

# 进程退出

进程退出函数没有返回值，他的参数是退出状态，会给到父进程，父进程可以获取到。

```
#include <stdlib.h>
void exit(int status);    //标准C库

#include <unistd.h>
void _exit(int status);   //Linux系统调用
```

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main()
{
    //标准输出是行缓冲的，所以遇到换行符就会把用户区的缓冲刷到内核中。
    printf("Hello\n");
    printf("World");

    //_exit(int)该函数是Linux系统调用中的函数，用于终止进程。
    _exit(0);

    // exit(int)函数是标准C库的函数，内部也会调用_exit函数，但是在此之前会执行刷新缓冲IO、关闭文件描述符等等操作。
    // exit(0);

    //如果第三个语句执行_exit函数，那么第二句是打印不出来的。而执行exit函数就可以打印出来。

    return 0;
}
```

# 孤儿进程

## 概述

- 父进程运行结束，但子进程还在运行（未运行结束），这样的子进程就称为孤儿进程（Orphan Process）。
- 每当出现一个孤儿进程的时候，内核就把孤儿进程的父进程设置为 init（1号进程），而 init 进程会循环地 wait() 它的已经退出的子进程。这样，当一个孤儿进程凄凉地结束了其生命周期的时候，init 进程就会代表党和政府出面处理它的一切善后工作。
- 因此孤儿进程并不会有什么危害。

## 实例

```
#include <unistd.h>
#include <stdio.h>

int main()
{
    pid_t pid = fork();
```

```

if (pid > 0)
{
    //父进程
    printf("this is parent process \n");
}
else if (pid == 0)
{
    //子进程
    for (int i = 0; i < 3; i++)
    {
        printf("你好 --> %d\n", i);
    }
    printf("子进程 %d --> 的父进程 %d\n", getpid(), getppid());
}
}

```

两种结果:

```

kiko@kiko:~/lesson/myprocess$ ./Orphan
你好 --> 0
你好 --> 1
你好 --> 2
子进程 85109 --> 的父进程 85108
this is parent process 85108
kiko@kiko:~/lesson/myprocess$ ./Orphan
this is parent process 85110
你好 --> 0
你好 --> 1
你好 --> 2
子进程 85111 --> 的父进程 1

```

CSDN @C学习者n号

## 僵尸进程

### 概述

- 每个进程结束之后, 都会释放自己地址空间中的用户区数据, 内核区的 PCB 没有办法自己释放掉, 需要父进程去释放。
  - 子进程终止时, 父进程尚未回收, 子进程残留资源 (PCB) 存放于内核中, 变成僵尸 (Zombie) 进程。
  - 僵尸进程不能被 kill -9 杀死。这样就会导致一个问题, 如果父进程不调用 wait() 或 waitpid() 的话, 那么保留的那段信息就不会释放, 其进程号就会一直被占用, 但是系统所能使用的进程号是有限的, 如果大量的产生僵尸进程, 将因为没有可用的进程号而导致系统不能产生新的进程, 此即为僵尸进程的危害, 应当避免。
  - 僵尸进程的产生一般是: 父进程一直在执行, 基本不停止, 但是内部并没有调用wait或者waitpid函数对终止的子进程回收, 此时终止的子进程就是僵尸进程。
- 如果父进程不会执行很久, 很快也结束了, 但是里面也没有调用wait或者waitpid函数, 那么即使产生了僵尸进程, 最终init进程也会将僵尸进程清理。

### 实例

```

#include <unistd.h>
#include <stdio.h>

int main()
{
    // 创建子进程

```

```

pid_t pid = fork();

