1)查看整个Linux系统进程之间关系的命令

gec@ubuntu:~s pstree

// 所有的进程的祖先进程都是init，由操作系统镜像启动文件产生

说明:

可以看到，最开始的系统进程叫systemd，这个进程的诞生比较特别，

其身份信息在系统启动前就已经存在于系统分区之中，在系统启动时直接复制到内存。

2)查看进程ID号----》ps -ef (静态)常用P->process

gec@ubuntu:~s ps -ef

用户名 进程ID 父进程 cpu占用 进程启动时问 进程持续时问 进程名

UID PID PPID C STIME TTY TIME CMD

root 1 0 0 18:48 ? 00:00:01 /sbin/init auto noprompt

root 2 0 0 18:48 ? 00:00:00 [kthreadd]

root 4 2 0 18:48 ? 00:00:00 [kworker/0:0H]

root 6 2 0 18:48 ? 00:00:00 [mm\_percpu\_wq]

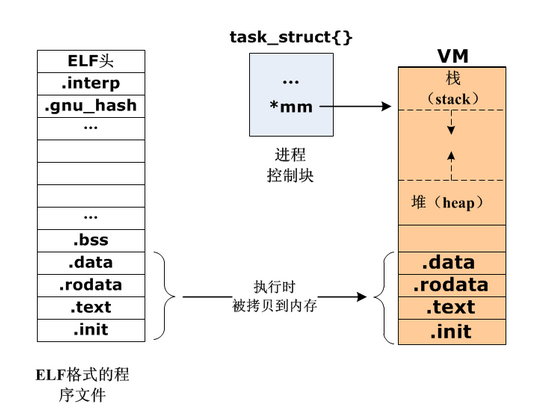
3)查看进程CPU的占用率---》top (动态)

gec@ubuntu:~$ top

程序：编译后产生的，格式为ELF的，存储于硬盘的文件

进程：程序中的代码和数据，被加载到内存中运行的过程

程序是静态的概念，进程是动态的概念



1.进程从诞生到死亡会经历哪些状态?

就绪态 TASK\_RUNNING 等待CPU资源不占用CPU资源，不运行代码。内核调用sched()

运行态 TASK\_RUNNING 占用CPU资源，运行代码。(执行态)

暂停态TASK\_STOPPED 占用CPU资源，不运行代码。可以到就绪态。

睡眠态/挂起态 占用CPU资源，运行代码。可以到就绪态。

TASK\_INTERRUPTIBLE 响应信号----》浅度睡眠 pause()--->—直等待下—信号

TASK\_UNINTERRUPTIBLE不响应信号----》深度睡眠

僵尸态 EXIT\_ZOMBIE 占用CPU资源，不运行代码。不可以到运行态。

进程退出的时候，就一定会变成僵尸态

死亡态 EXIT\_DEAD 不占用CPU资源，不运行代码。进程退出的时候，如果有人去帮自己回收资源，那么僵尸态就会变为死亡态

2.什么是僵尸态?(难点)

进程结束时，就从运行态变成僵尸态，所谓僵尸态，就是代表这个进程所占用的CPU资源和自身的任务结构体没有被释放，这个状态的进程就是僵尸态进程。

3.总结:

