

**数学与信息学院学生实验报告**

**实验课程名称：** 操作系统 **教师：**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **实验项目名称** | 进程控制和线程控制 | | | **实验成绩** |  |
| **学生姓名** |  | **学 号** |  | **年级专业班级** |  |
| **小组成员** |  | | | **实验日期** |  |

一、实验目的

懂得实现进程和线程的创建、运行和撤销。

能够理解并分析与进程、线程相关的系统调用。

二、实验软硬件环境

PC本机win10、VM虚拟机、Ubuntu16.04.3

三、实验内容及要求

实现进程和线程的创建、运行和撤销。

理解并分析与进程、线程相关的系统调用。

四、实验记录

1、进程的相关知识

### 1.1进程的定义及特征

进程是可并发执行的程序在某个数据集合上的一次计算活动，也是操作系统进行资源分配和调度的基本单位（一个正在执行的程序实例）

结构特征：进程由程序块 、数据块、进程控制块三部分组成。

▽动态性：进程的实质是进程实体的执行过程，同时进程有生命周期

▽并发性：多个进程可同存于内存中，能在一段时间内同时执行。

▽独立性：资源分配和调度的基本单位。

▽制约性：并发进程间存在制约关系，造成程序执行速度不可预测性，必须对进程的并发执行次序、相对执行速度加以协调。

▽异步性：进程是按异步方式运行的，即按各自独立、不可预知的速度向前推进。（书36页补充）

### 1.2进程的状态及转换图

1.2.1基本状态

运行态(running)：当进程得到处理机，其执行程序正在处理机上运行时的状态称为运行状态。在单CPU系统中，任何时刻最多只有一个进程处于运行状态。在多CPU系统中，处于运行状态的进程数最多为处理机的数目。

就绪状态(ready)：当一个进程已经准备就绪，一旦得到CPU，就可立即运行，这时进程所处的状态称为就绪状态。系统中有一个就绪进程队列，处于就绪状态进程按某种调度策略存在于该队列中。

等待态（阻塞态）（Wait / Blocked ）：若一个进程正等待着某一事件发生(如等待输入输出操作的完成)而暂时停止执行的状态称为等待状态。处于等待状态的进程不具备运行的条件，即使给它CPU，也无法执行。系统中有几个等待进程队列（按等待的事件组成相应的等待队列）。

1.2.2进程状态转换图

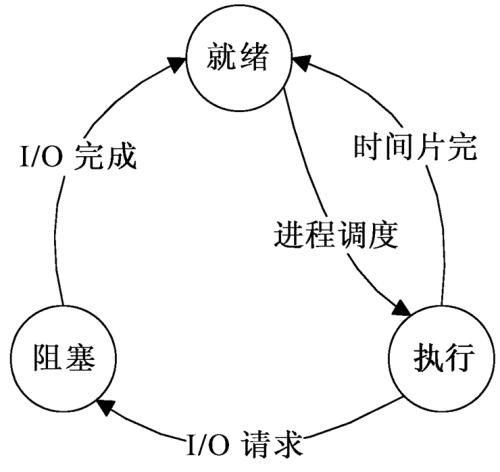


图1-1：三种基本状态的转换

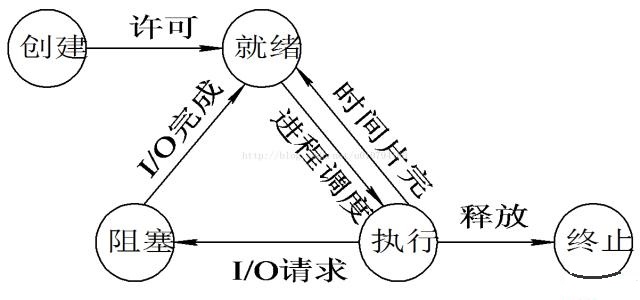


图1-2：进程的五种基本状态及转换

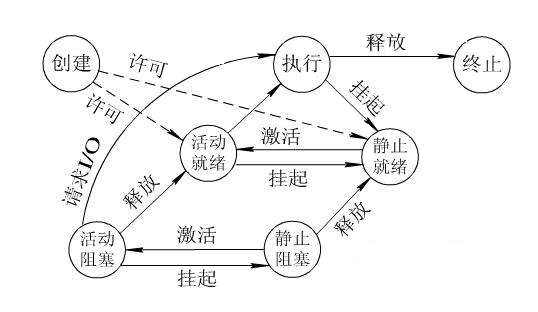


图1-3：具有创建、终止和挂起状态的进程状态图

2、线程的相关知识

### 2.1线程的概念

线程，有时被称为轻量级进程(Lightweight Process，LWP），是程序执行流的最小单元。一个标准的线程由线程ID，当前指令[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88" \t "_blank)(PC），[寄存器](https://baike.baidu.com/item/%E5%AF%84%E5%AD%98%E5%99%A8" \t "_blank)集合和[堆栈](https://baike.baidu.com/item/%E5%A0%86%E6%A0%88)组成。另外，线程是进程中的一个实体，是被系统独立调度和分派的基本单位，线程自己不拥有系统资源，只拥有一点儿在运行中必不可少的资源，但它可与同属一个进程的其它线程共享进程所拥有的全部资源。一个线程可以创建和撤消另一个线程，同一进程中的多个线程之间可以并发执行。由于线程之间的相互制约，致使线程在运行中呈现出间断性。线程也有[就绪](https://baike.baidu.com/item/%E5%B0%B1%E7%BB%AA)、[阻塞](https://baike.baidu.com/item/%E9%98%BB%E5%A1%9E)和[运行](https://baike.baidu.com/item/%E8%BF%90%E8%A1%8C)三种基本状态。就绪状态是指线程具备运行的所有条件，逻辑上可以运行，在等待处理机；运行状态是指线程占有处理机正在运行；阻塞状态是指线程在等待一个事件（如某个信号量），逻辑上不可执行。每一个程序都至少有一个线程，若程序只有一个线程，那就是程序本身。

线程是程序中一个单一的顺序控制流程。进程内一个相对独立的、可调度的执行单元，是系统独立调度和分派CPU的基本单位指[运行](https://baike.baidu.com/item/%E8%BF%90%E8%A1%8C" \t "_blank)中的程序的调度单位。在单个程序中同时运行多个线程完成不同的工作，称为[多线程](https://baike.baidu.com/item/%E5%A4%9A%E7%BA%BF%E7%A8%8B)。

**（来自百度词条的详细解释）**

### 2.2线程的状态及工作原理

执行状态：表示线程已获得处理机而正在运行。

就绪状态：指线程已具备了各种执行条件，只须再获得CPU便可立即执行；

阻塞状态：指线程在执行中因某事件受阻而处于暂停状态。

线程状态之间的转换和进程之间的转换是一样的 如图1-1

线程的工作原理

线程是[进程](https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B)中的实体，一个进程可以拥有多个线程，一个线程必须有一个[父进程](https://baike.baidu.com/item/%E7%88%B6%E8%BF%9B%E7%A8%8B)。线程不拥有[系统资源](https://baike.baidu.com/item/%E7%B3%BB%E7%BB%9F%E8%B5%84%E6%BA%90)，只有[运行](https://baike.baidu.com/item/%E8%BF%90%E8%A1%8C)必须的一些[数据结构](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84)；它与父进程的其它线程共享该进程所拥有的全部资源。线程可以创建和[撤消](https://baike.baidu.com/item/%E6%92%A4%E6%B6%88)线程，从而实现程序的并发执行。一般，线程具有[就绪](https://baike.baidu.com/item/%E5%B0%B1%E7%BB%AA)、[阻塞](https://baike.baidu.com/item/%E9%98%BB%E5%A1%9E)和[运行](https://baike.baidu.com/item/%E8%BF%90%E8%A1%8C)三种基本状态。