// 判断是父进程还是子进程
if (pid > 0)
{
    while (1)
    {
        printf("i am parent process, pid : %d, ppid : %d\n", getpid(),
getppid());
        sleep(2);
    }
}
else if (pid == 0)
{
    // 当前是子进程
    printf("i am child process, pid : %d, ppid : %d\n", getpid(),
getppid());
}

return 0;
}

```

执行结果如下，父进程一直在执行，子进程执行完就结束了。

```

kiko@Hkiko:~/lesson/myprocess$ ./zombie
i am parent process, pid : 85239, ppid : 84274
i am child process, pid : 85240, ppid : 85239
i am parent process, pid : 85239, ppid : 84274
i am parent process, pid : 85239, ppid : 84274
i am parent process, pid : 85239, ppid : 84274
i am parent process, pid : 85239, ppid : 84274
i am parent process, pid : 85239, ppid : 84274
i am parent process, pid : 85239, ppid : 84274
i am parent process, pid : 85239, ppid : 84274
i am parent process, pid : 85239, ppid : 84274

```

CSDN @C学习者n号

根据打印的结果，子进程的pid是85240。此时查看系统的进程 `ps -aux`

```

root      84587  0.6  0.0      0      0 ?        D   18:50   0:18 [kworker/2:1+usb_hub_wq]
root      84868  0.0  0.0      0      0 ?        I   19:14   0:00 [kworker/u256:1-events_unbound]
root      85151  0.0  0.0      0      0 ?        I   19:23   0:00 [kworker/3:0-events]
root      85153  0.0  0.0      0      0 ?        I   19:25   0:00 [kworker/u256:0-events_unbound]
root      85163  0.0  0.0      0      0 ?        I   19:31   0:00 [kworker/2:2-events]
root      85169  0.0  0.0      0      0 ?        I   19:37   0:00 [kworker/2:0-events]
root      85174  0.0  0.0      0      0 ?        I   19:37   0:00 [kworker/1:1-events]
root      85188  0.0  0.0      0      0 ?        I   19:38   0:00 [kworker/u256:2-events_unbound]
kiko      85227  0.1  0.1 4950584 13604 ?        Sl   19:38   0:00 /home/kiko/.vscode-server/extensions/ms-vscode.c
kiko      85239  0.0  0.0    2496    580 pts/5    S+   19:39   0:00 ./zombie
kiko      85240  0.0  0.0      0      0 pts/5    Z+   19:39   0:00 [zombie] <defunct>
kiko      85252  0.0  0.0   11160    576 ?        S    19:39   0:00 sleep 180
kiko      85350  0.0  0.0   14776   3500 pts/0    R+   19:39   0:00 ps -aux
kiko@Hkiko: $

```

CSDN @C学习者n号

## 进程回收

### 概述

进程回收主要是 `wait` 和 `waitpid` 函数。

所有这些系统调用都用于等待调用进程的子进程的状态变化，并获取有关其状态已更改的子进程的信息。状态变化被认为是：孩子终止；孩子被信号拦住；或者孩子被一个信号恢复了。在终止子进程的情况下，执行等待允许系统释放与子关联的资源。

如果一个孩子已经改变了状态，那么这些调用会立即返回。否则，它们会阻塞直到子进程改变状态或信号处理程序中断调用。

注意：一次 `wait` 或 `waitpid` 调用只能清理一个子进程，清理多个子进程应使用循环。

# wait函数

## 函数介绍

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *wstatus);
```

wait() 系统调用暂停调用线程的执行，直到其子线程之一终止。

参数：用于获取进程结束的状态码，是int值的地址。一个输出参数。

返回值：成功时，返回终止子进程的进程ID；所有的子进程都结束或者调用函数失败，返回 -1。

## 实例1

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
    //创建5个进程
    pid_t pid;
    for (size_t i = 0; i < 5; i++)
    {
        pid = fork();
        if (pid == 0)
            break;
    }

    if (pid > 0)
    {
        //父进程
        while (1)
        {
            printf("parent, pid = %d\n", getpid());

            //会阻塞，等待回收子进程
            int ret = wait(NULL); // NULL参数表示不需要获取进程结束的状态码

            printf("child die, pid = %d\n", ret);
        }
    }
    else if (pid == 0)
    {
        //子进程
        while (1)
        {
            printf("child, pid = %d\n", getpid());
            sleep(1);
        }
    }
}
```