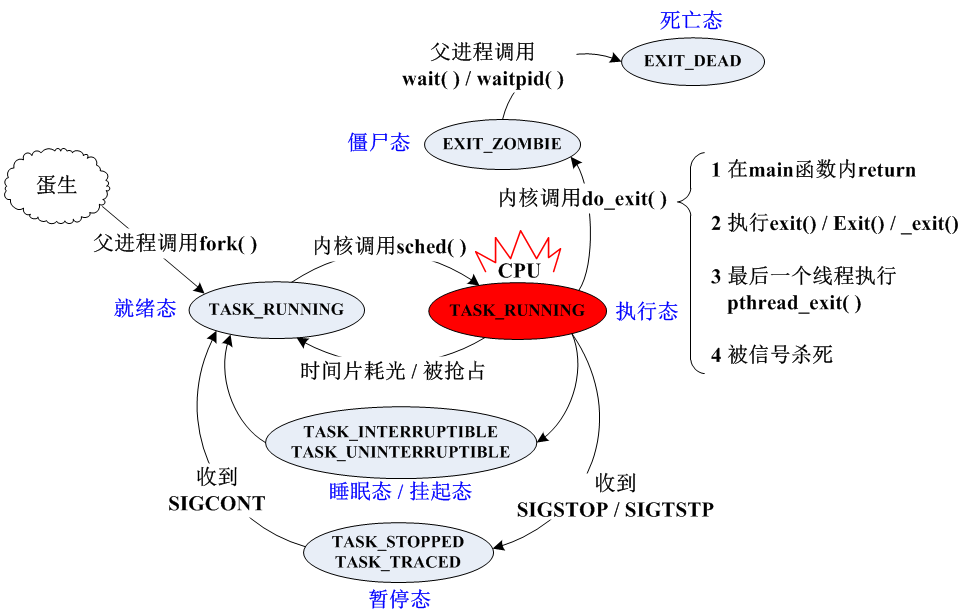
1)进程在暂停态时，收到继续 SIGSTOP / SIGTSTP 的信号时，是切换到就绪态，而不是运行态。(重点)

2）程序的main函数执行return 0就会导致进程的退出，一定会变成儡尸态。

3）进程不可以没有父进程，也不能同时拥有两个父进程。

4）祖先进程一定要帮其他的进程回收资源。

5）孤儿进程特征就是失去父进程时，会马上寻找继父，而不是等到孤儿进程变成僵尸态再找。(孤儿进程和僵尸进程的区别)



所有进程（除了系统初始进程systemd之外），都有一个父进程。

父进程通过调用fork()函数，将自身复制一份形成一个子进程。

新创建的子进程拥有与父进程一样的执行代码、内存空间（父子进程的内存空间的内容是一致的，但分属不同的区域各自独立）等信息，并处于就绪态（TASK\_RUNNING）。

当进程退出时（不管是主动退出还是被动退出），进入僵尸态（EXIT\_ZOMBIE），僵尸态下的进程无法运行，也无法被调度，但其所占据的系统资源未被释放。僵尸态是进程的必经状态，编程过程中不能避免僵尸态，但要避免进程长时间处于僵尸态。

僵尸态进程要等待其父进程对其资源进程回收后，才能变成死亡态（EXIT\_DEAD），死亡态的进程所有占据的系统资源可以被系统随时回收。

**fork函数原型**

pid\_t fork( void);

（pid\_t 是一个[宏定义](https://baike.baidu.com/item/%E5%AE%8F%E5%AE%9A%E4%B9%89?fromModule=lemma_inlink)，其实质是int 被定义在#includesys/types.h>中）

返回值： 若成功调用一次则返回两个值，子进程返回0，[父进程](https://baike.baidu.com/item/%E7%88%B6%E8%BF%9B%E7%A8%8B?fromModule=lemma_inlink)返回子进程ID；否则，出错返回-1

int main()

{

    fork();

    printf("hello world.\n");

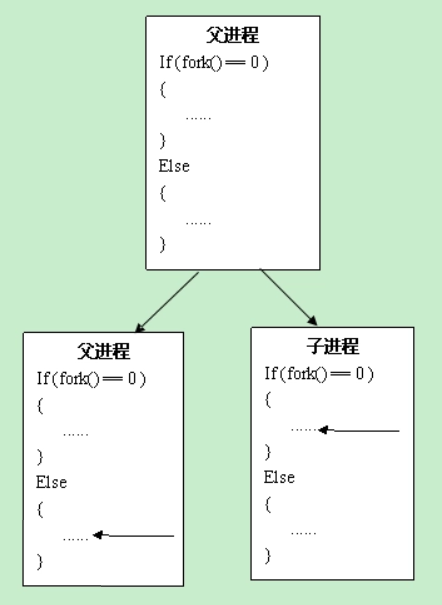
    return 0;

