

第二次作业

题目一解答：

三种传递参数方式以及适合情况：

- 1) 寄存器：在参数数量少于寄存器数目时，采用寄存器传参；
- 2) 由寄存器传递内存块地址：参数数目多于寄存器时，参数会存在于内存的块和表中，这时将块的地址通过寄存器来传递；
- 3) 参数也可通过程序放在或压入栈中，并通过操作系统弹出：此方法和内存块地址法传参都不限制传递参数的数量和长度，可以用于参数数量多与寄存器数目、或长度超过寄存器大小的情况。

题目二解答：

a) 操作系统中，“需要提供什么功能”即机制，“如何使用这些功能”即策略。区分机制和策略，使得操作系统能灵活地被修改——策略，即需要的功能可能随着时间、位置等环境而变化。底层设备是不尽相同的，而且都想让操作系统更加的“适合”自己，在机制和策略分离时，策略能被任意地修改以适应设备，但机制保持不变，这样的原则提供了更灵活的制度。

b) 案例：“操作系统中的进程调度”

WINDOWS 进程调度采用的是机制与策略分离的方式：在 WINDOWS 多任务系统中，进程的调度算法包括：轮询、中断、时间片等，这可看作具体“策略”，由进程调度模块实现；而提供从进程 A 切换到进程 B 这种功能，是靠硬件、数据结构（比如上下文切换时内核数据结构 PCB）支撑的，由 WINDOWS 内核中底层实现，这可看作是“机制”。

这样带来的好处是：

- ①可扩展性：随时可添加新的调度算法，而对原有实现不会有太大影响；
- ②可靠性：模块间通过接口通信，相互之间独立，一个模块的修改不会对另一个模块产生影响；

题目三解答：

a) 微内核设计主要优点？

- ①便于移植：由于这种设计将所有非基本部分从内核移走转而时限为系统/用户程序，使得内核更小；
- ②便于扩充操作系统：所有的新服务可在用户空间增加，因而无需修改内核，即使内核需要被修改时，由于内核本身很小，所做的改变也是会很小；
- ③更好的安全性、可靠性：绝大多数服务是作为用户而不是作为内核进程运行，这

