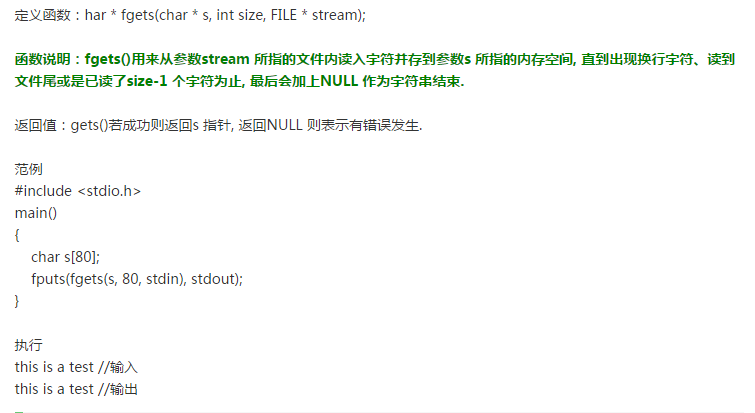
## Shell概述

Linux系统提供给用户的最重要的系统程序是Shell命令语言[解释程序](http://wenwen.soso.com/z/Search.e?sp=S%E8%A7%A3%E9%87%8A%E7%A8%8B%E5%BA%8F&ch=w.search.yjjlink&cid=w.search.yjjlink" \t "_blank)。它不属于内核部分，而是在核心之外，以用户态方式运行。其基本功能是解释并执行用户打入的各种命令，实现用户与Linux核心的接口。系统初启后，核心为每个[终端用户](http://wenwen.soso.com/z/Search.e?sp=S%E7%BB%88%E7%AB%AF%E7%94%A8%E6%88%B7&ch=w.search.yjjlink&cid=w.search.yjjlink" \t "_blank)建立一个进程去执行Shell解释程序。它的执行过程基本上按如下步骤：

1. 读取用户由[键盘](http://wenwen.soso.com/z/Search.e?sp=S%E9%94%AE%E7%9B%98&ch=w.search.yjjlink&cid=w.search.yjjlink" \t "_blank)输入的命令行。



1. 分析命令，以命令名作为文件名，并将其它参数改造为[系统调用](http://wenwen.soso.com/z/Search.e?sp=S%E7%B3%BB%E7%BB%9F%E8%B0%83%E7%94%A8&ch=w.search.yjjlink&cid=w.search.yjjlink" \t "_blank)execve( )内部处理所要求的形式；解析过程：跳过开头空格，第一项为命令名字，作为execve的第一个参数，后面的解析通过空格，直到结尾为止。



1. 终端进程调用fork( )建立一个子进程；

调用fork，产生子进程。

(4) 终端进程本身用系统调用wait4( )来等待子进程完成（如果是后台命令，则不等待）。当子进程运行时调用execve( )，子进程根据文件名（即命令名）到目录中查找有关文件（这是[命令解释程序](http://wenwen.soso.com/z/Search.e?sp=S%E5%91%BD%E4%BB%A4%E8%A7%A3%E9%87%8A%E7%A8%8B%E5%BA%8F&ch=w.search.yjjlink&cid=w.search.yjjlink)构成的文件），将它调入内存，执行这个程序（解释这条命令）。

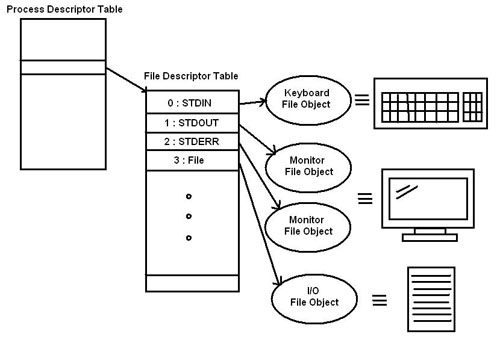
(5) 如果命令末尾有&号（后台命令符号），则终端进程不用系统调用wait4( )等待，立即发提示符，让用户输入下一个命令，转⑴。如果命令末尾没有&号，则终端进程要一直等待，当子进程（即[运行命令](http://wenwen.soso.com/z/Search.e?sp=S%E8%BF%90%E8%A1%8C%E5%91%BD%E4%BB%A4&ch=w.search.yjjlink&cid=w.search.yjjlink" \t "_blank)的进程）完成处理后终止，向父进程（终端进程）报告，此时终端进程醒来，在做必要的判别等工作后，终端进程发提示符，让用户输入新的命令，重复上述处理过程。