}

注意:只有父进程退出，才会出现命令行，子进程退出是不会出现命令行。

**提问：为什么fork会返回两次？？？**

由于在复制时复制了[父进程](https://baike.baidu.com/item/%E7%88%B6%E8%BF%9B%E7%A8%8B?fromModule=lemma_inlink)的[堆栈段](https://baike.baidu.com/item/%E5%A0%86%E6%A0%88%E6%AE%B5?fromModule=lemma_inlink)，所以两个进程都停留在fork函数中，等待返回。因此fork函数会返回两次，一次是在父进程中返回，另一次是在子进程中返回，这两次的返回值是不一样的。过程如下图。



调用fork之后，数据、堆、栈有两份，代码仍然为一份但是这个[代码段](https://baike.baidu.com/item/%E4%BB%A3%E7%A0%81%E6%AE%B5?fromModule=lemma_inlink)成为两个进程的共享代码段都从fork函数中返回，箭头表示各自的执行处。当父子进程有一个想要修改数据或者堆栈时，两个进程真正分裂。

总结：

子进程不共享父进程的存储空间是因为在创建子进程时，操作系统会为其分配新的存储空间，包括数据空间、堆和栈。这样做是为了保证父进程和子进程可以并发执行而不相互干扰。

当父进程创建子进程时，它会将自己的存储空间的副本复制给子进程，但这个副本是独立的，与父进程的存储空间完全隔离。这样做的好处是，父进程和子进程可以同时运行并独立地进行各自的操作，而不会相互影响。

子进程的存储空间副本是通过进程间的复制机制实现的。操作系统会为子进程创建新的页表，将父进程的虚拟地址映射到新分配的物理内存上。这样，子进程就可以在自己的存储空间中进行操作，而不会对父进程的存储空间造成任何影响。

总之，子进程不共享父进程的存储空间是为了确保进程之间的隔离性和并发执行的能力。每个进程都拥有自己独立的存储空间，以便进行独立的操作和数据管理。

**获取进程的PID**

#include <unistd.h>

#include <assert.h>

int main(int argc, char argv[])

{

    pid\_t pid = fork();

    assert(pid != -1); // 如果触发断言，则下面这行代码不会被执行

    // 在父进程中，pid将是子进程的PID

    // 在子进程中，pid将是0

    printf("[%d]:Process pid = %d.\n", getpid(), pid); // 函数 getpid() 的功能是获取自身进程的PID

    return 0;

}

**僵尸进程**

僵尸进程（Zombie Process）是指一个子进程已经终止，但是其父进程还没有调用wait()或waitpid()等系统调用来获取该子进程的退出状态，

导致子进程的进程描述符仍然保留在系统进程表中，并且在进程表中的部分信息（如进程ID、退出状态等）仍然可见。

僵尸进程不会占用实际的内存空间，但会占用一个唯一的进程ID。

僵尸进程通常发生在一个子进程比父进程先退出的情况下。父进程应该负责回收子进程的资源和获取其退出状态，

但如果父进程没有处理僵尸进程，那么子进程就会成为僵尸进程。

void create\_zombie() {

    pid\_t pid = fork();

    if (pid == 0) {

        // 子进程

        printf("子进程正在执行\n");

        sleep(2);

        printf("子进程执行完毕\n");

    } else if (pid > 0) {

        // 父进程

        printf("父进程创建了子进程（PID: %d）\n", pid);

        sleep(5);  // 父进程暂停一段时间，不回收子进程资源

        // 此处应该调用 wait() 或 waitpid() 来获取子进程的退出状态，

        // 但在示例中省略了这一步，导致子进程成为僵尸进程

    } else {

        // fork() 失败

        fprintf(stderr, "无法创建子进程\n");

        exit(1);

    }

}

int main() {

    create\_zombie();// 创建僵尸进程

    return 0;