在多[中央处理器](https://baike.baidu.com/item/%E4%B8%AD%E5%A4%AE%E5%A4%84%E7%90%86%E5%99%A8)的[系统](https://baike.baidu.com/item/%E7%B3%BB%E7%BB%9F)里，不同线程可以同时在不同的中央处理器上[运行](https://baike.baidu.com/item/%E8%BF%90%E8%A1%8C)，甚至当它们属于同一个[进程](https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B)时也是如此。大多数支持多处理器的[操作系统](https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F)都提供[编程接口](https://baike.baidu.com/item/%E7%BC%96%E7%A8%8B%E6%8E%A5%E5%8F%A3)来让[进程](https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B)可以控制自己的线程与各处理器之间的关联度。

有时候，线程也称作[轻量级进程](https://baike.baidu.com/item/%E8%BD%BB%E9%87%8F%E7%BA%A7%E8%BF%9B%E7%A8%8B)。就象[进程](https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B)一样，线程在程序中是独立的、并发的执行路径，每个线程有它自己的[堆栈](https://baike.baidu.com/item/%E5%A0%86%E6%A0%88)、自己的[程序计数器](https://baike.baidu.com/item/%E7%A8%8B%E5%BA%8F%E8%AE%A1%E6%95%B0%E5%99%A8)和自己的[局部变量](https://baike.baidu.com/item/%E5%B1%80%E9%83%A8%E5%8F%98%E9%87%8F)。但是，与分隔的[进程](https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B)相比，进程中的线程之间的隔离程度要小。它们[共享内存](https://baike.baidu.com/item/%E5%85%B1%E4%BA%AB%E5%86%85%E5%AD%98)、[文件](https://baike.baidu.com/item/%E6%96%87%E4%BB%B6)句柄和其它每个[进程](https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B)应有的状态。

[进程](https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B)可以支持多个线程，它们看似同时执行，但互相之间并不同步。一个进程中的多个线程共享相同的内存地址空间，这就意味着它们可以访问相同的[变量](https://baike.baidu.com/item/%E5%8F%98%E9%87%8F)和[对象](https://baike.baidu.com/item/%E5%AF%B9%E8%B1%A1)，而且它们从同一堆中分配对象。尽管这让线程之间共享信息变得更容易，但您必须小心，确保它们不会妨碍同一[进程](https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B)里的其它线程。

### 2.3线程的三种类型

**内核线程：**它的创建和撤消是由内核的内部需求来决定的，用来负责执行一个指定的函数，一个内核线程不需要和一个用户进程联系起来。它共享内核的正文段核全局数据，具有自己的内核堆栈。它能够单独的被调度并且使用标准的内核同步机制，可以被单独的分配到一个处理器上运行。内核线程的调度由于不需要经过态的转换并进行地址空间的重新映射，因此在内核线程间做上下文切换比在进程间做上下文切换快得多。

**轻量级进程：**轻量级进程是核心支持的用户线程，它在一个单独的进程中提供多线程控制。这些轻量级进程被单独的调度，可以在多个处理器上运行，每一个轻量级进程都被绑定在一个内核线程上，而且在它的生命周期这种绑定都是有效的。轻量级进程被独立调度并且共享地址空间和进程中的其它资源，但是每个LWP都应该有自己的程序计数器、寄存器集合、核心栈和用户栈。

**用户线程：**用户线程是通过线程库实现的。它们可以在没有内核参与下创建、释放和管理。线程库提供了同步和调度的方法。这样进程可以使用大量的线程而不消耗内核资源，而且省去大量的系统开销。用户线程的实现是可能的，因为用户线程的上下文可以在没有内核干预的情况下保存和恢复。每个用户线程都可以有自己的用户堆栈，一块用来保存用户级寄存器上下文以及如信号屏蔽等状态信息的内存区。库通过保存当前线程的堆栈和寄存器内容载入新调度线程的那些内容来实现用户线程之间的调度和上下文切换。

内核仍然负责进程的切换，因为只有内核具有修改内存管理寄存器的权力。用户线程不是真正的调度实体，**内核对它们一无所知，而只是调度用户线程下的进程或者轻量级进程，这些进程再通过线程库函数来调度它们的线程。当一个进程被抢占时，它的所有用户线程都被抢占，当一个用户线程被阻塞时，它会阻塞下面的轻量级进程，如果进程只有一个轻量级进程，则它的所有用户线程都会被阻塞。**

3、进程的创建及运行

### 3.1进程的创建概念引入

3.1.1进程的层次结构

在OS中，允许一个进程创建另一个进程，通常把进程的进程称为父进程，而把被创建的进程称为子进程。子进程可以创建更多的孙进程，由此便形成了一个进程的层次结构。

3.1.2引起创建进程的事件

（为使程序之间能并发进行，应先为它们分别创建进程。）导致一个进程去创建另一个进程的典型事件有四类：用户登录、作业调度、提供服务、应用请求。

3.1.3 在系统中每当出现了闯进新进程的请求后，OS便调用进程创建原语Create按以下步骤创建一个新进程：

1. 申请空白PCB，为新进程申请获得唯一的数字标识符，并从PCB集合中索取一个空白PCB。
2. 为新进程分配其运行所的资源，包括各种物理和逻辑资源，如内存、文件、I/O设备和CPU时间等。
3. 初始化进程控制块（PCB）
4. 如果进程就绪队列能够接纳新进程，便将新进程插入就绪队列。

