**四、避免僵死进程的方法**

网上搜了下，总结有三种方方法：

① 程序中显示的调用signal(SIGCHLD, SIG\_IGN)来忽略SIGCHLD信号，这样子进程结束后，由内核来wai和释放资源

② fork两次，第一次fork的子进程在fork完成后直接退出，这样第二次fork得到的子进程就没有爸爸了，它会自动被老祖宗init收养，init会负责释放它的资源，这样就不会有“僵尸”产生了

③ 对子进程进行wait，释放它们的资源，但是父进程一般没工夫在那里守着，等着子进程的退出，所以，一般使用信号的方式来处理，在收到SIGCHLD信号的时候，在信号处理函数中调用wait操作来释放他们的资源。

**五、对每个避免僵死进程方法的解析与总结**

首先我们让我们来看一个生成僵尸进程的程序zombie.c如下：

#include <stdio.h>    
#include <stdlib.h>    
#include <unistd.h>    
    
int main(int argc, const char \*argv[])    
{    
    int i;    
    pid\_t pid;    
    
    for (i = 0; i < 10; i++) {    
        if ((pid = fork()) == 0)    /\* child \*/    
            \_exit(0);    
    }    
    sleep(10);    
    
    exit(EXIT\_SUCCESS);    
}

运行程序，在10s睡眠期间使用ps查看进程，你会发现有10个标记为“defunct”的僵尸进程：

图形用户界面

描述已自动生成

接下来看第一种方法，程序avoid\_zombie1.c如下:

#include <stdio.h>    
#include <stdlib.h>    
#include <signal.h>    
#include <unistd.h>    
#include <errno.h>    
    
int main(int argc, const char \*argv[])    
{    
    pid\_t pid;    
    
    if (SIG\_ERR == signal(SIGCHLD, SIG\_IGN)) {    
        perror("signal error");    
        \_exit(EXIT\_FAILURE);    
    }    
    
    while (1) {    
        if ((pid = fork()) == 0)    /\* child \*/    
            \_exit(0);    
    }    
    
    exit(EXIT\_SUCCESS);    
}

程序运行期间通过ps命令的确没有发现僵尸进程的存在。

在man文档中有这段话：

Note that even though the default disposition of SIGCHLD is "ignore", explicitly setting the disposition to SIG\_IGN results in different treatment of zombie process children.

意思是说尽管系统对信号SIGCHLD的默认处理就是“ignore”，但是显示的设置成SIG\_IGN的处理方式在在这里会表现不同的处理方式（即子进程结束后，资源由系统自动收回，所以不会产生僵尸进程），这是信号SIGCHLD与其他信号的不同之处。

在man文档中同样有这样一段话：

The original POSIX standard left the behavior of setting SIGCHLD to SIG\_IGN unspecified. 看来这个方法不是每个平台都使用，尤其在一些老的系统中，兼容性不是很好，所以如果你在写一个可移植的程序的话，不推荐使用这个方法。

第二种方法，即通过两次fork来避免僵尸进程，我们来看一个例子avoid\_zombie2.c：

#include <stdio.h>    
#include <stdlib.h>    
#include <signal.h>    
#include <unistd.h>    
#include <errno.h>    
    
int main(int argc, const char \*argv[])    
{    
    pid\_t pid;    
    
    while (1) {    
        if ((pid = fork()) == 0) {  /\* child \*/    
            if ((pid = fork()) > 0)    
                \_exit(0);    
            sleep(1);    
            printf("grandchild, parent id = %ld\n",    
                            (long)getppid());    
            \_exit(0);    
        }    
        if (waitpid(-1, NULL, 0) != pid) {    
            perror("waitpid error");    
            \_exit(EXIT\_FAILURE);    
        }    
    }    
    
    exit(EXIT\_SUCCESS);    
}

这的确是个有效的办法，但是我想这个方法不适宜网络并发服务器中，应为fork的效率是不高的。

最后来看第三种方法， 也是最通用的方法

先看我们的测试程序avoid\_zombie3.c

#include <stdio.h>    
#include <stdlib.h>    
#include <errno.h>    
#include <string.h>    
#include <libgen.h>    
#include <signal.h>    
#include <unistd.h>    
#include <sys/wait.h>    
#include <sys/types.h>    
    
    
void avoid\_zombies\_handler(int signo)    
{    
    pid\_t pid;    
    int exit\_status;    
    int saved\_errno = errno;    
    
    while ((pid = waitpid(-1, &exit\_status, WNOHANG)) > 0) {    
        /\* do nothing \*/    
    }    
    
    errno = saved\_errno;    
}    
    
int main(int argc, char \*argv[])    
{    
    pid\_t pid;    
    int status;    
    struct sigaction child\_act;     
    
    memset(&child\_act, 0, sizeof(struct sigaction));    
    child\_act.sa\_handler = avoid\_zombies\_handler;    
    child\_act.sa\_flags = SA\_RESTART | SA\_NOCLDSTOP;     
    sigemptyset(&child\_act.sa\_mask);    
    if (sigaction(SIGCHLD, &child\_act, NULL) == -1) {    
        perror("sigaction error");    
        \_exit(EXIT\_FAILURE);    
    }    
    
    while (1) {    
        if ((pid = fork()) == 0) {  /\* child process \*/    
            \_exit(0);    
        } else if (pid > 0) {        /\* parent process \*/    
        }    
    }    
        