样一个服务器出错，操作系统的其他部分并不会受影响。

b) 用户程序和系统服务在微内核结构内如何相互影响？

用户程序和系统服务通过微内核提供的功能进行通信（通信以消息传递的形式提供），从而实现间接交互。

c) 微内核设计的缺点？

①通过进程通信的方式交换数据或者调用系统服务，而不是使用系统调用，消息传递的频繁使用造成额外的操作系统开销；

②使用一些频繁使用的系统服务时，比如网络收发数据，造成的进程上下文切换对操作系统来说也是一个负担；

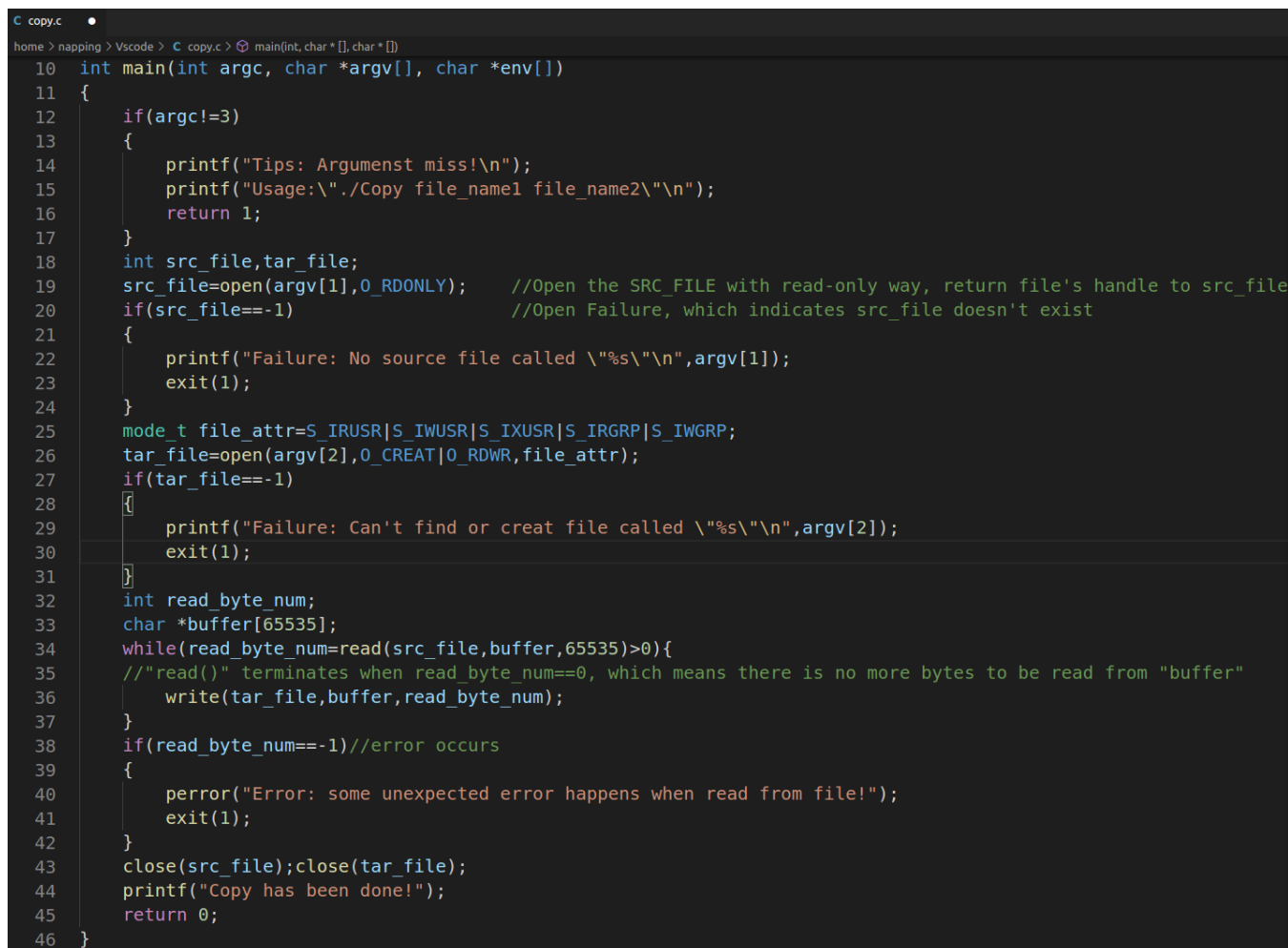
③由于系统服务高度模块化，系统服务之间存在大量的内存复制；

④对互相之间存在复杂调用关系的系统服务，难以设计通信接口；

⑤系统服务与内核在地址空间上分离，造成代码局部性差，降低了 cache 命中率。

题目四解答：

Linux 中 c 程序截图如下：

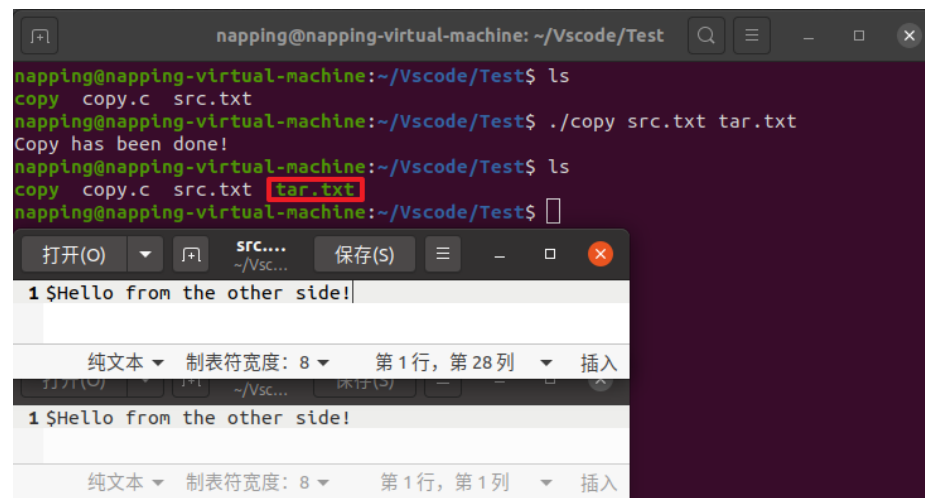


```
10 int main(int argc, char *argv[], char *env[])
11 {
12     if(argc!=3)
13     {
14         printf("Tips: Argumenst miss!\n");
15         printf("Usage: \"./Copy file_name1 file_name2\"\n");
16         return 1;
17     }
18     int src_file,tar_file;
19     src_file=open(argv[1],O_RDONLY);    //Open the SRC_FILE with read-only way, return file's handle to src_file
20     if(src_file==-1)                    //Open Failure, which indicates src_file doesn't exist
21     {
22         printf("Failure: No source file called \"%s\"\n",argv[1]);
23         exit(1);
24     }
25     mode_t file_attr=S_IRUSR|S_IWUSR|S_IXUSR|S_IRGRP|S_IWGRP;
26     tar_file=open(argv[2],O_CREAT|O_RDWR,file_attr);
27     if(tar_file==-1)
28     {
29         printf("Failure: Can't find or creat file called \"%s\"\n",argv[2]);
30         exit(1);
31     }
32     int read_byte_num;
33     char *buffer[65535];
34     while(read_byte_num=read(src_file,buffer,65535)>0){
35         // "read()" terminates when read_byte_num==0, which means there is no more bytes to be read from "buffer"
36         write(tar_file,buffer,read_byte_num);
37     }
38     if(read_byte_num==-1)//error occurs
39     {
40         perror("Error: some unexpected error happens when read from file!");
41         exit(1);
42     }
43     close(src_file);close(tar_file);
44     printf("Copy has been done!");
45     return 0;
46 }
```

