的数据传输，而同一个时刻只能最多有一个方向的传输，不能两个方向同时进行。管道连接一个写进程、一个读进程，在一个固定大小的缓冲区存放信息。当管道满时，进程在写管道会被阻塞，而当管道空时，进程读管道会被阻塞。

具体而言，管道可以分为匿名管道和命名管道两种。

匿名管道更为常用，匿名管道只能实现有血缘关系的进程间通信（如父子进程，兄弟进程等），它的生命周期随进程的结束结束，通常在父进程和子进程之间传输数据。匿名管道的标准shell语法是列出多 个命令，用竖线分隔。如：ls -l | grep key | less。

匿名管道很好的体现了Linux的设计哲学哲学构建简单且可扩展的软件。每个软件都尽做一件事，但是要把这件事做的极致，这些小而美的软件通过管道即可协同工作，实现各种强大的功能。

另一种是命名管道，它需要手动创建但只要系统启动它都一直存在，超过进程的生命周期。如果不再使用，可以将其删除。使用有名管道既可以进行有血缘关系 的进程间通信，也可以进行没有血缘关系的进程间通信。

**②修改睡眠时机、睡眠长度，看看会有什么变化。请解释。**

当修改睡眠时机与睡眠长度时，会导致子进程1和子进程2结束时间不同，则父进程的被唤醒时间不同。如果子进程1先在管道中写入，则父进程被唤醒后，先读出子进程1在管道中写入的内容，子进程2同理。

**③加锁、解锁起什么作用？不用它行吗？**

加锁、解锁是为了对于临界资源形成保护。如果不使用，理论上如果多个线程同时对管道进行写操作会产生异常。

但是我在实验中修改代码，尝试过不加锁、不解锁、既不加锁也不解锁等多种方式进行验证，并且修改休眠时间来确保进程的访问顺序，不过实验结果都不变，即可以正常读入读出，出现这个结果我比较疑惑。

我查过了lockf函数的man文档，不过文档内容大多是介绍lockf函数的用法，并没有对与不使用lockf这样的锁机制对于并发文件读写的影响。不过，我在一篇关于File locking的帮助文档中找到了下面的描述。[(超链接在此处)](https://www.ict.griffith.edu.au/teaching/2501ICT/archive/guide/ipc/flock.html)

|  |
| --- |
| There are two types of locking mechanisms: mandatory and advisory. Mandatory systems will actually prevent read()s and write()s to file. Several Unix systems support them. Nevertheless, I'm going to ignore them throughout this document, preferring instead to talk solely about advisory locks. With an advisory lock system, processes can still read and write from a file while it's locked. Useless? Not quite, since there is a way for a process to check for the existence of a lock before a read or write. See, it's a kind of cooperative locking system. This is easily sufficient for almost all cases where file locking is necessary. |

根据文档的介绍，进程有一种特殊机制，在读取或写入文件之前检查锁的存在。只是建议在写代码时“手动”使用lockf加锁、解锁，即使没有这样做也不会导致错误。

由于知识欠缺，暂时难以对以上说法做出验证，随着今后学习知识增多，再进行解释。

**问题七：编程验证：实现父子进程通过管道进行通信。进一步编程，验证子进程结束，由父进程执行撤消进程的操作。测试父进程先于子进程结束时，系统如何处理“孤儿进程” 的。**

**思考题：对此作何感想，自己动手试一试？解释一下你的实现方法。**

**答：**把问题六中的程序的父进程部分去掉wait，不等待子进程结束，在子进程部分添加sleep，保证父进程提前于子进程结束。

一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被init进程(进程号为1)所收养，并由init进程对它们完成状态收集工作，对于init进程，他循环的进行wait操作，从而保证子进程的结束。

文本

描述已自动生成

**图13. 孤儿进程的处理**

**问题八：编写两个程序一个是服务者程序，一个是客户程序。执行两个进程之间通过消息机制通信。消息标识MSGKEY 可用常量定义，以便双方都可以利用。客户将自己的进程标识（pid）通过消息机制发送给服务者进程。服务者进程收到消息后，将自己的进程号和父进程号发送给客户，然后返回。客户收到后显示服务者的pid 和ppid，结束。以下例程6基本实现以上功能。这部分内容涉及《实验指导》第五部分中“IPC系统调用”。先熟悉一下，再调试程序。**

**答：**此处的通信方式，实际上为消息队列的方式。

消息队列是在两个不相关进程间传递数据的一种简单、高效方式的方式。它独立于发送进程、接受进程而存在。消息队列提供了一种从一个进程向另一个进程发送一个数据块的方法。每个数据块都被认为是一个管道，接收进程可以独立地接收含有不同管道的数据结构。我们可以通过发送消息来避免命名管道的同步和阻塞问题。