## 需要知识储备

1. IO重定向原理

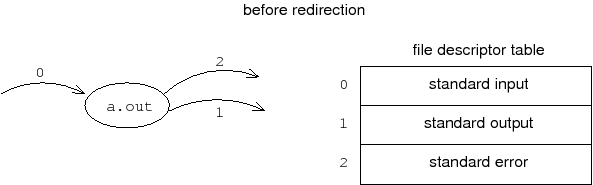
<http://www.cnblogs.com/weidagang2046/p/io-redirection.html>

理解I/O重定向的原理需要从Linux内核为进程所维护的关键数据结构入手。对Linux进程来讲，每个打开的文件都是通过文件描述符(File Descriptor)来标识的，内核为每个进程维护了一个文件描述符表，这个表以FD为索引，再进一步指向文件的详细信息。在进程创建时，内核为进程默认创建了0、1、2三个特殊的FD，这就是STDIN、STDOUT和STDERR，如下图所示意：

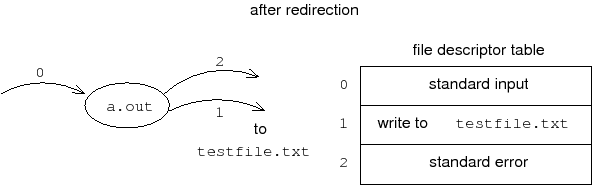


所谓的I/O重定向也就是让已创建的FD指向其他文件。比如，下面是对STDOUT重定向到testfile.txt前后内核文件描述符表变化的示意图

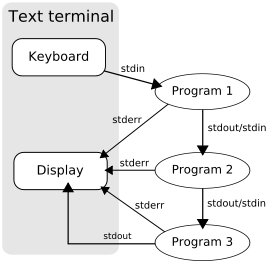
重定向前：

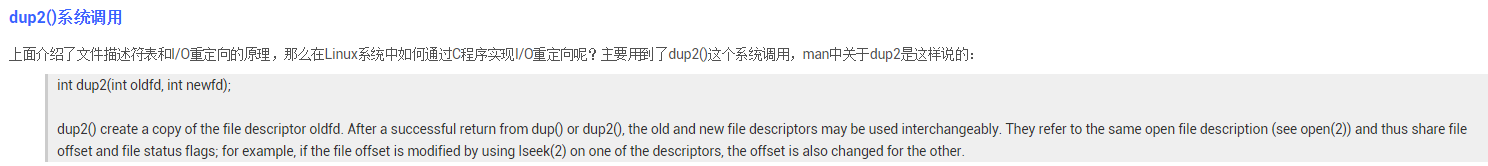


重定向后：



在I/O重定向的过程中，**不变的是FD 0/1/2代表STDIN/STDOUT/STDERR，变化的是文件描述符表中FD 0/1/2对应的具体文件**，应用程序只关心前者。本质上这和接口的原理是相通的，通过一个间接层把功能的使用者和提供者解耦。





1.描述符表（descriptor table）– 每个进程都有各自的表

file descriptor（fd），当调用open函数时会返回这个文件的文件描述符fd，这个fd相当于一个索引，而一个进程可以打开多个文件，所以每个进程都有一个描述符表（fd就是这个表的索引）用于保存该进程打开的文件的索引。简而言之，fd就是描述符表的索引，而描述符表项中存的是每个文件表项的起始地址。