### 3.2以fork函数创建进程为例

调用fork创建的子进程，将共享父进程的代码空间，复制父进程数据空间，此时子进程会获得父进程的所有变量的一份拷贝。

系统调用fork（），在fork函数调用完，如果创建成功就会返回两个进程，一个是father进程和一个新的进程son1（fork的一次调用两次返回）。

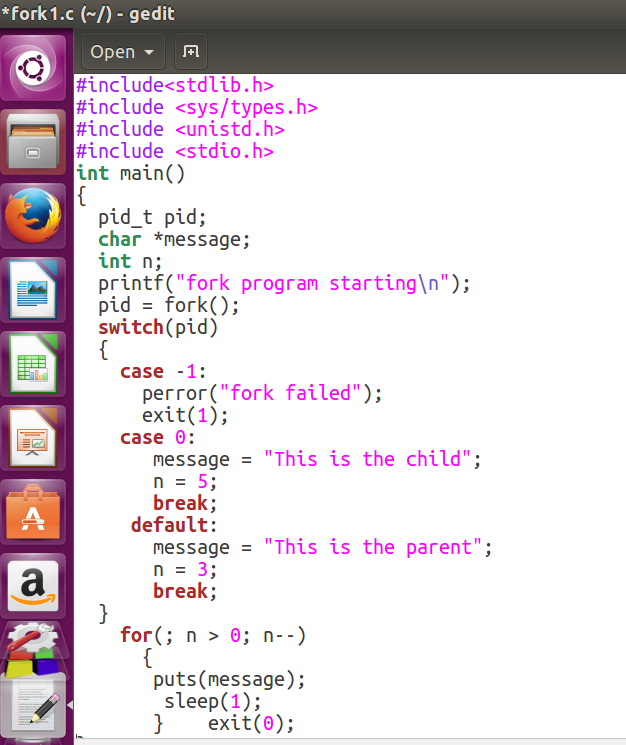
fork()返回值意义如下：

=0：在子进程中，表示当前进程是子进程。

>0：在父进程中，返回值为子进程的id值（唯一标识号）。

-1：创建失败。

**3.2.1 例子1：**



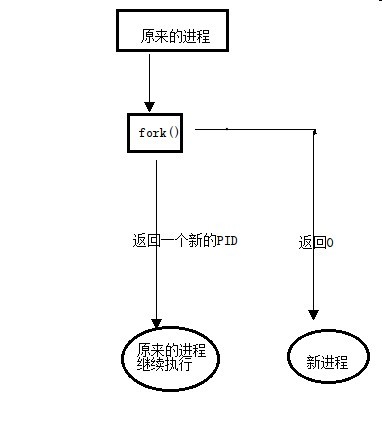
(1)pid\_t是进程号，是唯一表示进程的ID。

(2) pid\_t fork(void) 函数

包含的头文件：#include <sys/types.h> #include <unistd.h>

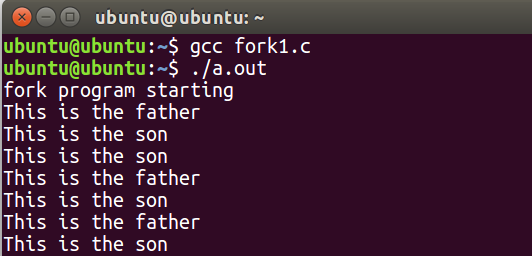
调用fork可以创建一个全新的进程。这个系统调用对当前进程进行复制。在进程表里创建一个新的项目，新项目的许多属性与当前进程是相同的。新进程和原进程几乎一模一样，执行的也是相同的代码，但新进程有自己的数据空间、自己的环境等。

(3)程序调用了fork函数的时候被分成了两个进程。在父进程里，fork函数返回新进程的PID进程号，新进程则返回0，这个可以做为区分父子进程的依据。



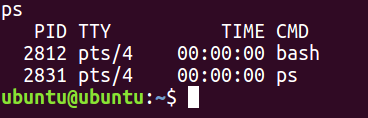
父进程和子进程的执行的代码都和fork1.c里的代码一致。但是，fork根据不同进程返回不同的PID，那么父子进程的实际有效代码部分是不同的。就是说，进程会根据PID的不同，有选择的执行各自的代码。

这个程序将产生两个进程，新进程（子进程）会输出消息5次，而父进程之输出3次。父进程会在子进程打印完它的全部消息之前退出。运行一下这个程序，我们可以看到如下交替输出的消息：



源文件虽然只有一个，但是运行的进程是2个，而不是1个。fork函数在父进程里创建了新的进程，两个进程执行的是相同的代码，即代码段是一样的。

使用ps命令输出当前运行的进程：



2812和2831是两个不同的进程，和上面说的一样。

**3.2.2 例子2：**

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

int main()

{

int count = 1;

int pid1;

printf("Recently pid is %d \n", getpid());

pid1 = fork();

if (pid1 < 0)

{

printf("error in fork!");

}

else if (pid1==0)

{

count++;

printf("My father's pid is %d I'm son1，my pid is %d B\n", getppid(), getpid());

}

else

{

int pid2 = fork();

if (pid2 < 0)

{

printf("error in fork!");

}

if (pid2==0)

{

count++;

printf("My father's pid is %d I'm son2，my pid is %d C \n", getppid(), getpid());

}

else

{

count++;

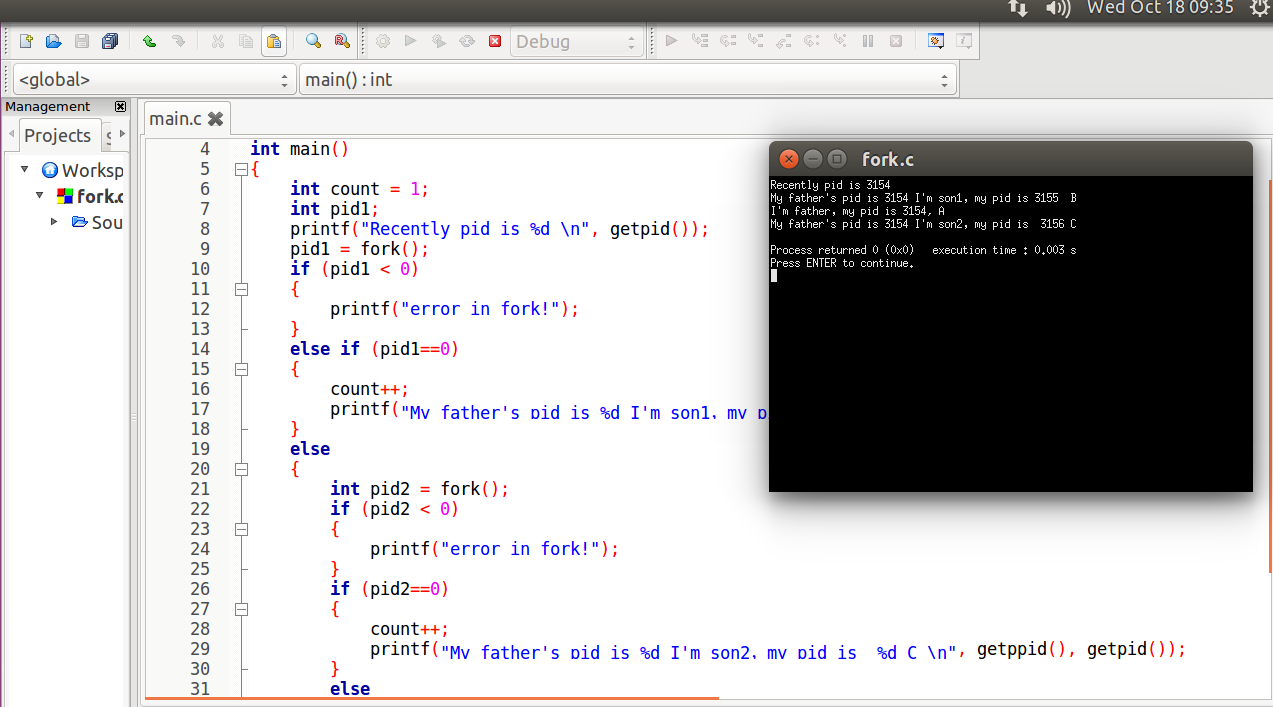
printf("I'm father，my pid is %d, A\n", getpid());

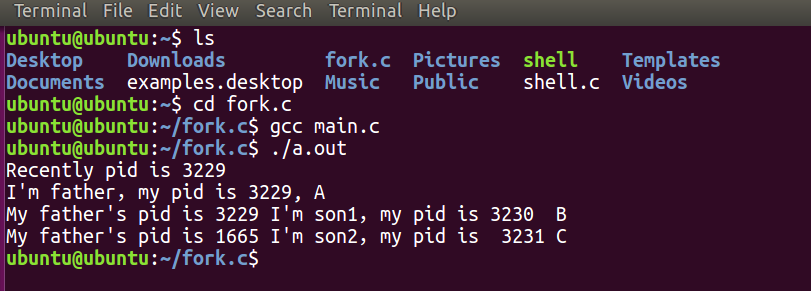
}

}

return 0;

