# 实验四 进程的创建和简单控制

**实验目的：**

1. 理解系统调用的概念；
2. 认识进程的并发执行，了解进程族之间各种标识及其存在的关系；
3. 熟悉进程的创建、阻塞、唤醒、撤销等控制方法。

**实验内容：**

1. 掌握进程创建的系统调用fork ( )；
2. 了解并发程序的不可确定性，进行简单并发程序设计。
3. 使用系统调用：进程的创建fork ( )、阻塞wait ( )、睡眠sleep ( )、终止exit( )等。

**实验步骤：**

**(一)系统调用**

系统功能调用（system call）是操作系统提供给程序设计人员的一种服务。程序设计人员在编写程序时，可以利用系统调用来请求操作系统的服务。

**(二)本实验涉及的系统调用**

1. 创建一个新进程：pid\_t fork( void);

函数说明：

pid\_t是一个[宏定义](https://baike.baidu.com/item/%E5%AE%8F%E5%AE%9A%E4%B9%89" \t "_blank)，其实质是int，被定义在#include<sys/types.h>中。

系统调用fork用于创建一个新进程。调用者称为父进程，生成的新进程称为子进程。创建新进程后，父子两个进程将执行fork（）系统调用之后的下一条指令。子进程使用相同的PC（程序计数器），相同的CPU寄存器，以及相同的打开文件。

fork调用的一个奇妙之处就是它仅仅被调用一次，却能够返回两次，它可能有三种不同的返回值：

1）在父进程中，fork返回新创建子进程的进程ID；

2）在子进程中，fork返回0；

3）如果出现错误，fork返回一个负值。

需要的头文件：

1）#include<unistd.h>

2）#include<sys/types.h>

1. 获取进程标识号：pid\_t getpid(void); pid\_t getpid(void);

函数说明：

系统调用getpid（）用来取得当前进程的进程ID，系统调用getppid（）用来取得当前进程的父进程ID。

返回值：

getpid（）返回当前进程的进程ID；getppid（）返回当前进程的父进程ID。

1. 等待子进程终止：pid\_t wait(int \*status);

函数说明：

系统调用wait( )用于使父进程阻塞，直到一个子进程结束或者该进程接收到了一个指定的信号为止。如果该父进程没有子进程或者它的子进程已经结束，则wait( )就会立即返回。

返回值：

成功返回已运行结束的子进程号；失败返回-1。

需要的头文件：

1）#include<sys/wait.h>

2）#include<sys/types.h>

1. 终止进程：void exit(int status);

函数说明：

使调用本函数的进程正常终止，然后把形参的值status&0377（八进制）返回给父进程，父进程可以通过wait函数族来获取这个返回值。

exit(status)函数执行之后，形参的值会被传递到父进程，这时有3种情况：

1）如果父进程设置了SA\_NOCLDWAIT标志，或者 把SIGCHLD信号的处理函数设置为SIG\_IGN，那么子进程的返回值被丢弃，子进程立即消亡；

2）如果父进程在等待子进程，他将被通知子进程的退出状态；子进程立即消亡；

3）如果父进程没有被设置为“忽略子进程的退出值”，这时父进程应当使用wait或者waitpid等待子进程的结束，如果父进程不等待，那么子进程结束之后会变成僵尸进程。

**(三)孤儿进程和僵尸进程**

孤儿进程：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/systemd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。孤儿进程不会浪费资源。

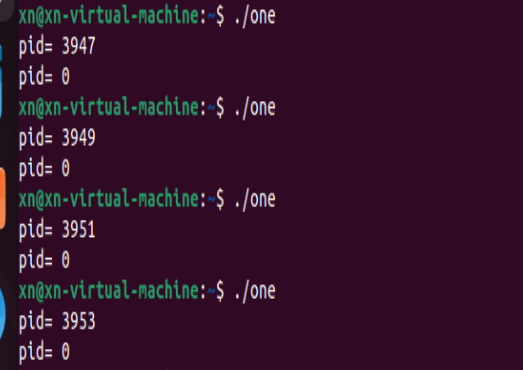
僵尸进程：一个进程使用fork创建子进程，如果子进程退出，而父进程并没有调用wait或waitpid获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中。这种进程称之为僵尸进程。僵尸进程浪费系统资源（进程描述符task\_struct存在）。