```

return 0; // exit(0)
}

```

```

kiko 96837 0.0 0.0 11160 580 ? S 20:51 0:00 sleep 180
kiko 96838 0.0 0.0 2496 512 pts/5 S+ 20:51 0:00 ./mywait
kiko 96839 0.0 0.0 2496 72 pts/5 S+ 20:51 0:00 ./mywait
kiko 96840 0.0 0.0 2496 72 pts/5 S+ 20:51 0:00 ./mywait
kiko 96841 0.0 0.0 2496 72 pts/5 S+ 20:51 0:00 ./mywait
kiko 96842 0.0 0.0 2496 72 pts/5 S+ 20:51 0:00 ./mywait
kiko 96843 0.0 0.0 2496 72 pts/5 S+ 20:51 0:00 ./mywait
kiko 96906 2.5 0.2 364344 22616 ? Ss1 20:51 0:00 /usr/libexec/tracker-store
kiko 96940 0.0 0.0 14776 3580 pts/0 R+ 20:51 0:00 ps -aux
kiko@Hkiko: $
39 }
问题 输出 调试控制台 终端
child, pid = 96841
child, pid = 96839
child, pid = 96842
child die, pid = 96840
parent, pid = 96838
child, pid = 96843
child, pid = 96841
child, pid = 96839
kiko 96837 0.0 0.0 11160 580 ? S 20:51 0:00 sleep 180
kiko 96838 0.0 0.0 2496 512 pts/5 S+ 20:51 0:00 ./mywait
kiko 96839 0.0 0.0 2496 72 pts/5 S+ 20:51 0:00 ./mywait
kiko 96840 0.0 0.0 2496 72 pts/5 S+ 20:51 0:00 ./mywait
kiko 96841 0.0 0.0 2496 72 pts/5 S+ 20:51 0:00 ./mywait
kiko 96842 0.0 0.0 2496 72 pts/5 S+ 20:51 0:00 ./mywait
kiko 96843 0.0 0.0 2496 72 pts/5 S+ 20:51 0:00 ./mywait
kiko 96906 2.5 0.2 364344 22616 ? Ss1 20:51 0:00 /usr/libexec/tracker-store
kiko 96940 0.0 0.0 14776 3580 pts/0 R+ 20:51 0:00 ps -aux
kiko@Hkiko: $ kill -9 96840
kiko@Hkiko: $
CSDN @C学习者n号
CSDN @C学习者n号

```

## 实例2

```

#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
    //创建5个进程
    pid_t pid;
    for (size_t i = 0; i < 5; i++)
    {
        pid = fork();
        if (pid == 0)
            break;
    }

    if (pid > 0)
    {
        //父进程
        while (1)
        {
            printf("parent, pid = %d\n", getpid());

            //会阻塞，等待回收子进程
            int status;
            int ret = wait(&status); // NULL参数表示不需要获取进程结束的状态码

            printf("child die, pid = %d --- status = %d\n", ret, status);

            if (ret == -1)
                break;
        }
    }
    else if (pid == 0)
    {
        //子进程
    }
}

```

```

        // while (1)
        // {
        printf("child, pid = %d\n", getpid());
        // sleep(3);
        exit(1);
        // }
    }

    return 0; // exit(0)
}

```

```

kiko@Hkiko:~/lesson/myprocess$ ./mywait
child, pid = 99562
parent, pid = 99561
child, pid = 99563
child die, pid = 99562 --- status = 256
parent, pid = 99561
child, pid = 99566
child die, pid = 99563 --- status = 256
parent, pid = 99561
child die, pid = 99566 --- status = 256
parent, pid = 99561
child, pid = 99565
child die, pid = 99565 --- status = 256
parent, pid = 99561
child, pid = 99564
child die, pid = 99564 --- status = 256
parent, pid = 99561
child die, pid = -1 --- status = 256

```

CSDN @C学习者n号

### 实例3

```

#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
    //创建5个进程
    pid_t pid;
    for (size_t i = 0; i < 5; i++)
    {
        pid = fork();
        if (pid == 0)
            break;
    }

    if (pid > 0)
    {
        //父进程
        while (1)
        {
            printf("parent, pid = %d\n", getpid());