}





**程序实现的是：使用系统调用fork( )创建两个子进程，这样在此程序运行时，在系统中就有一个父进程和两个子进程在活动。每一个进程在屏幕上显示一个字符，其中父进程显示字符A，子进程分别显示字符B和字符C。**

孤儿进程：父进程在子进程结束之前死亡（return或者exit）。在一定时间内，当系统发现孤儿进程时，init进程就收养孤儿进程，成为它的father，child进程exit后资源回收都又init进程完成。

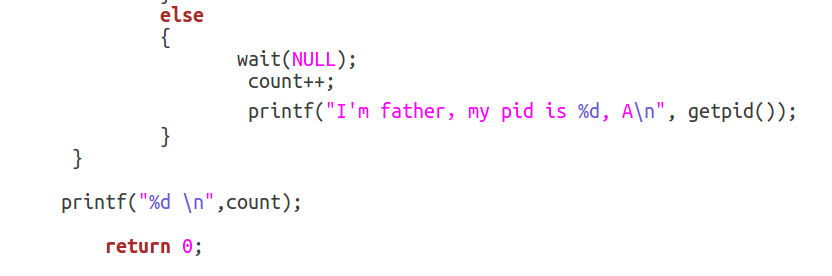
僵死进程：子进程在父进程之前结束了，但是父进程没有用wait或waitpid回收子进程

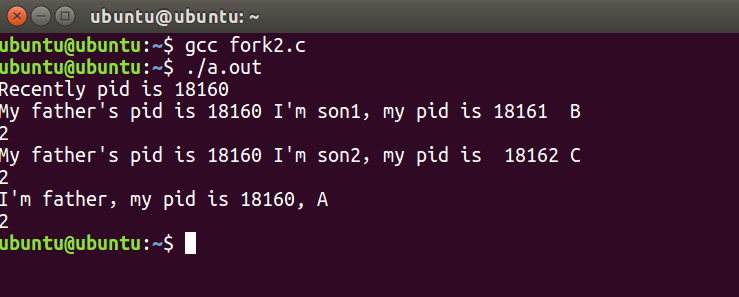
三个进程都是在return后死亡，而他们的死亡顺序是又操作系统控制。所以父进程在son2结束 死亡。son2成了孤儿进程，被init进程回收。在大多数linux上面init进程的pid为1，但是据说ubuntu为了减轻pid为1进程的压力，将其内核修改过。我init进程的pid为1965。

wait挂起调用它的进程，直到它的任一子进程退出或者收到一个不能忽略的信号为止，如果在父进程执行wait调用前就已经有子进程退出，则立刻返回。函数成功调用后返回结束运行子进程的ID，否则返回-1。进程死掉以后，会留下一具僵尸（一些数据），wait充当了“殓尸工”，把僵尸推去火化，使其最终归于无形。

程序消息的输出是父子进程交替输出，且父进程在子进程之前结束。如果要安排父进程在子进程结束之后才结束。可以调用wait函数。

这里修改下代码，在父进程里面调用wait（）函数收回子进程资源避免僵死进程。





在实验中，我还设置了一个数count，是为了实验fork创建的子进程，每个进程是父进程的一份复制，但是又有自己的空间，每个进程都有自己的变量。

son1创建时，会复制变量count=0，但是count是存在son1的数据段中，所以son1的count和father以及son2拥有不同的地址，每个count是独立的，不是共用的。

(这个程序目前存在，有两个子进程但是只**wait（）**了一次的问题，如果有一个子进程返回，wait调用就结束。父进程可能不会回收第二个返回的子进程，所以还是存在僵死进程的问题。可以使用**waitpid（）**函数来指定回收子进程）

**注：博客Jason的专栏** [linux中fork（）函数详解（原创！！实例讲解）](http://blog.csdn.net/jason314/article/details/5640969)

这里关于fork创建进程由简单到进阶讲述的比较直白可作为理解参考。

### 3.3创建进程的三种方法相关介绍

在Linux中主要提供了fork、vfork、clone三个进程创建方法。   
在linux源码中这三个调用的执行过程是执行fork(),vfork(),clone()时，通过一个系统调用表映射到sys\_fork(),sys\_vfork(),sys\_clone(),再在这三个函数中去调用do\_fork()去做具体的创建进程工作。   
  
　　一、fork   
　　1. 调用方法   
　 　#include <sys/types.h>   
    #include <unistd.h>   
    pid\_t fork(void);   
   正确返回：在父进程中返回子进程的进程号，在子进程中返回0   
　　错误返回：-1   
　　2. fork函数调用的用途   
　　一个进程希望复制自身，从而父子进程能同时执行不同段的代码。   
  
　　二、vfork   
　　1. 调用方法   
　　与fork函数完全相同   
　　#include <sys/types.h>   
　　#include <unistd.h>   
　 　pid\_t vfork(void);   
　　正确返回：在父进程中返回子进程的进程号，在子进程中返回0   
　　错误返回：-1   
　　2. vfork函数调用的用途   
　　用vfork创建的进程主要目的是用exec函数执行另外的程序。   
  
　　三、clone   
　　1.调用方法   
　　#include <sched.h>   
　　int clone(int (\*fn)(void \*), void \*child\_stack, int flags, void \*arg);   
　　正确返回：返回所创建进程的PID，函数中的flags标志用于设置创建子进程时 的相关选项，   
　　错误返回：-１   
　　2.clone()函数调用的用途   
　　用于有选择地设置父子进程之间需共享的资源   
  
　　四、fork，vfork，clone的区别   
　　1. fork出来的子进程是父进程的一个拷贝，即，子进程从父进程得到了数据段和堆栈段的拷贝，这些需要分配新的内存；而对于只读的代码段，通常使用共享内存的方式访问；而vfork则是子进程与父进程共享内存空间, 子进程对虚拟地址空间任何数据的修改同样为父进程所见；clone则由用户通过参clone\_flags 的设置来决定哪些资源共享，哪些资源拷贝。   
　　2. fork不对父子进程的执行次序进行任何限制，fork返回后，子进程和父进程都从调用fork函数的下一条语句开始行，但父子进程运行顺序是不定的，它取决于内核的调度算法；而在vfork调用中，子进程先运行，父进程挂起，直到子进程调用了exec或exit之后，父子进程的执行次序才不再有限制；clone中由标志CLONE\_VFORK来决定子进程在执行时父进程是阻塞还是运行，若没有设置该标志，则父子进程同时运行，设置了该标志，则父进程挂起，直到子进程结束为止。

系统调用fork()和vfork()是无参数的，而clone()则带有参数。fork()是全部复制，vfork()是共享内存，而clone() 是则可以将父进程资源有选择地复制给子进程，而没有复制的数据结构则通过指针的复制让子进程共享，具体要复制哪些资源给子进程，由参数列表中的 clone\_flags来决定。另外，clone()返回的是子进程的pid。