**(四)例程，使用fork( )创建进程**

1. 编辑下述代码，详见教材P98。

|  |
| --- |
| #include<stdio.h>  main()  {  int pid;  pid = fork();  printf("pid= %d\n",pid); ➀  //printf("pid= %d\n",getpid()); ➁  sleep(1);  } |

编译链接通过后，多次运行例程，观察进程并发执行结果，并思考下述问题：



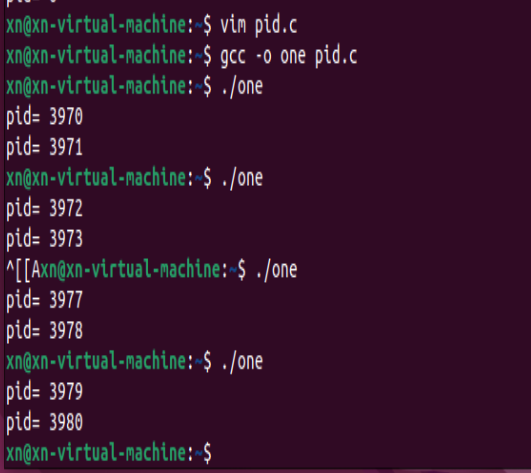
1. 为什么会有两行输出？理解fork（）的作用；

当程序运行时，fork() 函数被调用。fork() 函数创建一个新的进程，这个新进程几乎是当前进程的副本，包括程序计数器、寄存器集合和内存映像。 在父进程中，fork() 返回新创建的子进程的进程标识符（PID），而在子进程中，fork() 返回0。因此，会看到两个不同的pid值被打印出来，一个是正数（子进程的PID），另一个是0（父进程中的返回值）。

1. 多次运行，观察输出内容的变化，理解系统给进程随机分配进程号；

每次程序运行并调用fork()时，操作系统都会为新创建的子进程分配一个新的PID。这个PID是由操作系统动态分配的，通常是基于系统内部的调度和资源管理策略。 当多次运行程序时，会看到每次输出的PID值都是不同的，这说明操作系统在每次创建新进程时都在使用不同的PID,但是子进程id号一直为0；

1. 将语句➀替换成语句➁，再次运行程序。观察输出的改变，理解fork（）的返回值。



程序的输出将会显示父进程的PID。在父进程中，getpid() 会返回当前进程的PID，而在子进程中，由于fork() 已经返回了子进程的PID，所以getpid() 也会返回子进程的PID。 当运行修改后的程序时，会看到两次输出，一次是父进程的PID，另一次是子进程的PID。这进一步说明了fork() 函数在父进程和子进程中返回不同的值。

1. 编辑下面的程序，要求实现父进程产生两个子进程，父进程显示字符“a”、两个子进程，分别显示字符“b”、“c”，如下所示。

|  |
| --- |
| #include<stdio.h>  main( )  {  int p1,p2;  while ((p1 = fork( )) == -1); //父进程创建第一个进程，直到成功  if(p1 == 0) //0返回给子进程 1  {  putchar('b'); //P1的处理过程  }  else ➀  { //正数返回给父进程(子进程号)  while ((p2 = fork( )) == -1); //父进程创建第二个进程，直到成功  if(p2 == 0) //0返回给子进程2  {  putchar('c'); //P2的处理过程  }  else ➁  {  putchar('a'); //P2创建完成后，父进程的处理过程  }  }  } |
|  |