2.文件表（file table）– 所有进程共享

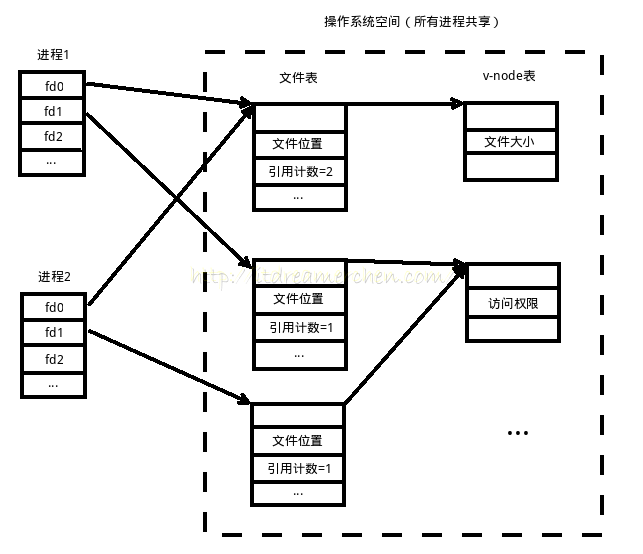
每个表项对应一个文件操作记录，其中记录了当前的访问位置，引用次数等。多个fd可以指向同一个文件表项（共享），多个文件表项可以指向同一个v-node（两个不相关的进程打开同一个文件）。

3.v-node表 — 所有进程共享

存在磁盘上面的文件有两个部分，一部分是数据，另一部分就是v-node，v-node记录了这个文件的相关信息，如文件大小，访问权限，上次访问时间等。当打开文件时操作系统会将这个文件的v-node加载进内存，对一个文件而言v-node只会被加载一次， 所有的进程都共享这些v-node。

4.三者之间的关系

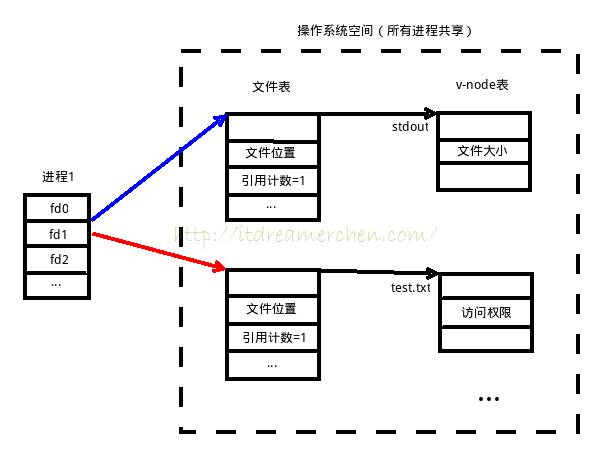
为了更好理解上面所说的，先来看看下面这张图：

[](http://burningcodes.net/wp-content/uploads/2015/01/fd.png)

多个fd可以指向同一个文件表项，这时会增加引用计数，当引用计数为0时，操作系统会删除这个表项。每个文件表项中存储了当前的文件指针指向的位置。不同的文件表项若打开了相同的文件，则一定指向相同的v-node表项。

5.IO重定向的原理

一般情况下，fd=0则表示标准输入，fd=1表示标准输出，fd=2表示标准错误输出。 使用“>”进行输出重定向的原理即将指向stdout的fd1改成指向test.txt即可，如下图：

[](http://burningcodes.net/wp-content/uploads/2015/01/fd1.png)