总结部分选自http://blog.csdn.net/lyc417356935/article/details/45021345

4、进程的撤销

### 4.1 release\_task()函数的处理过程

进程终止运行后，一般会处于僵死状态，需要由父进程来执行wait操作来回收进程的进程描述符及内核栈所占内存，同时把僵死进程从进程相关的各个表上摘除下来。对应的处理函数是release\_task(),该函数的处理过程是：

1递减进程拥有者进程的个数， atomic\_dec(&p->user->processes);

2如果进程被跟踪，把它从调试程序的ptrace\_children链表中删除\_ ptrace\_unlink(p);

3调用\_\_exit\_signal()删除所有的挂起信号并释放signal\_struct描述符，若没有其它轻量级进程使用该signal\_struct的话，会删除该数据结构: \_\_exit\_signal(p);

4调用\_\_exit\_sighand()删除信号处理函数： \_\_exit\_sighand(p);

5调用\_\_unhash\_process()，该函数依次执行下列操作：

a变量nr\_threads减1.

b两次调用detach\_pid(),分别从PIDTYPE\_PID和PIDTYPE\_TGID类型的PID散列表中删除进程描述符。

c如果进程是线程组的领头进程，那么调用两次detach\_pid()从PIDTYPE\_PGID, PIDTYPE\_SID类型的散列表中删除该进程描述符。

d用RMOVE\_LINKS从进程链表中删除该进程。

6如果进程不是线程组的领头进程，领头进程处于僵死状态，且该进程时线程组中的最后一个成员，该进程向领头进程的父进程发出一个信号，通知该进程已经死亡。

7调用sched\_exit()函数来调整父进程的时间片。

8调用put\_task\_struct()递减进程描述符的引用计数，当计数器变为0时，则函数终止所有残留的对进程的引用.

a递减进程所有者的user\_struct 数据结构的使用计数，如果该引用计数变为0，释放该结构。

b释放进程描述符以及thread\_info描述符和内核态堆栈。

### 4.2Linux进程终止的5种形式

1、main函数的自然返回，return   
2、调用exit函数   
3、调用\_exit函数   
4、调用abort函数   
5、接受能导致进程终止的信号:

ctrl+c (^C)

SIGINT(SIGINT中断进程)

SIGQUIT （进程在因收到SIGQUIT退出时会产生core文件, 在这个意义上类似于一个程序错误信号）

ctrl + \ (^\Quit)

其中1、2、3为正常终止，4、5异常终止

exit和\_exit函数都是用来终止进程的。当程序执行到exit和\_exit时，进程会**无条件的停止剩下的所有操作**，清除包括PCB在内的各种数据结构，并终止本程序的运行。

exit函数和\_exit函数的**最大区别在于exit函数在退出之前会检查文件的打开情况，把文件缓冲区中的内容写回文件**，也就是清理I/O缓冲。

exit可输出缓冲区数据

#include <stdio.h>

#include <stdlib.h>

int main()

{

printf("Using exit\n");

printf("This is the content in buffer");

exit(0);

}

//运行结果：

Using exit

This is the content in buffer

**\_exit无法输出缓冲区数据**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main()

{

printf("Using exit\n");

printf("This is the content in buffer");

\_exit(0);

}

//运行结果：

Using exit

abort()是**使异常程序终止**，同时发送SIGABRT信号给调用进程。   
使用实例：

#include <stdio.h>

#include <stdlib.h>

int main()

{

FILE \*stream;

if((stream = fopen("nofilehere", "r")) == NULL)

{

perror("Can not open");

abort();

}

else

{

fclose(stream);

}

return 0;

运行结果：

Can not open: No such file or directory

Aborted (core dumped)

5、线程的创建及运行

Linux的线程实现是在核外进行的，核内提供的是创建进程的接口do\_fork()。内核提供了两个系统调用\_\_clone()和 fork()，最终都用不同的参数调用do\_fork()核内API。当然，要想实现线程，没有核心对多进程（其实是轻量级进程）共享数据段的支持是不行 的，因此，do\_fork()提供了很多参数，包括CLONE\_VM（共享内存空间）、CLONE\_FS（共享文件系统信息）、 CLONE\_FILES（共享文件描述符表）、CLONE\_SIGHAND（共享信号句柄表）和CLONE\_PID（共享进程ID，仅对核内进程，即0号 进程有效）。当使用fork系统调用时，内核调用do\_fork()不使用任何共享属性，进程拥有独立的运行环境，而使用 pthread\_create()来创建线程时,则最终设置了所有这些属性来调用\_\_clone()，而这些参数又全部传给核内的do\_fork()，从 而创建的"进程"拥有共享的运行环境，只有栈是独立的，由\_\_clone()传入。

Linux线程在核内是以轻量级进程的形式存在的，拥有独立的进程表项，而所有的创建、同步、删除等操作都在核外pthread库中进行。pthread 库使用一个管理线程（\_\_pthread\_manager()，每个进程独立且唯一）来管理线程的创建和终止，为线程分配线程ID，发送线程相关的信号 （比如Cancel），而主线程（pthread\_create()）的调用者则通过管道将请求信息传给管理线程。

### 5.1pthread的实现

Linux系统下的多线程遵循POSIX线程接口，称为pthread。编写Linux下的多线程程序，需要使用头文件pthread.h，连接时需要使用库libpthread.a（也可以直接选择终端命令行调用就不需要在编译器选择libpthread的环境）顺便说一下，Linux下pthread的实现是通过系统调用clone（）来实现的。clone（）是Linux所特有的系统调用，它的使用方式类似fork。

#include <stdio.h>

#include <pthread.h>

#include <stdlib.h>

void thread(void)

{

int i;

for(i=0;i<3;i++)

printf("This is a pthread.\n");

}

int main(void)

{

pthread\_t id;

int i;

int ret;

ret =pthread\_create(&id,NULL,(void \*) thread,NULL);

if(ret!=0){

printf ("Create pthread error!\n");

exit (1);

}

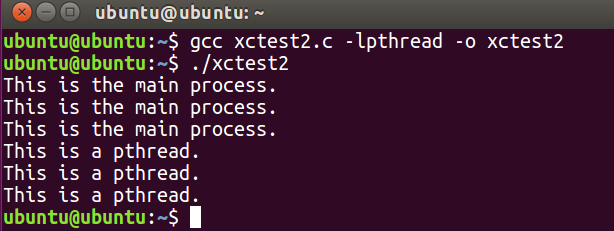
for(i=0;i<3;i++)

printf("This is the main process.\n");

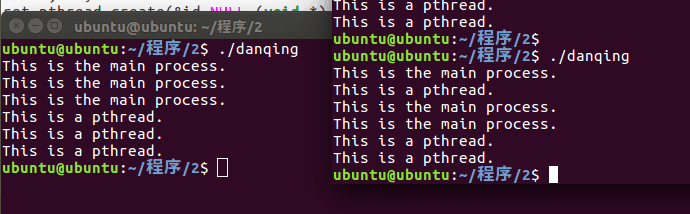
pthread\_join(id,NULL);

return (0);