编译链接通过后，多次运行例程，观察进程并发执行结果，并思考下述问题：

1. 分析为何例程中三个分支都运行了？

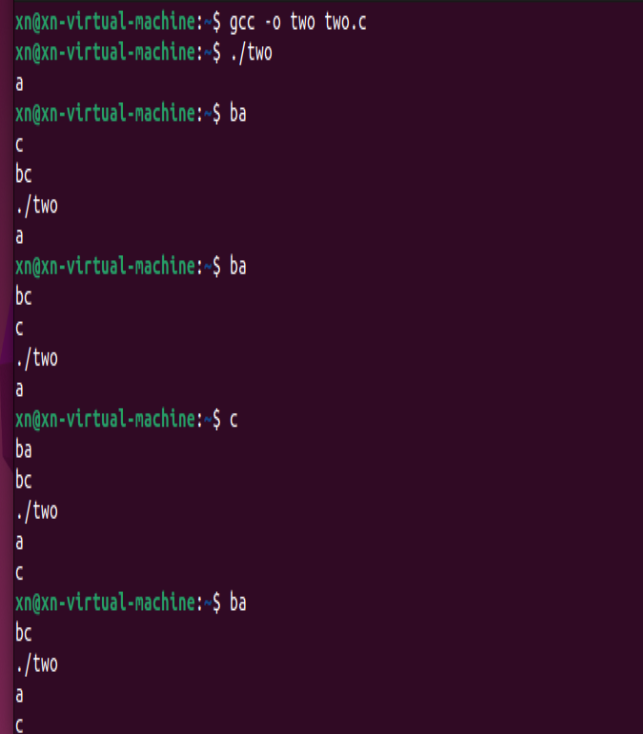
因为在每次调用 fork() 函数后，会创建两个进程：一个父进程的副本和一个子进程。这意味着每次 fork() 调用都会在父进程和子进程中各执行一次。由于 fork() 调用是在 while 循环中进行的，所以即使第一次 fork() 失败，循环会继续执行直到 fork() 成功。这就导致了每个分支都被执行。

1. ./f1 运行结果为什么不一样？每种结果的产生原因。

运行结果不一样是因为进程调度是由操作系统控制的，而且每次运行程序时，进程的执行顺序和调度可能会有所不同。这就导致了每次运行程序时，父进程和子进程的执行顺序可能会变化，从而产生不同的输出结果

1. 删除语句➀或➁，观察输出的内容，体会fork的使用。

删除语句1 后



1. 运行命令为什么是“./command”？理解Linux的PATH环境变量的作用。

Linux 系统使用 PATH 环境变量来确定可执行文件的搜索路径。

1. . 和 .. 什么含义？理解Linux当前目录和父目录的概念。

. 表示当前目录，.. 表示父目录。

1. shell 提示为什么不换行，而是紧接着输出内容显示

shell 提示通常在命令执行完毕后显示，如果命令紧接着执行，shell 会立即显示提示，而不是等待换行。

1. 输出字母为什么和提示交错？

由于进程调度的不确定性，父进程和子进程可能会几乎同时执行，导致它们的输出交错在一起

1. ./f1|pstree|grep f1什么含义？理解命令中管道的作用和使用方法。

这个命令链使用管道 | 将 pstree 命令的输出传递给 grep f1 命令，以便过滤出包含 f1 的进程信息。

1. 第8问中有时组合命令没有输出，请分析原因？

可能是因为 f1 进程已经结束，或者 pstree 命令的输出格式与 grep 命令不匹配。

1. 如果想保留第8问的./f1的输出内容，该如何操作？理解Linux文件重定向的概念和使用方法。

使用输出重定向操作符 > 将输出保存到文件中，./f1 > output.txt。这样，./f1 的输出将被保存到 output.txt 文件中。

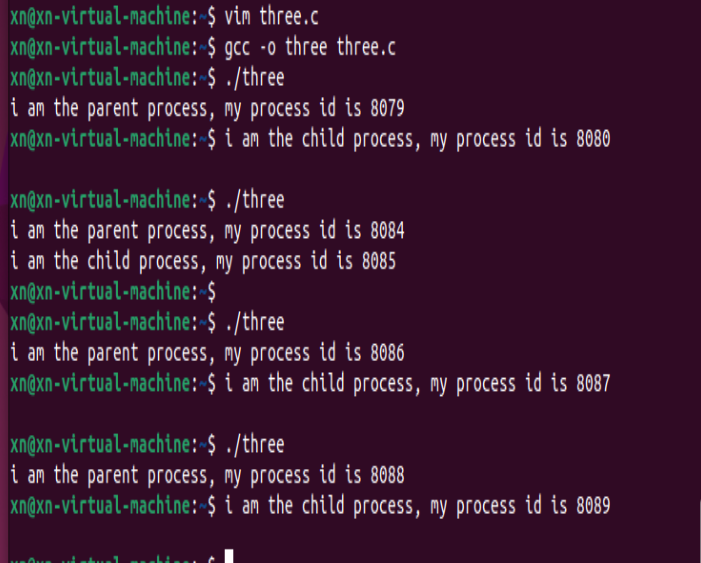
**扩展编程：修改代码，产生祖孙三代的进程。**

**(五)例程，使用getpid（）和getppid（）查看进程号。**

1. 理解系统调用fork（）的两个返回值，理解获取进程号的系统调用getpid( )和getppid( )。

|  |
| --- |
| #include <unistd.h>  #include <stdio.h>  #include <sys/types.h>  main ()  {  pid\_t pid;  pid=fork();  if (pid < 0)  printf("error in fork!");  else if (pid == 0)  {  printf("i am the child process, my process id is %d\n",getpid());  }  else  {  printf("i am the parent process, my process id is %d\n",getpid());  }  } |