具体到这一道题的代码中，客户端建立消息队列之后，需要定义消息类型，则系统会根据消息类型创建一条对立的消息队列，可以类比上面提到的管道。并进行消息发送操作。如下图所示：

文本

描述已自动生成

**图14. 客户程序**

而服务者程序需要与客户程序使用相同的消息队列，并等待客户程序发送消息，接收到消息发送收到的消息。

文本

描述已自动生成

**图15. 服务者程序**

这里的消息队列可以与问题七中提到的命名管道类比，每个数据块都有一个最大长度的限制。每一种发送的数据块都有一个消息队列，相当于一个独立的管道，相互之间互不影响。

不过，消息队列通信与管道通信相比有以下区别：

1. 匿名管道是跟随进程的，消息队列是跟随内核的，也就是说进程结束之后，匿名管道就消亡了，但是消息队列还会存在。
2. 管道是文件，存放在磁盘上，访问速度慢，消息队列是数据结构，存放在内存，访问速度快。
3. 管道是数据流式存取，消息队列是数据块式存取。

**问题九：**这部分内容涉及《实验指导》第五部分中“有关信号处理的系统调用”。编程实现软中断信号通信。父进程设定软中断信号处理程序，向子进程发软中断信号。子进程收到信号后执行相应处理程序。

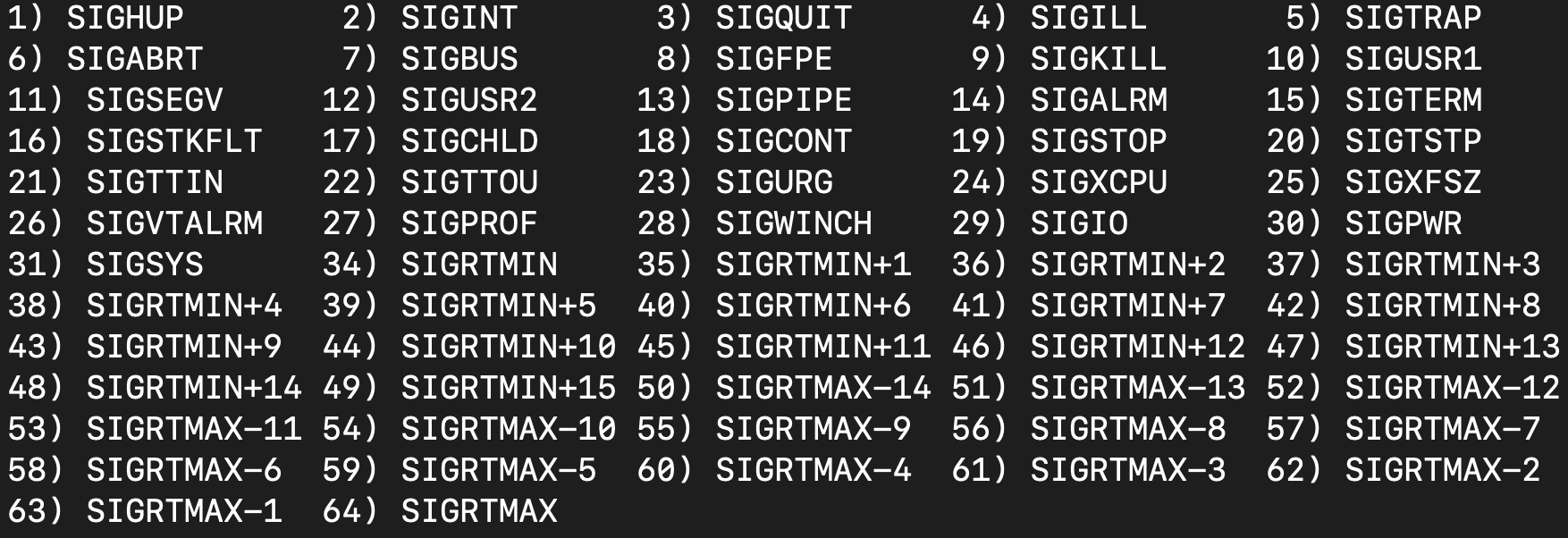
**思考题：**这就是软中断信号处理，有点儿明白了吧？讨论一下它与硬中断有什么区别？看来还挺管用，好好利用它。

**答：**软中断是执行中断指令产生的，而硬中断是由外设引发的。

硬中断的中断号是由中断控制器提供的，而软中断的中断号由指令直接指出，无需使用中断控制器。并且硬中断是可屏蔽的，软中断不可屏蔽。

软中断信号(signal)是Linux下用来在进程间传递消息的方式之一,也是进程间唯一的异步通信方式。对于信号阻塞而言，他可以帮助操作系统不中断正在处理的信号、不忽略当前心得信号。