}



再次运行，有时会得到以下结果



前后两次结果不一样，这是两个线程争夺CPU资源的结果

上面的示例中，我使用到了两个函pthread\_create和pthread\_join并声明了一个pthread\_t型的变量。  
　　pthread\_t在头文件/usr/include/bits/pthreadtypes.h中定义：  
　　typedef unsigned long int pthread\_t;  
　　它是一个线程的标识符。函数pthread\_create用来创建一个线程，它的原型为：  
　　extern int pthread\_create \_\_P ((pthread\_t \*\_\_thread, \_\_const pthread\_attr\_t \*\_\_attr,  
　　void \*(\*\_\_start\_routine) (void \*), void \*\_\_arg));  
　　第一个参数为指向线程标识符的指针，第二个参数用来设置线程属性，第三个参数是线程运行函数的起始地址，最后一个参数是运行函数的参数。这里函数thread不需要参数，所以最后一个参数设为空指针。第二个参数我也设为空指针，这样将生成默认属性的线程。

第一个参数为被等待的线程标识符，第二个参数为一个用户定义的指针，它可以用来存储被等待线程的返回值。这个函数是一个线程阻塞的函数，调用它的函数将一直等待到被等待的线程结束为止，当函数返回时，被等待线程的资源被收回。一个线程的结束有两种途径，一种是象我们上面的例子一样，函数结束了，调用它的线程也就结束了；另一种方式是通过函数pthread\_exit来实现。它的函数原型为：  
　　extern void pthread\_exit \_\_P ((void \*\_\_retval)) \_\_attribute\_\_ ((\_\_noreturn\_\_));  
　　唯一的参数是函数的返回代码，只要pthread\_join中的第二个参数thread\_return不是NULL，这个值将被传递给thread\_return。最后要说明的是，一个线程不能被多个线程等待，否则第一个接收到信号的线程成功返回，其余调用pthread\_join的线程则返回错误代码ESRCH。

### 5.2等待线程

由于一个进程中的多个线程是共享数据段的，通常在线程退出之后，退出线程所占用的资源并不会随着线程的终止而得到释放pthread\_join()函数是类似进程的wait()/waitpid()函数，用于将当前线程挂起来等待线程的结束。是一个线程阻塞的函数，调用它的线程一直等待到被等待的线程结束为止，函数返回时，被等待线程的资源就被收回

代码：

#include<stdio.h>

#include<sys/types.h>

#include<unistd.h>

#include<pthread.h>

pthread\_t pth1\_id,pth2\_id;

void \*fun1(void \*arg)

{

int i=3;

while(i--)

{

printf("pthread1 is running \n");

sleep(1);

}

pthread\_exit((void\*)1);

//return 0;

}

void \*fun2(void\*arg)

{

int i=5;

while(i--)

{

printf("pthread2 is running \n");

sleep(1);

}

pthread\_exit((void \*)2);

}

int main()

{

void \*tmp;

if(pthread\_create(&pth1\_id,NULL,fun1,NULL)!=0)

{

printf("Create thread1 error ! \n");

}

if(pthread\_create(&pth2\_id,NULL,fun2,NULL)!=0)

{

printf("Create thread2 error ! \n");

}

printf("waitting pthread1 end....\n");

if(pthread\_join(pth1\_id,&tmp)!=0)

{

printf("joining1 error! \n");

}

printf("pthread1 exit ret is %d\n",(int \*)tmp);

printf("waitting pthread2 end....\n");

if(pthread\_join(pth2\_id,&tmp)!=0)

{

printf("joining2 error! \n");

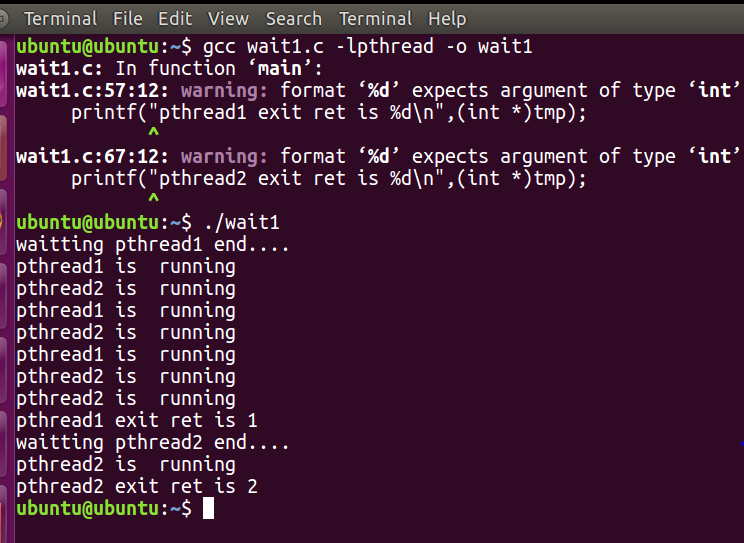
}

printf("pthread2 exit ret is %d\n",(int \*)tmp);

sleep(1);

return 0;

}



从运行结果可以看出pthread\_join 是一个阻塞函数。

### 5.3****Linux中线程与进程关系****

**进程和线程的关系：**

（1）一个线程只能属于一个进程，而一个进程可以有多个线程，但至少有一个线程。

（2）资源分配给进程，同一进程的所有线程共享该进程的所有资源。

（3）处理机分给线程，即真正在处理机上运行的是线程。

（4）线程在执行过程中，需要协作同步。不同进程的线程间要利用消息通信的办法实现同步。

线程是指进程内的一个执行单元,也是进程内的可调度实体.

**与进程的区别:**

(1)调度：线程作为调度和分配的基本单位，进程作为拥有资源的基本单位

(2)并发性：不仅进程之间可以并发执行，同一个进程的多个线程之间也可并发执行

(3)拥有资源：进程是拥有资源的一个独立单位，线程不拥有系统资源，但可以访问隶属于进程的资源.

(4)系统开销：在创建或撤消进程时，由于系统都要为之分配和回收资源，导致系统的开销明显大于创建或撤消线程时的开销。

    在Linux系统中，进程虽然定义为程序的执行实例, 它并不执行什么, 只是维护应用程序所需的各种资源，而线程则是真正的执行实体。为了让进程完成一定的工作, 进程必须至少包含一个线程。进程所维护的是程序所包含的资源(静态资源)，比如：虚拟地址空间、打开的文件描述符集合、文件系统状态和信号处理程序等；线程所维护的运行相关的资源(动态资源)，比如：运行栈、调度相关的控制信息、待处理的信号集等。在Linux内核中并没有线程的概念，每一个执行实体都是一个task\_struct结构, 通常称之为进程。进程是一个执行单元，维护着执行相关的动态资源。同时它又引用着程序所需的静态资源。通过系统调用clone创建子进程时，可以有选择性地让子进程共享父进程所引用的资源。这样的子进程通常称为轻量级进程。linux上的线程就是基于轻量级进程，由用户态的pthread库实现的。

    使用pthread时, 在用户看来, 每一个task\_struct就对应一个线程, 而一组线程以及它们所共同引用的一组资源就是一个进程，但是一组线程并不仅仅是引用同一组资源就够了，它们还必须被视为一个整体，即所谓的线程组。

    总之，一个进程里面可以有多个线程，这些线程由内核自动调度，并且每个线程有它自己的线程上下文（thread context），包括线程ID、栈、栈指针、程序计数器、通用目的寄存器和条件码。而进程中其他资源是所有的线程共享的，包括虚拟地址空间（代码、数据、堆、共享库）、文件系统信息、文件描述符表和信号处理程序。