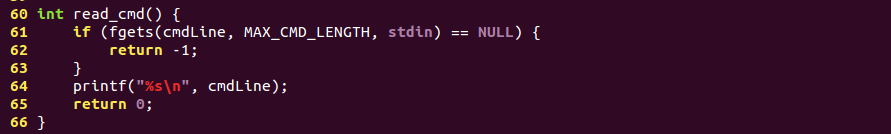
将蓝线改成红线即可完成输出重定向至test.txt

1. 管道原理

1. 命令解析

## 实现功能

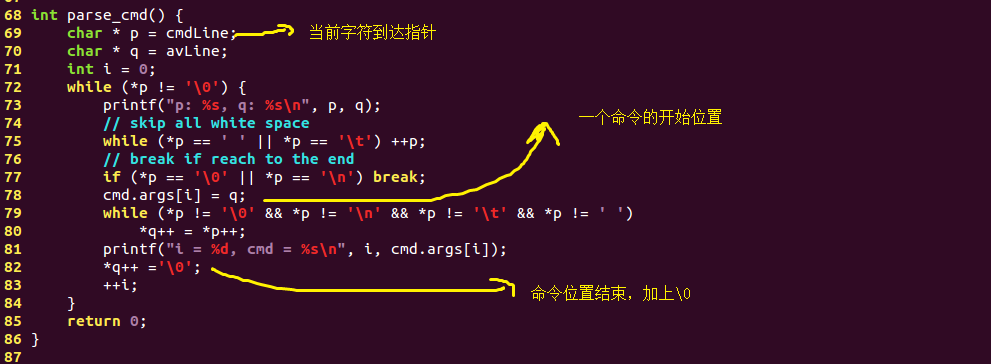
### 命令行的读取



### 基本命令解析

像,ls –al之类的简单命令，解析会跳过相应的空白，tab知道结束’\n’、’\0’为止。

存到一个char \* args[]数组中。



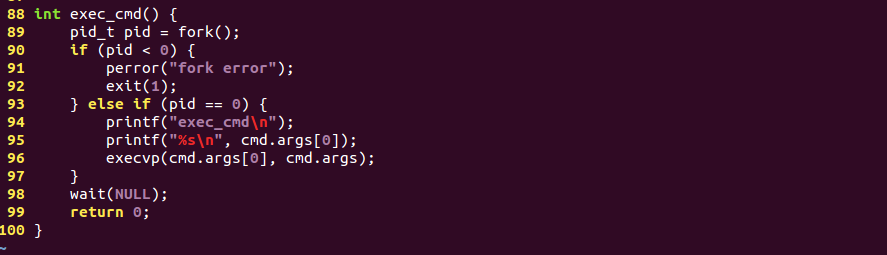
### 简单命令执行

建立子进程并执行execvp，父进程等待子进程结束

int execvp(const char \*file, char \*const argv[]);

path参数表示你要启动程序的名称包括路径名（表示第一个参数path不用输入完整路径，只有给出命令名即可，它会在环境变量PATH当中查找命令）；arg参数表示启动程序所带的参数，一般第一个参数为要执行命令名，不是带路径且arg必须以NULL结束。

http://www.cnblogs.com/mickole/p/3187409.html



如果一个进程想执行另一个程序，它就可以fork或vfork出一个新进程，然后调用任何一个exec函数。为此，Linux还专门对fork作了优化：通常fork会将调用进程的所有内容原封不动的拷贝到新产生的子进程中去，这些拷贝的动作很消耗时间，而如果fork完之后我们马上就调用exec，那这些辛辛苦苦拷贝来的东西就会被立刻抹掉，这看起来非常不划算，于是人们设计了一种"写时复制（copy-on-write）"技术，使得fork结束后并不立刻复制父进程的内容到子进程，而是到了真正使用时才复制，这样如果下一条语句是exec，它就不会作无用功了。其实"写时复制"还是有复制，进程的mm结构、页表都还是被复制了（"写时复制"也必须由这些信息来支撑。否则内核捕捉到CPU访存异常，怎么区分 这是“写时复制”引起的，还是真正的越权访问呢？）。