其中，对于执行时函数参数错误、源文件文件不存在、目标文件无法创建、写入目标文件出错等错误进行了检测且有相应错误提示；其中，如果目标文件未找到，则默认为其创建

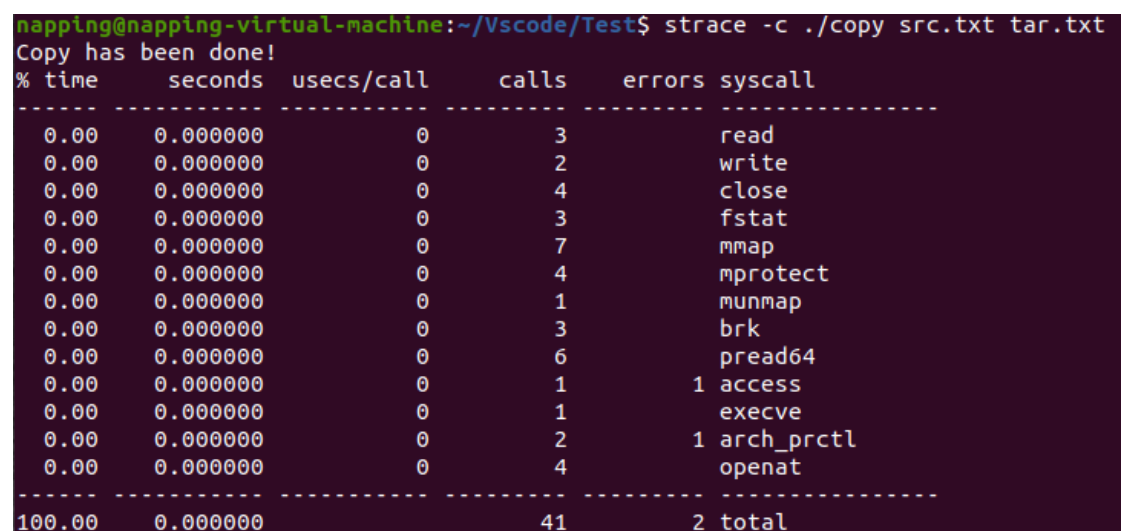
一个同名文件，创建过程中失败的话则报错。

执行结果如下：



```
napping@napping-virtual-machine: ~/Vscode/Test
napping@napping-virtual-machine:~/Vscode/Test$ ls
copy copy.c src.txt
napping@napping-virtual-machine:~/Vscode/Test$ ./copy src.txt tar.txt
Copy has been done!
napping@napping-virtual-machine:~/Vscode/Test$ ls
copy copy.c src.txt tar.txt
napping@napping-virtual-machine:~/Vscode/Test$
```