**选自：** 浅析Linux线程的创建 - MaximusZhou的专栏 - CSDN博客 <http://blog.csdn.net/maximuszhou/article/details/41449847>

6、线程的撤销

### 6.1线程撤销的相关知识

线程取消的定义：一般情况下，线程在其主体函数退出的时候会自动终止，但同时也可以因为接收到另一个线程发来的终止（取消）请求而强制终止。

线程取消的语义：线程取消的方法是向目标线程发Cancel信号，但如何处理Cancel信号则由目标线程自己决定，或者忽略、或者立即终止、或者继续运行至Cancelation-point（取消点），由不同的Cancelation状态决定。

线程接收到CANCEL信号的缺省处理（即pthread\_create()创建线程的缺省状态）是继续运行至取消点，也就是说设置一个CANCELED状态，线程继续运行，只有运行至Cancelation-point的时候才会退出。

取消点：根 据POSIX标准，pthread\_join()、pthread\_testcancel()、pthread\_cond\_wait()、 pthread\_cond\_timedwait()、sem\_wait()、sigwait()等函数以及read()、write()等会引起阻塞的系 统调用都是Cancelation-point，而其他pthread函数都不会引起Cancelation动作。但是pthread\_cancel的手 册页声称，由于LinuxThread库与C库结合得不好，因而目前C库函数都不是Cancelation-point；但CANCEL信号会使线程从阻 塞的系统调用中退出，并置EINTR错误码，因此可以在需要作为Cancelation-point的系统调用前后调用 pthread\_testcancel()，从而达到POSIX标准所要求的目标。代码段如下:

pthread\_testcancel();

retcode = read(fd, buffer, length);

pthread\_testcancel();

任一线程调用了exit函数，整个进程就会终止。

信号默认动作是终止进程，那么信号发送到该进程，整个进程也会被终止。

单个线程通过以下三种方式退出

（1）线程只是从启动函数中返回，返回值是线程的退出码

（2）线程可以被同一进程中的其他线程取消。

（3）线程调用pthread\_exit。

void pthread\_exit(void \* arg);

arg是个无类型指针，该指针会被其他线程调用pthread\_join捕捉。在线程的子函数中调用pthread\_exit()函数，线程也会退出，这点跟exit()函数相同

线程之间是异步的，无法确定哪个线程先执行。

进程内的信号捕捉一般在控制线程内进行。

### 6.2线程撤销实例分析

2．5 与线程取消相关的pthread函数

int pthread\_cancel(pthread\_t thread)   
发送终止信号给thread线程，如果成功则返回0，否则为非0值。发送成功并不意味着thread会终止。

int pthread\_setcancelstate(int state, int \*oldstate)   
设置本线程对Cancel信号的反应，state有两种值：PTHREAD\_CANCEL\_ENABLE（缺省）和 PTHREAD\_CANCEL\_DISABLE，分别表示收到信号后设为CANCLED状态和忽略CANCEL信号继续运行；old\_state如果不为 NULL则存入原来的Cancel状态以便恢复。

int pthread\_setcanceltype(int type, int \*oldtype)   
设置本线程取消动作的执行时机，type由两种取值：PTHREAD\_CANCEL\_DEFFERED和 PTHREAD\_CANCEL\_ASYCHRONOUS，仅当Cancel状态为Enable时有效，分别表示收到信号后继续运行至下一个取消点再退出和 立即执行取消动作（退出）；oldtype如果不为NULL则存入运来的取消动作类型值。

void pthread\_testcancel(void)   
检查本线程是否处于Canceld状态，如果是，则进行取消动作，否则直接返回。

   一个实例的分析：

#include<iostream>

#include<unistd.h>

#include<pthread.h>

using namespace std;

pthread\_t pid[3];

void\* thread\_run\_1(void\* arg){

cout<<"Now in the thread 1"<<endl;

int sum = 0;

int state, oldstate;

state = PTHREAD\_CANCEL\_DEFERRED;

pthread\_setcancelstate(state, &oldstate);

cout<<"oldstate is "<<(state == oldstate? "Deferred":"Async")<<endl;

//耗时间的循环

for (int i = 1; i<=5; ++i)

cout<<"before testcancel"<<endl;

pthread\_testcancel();

cout<<"after testcancel"<<endl;

cout<<"thread 1 done!"<<endl;

}

void\* thread\_run\_2(void\* arg){

cout<<"Now in the thread 2"<<endl;

pthread\_cancel(pid[1]);

sleep(2);

cout<<"thread 2 done!"<<endl;

}

int main(){

pid[0] = pthread\_self();

if (pthread\_create(&pid[1],NULL,thread\_run\_1,NULL) != 0){

cout<<"error create thread 1"<<endl;

return -1;

}

if (pthread\_create(&pid[2],NULL,thread\_run\_2,NULL) != 0){

cout<<"error create thread 2"<<endl;

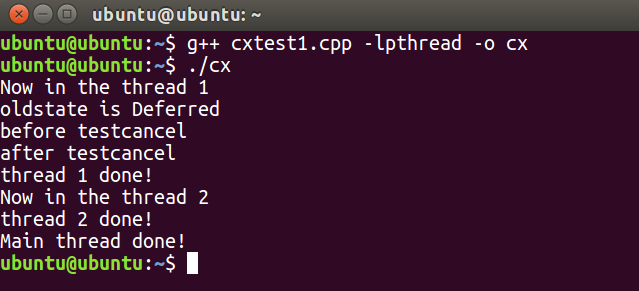
return -1;

}

sleep(5);

cout<<"Main thread done!"<<endl;

}

在这个程序中，我们在main线程中生成两个线程:thread1和thread2，并使用一个全局数组pid来保存线程的id。在这个程序中，我们在thread2中执行取消其他线程的操作。  
  
运行结果如下：  
  
thread1后面几个输出都没有进行，可以看到是确实的被thread2取消了。

但是很明显，thread1并没有运行到pthread\_testcancel所指示的地方

如果我们把main thread中sleep的时间增加，就可以使得程序运行到thread1的pthread\_testcancel()处。如果thread1确实被取消 了，则“after testcancel”不会被输出。为了在main thread中确保thread1返回，我们使用pthread\_join函数来阻塞整个程序直到thread1返回，这样，main函数的结束标志着thread1必然已经结束。

7、进程、线程相关的系统调用

  进程函数和线程函数的相关性：

|  |  |  |
| --- | --- | --- |
| **Process Primitive** | **Thread Primitive** | **Description** |
| fork | pthread\_create | 创建新的控制流 |
| exit | pthread\_exit | 退出已有的控制流 |
| waitpid | pthread\_join | 等待控制流并获得结束代码 |
| atexit | pthread\_cleanup\_push | 注册在控制流退出时候被调用的函数 |
| getpid | pthread\_self | 获得控制流的id |
| abort | pthread\_cancel | 请求非正常退出 |