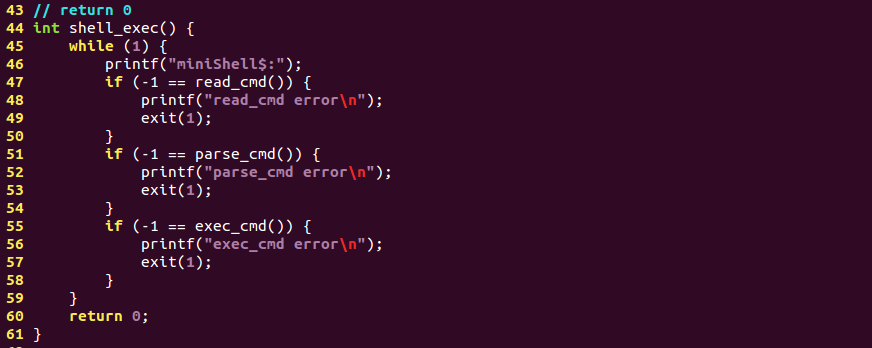
而vfork就把事情做绝了，所有有关于内存的东西都不复制了，父子进程的内存是完全共享的。但是这样一来又有问题了，虽然用户程序可以设计很多方法来避免父子进程间的访存冲突。但是关键的一点，父子进程共用着栈，这可不由用户程序控制的。一个进程进行了关于函数调用或返回的操作，则另一个进程的调用栈 （实际上就是同一个栈）也被影响了。这样的程序没法运行下去。所以，vfork有个限制，子进程生成后，父进程在vfork中被内核挂起，直到子进程有了自己的内存空间（exec\*\*）或退出(\_exit)。并且， 在此之前，子进程不能从调用vfork的函数中返回（同时，不能修改栈上变量、不能继续调用除\_exit或exec系列之外的函数，否则父进程的数据可能 被改写）。

尽管限制很多，vfork后马上exec效率会比fork高不少。

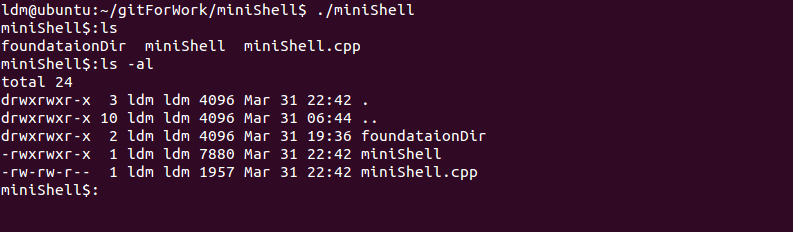
对于**shell命令，它本身就是一个进程**，要想执行ls程序，则需去加载ls程序，这时shell命令进程则需fork()创建一个新进程，而我们知道**新创建的进程与原进程几乎是一样的**，也就意味着新的进程的代码还是跟shell程序本身是一样的，也就**无法执行ls程序**，所以，这时我们只有将新进程用ls程序替换，也就是用**exec系列函数**来替换，这也就是它的意义所在。

### 整体循环不断处理shell

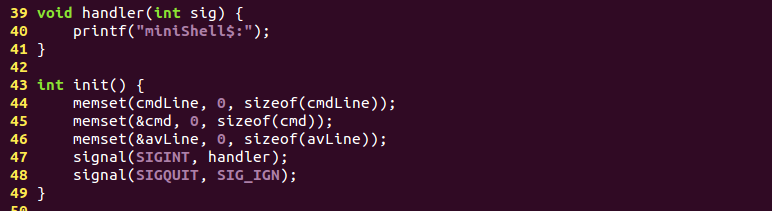
循环操作，依次读入命令，解析命令，执行命令。

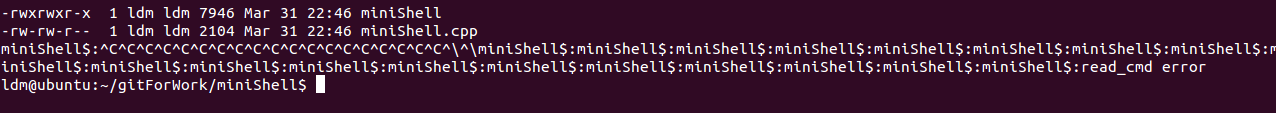


### 运行结果



### 添加中断与取消信号处理





没有进行fflush(stdout).

使用printf()函数后，内容存入输出缓冲区。通常，如果有'\n'时，才进行一次输出，或者，等到时间片轮转到系统的输出程序时，才将其输出。在printf()函数后使用fflush( stdout )函数，先将即将输出的内容输出，然后，再将输出缓冲区清空。