实验四 进程的创建和简单控制

鲍竹涵 软件工程 2302 班 32301227 2025 年 3 月 20 日星期四
OS:Ubuntu 24.10, GECN24WW(V1.08), x64, GNOME x47, Kernal : GECN24WW(V1.08)
IDE:Visual Studio Code for Liunx 1.98.2
Compiler: gcc/g++/g++14 (Ubuntu 14.2.0-4ubuntu2) 14.2.0

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，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>

2. 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。

3. 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>

4. 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);                    ①
    //printf("pid= %d\n",getpid());             ②
    sleep(1);
}
```

08操作系统原理与实验 > Experiment4 > Demo1.cpp > main()

```cpp
#include <bits/stdc++.h>
#include <unistd.h>
#include <sys/types.h>

int main(){
    int pid;
    pid = fork();
    // std::cout << "pid = " << pid << std::endl;
    std::cout << "fun() getpid = " << getpid() << std::endl;
    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；

```
问题  输出  调试控制台  终端  端口  评论

pid = 34326
pid = 0
[1] + Done                       "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<"
/tmp/Microsoft-MIEngine-In-iymayb4y.z12" 1>"/tmp/Microsoft-MIEngine-Out-ys2lohd5.cye"
baozhuhan@ubuntu24:~/Documents/Awesome-SE-Box$
```

在代码中，fork() 函数被调用后，进程会被复制，生成一个子进程。fork() 函数在父进程中返回子进程的 PID，在子进程中返回 0。因此，std::cout << "pid = " << pid << std::endl; 这行代码会在父进程和子进程中各执行一次。

(2) 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；

第一次运行：

```
问题  输出  调试控制台  终端  端口  评论

pid = 39203
pid = 0
[1] + Done                       "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<"
/tmp/Microsoft-MIEngine-In-puhgp0vr.gaq" 1>"/tmp/Microsoft-MIEngine-Out-0p0pgmty.3ng"
baozhuhan@ubuntu24:~/Documents/Awesome-SE-Box$
```

第二次运行：

```
问题  输出  调试控制台  终端  端口  评论

pid = 39813
pid = 0
[1] + Done                       "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<"
/tmp/Microsoft-MIEngine-In-0natqvt0.vpv" 1>"/tmp/Microsoft-MIEngine-Out-dpbvbhd2.0be"
baozhuhan@ubuntu24:~/Documents/Awesome-SE-Box$
```

观察到系统给进程随机分配进程号

(3) 将语句①替换成语句②，再次运行程序。观察输出的改变，理解 fork（）的返回值。

```
问题  输出  调试控制台  终端  端口  评论

fun() getpid = 41035
fun() getpid = 41107
[1] + Done                       "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<"
/tmp/Microsoft-MIEngine-In-rnqa1mv3.fyp" 1>"/tmp/Microsoft-MIEngine-Out-lmtvlq11.whe"
baozhuhan@ubuntu24:~/Documents/Awesome-SE-Box$
```

当前代码中，std::cout << "fun() getpid = " << getpid() << std::endl; 输出的是当

前进程的 PID，而不是 fork() 返回的值。因此，无论是父进程还是子进程，都会输出各自的 PID。

2. 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                    ①
    {                                       //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                    ②
            {
            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：

编写代码如下：

```
08操作系统原理与实验 > Experiment4 > Demo2.cpp > main()
int main(){
    int p1, p2;
    while((p1 = fork()) == -1);
    if(p1 == 0){
        std::cout << 'b' << std::endl;
    }else{
        while((p2 = fork()) == -1);
        if(p2 == 0){
            std::cout << 'c' << std::endl;
        }else{
            std::cout << 'a' << std::endl;
        }
    }
}
```

运行结果如下：

```
问题  输出  调试控制台  终端  端口  评论

b
a
c
[1] + Done                       "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<"
/tmp/Microsoft-MIEngine-In-dfgsqu03.lsl" 1>"/tmp/Microsoft-MIEngine-Out-0d0h03vl.btr"
baozhuhan@ubuntu24:~/Documents/Awesome-SE-Box$
```

1) 分析为何例程中三个分支都运行了？

fork() 函数被调用两次，每次都会创建一个新的子进程。由于每个子进程都会从 fork() 调用之后的代码继续执行，因此三个分支都运行的情况。

2) ./f1 运行结果为什么不一样？每种结果的产生原因。

都是一样的，OMG

3) 删除语句①或②，观察输出的内容，体会 fork 的使用。

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

在 Linux 系统中，PATH 环境变量用于指定系统查找可执行文件的目录。当你在终端中输入一个命令时，系统会按照 PATH 环境变量中列出的目录顺序查找对应的可执行文件。如果找到了，就执行该文件；如果找不到，就会返回“命令未找到”的错误。