### 7.1Linux进程控制函数

**1.fork**

在linux系统中，用户创建进程的唯一方法就是使用系统调用fork，大概要进行下面的操作。

<1>.分配表项，一个用户的进程项是有限的；

<2>.创建子进程的进程标识号；

<3>.复制父进程中的项目给子进程；

<4>.与父进程相连的文件表和索引值加1，与子进程相连；

<5>.内核为子进程创建用户级上下文；

<6>.生成进程的动态部分。

#include <sys/types.h>

#include <unistd.h>

pid\_t fork(void);

return:调用成功，对父进程返回子进程的PID，对子进程返回-1；调用失败，返回-1至父进程。

fork函数最重要的就是返回值，非常特殊，返回两个值，分别给父进程和子进程，在fork创建进程时起到至关重要的地方。

**2.exec**

这个函数最大的作用在于可以取代调用进程的内容，也是linux可以执行新程序的关键。当fork完后调用exec便会进行新进程的进行，这也是为何fork复制出来的子进程执行之后与父进程不一样的根本原因所在。这里还有个“copy-on-write”的技术，提高进程的效率的。

exec系统调用，是一个函数族。

#include<unistd.h>

int execl(const char \*path,const char \*arg);

int execlp(const char \*file,const char \*arg);

int execle(const char \*path,const char \*arg,char \*const envp[]);

int execv(const char \*path,const char \*argv[]);

int execvp(const char \*file,const char \*argv[]);

int execve(const char \*path,const char \*argv[],char \*const envp[]); //真正意义上的调用函数

除了execlp,execvp之外，其他四个函数必须是一个完整的path

envp存放的是环境变量，何为环境变量？简单来说，就是一组用来确定应用程序在系统中一些细节的值，一般以“XXX=xxx”形式存在。还有一点要注意的是凡是数组都是以NULL结尾。

**3.exit**

简单一点说，exit函数就是进程结束函数，就是一个进程结束的标识。

这里还要提到一个\_exit函数，主要区别在于在linux中的库函数原型不一样，exit()定义在stdlib.h中，\_exit定义在unistd.h中。

这里用一句话说明区别，它们都是写入内存，但\_exit只有在满足特点条件下才能写入文件，如何要保存数据的完整性，一般使用exit函数。

**4.wait的 5个知识点：**

<1>.僵尸进程：在一个进程调用了exit之后，该进程并非马上消失还留下一些参与的信息

<2>.用wait结束僵尸进程.

#include <sys/types.h>

#include <sys/wait.h>

pid\_t wait (int \*stutas)

原理：进程一旦调用了wait，就立即阻塞自己，当分析到当前进程的子进程已经exit，便会收集这个子进程的信息，然后彻底销毁，如果没有找到这样的子进程，就会一直阻塞在这里，直到有一个出现。

<3>.参数status

在wait中，这是一个指向一个整型数据的指针，用来存放子进程退出时的状态，后来又定义了一个WIFEXITED（status)的宏来完成这项工作，这里的status就是一个指针指向的整数。

<4>进程同步。wait可以解决进程的同步问题，通过wait（&status）语句在父进程中等待子进程的返回值。

<5>waitpid函数。其实，wait是waitpid的包装后的形式。waitpid有三个参数（pid，\*status，options）

pid>0,只等待PID=pid的子进程；=0,等待同一进程组中的任何子进程；=-1，等待任何子进程退出，此时=wait；<-1，等待指定进程组中的任何子进程

options：WNOHANG|WUNTRACED，也可设为0，此时=wait。

返回值：1.正常返回子进程的PID；2.若设置了WNOHANG，可能返回0；3.error，返回-1。

**5.sleep**

就是挂起进程指定的秒数，时间到了返回0。

当然除了这些进程的基本操作以外，还有一些特殊操作。比如，setuid和setgid，setpgrp和setpgid设置进程组ID，chdir修改指定的目录，chroot修改根目录，nice修改权限等等。

注：选自CSDN，博客YJFinHZ

### 7.2进程线程实现方式的差异---相关系统调用

线程是更小的可以调度的单位，也就是说，只要达到线程的水平就可以被调度了，进程自然可以被调度。它强调的是分配资源时的对象必须是进程，不会给一个线程单独分配系统管理的资源。若要运行一个任务，想要获得资源，最起码得有进程，其他子任务可以以线程身份运行，资源共享就行了。

  简而言之，进程的个体间是完全独立的，而线程间是彼此依存的。多进程环境中，任何一个进程的终止，不会影响到其他进程。而多线程环境中，父线程终止，全部子线程被迫终止(没有了资源)。而任何一个子线程终止一般不会影响其他线程，除非子线程执行了exit()系统调用。任何一个子线程执行exit()，全部线程同时灭亡。

其实，也没有人写出只有线程而没有进程的程序。多线程程序中至少有一个主线程，而这个主线程其实就是有main函数的进程。它是整个程序的进程，所有线程都是它的子线程。我们通常把具有多线程的主进程称之为主线程。

从系统实现角度讲，进程的实现是调用fork系统调用：pid\_t fork(void);

线程的实现是调用clone系统调用：

int clone(int (\*fn)(void \*), void \*child\_stack, int flags, void \*arg, .../\* pid\_t \*ptid, struct user\_desc \*tls, pid\_t \*ctid \*/);

其中，fork()是将父进程的全部资源复制给了子进程。而线程的clone只是复制了一小部分必要的资源。在调用clone时可以通过参数控制要复制的对象。可以说，fork实现的是clone的加强完整版。当然，后来操作系统还进一步优化fork实现——写时复制技术。在子进程需要复制资源(比如子进程执行写入动作更改父进程内存空间)时才复制，否则创建子进程时先不复制。

实际中，编写多进程程序时采用fork创建子进程实体。而创建线程时并不采用clone系统调用，而是采用线程库函数。常用线程库有Linux-Native线程库和POSIX线程库。其中应用最为广泛的是POSIX线程库。因此在多线程程序中看到的是pthread\_create而非clone。我们知道，库是建立在操作系统层面上的功能集合，因而它的功能都是操作系统提供的。由此可知，线程库的内部很可能实现了clone的调用。不管是进程还是线程的实体，都是操作系统上运行的实体。

  最后，说一下vfork()，这也是一个系统调用，用来创建一个新的进程。它创建的进程并不复制父进程的资源空间，而是共享，也就说实际上vfork实现的是一个接近线程的实体，只是以进程方式来管理它。并且，vfork()的子进程与父进程的运行时间是确定的：子进程“结束”后父进程才运行。【这里的结束并非子进程完成退出之意，而是子进程返回时。】

五、实验心得与问题

1、感觉这次实验报告折腾好久虽然加入导航栏什么的但最后做的也是够烂的了（希望不要成为反例，不要出现羞耻的实验心得部分），但是疯狂爱上CSDN博客平台，有一种大佬们指点江山的感觉，同时觉得也是很值得学习和查阅的选择，相关学习的小结都放在报告里了，实验的最后主要是说明实验过程中的问题。

2、此次试验我安装了codeblocks编译器来编写程序，方便直接调试，同时用了终端命令行的测试。写程序的话，果然还是比较习惯用编译器来调试。

3、进程的撤销和终止，始终觉得不是同一个概念，但是又好像进行了一个无用功的查询，始终没有得出个肯定的结果，报告中就索性按两者一样处理了，另外发现维基百科并不能打开，貌似因某些政治因素，特此附上解决办法<https://jingyan.baidu.com/article/49ad8bcebd967c5834d8faa4.html>

4、听说课堂上有演示过线程的创建的程序，然而一点印象也没有，罪过罪过。

5、不得不反思自己的学习方法问题了，效率实在太低，在一个实验上消耗的时间过多了点，又总觉得自己是错的，另外！强势提醒自己不要再纠结于程序里的警告，看错误就好！