可以使用命令kill -l查看具体信号与数字代号的关联。其中，非可靠信号为1～31号信号，其中的信号可能会丢失。而34～64号信号，信号不可能丢失。而本题给出的范例代码正是在父进程中使用kill发送软中断18号指令给子进程。



**图16. 各个信号与其编号**

**问题十：**怎么样，试一下吗？用信号量机制编写一个解决生产者—消费者问题的程序，这可是受益匪浅的事。本《实验指导》第五部分有关进程通信的系统调用中介绍了信号量机制的使用。

**答：**生产者-消费者模型中，两个进程共用临界资源，所以它们首先互斥；此外，如果缓冲区满，生产者是不能生产的，所以生产者进程受到消费者进程的制约；如果缓冲区空，消费者是不能消费的，所以消费者进程受到生产者进程的制约。一个信号量只能限制一端，要限制两端（空和满）至少要两个信号量。

当分析清楚同步与互斥关系之后，可以确定以下三个信号量：

1. mutex限制缓冲区资源，同时只可以有1个进程访问缓冲区。
2. full限制生产者，只有当缓冲区不满才可以生产。
3. empty限制消费者，只有当缓冲区不空才可以消费。

基于上述的分析，使用POSIX的pthread线程库与semaphore信号量库进行编程与测试。下图左设置了三个信号量，在下图右main函数中分别初始化信号量full为0，信号量empty为缓冲区大小，信号量mutex为1，并且创建了多个消费者与生产者线程，进行模拟。

文本

描述已自动生成文本

低可信度描述已自动生成

**图17. 库引用以及信号量设置(图左) main函数(图右)**

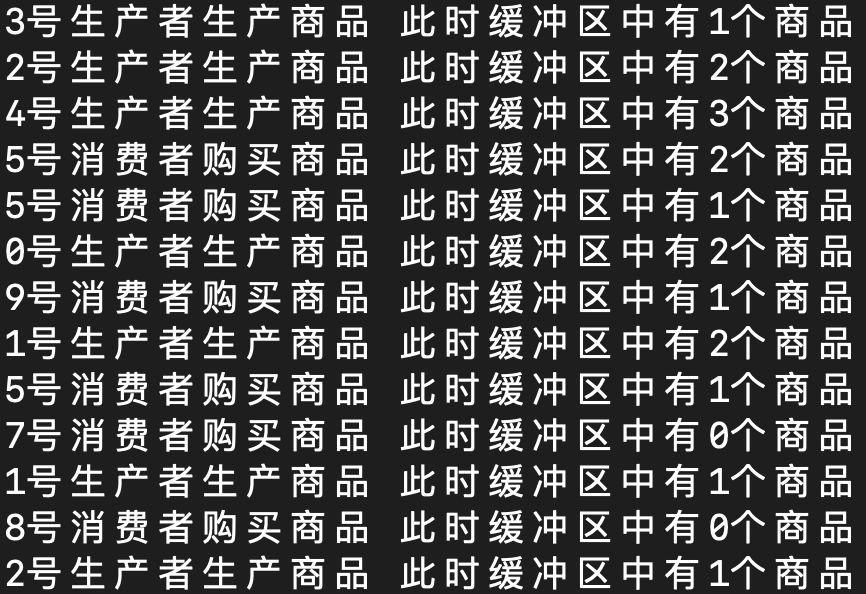
对于消费者而言，由于设置了信号量empty，只有当缓冲区不为空时才可以继续访问。mutex的设置保护了互斥资源商品，此处用变量cache\_来模拟，即同时只有一个进程可以访问cache\_变量。

文本, 日程表

描述已自动生成 文本

描述已自动生成

**图18. 生产者(图左) 消费者(图右)**



**图19. 生产者(图左) 消费者(图右)**

程序运行结果如上，可以发现，初始时刻没有商品，因此消费者需要等到生产者生产后才可以消费。程序结果符合预期。

**4 收获与体会**

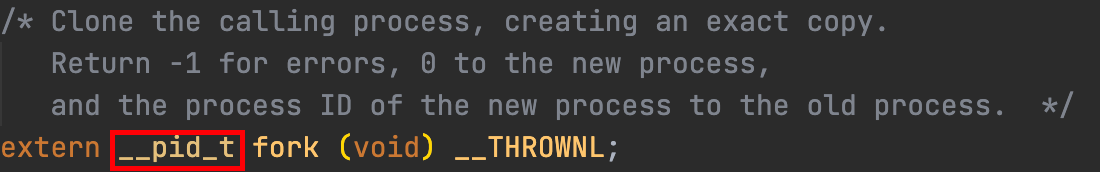
本次的两个实验主要接触了Linux操作系统的CLI，由于centOS的官方Red Hat已经声明不在对于centOS进行版本的更新，Linux发行版众多，但不同版本间相似，本实验使用Ubuntu22.04版本进行实验。