            //会阻塞，等待回收子进程
            int status;

```

```

        int ret = wait(&status); // NULL参数表示不需要获取进程结束的状态码

        printf("child die, pid = %d --- status = %d\n", ret, status);

        if (ret == -1)
            break;
    }
}
else if (pid == 0)
{
    //子进程
    while (1)
    {
        printf("child, pid = %d\n", getpid());
        sleep(3);
        // exit(1);
    }
}

return 0; // exit(0)
}

```

```

1 #include <sys/types.h>
2 #include <sys/wait.h>
3 #include <stdio.h>
4 #include <unistd.h>
5 #include <stdlib.h>
6
7 int main()
8 {
9     //创建5个进程
10    pid_t pid;
11    for (size_t i = 0; i < 5; i++)
12    {
13        pid = fork();
14        if (pid == 0)
15            break;
16    }
17
18    if (pid > 0)
19    {

```

kiko@Hkiko: ~/lesson/myprocess

root	101018	0.4	0.0	0	0	?	I	18:00
root	101020	0.0	0.0	0	0	?	I	18:00
root	101021	0.0	0.0	0	0	?	I	18:00
root	101307	0.0	0.1	14008	9088	?	Ss	18:07
kiko	101384	0.7	0.0	14008	6100	?	S	18:07
kiko	101386	0.0	0.0	12708	3812	?	Ss	18:07
kiko	101431	0.0	0.0	2616	528	?	S	18:07
kiko	101441	10.9	0.9	936072	75140	?	S1	18:07
kiko	101483	0.0	0.0	11160	580	?	S	18:07
kiko	101484	1.8	0.6	867960	54636	?	S1	18:07
kiko	101512	7.7	1.6	931824	129828	?	S1	18:07
kiko	101524	0.4	0.5	834032	41004	?	S1	18:07
root	101537	0.1	0.0	0	0	?	R	18:07
kiko	101570	0.1	0.0	13948	5420	pts/4	Ss	18:07
kiko	101594	1.0	0.4	1196312	35236	?	S1	18:07
kiko	101630	0.4	0.4	588904	40356	?	S1	18:07
kiko	101644	0.1	0.1	4950584	16148	?	S1	18:07
kiko	101777	0.0	0.0	2496	516	pts/4	S+	18:07
kiko	101778	0.0	0.0	2496	76	pts/4	S+	18:07
kiko	101779	0.0	0.0	2496	76	pts/4	S+	18:07
kiko	101780	0.0	0.0	2496	76	pts/4	S+	18:07
kiko	101781	0.0	0.0	2496	76	pts/4	S+	18:07
kiko	101782	0.0	0.0	2496	76	pts/4	S+	18:07
root	101813	0.0	0.0	0	0	?	I	18:08
root	101904	0.0	0.1	14012	9168	?	Ss	18:08
kiko	101988	0.4	0.0	14012	5848	?	R	18:08
kiko	101989	0.2	0.0	13976	5316	pts/5	Ss	18:08
kiko	102180	0.0	0.0	14776	3592	pts/5	R+	18:08

child die, pid = 101781 --- status = 9

parent, pid = 101777

child, pid = 101779

child, pid = 101782

kiko@Hkiko: ~/lesson/myprocess\$ kill -9 101781

kiko@Hkiko: ~/lesson/myprocess\$

CSDN @C学习者n号

## waitpid函数

## 概述

### 函数原型

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *wstatus, int options);
```

功能：回收指定进程号的子进程，可以设置是否阻塞，默认是阻塞的。

参数：

```
pid
    pid > 0 : 某个子进程的pid
    pid = 0 : 回收当前进程组的所有子进程
    pid = -1 : 回收所有的子进程，相当于 wait() （最常用）
    pid < -1 : 某个进程组的组id的绝对值，回收指定进程组中的子进程
options
    0 : 阻塞
    WNOHANG : 非阻塞
wstatus
    输出参数，用于获取终止进程的状态。和wait函数一样。
```

### 返回值

```
> 0 : 返回被终止的子进程的id
= 0 : 如果 options=WNOHANG，表示还有子进程未终止
= -1 : 错误，或者没有子进程了
```

### wait和waitpid函数

```
wait(&wstatus)
等价于：
waitpid(-1, &wstatus, 0);
```

## 实例1

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
    //先创建5个进程
    pid_t pid;
    for (size_t i = 0; i < 5; i++)
    {
        pid = fork();
        if (pid == 0)
            break;
    }
}
```