5) . 和 .. 什么含义？理解 Linux 当前目录和父目录的概念。

在 Linux 文件系统中，. 和 .. 是两个特殊的目录名，用于表示当前目录和父目录。

6) shell 提示为什么不换行，而是紧接着输出内容显示？

在 shell 中，如果输出内容没有包含换行符 \n，那么 shell 提示符会紧接着输出内容显示，而不会换行。这是因为 shell 提示符默认是在当前行的末尾显示的。

7) 输出字母为什么和提示交错？

标准输出通常是行缓冲的，这意味着它会在遇到换行符时刷新缓冲区，而标准错误输出是无缓冲的，立即输出。

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

命令 ./f1 | pstree | grep f1 的含义是执行 f1 程序，显示进程树，并筛选出包含 f1 的进程信息。

9) 第 8 问中有时组合命令没有输出，请分析原因？

组合命令 ./f1 | pstree | grep f1 有时没有输出的原因可能是 pstree 命令在 f1 进程结束之前生成了进程树，导致 f1 进程没有出现在 pstree 的输出中。由于 pstree 显示的是命令执行时的进程树快照，如果 f1 进程执行得非常快，可能在 pstree 捕获进程树之前就已经结束了。

10) 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。

It is very easy, just use "./f1 > out.log 2>&1" :)

扩展编程：修改代码，产生祖孙三代的进程。

```
#include <bits/stdc++.h>
#include <unistd.h>
#include <sys/types.h>

int main(){
int p1, p2, p3;
while((p1 = fork()) == -1);
if(p1 == 0){
while((p2 = fork()) == -1);
if(p2 == 0){
std::cout << 'c' << std::endl;
}else{
std::cout << 'b' << std::endl;
}
}else{
std::cout << 'a' << std::endl;
}
}
```

(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：

编写程序如下：

```
08操作系统原理与实验 > Experiment4 > Demo3.cpp > main()
#include <bits/stdc++.h>
#include <unistd.h>
#include <sys/types.h>

int 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());
    }
}
```

第一次运行结果：

```
问题  输出  调试控制台  终端  端口  评论

I am the parent process, my process id is 85491
I am the child process, my process id is 85507
[1] + Done                       "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<"/tmp/Microsoft-MIEng
b" 1>"/tmp/Microsoft-MIEngine-Out-u5ejiwi2.tbt"
baozhuhan@ubuntu24:~/Documents/Awesome-SE-Box$
```

第二次运行结果：

```
问题  输出  调试控制台  终端  端口  评论

I am the parent process, my process id is 86326
I am the child process, my process id is 86333
[1] + Done                       "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<"/tmp/Microsoft-MIEng
i" 1>"/tmp/Microsoft-MIEngine-Out-eqfoex4t.x35"
baozhuhan@ubuntu24:~/Documents/Awesome-SE-Box$
```

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因为父进程和子进程是并发执行的。fork() 函数调用后，父进程和子进程会同时运行，并且它们的执行顺序是不确定的。这意味着父进程和子进程的输出可能会交替出现，具体顺序取决于操作系统的调度。

2) 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。

改写的程序：

```c
int 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", pid);
    } else {
        printf("I am the parent process, my process id is %d\n", pid);
    }
}
```

改写后的程序运行结果：

```
问题  输出  调试控制台  终端  端口  评论

I am the child process, my process id is 0
I am the parent process, my process id is 88338
[1] + Done                       "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<"
/tmp/Microsoft-MIEngine-In-rr5gbsym.avs" 1>"/tmp/Microsoft-MIEngine-Out-x0psexqv.3l0"
baozhuhan@ubuntu24:~/Documents/Awesome-SE-Box$
```

观察到子进程的返回值是不同的！

3) 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。

孤儿进程是指其父进程已经终止，但它仍在运行的进程。当一个进程终止时，操作系统会将其所有的子进程重新分配给 init 进程（PID 为 1），init 进程会成为这些孤儿进程的新父进程，并负责清理它们的资源。

2. 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：

第一次运行程序：

第二次运行程序：

第三次运行程序：

1) 分析父子进程输出内容交替的原因；

在这个程序中，父进程和子进程通过调用 fork() 函数创建。fork() 函数会创建一个新的进程（子进程），这个子进程是父进程的副本。父进程和子进程会并行执行，导致它们的输出内容可能交替出现。