编译链接通过后，多次运行例程，观察进程并发执行结果，并思考下述问题：



1. 请分析父子进程输出内容交替的原因。

父子进程输出内容交替的原因是由于进程调度的结果。在任何给定的时间内，操作系统的调度器选择哪个进程运行是不确定的。因此，父进程和子进程的执行顺序可能会交替出现，这取决于操作系统的调度策略和当前系统的工作负载。 当 fork() 成功时，它会创建两个几乎相同的进程副本：父进程和子进程。这两个进程都将继续执行 fork() 之后的代码。由于进程调度的并发性，可能会看到父进程和子进程的输出交错显示。

1. 改写原程序，用变量pid替换getpid()，再次观察运行情况，理解fork()在父子进程中有不同的返回值。



pid 变量来存储 fork() 的返回值，并在之后的 printf() 调用中使用它。可以清楚地看到 fork() 在父进程和子进程中返回不同的值：在子进程中，pid 为 0；在父进程中，pid 为子进程的 PID。

1. 理解上图中两次运行后，子进程输出的差异，理解孤儿进程的概念。

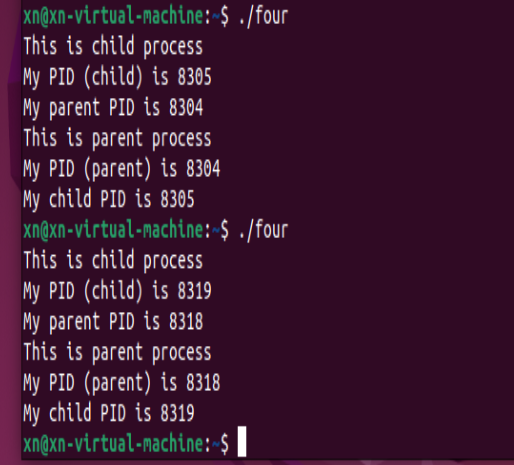
每次运行程序时，子进程的 PID 都可能不同，因为操作系统会为每个新创建的进程分配一个唯一的 PID。两次运行程序后，子进程输出的 PID 可能会有所不同。

孤儿进程是指其父进程在子进程结束之前已经终止的进程。在这种情况下，子进程将成为系统进程的子进程，通常由 init 进程（PID 为 1 的进程）接管。在提供的程序中，父进程在子进程执行完毕后立即结束，所以子进程不会成为孤儿进程。但是，如果父进程在子进程之前结束，那么子进程将成为孤儿进程。

1. 理解系统调用wait( )、getpid( )和getppid( )的使用。程序代码如下所示。

|  |
| --- |
| #include <stdio.h>  #include <sys/types.h>  #include <sys/wait.h>  #include <sys/stat.h>  #include <fcntl.h>  #include <unistd.h>  #include <errno.h>  int main()  {  char buf[100];  pid\_t cld\_pid;  int fd;  if((fd=open("temp",O\_CREAT|O\_TRUNC|O\_RDWR,S\_IRWXU))==-1)  {  printf("open error%d",errno);  exit(1);  }  strcpy(buf,"This is parent process write\n");  if((cld\_pid=fork())==0)  { //这里是子进程执行的代码  strcpy(buf,"This is child process write\n");  printf("This is child process\n");  sleep(1);  printf("My PID (child) is%d\n",getpid()); //打印出本进程的ID  sleep(1);  printf("My parent PID is %d\n",getppid()); //打印出父进程的ID  sleep(1);  write(fd,buf,strlen(buf));  close(fd);  exit(0);  }  else  { //这里是父进程执行的代码  wait(0); //如果此处没有这一句会如何？  printf("This is parent process\n");  sleep(1);  printf("My PID (parent) is %d\n",getpid()); //打印出本进程的ID  sleep(1);  printf("My child PID is %d\n",cld\_pid); //打印出子进程的ID  sleep(1);  write(fd,buf,strlen(buf));  close(fd);  }  return 0;  } |