```

if (pid > 0)
{
    //父进程
    while (1)
    {
        printf("parent, pid = %d\n", getpid());

        int status;
        int ret = waitpid(-1, &status, 0); // 0, 阻塞; -1, 清理所有子进程

        printf("child die, pid = %d --- status = %d\n", ret, status);

        if (ret == -1)
            break;
    }
}
else if (pid == 0)
{
    //子进程
    while (1)
    {
        printf("child, pid = %d\n", getpid());
        sleep(3);
        // exit(1);
    }
}

return 0; // exit(0)
}

```

```

34  ...else if (pid == 0)
35  ...{
36  ...//子进程
37  ...while (1)
38  ...{
    kiko 102755 0.0 0.0 11160 584 ? S 20:28 0:00 sleep 180
    root 102759 0.0 0.0 0 0 ? I 20:28 0:00 [kworker/u256:1-events_unbou
    kiko 102814 0.2 0.1 4950584 15280 ? S1 20:28 0:00 /home/kiko/.vscode-server/ex
    kiko 102835 0.0 0.0 2496 576 pts/4 S+ 20:28 0:00 ./mywaitpid
    kiko 102836 0.0 0.0 2496 72 pts/4 S+ 20:28 0:00 ./mywaitpid
    kiko 102837 0.0 0.0 2496 72 pts/4 S+ 20:28 0:00 ./mywaitpid
    kiko 102838 0.0 0.0 2496 72 pts/4 S+ 20:28 0:00 ./mywaitpid
    kiko 102839 0.0 0.0 2496 72 pts/4 S+ 20:28 0:00 ./mywaitpid
    kiko 102840 0.0 0.0 2496 72 pts/4 S+ 20:28 0:00 ./mywaitpid
    root 102904 0.0 0.0 0 0 ? I 20:28 0:00 [kworker/2:2-events]
    kiko 102920 1.0 0.2 364340 22436 ? Ssl 20:28 0:00 /usr/libexec/tracker-store
    kiko 102971 0.0 0.0 14776 3588 pts/5 R+ 20:28 0:00 ps -aux
    kiko@Hkiko: /lesson/myprocess$ ps -9 102839
    PID TTY STAT TIME COMMAND
    102839 pts/4 S+ 0:00 ./mywaitpid
    kiko@Hkiko: /lesson/myprocess$ kill -9 102839
    kiko@Hkiko: /lesson/myprocess$
    child, pid = 102838
    child, pid = 102840
    child, pid = 102837
    child, pid = 102836
    child, pid = 102839
    child, pid = 102838
    child, pid = 102840
    child die, pid = 102839 --- status = 9
    parent, pid = 102835
    child, pid = 102837

```

## 实例2

```

#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
    //先创建5个进程
    pid_t pid;
    for (size_t i = 0; i < 5; i++)
    {
        pid = fork();
        if (pid == 0)

```

```

        break;
    }

    if (pid > 0)
    {
        //父进程
        while (1)
        {
            printf("parent, pid = %d\n", getpid());
            sleep(2);

            int status;
            int ret = waitpid(-1, &status, WNOHANG); // WNOHANG, 阻塞; -1, 清理所有
子进程

            printf("child die, pid = %d --- status = %d\n", ret, status);

            if (ret == -1)
                break;
        }
    }
    else if (pid == 0)
    {
        //子进程
        while (1)
        {
            printf("child, pid = %d\n", getpid());
            sleep(3);
            // exit(1);
        }
    }

    return 0; // exit(0)
}

```

父进程不会被阻塞。如果没有被杀死的进程，waitpid返回0，获取的状态也是0。

```

kiko@hkiko:~/lesson/myprocess$ ./mywaitpid2
parent, pid = 104212
child, pid = 104216
child, pid = 104215
child, pid = 104217
child, pid = 104214
child, pid = 104213
child die, pid = 0 --- status = 0
parent, pid = 104212
child, pid = 104216
child, pid = 104217
child, pid = 104214
child, pid = 104213

```

CSDN @C学习者n号