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

语句的作用是让当前进程暂停执行 1 秒钟。这可以让输出内容更容易观察，并且可以模拟一些实际应用中的延迟。

删除所有 sleep 后运行结果：

```
问题  输出  调试控制台  终端  端口  评论

This is child process
My PID (child) is 107476
My parent PID is 107460
This is parent process
My PID (parent) is 107460
My child PID is 107476
[1] + Done                       "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<"
/tmp/Microsoft-MIEngine-In-edduio2t.nsa" 1>"/tmp/Microsoft-MIEngine-Out-zohky2ks.2ol"
baozhuhan@ubuntu24:~/Documents/Awesome-SE-Box$
```

3) 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。

删除 wait（0）后运行结果：

```
问题  输出  调试控制台  终端  端口  评论

This is parent process
My PID (parent) is 107897
My child PID is 107901
This is child process
My PID (child) is 107901
My parent PID is 107897
[1] + Done                       "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<"
/tmp/Microsoft-MIEngine-In-at03velu.hsv" 1>"/tmp/Microsoft-MIEngine-Out-dlmds2nd.i4z"
baozhuhan@ubuntu24:~/Documents/Awesome-SE-Box$
```

删除 wait(0)；语句后，父进程将不会等待子进程结束，而是直接继续执行自己的代码。这样，父进程和子进程将并行执行，导致它们的输出内容更加混杂。

3. 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：

第一次运行：

第二次运行：

第三次运行：

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。

在这个例程中，fork() 函数被调用了两次，每次调用都会创建一个新的进程。因此，运行这个程序后，共产生了 4 个进程。我们可以通过分析每次 fork() 调用后的进程关系来理

解它们的宗族关系。

2) 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。分别是：

"a"：由原始父进程 P 输出两次。

"b"：由第一次 fork() 创建的子进程 P1 输出一次。

"a"：由第一次 fork() 创建的子进程 P1 输出一次。

"c"：由第二次 fork() 创建的子进程 P2 输出一次。

"c"：由第二次 fork() 创建的子进程 C2 输出一次。

3) 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。

删除后的输出结果：

删除输出语句中的回车符后，输出结果可能会有所不同。这是因为标准输出（stdout）通常是行缓冲的，这意味着输出会在遇到换行符（\n）时刷新缓冲区。如果没有换行符，输出可能会被缓冲，直到缓冲区满或程序结束时才会输出。

(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：

```
问题  输出  调试控制台  终端  端口  评论

I am the father process.
child:I am the child process.
child:pid: 123431       ppid:123424
child:I will sleep for five seconds.
Father process exited.
[1] + Done                       "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<"
/tmp/Microsoft-MIEngine-In-mofu30ot.ksl" 1>"/tmp/Microsoft-MIEngine-Out-kv11wn50.0sd"
baozhuhan@ubuntu24:~/Documents/Awesome-SE-Box$
```

1) 删除所有 sleep（），观察父子进程的宗族关系及各自的进程号；

```
问题  输出  调试控制台  终端  端口  评论

I am the father process.
child:I am the child process.
Father process exited.
child:pid: 124189       ppid:124185
child:I will sleep for five seconds.
child:pid: 124189       ppid:124185
Child process exited.
[1] + Done                       "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<"
/tmp/Microsoft-MIEngine-In-vwidrffu.xcp" 1>"/tmp/Microsoft-MIEngine-Out-adyy2tsa.01u"
baozhuhan@ubuntu24:~/Documents/Awesome-SE-Box$
```

2) 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。

2. 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：

```
I am the father process.I will sleep for two seconds
I am the child process. I existed.
    PID    PPID S TT       COMMAND
   3147       1 S ?        /usr/lib/systemd/systemd --user
   3151    3147 S ?        (sd-pam)
   3171    3147 S ?        /usr/bin/pipewire
   3172    3147 S ?        /usr/bin/pipewire -c filter-chain.conf
   3177    3147 S ?        /usr/bin/wireplumber
   3178    3147 S ?        /usr/bin/pipewire-pulse
   3180    3147 S ?        /usr/bin/gnome-keyring-daemon --foreground --components=pk
   3196    3147 S ?        /usr/bin/dbus-daemon --session --address=systemd: --nofork
   3228    3147 S ?        /usr/libexec/xdg-document-portal
   3232    3147 S ?        /usr/libexec/xdg-permission-store
   3510    3147 S ?        /usr/libexec/gvfsd
   3511    3147 S ?        /snap/prompting-client/87/bin/prompting-client-daemon
   3522    3147 S ?        /usr/libexec/gvfsd-fuse /run/user/1000/gvfs -f
   3530    3147 S ?        /usr/libexec/at-spi-bus-launcher
   3537    3530 S ?        /usr/bin/dbus-daemon --config-file=/usr/share/defaults/at-
   3594    3147 S ?        /snap/snapd-desktop-integration/253/usr/bin/snapd-desktop-
   3607    3147 S ?        /usr/libexec/gcr-ssh-agent --base-dir /run/user/1000/gcr
   3608    3147 S ?        /usr/libexec/gnome-session-ctl --monitor
   3628    3147 S ?        /usr/libexec/gnome-session-binary --systemd-service --sess
   3685    3594 S ?        /snap/snapd-desktop-integration/253/usr/bin/snapd-desktop-
   3715    3147 S ?        /usr/bin/gnome-shell
   3893       1 S ?        /usr/share/sangfor/aTrust/resources/bin/aTrustAgent --plug
   3905    3715 S ?        /usr/libexec/mutter-x11-frames
   3971    3147 S ?        /usr/libexec/at-spi2-registryd --use-gnome-session
   4031    3147 S ?        /usr/libexec/xdg-desktop-portal
   4110    3147 S ?        /usr/libexec/gnome-shell-calendar-server
   4117    3147 S ?        /usr/libexec/evolution-source-registry
   4126    3147 S ?        /usr/libexec/dconf-service
   4146    3147 S ?        /usr/bin/gjs -m /usr/share/gnome-shell/org.gnome.Shell.Not
   4171    3147 S ?        /usr/bin/ibus-daemon --panel disable --xim
   4172    3147 S ?        /usr/libexec/gsd-a11y-settings
   4173    3147 S ?        /usr/libexec/gsd-color
```