编译链接通过后，多次运行例程，观察进程并发执行结果，并思考下述问题：



1. 分析父子进程输出内容交替的原因；

父子进程输出内容交替的原因是由于操作系统的进程调度策略。在多进程环境中，操作系统的调度器负责决定哪个进程在何时运行。由于进程调度通常是抢占式的，父进程和子进程可能会交替获得CPU时间片来执行它们的代码。这种交替执行的现象是并发程序的一个典型特征，它展示了操作系统如何管理和调度多个同时运行的进程。

1. 语句sleep(1);起什么作用？删除所有sleep(1);语句，并观察运行结果；

sleep(1); 语句使得当前运行的进程暂停执行1秒钟。这个调用在本程序中用于模拟进程的执行时间，以及在打印信息之间引入延迟，使得输出结果更容易观察和理解。如果删除所有的 sleep(1); 语句，父进程和子进程的输出可能会几乎同时发生，没有明显的延迟，这使得观察和区分它们的输出变得更加困难。此外，没有 sleep(1); 语句，父进程可能会在子进程完成其工作之前就结束，导致输出顺序混乱。

1. 删除wait(0);语句，并观察运行结果，并请分析两次结果不同的原因，理解wait的作用。

wait() 系统调用用于等待子进程结束。在本程序中，父进程中的 wait(0); 语句使得父进程等待子进程结束之后再继续执行。如果删除这个 wait(0); 语句，父进程将不会等待子进程完成，而是继续执行其后续的代码。这可能导致父进程在子进程完成写入文件操作之前就尝试写入相同的文件，从而引发竞态条件，导致输出结果不一致。 此外，没有 wait() 调用，父进程可能会先于子进程结束，使得子进程成为孤儿进程。孤儿进程通常由 init 进程接管，这可能会导致程序的非预期行为和资源管理问题。

总结来说，wait() 调用确保了父进程在子进程完成后再继续执行，这有助于避免竞态条件和确保资源的正确管理。通过删除 wait(0); 语句，我们可以观察到进程同步的重要性，以及在并发编程中正确管理进程间通信和同步的必要性。

1. 扩充：关于父子进程各自又再生成子进程的例子。

|  |
| --- |
| #include <unistd.h>  #include <stdio.h>  #include <sys/types.h>  main()  {  pid\_t a\_pid,b\_pid;  if((a\_pid=fork())<0)  printf("error!");  else  if(a\_pid==0)  printf("b\n");  else  printf("a\n");    if((b\_pid=fork())<0)  printf("error!");  else  if(b\_pid==0)  printf("c\n");  else  printf("a\n");  } |