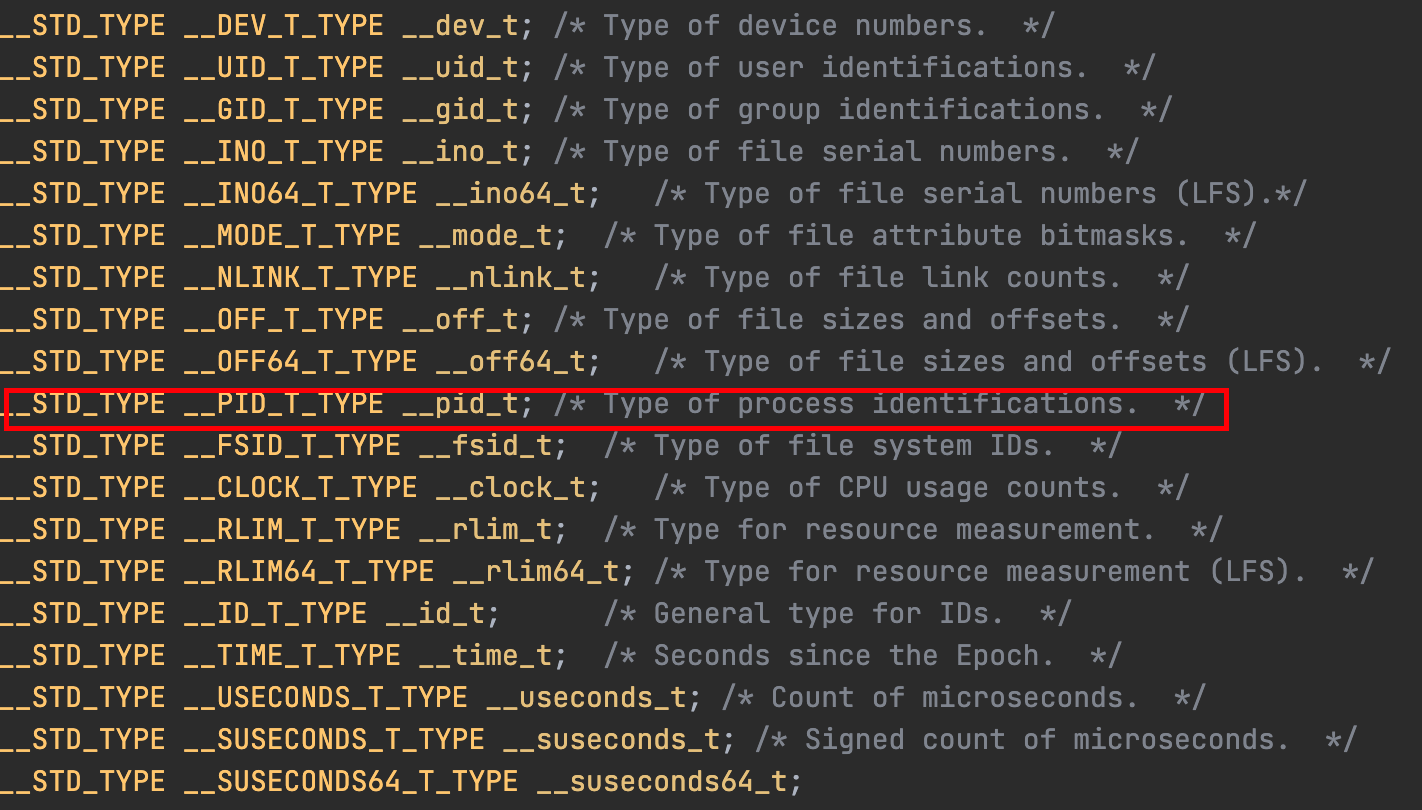
对于Linux的CLI熟悉是必要的，无论是在今后的工作中接触到的操作系统一定是Linux，还是使用服务器GPU进行计算时，基本的命令都是必不可少的。又比如被誉为“最好的编辑器”Vim，当熟练使用后可以大大提升工作效率等等。这两次实验帮助重温了命令行基本操作，为之后的实验打好了基础。

# **5 附录**

fork函数返回值\_\_pid\_t层层宏定义到int的过程如下图所示

\_\_pid\_t 🡪 \_\_PID\_T\_TYPE 🡪 \_\_S32\_TYPE 🡪 int

****



**图20. \_\_pid\_t宏定义为\_\_PID\_TTYPE**

一些文字和图案

描述已自动生成

**图21. \_\_PID\_TTYPE宏定义为\_\_S32\_TYPE**

文本

描述已自动生成

**图22. \_\_S32\_TYPE宏定义为int**