}

使用命令

ps-ef| grep zombie\_example 查看进程状态

UID        PID  PPID  C STIME TTY          TIME CMD

user     1234  5678  0 08:00 pts/0    00:00:00 ./zombie\_example

user     1235  1234  0 08:00 pts/0    00:00:00 [zombie\_example] <defunct>

<defunct> 僵尸进程的标志

**释放僵尸进程**

**3.1 方法一：父进程直接退出**

这应该不算什么方法，父进程的退出只是便于让系统中的孤儿进程组收养其孤儿进程，进而在该孤儿进程编程僵尸后由系统负责回收并释放其系统资源。

**3.2 方法二：子进程等待父进程对其执行wait()/waitpid()**

这是最浅显的做法，wait()/waitpid()函数有如下三个功效：

释放对应僵尸子进程的系统资源

获取对应僵尸子进程的退出状态

阻塞父进程（可选）

上述功效中的第一项即可满足我们目前的需求，比如在如上代码中，只要父进程代码做如下修改即可避免僵尸的产生：

int main(){  
 // 子进程退出（变僵尸）if(fork() == 0)  
 return0;

// 父进程调用wait()释放子进程资源wait(NULL);  
 pause();  
 return0;  
}

此办法看似简单，但并不实用，因为一般而言父进程很难预知子进程退出的时机，当父进程执行wait()/waitpid()的时候，要么进入漫长的等待，要么子进程尚未变僵尸，因此让父进程去时刻主动关注子进程的状态和回收资源是不切实际的，而且父进程的后代进程可能不止一个，大家任务各有长短，父进程本身也可能处于循环任务之中，因此该办法仅供参考。

**3.3 方法三：子进程主动告知父进程前来收尸**

这是最常见的做法，让变成僵尸的子进程主动通知其父进程，既省却了父进程监控子进程的麻烦，又可以让子进程的僵尸状态尽可能保证在最短时间内处理。

具体而言，子进程在进入僵尸态时，会自动向父进程发送信号SIGCHILD，而父进程可以利用异步信号响应函数来及时处理这些僵尸子进程。参考代码如下：

void cleanup(intsig){  
 // 僵尸子进程会被自动清除wait(NULL);  
}

int main(){  
 // 在产生子进程之前，准备好处理它们的SIGCHILD信号signal(SIGCHLD, cleanup);

// 子进程退出，成为僵尸进程if(fork() == 0)  
 return0;

// 父进程干自己的活，无需关注子进程while(1)  
 pause();

return0;  
}

注意，由于pause()函数会在收到信号后退出，为了能查看父进程在清除子进程资源后仍在运行，上述代码中使用了循环来执行pause()函数。

**孤儿进程**

孤儿进程是指在父进程结束或被终止后，子进程仍然在运行的进程。这些子进程将被操作系统的init进程（PID为1）接管，并由它来完成子进程的资源回收。

void create\_orphan() {

    pid\_t pid = fork();

    if (pid == 0) {

        // 子进程

        printf("子进程开始执行\n");

        printf("子进程的父进程ID：%d\n", getppid());

        sleep(5);

        printf("子进程执行完毕\n");

    } else if (pid > 0) {

        // 父进程

        printf("父进程开始执行\n");

        printf("父进程ID：%d\n", getpid());

        sleep(2);

        printf("父进程执行完毕\n");

        // 父进程终止，子进程成为孤儿进程

    } else {

        // fork() 失败

        fprintf(stderr, "无法创建子进程\n");

        exit(1);

    }

}

int main() {

    create\_orphan();

    return 0;

}

使用命令

ps-ef| grep orphan\_example

UID        PID  PPID  C STIME TTY          TIME CMD

user     1234     1  0 08:00 pts/0    00:00:00 ./orphan\_example

user     5678  1234  0 08:00 pts/0    00:00:00 [orphan\_example]

**回收进程**

函数原型

    #include <sys/types.h>

    #include <sys/wait.h>

    pid\_t wait(int \*wstatus);