编译链接通过后，多次运行例程，观察进程并发执行结果，并思考下述问题：



1. 例程运行后，共产生了几个进程？请分析出它们的宗族关系。

3个进程

初始进程（父进程）启动后，通过第一次 fork() 创建了一个子进程（子进程A）。此时，父进程和子进程A都存在。

子进程A执行第二次 fork()，创建了另一个子进程（子进程B）。此时，子进程A成为子进程B的父进程。

父进程、子进程A和子进程B都独立执行，因为每个进程都会执行 fork() 后的代码，所以它们都会尝试输出字符。

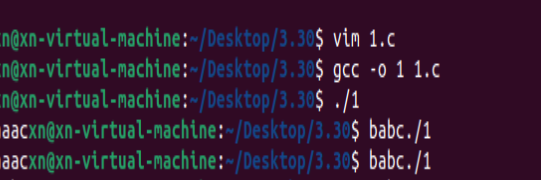
1. 例程运行后，共输出几个字符？分别是什么字符？分别由哪个进程输出的？

共输出6个字符

父进程输出字符 'a'。

子进程A输出字符 'b'。

子进程B输出字符 'c'。

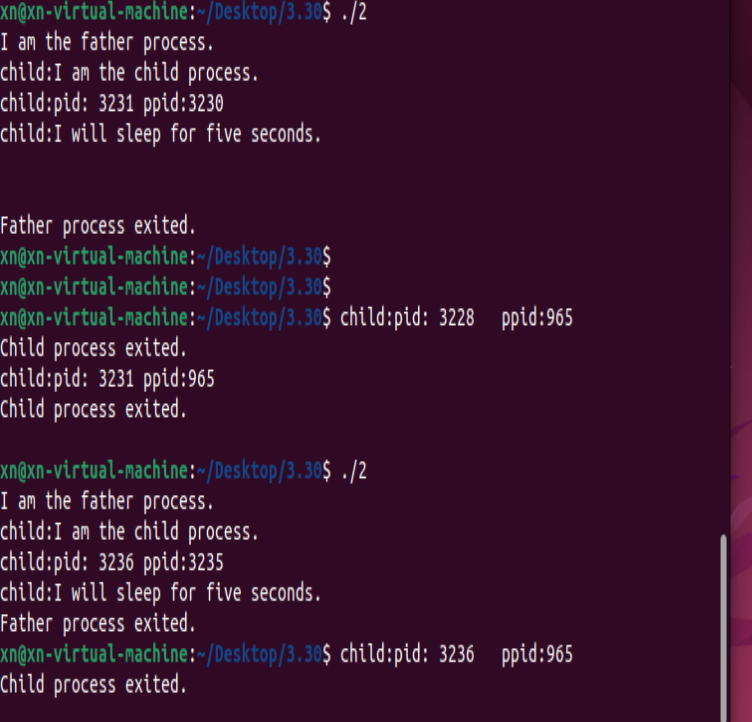
1. 删除输出语句中的回车符，输出结果有何改变？试分析原因，理解输出缓冲的概念。
   1. 如果删除了 printf 函数中的换行符 \n，那么所有的输出将会在同一行上连续显示，而不是每个字符后都有一个新行。
   2. 由于输出缓冲区的存在，当标准输出（通常是终端或控制台）需要显示输出时，缓冲区中的内容会被一次性输出。如果没有换行符，所有的字符会连续输出，不会有明显的分隔。
   3. 在某些情况下，如果输出缓冲区没有被刷新（例如，没有遇到换行符或其他刷新缓冲区的操作），那么输出可能会延迟，直到缓冲区被填满或者程序结束。删除换行符后，由于输出是连续的，缓冲区可能会更频繁地被刷新，这可能会导致输出看起来更加连续。

**(六)例程，观察僵尸进程和孤儿进程。**

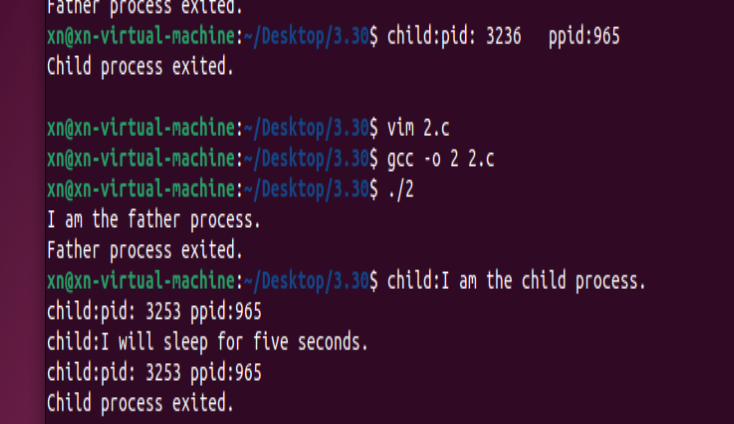
1. 编辑运行下述程序，观察孤儿进程：

|  |
| --- |
| #include <stdio.h>#include <stdlib.h>#include <errno.h>#include <unistd.h>  int main(){ pid\_t pid;  pid = fork(); //创建一个进程 if (pid == 0) //子进程 { printf("child:I am the child process.\n"); printf("child:pid: %d\tppid:%d\n", getpid(), getppid());//输出进程ID和父进程ID printf("child:I will sleep for five seconds.\n"); sleep(5);//睡眠5s，保证父进程先退出 printf("child:pid: %d\tppid:%d\n", getpid(), getppid()); printf("Child process exited.\n"); } else//父进程 { printf("I am the father process.\n"); sleep(1);//父进程睡眠1s，保证子进程输出进程id printf("Father process exited.\n"); } exit(0);} |