    \_exit(EXIT\_SUCCESS);    
}

首先需要知道三点：

1. 当某个信号的信号处理函数被调用时，该信号会被操作系统阻塞（默认sa\_flags不设置SA\_NODEFER标志）。

2.当某个信号的信号处理函数被调用时，该信号阻塞时，该信号又多次发生，那么操作系统并不将它们排队，而是只保留第一次的，后续的被抛弃。

还有一点我们必须清楚的是

3. wait系列函数与信号SIGCHLD是没有任何关系的，即wait系列函数并不是信号SIGCHLD驱动的。

这个时候，肯定有人有疑问了，既然会丢弃信号，那怎么保证可以收回所有的僵尸进程呢？

关于这个问题，我们可以这样来理解，当子进程结束时，不管有没有产生SIGCHLD信号，或者子进程产生了SIGCHLD信号，而不管父进程有没有收到SIGCHLD信号，这都与子进程已经终止这个事实无关，就是说，子进程终止与信号其实没有任何关系，只是操作系统在子进程终止时会发送信号SIGCHLD给父进程，告之其子进程终止的消息，这样的话，父进程就可以做相应的操作了。

而wait系列函数的目的就是收回子进程终止时残留在进程列表中的信息，所以任何时候调用while ((pid = waitpid(-1, &exit\_status, WNOHANG)) > 0)都可以收回所有的僵尸进程信息（可以参考下面的程序）。但是这里为什么放在信号处理函数中处理了，这样做的原因是：子进程什么时候结束是个异步事件，而信号机制就是用来处理异步事件的，所以当子进程结束时，可以迅速的收回其残余信息，这样系统中就不会积累大量的僵尸进程了。

也可以这样来理解：系统把所有的僵尸进程串在一起形成一个僵尸进程链表，而while ((pid = waitpid(-1, &exit\_status, WNOHANG)) > 0)就是来清空这个链表的，直到waitpid()返回0，表明已经没有僵尸进程了，或者返回-1，表明出错（当错误码errno为ECHILD的时候同样表明已经不存在僵尸进程了）。

了解了以上知识点，就能理解为什么while ((pid = waitpid(-1, &exit\_status, WNOHANG)) > 0)能够回收所有的僵尸进程了。

我们可以在上面的信号处理函数中加入相应的打印信息：

static int num1 = 0    
static int num2 = 0;    
void avoid\_zombies\_handler(int signo)    
{    
    pid\_t pid;    
    int exit\_status;    
    int saved\_errno = errno;    
    
    printf("num1 = %d\n", ++num1);    
    while ((pid = waitpid(-1, &exit\_status, WNOHANG)) > 0) {    
        printf("num2 = %d\n", ++num2);    
    }    
    
    errno = saved\_errno;    
}

打印的结果你会发现，当num1递增1的时候，即每调用一次信号处理函数，num2一般会递增很多，即while循环了很多次，所以尽管有的SIGCHLD信号被丢弃了，但是我们不用担心子进程的残余信息会收不回来。退出while循环时，证明此时系统中已经没有僵尸进程了，所以退出信号处理函数后，阻塞的唯一SIGCHLD信号会再次触发该信号处理函数，这样我们就不用担心了。

我们不防做个最坏的打算，即之前的信号全部被丢弃了，只有最后一次的SIGCHLD信号被捕获，从而触发了信号处理函数，这样我们也不用担心，因为while循环会一次性收回全部的僵尸进程信息，只是这次循环的次数要多得多罢了，当然这只是假设，一般系统不会出现这样的情况（可以参考本文最后一个程序事例）。

为了证明wait系统函数与信号SIGCHLD没有任何关系，我们可以做个简单的实验,代码如下：

#include <stdio.h>    
#include <stdlib.h>    
#include <unistd.h>    
#include <sys/wait.h>    
#include <sys/types.h>    
    
int main(int argc, char \*argv[])    
{    
    int i;    
    pid\_t pid;    
    
    for (i = 0; i < 5; i++) {    
        if ((pid = fork()) == 0)    /\* child \*/    
            \_exit(0);    
    }    
    sleep(10);    
    while (waitpid(-1, NULL, WNOHANG) > 0) {    
        /\* do nothing \*/    
    }    
    sleep(10);    
    
    \_exit(EXIT\_SUCCESS);    
}

以下是打印结果：

电子设备的屏幕

中度可信度描述已自动生成

可以看到第一次sleep时系统中积累了5个僵尸进程，第二次sleep时，那5个僵尸进程都被收回了。这个也明显的看到了使用信号处理函数的优势，即可以保证系统不会积累大量的僵尸进程，它可以迅速的清理掉系统中的僵尸进程。