主要功能：

    阻塞当前进程。

    等待其子进程退出并回收其系统资源

接口解析：

    如果当前进程没有子进程，则该函数立即返回。

    如果当前进程有不止1个子进程，则该函数会回收第一个变成僵尸态的子进程的系统资源。

    子进程的退出状态（包括退出值、终止信号等）将被放入wstatus所指示的内存中，

    若wstatus指针为NULL，则代表当前进程放弃其子进程的退出状态。

    参数可以设为以下两种情况之一：

    1.int\* wstatus：传递一个指向整型变量的指针，用于存储子进程的退出状态。父进程将通过这个指针获取子进程的退出状态信息。

    2.NULL：表示父进程放弃获取子进程的退出状态，不关心子进程的结束状态和相关信息。

    需要注意的是，wait()函数在等待子进程结束时会阻塞父进程，直到有一个子进程结束或被信号中断。

    如果父进程有多个子进程，可以通过多次调用wait()函数来依次处理每个子进程的退出状态。

#include <unistd.h>

#include <assert.h>

#include <sys/types.h>

#include <sys/wait.h>

int main(int argc, char argv[])

{

    pid\_t pid = fork();

    assert(pid != -1); // 如果触发断言，则下面这行代码不会被执行

    // 在父进程中，pid将是子进程的PID

    // 在子进程中，pid将是0

    // 加上如下语句，确保子进程先退出

    wait(NULL);

    printf("[%d]:Process pid = %d.\n", getpid(), pid); // 函数 getpid() 的功能是获取自身进程的PID

    return 0;

}

|  |  |
| --- | --- |
| **宏** | **功能** |
| WIFEXITED(status) | 判断子进程是否正常退出 |
| WEXITSTATUS(status) | 获取正常退出的子进程的退出值 |
| WIFSIGNALED(status) | 判断子进程是否被信号杀死 |
| WTERMSIG(status) | 获取杀死子进程的信号的值 |

回收僵尸子进程资源除了可以使用上述接口之外，以下函数接口也经常被用到：

#include <sys/types.h>

#include <sys/wait.h>

pid\_t waitpid(pid\_t pid, int \*wstatus, int options);

pid：要等待的进程ID。有几种特殊取值：

     -1：等待任意子进程；

     0：等待与调用进程在同一个进程组的任意子进程；

     正整数：等待进程ID等于pid的子进程。

wstatus：保存进程状态信息的指针，如果不关心进程的终止状态可以传入NULL。

options：选项，用于指定等待的类型和行为。

与wait()的区别：

可以通过参数 pid 用来指定想要回收的子进程。

可以通过 options 来指定非阻塞等待。

|  |  |  |  |
| --- | --- | --- | --- |
| **pid** | **作用** | **options** | **作用** |
| <-1 | 等待组ID等于pid绝对值的进程组中的任意一个子进程 | 0 | 阻塞等待子进程的退出 |
| -1 | 等待任意一个子进程 | WNOHANG | 若没有僵尸子进程，则函数立即返回 |
| 0 | 等待本进程所在的进程组中的任意一个子进程 | WUNTRACED | 当子进程暂停时函数返回 |
| >0 | 等待指定pid的子进程 | WCONTINUED | 当子进程收到信号SIGCONT继续运行时函数返回 |

**注意：**

options的取值，可以是0，也可以是上表中各个不同的宏的位或运算取值。

int main()

{

    pid\_t child\_pid;

    int status;

    // 创建子进程

    child\_pid = fork();

    if (child\_pid == 0)

    {

        // 在子进程中执行任务

        printf("Child process: PID=%d\n", getpid());

        sleep(2);

        exit(123); // 子进程退出，返回状态码123

    }

    else if (child\_pid > 0)

    {

        // 在父进程中等待子进程结束

        printf("Parent process: Child PID=%d\n", child\_pid);

        // 使用waitpid等待子进程

        waitpid(child\_pid, &status, 0);

        if (WIFEXITED(status))

        {

            // 子进程正常终止

            printf("Child process exited with status: %d\n", WEXITSTATUS(status));

        }

        else if (WIFSIGNALED(status))

        {

            // 子进程被信号终止

            printf("Child process terminated by signal: %d\n", WTERMSIG(status));

        }

    }

    else

    {

        perror("fork error");

        exit(1);

    }

    return 0;

}