1) 理解 system（），在程序中运行 shell 命令；

system() 函数用于在程序中运行 shell 命令。它会调用 /bin/sh -c 来执行传递给它的命令字符串。system() 函数会阻塞调用进程，直到命令执行完毕。

2) 观察输出内容，查看僵尸进程。

```
 125146   15734 S ?        /home/baozhuhan/.vscode/extensions/ms-vscode.cpptools-1.23
 125194  125181 S ?        /home/baozhuhan/Documents/Awesome-SE-Box/08操作系统原理与
 125198  125194 Z ?        [Demo7] <defunct>
 125228  125194 S ?        sh -c -- ps -o pid,ppid,state,tty,command
 125229  125228 R ?        ps -o pid,ppid,state,tty,command
Father process exited
```

在这个例子中，PID 为 125198 的进程是一个僵尸进程，因为它的状态是 Z，并且命令显示为 [Demo7] <defunct>。这是因为子进程已经退出，但父进程还没有调用 wait() 或 waitpid() 来获取子进程的终止状态，从而导致子进程成为僵尸进程。

(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；

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：

```
#include <bits/stdc++.h>
#include <unistd.h>
#include <sys/wait.h>
#include <cstdlib>

int main() {
pid_t pid = fork();
if (pid < 0) {
std::cerr << "Fork failed" << std::endl;
return 1;
} else if (pid == 0) {
// 子进程
for (int i = 0; i < 5; ++i) {
std::cout << "子进程 ID: " << getpid() << ", 父进程 ID: " << getppid() << std::endl;
system("/bin/ps");
sleep(3);
}
exit(0);
} else {
// 父进程
wait(NULL); // 等待子进程结束
std::cout << "子进程结束，父进程 ID: " << getpid() << std::endl;
}
return 0;
}
```

运行结果：

```
子进程 ID: 135139, 父进程 ID: 135138
PID TTY TIME CMD
134364 pts/0 00:00:00 bash
135138 pts/0 00:00:00 Homework
135139 pts/0 00:00:00 Homework
135140 pts/0 00:00:00 sh
135141 pts/0 00:00:00 ps
子进程 ID: 135139, 父进程 ID: 135138
PID TTY TIME CMD
134364 pts/0 00:00:00 bash
135138 pts/0 00:00:00 Homework
135139 pts/0 00:00:00 Homework
135237 pts/0 00:00:00 sh
135238 pts/0 00:00:00 ps
子进程 ID: 135139, 父进程 ID: 135138
PID TTY TIME CMD
134364 pts/0 00:00:00 bash
135138 pts/0 00:00:00 Homework
135139 pts/0 00:00:00 Homework
135258 pts/0 00:00:00 sh
135259 pts/0 00:00:00 ps
子进程 ID: 135139, 父进程 ID: 135138
PID TTY TIME CMD
134364 pts/0 00:00:00 bash
135138 pts/0 00:00:00 Homework
135139 pts/0 00:00:00 Homework
135370 pts/0 00:00:00 sh
135371 pts/0 00:00:00 ps
子进程 ID: 135139, 父进程 ID: 135138
PID TTY TIME CMD
134364 pts/0 00:00:00 bash
135138 pts/0 00:00:00 Homework
135139 pts/0 00:00:00 Homework
135445 pts/0 00:00:00 sh
135446 pts/0 00:00:00 ps
子进程结束，父进程 ID: 135138
```

实验圆满完成！