编译链接通过后，多次运行例程，观察进程并发执行结果，并思考下述问题：



1. 删除所有sleep（），观察父子进程的宗族关系及各自的进程号；



父进程会先于子进程退出·，父进程退出后，子进程成为孤儿进程，它的父进程ID会被设置为 1（init 进程的进程号）。

子进程会继续执行，打印剩余的 "child" 相关的信息

1. 恢复所有sleep（），观察父进程提前结束后，子进程成为孤儿进程转交给1号进程。

当 sleep() 调用被恢复后，父进程会睡眠 1 秒，这确保了子进程有足够的时间先打印 "child" 相关的信息。

父进程在睡眠 1 秒后退出，打印 "Father process exited."，并终止。

父进程退出后，子进程成为孤儿进程，它的父进程ID会被设置为 1（init 进程的进程号）。

init 进程是 UNIX 和类 UNIX 系统中的第一个进程，它的进程号通常是 1。init 进程负责接收和管理孤儿进程。

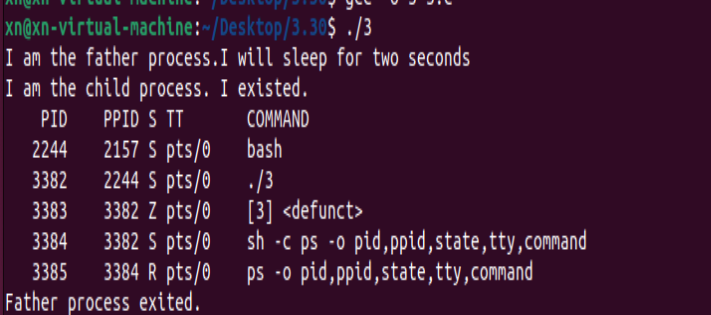
子进程继续执行，打印 "child" 相关的信息，包括它的进程ID和父进程ID（在父进程退出后变为 1）。

子进程最终也会退出，并打印 "Child process exited."

1. 编辑运行下述例程，观察孤儿进程：

|  |
| --- |
| #include <stdio.h>#include <unistd.h>#include <errno.h>#include <stdlib.h>  int main(){ pid\_t pid; pid = fork(); if (pid == 0) { printf("I am the child process. I existed.\n");  exit(0); } printf("I am the father process.I will sleep for two seconds\n"); //等待子进程先退出 sleep(2); //输出进程信息 system("ps -o pid,ppid,state,tty,command"); printf("Father process exited.\n"); exit(0);  } |

编译链接通过后，多次运行例程，观察进程并发执行结果，并思考下述问题：



1. 理解system（），在程序中运行shell命令；

system() 函数是 C 语言标准库中的一个函数，它用于在程序中运行 shell 命令。

该函数会创建一个 shell 进程，然后执行指定的命令字符串。

命令执行完毕后，system() 会等待 shell 进程结束，然后返回 shell 进程的退出状态。

1. 观察输出内容，查看僵尸进程。

ps -o pid,ppid,state,tty,command 命令的输出会显示进程的进程ID（pid）、父进程ID（ppid）、状态（state）、终端（tty）和命令（command）。

在输出中，可以查找子进程的进程ID和父进程ID，以及它们的状态。

如果父进程在子进程退出之前打印了 "Father process exited." 并退出，那么子进程将成为孤儿进程，并被 init 进程接管。

通常情况下，子进程会在父进程退出后很快退出，因此可能不会在 ps 命令的输出中看到子进程作为僵尸进程。僵尸进程是指已经执行完毕但仍然保留在进程表中的进程，通常是因为父进程没有调用 wait() 或 waitpid() 来读取子进程的退出状态。

观察僵尸进程，让父进程不调用 wait() 或 waitpid() 并且快速退出，这样子进程在退出后可能会暂时成为僵尸进程，直到下一个父进程（通常是 init 进程）读取其退出状态。

**(七)编程题：理解前述例程后，按要求完成程序编写。**

编写程序创建子进程。父子进程分别打印自己和父进程的进程ID，要求每3秒钟打印系统进程信息，重复5次后退出。父进程待子进程结束后退出。提示：

1. 用系统调用getpid和getppid获取进程ID；
2. 用系统调用fork进程创建；
3. 用系统调用wait控制父子进程同步；wait(NULL)
4. 用库函数system实现在一个进程内部运行另一个进程，即创建一个新进程；
5. Shell命令＂/bin/ps＂作为system的字符串参数，实现打印系统进程信息。

代码



结果