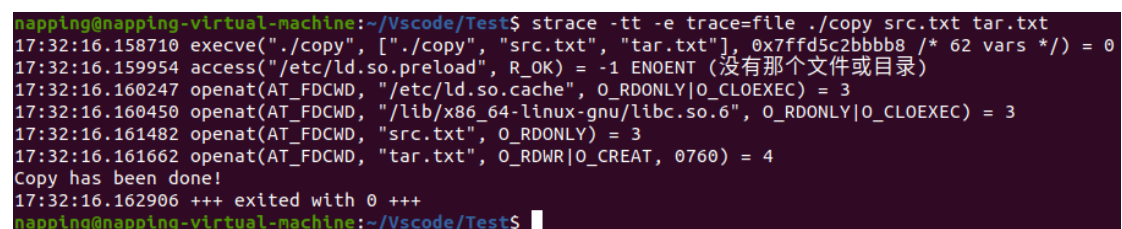
题目五解答：



```
napping@napping-virtual-machine:~/Vscode/Test$ strace -c ./copy src.txt tar.txt
Copy has been done!
% time      seconds  usecs/call   calls   errors syscall
-----
0.00      0.000000      0          3        read
0.00      0.000000      0          2        write
0.00      0.000000      0          4        close
0.00      0.000000      0          3        fstat
0.00      0.000000      0          7        mmap
0.00      0.000000      0          4        mprotect
0.00      0.000000      0          1        munmap
0.00      0.000000      0          3        brk
0.00      0.000000      0          6        pread64
0.00      0.000000      0          1        1 access
0.00      0.000000      0          1        execve
0.00      0.000000      0          2        1 arch_prctl
0.00      0.000000      0          4        openat
-----
100.00    0.000000          41        2 total
```

(-c 选项统计每一系统调用的所执行的时间,次数和出错的次数等)

Linux 下通过 strace 跟踪命令“./copy src.txt tar.txt”执行情况，这里只对系统调用进行跟踪（-e trace=all）：



```
napping@napping-virtual-machine:~/Vscode/Test$ strace -tt -e trace=file ./copy src.txt tar.txt
17:32:16.158710 execve("./copy", ["/.copy", "src.txt", "tar.txt"], 0x7ffd5c2bbbb8 /* 62 vars */) = 0
17:32:16.159954 access("/etc/ld.so.preload", R_OK) = -1 ENOENT (没有那个文件或目录)
17:32:16.160247 openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
17:32:16.160450 openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
17:32:16.161482 openat(AT_FDCWD, "src.txt", O_RDONLY) = 3
17:32:16.161662 openat(AT_FDCWD, "tar.txt", O_RDWR|O_CREAT, 0760) = 4
Copy has been done!
17:32:16.162906 +++ exited with 0 +++
napping@napping-virtual-machine:~/Vscode/Test$
```

(-tt 在输出中的每一行前加上时间信息,微秒级.)

这里是所有的文件系统调用，最开 execve 可执行文件 copy，之后的几个系统调用个人理解应该是向缓存访问输入文件名，访问失败后创建这些文件以便之后访问，然后再打开文

件。

通过修改 -e trace=read、write 追踪系统调用读和写：

```
napping@napping-virtual-machine:~/Vscode/Test$ strace -tt -e trace=read ./copy src.txt tar.txt
17:34:19.875422 read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\360q\2\0\0\0\0"... , 832) = 832
17:34:19.876886 read(3, "$Hello from the other side!\n", 65535) = 28
17:34:19.877169 read(3, "", 65535) = 0
Copy has been done!
17:34:19.878983 +++ exited with 0 +++
napping@napping-virtual-machine:~/Vscode/Test$ strace -tt -e trace=write ./copy src.txt tar.txt
17:34:31.422583 write(4, "$Hello from the other side!\n", 28) = 28
17:34:31.423973 write(1, "Copy has been done!\n", 20)Copy has been done!
) = 20
17:34:31.424265 +++ exited with 0 +++
napping@napping-virtual-machine:~/Vscode/Test$
```

可以看到最开始读入的是 ELF 文件格式，之后再 while 循环中对 src.txt 循环读取，由于我们的 src.txt 只有一句话，所以读取一次（一次性读取 65535）之后就读完了，下一次读取的字节数等于 0，所以总共 3 次 read。

对于 write，除了向 tar.txt 文件写入，还有我们调用的 printf 函数也会调用。

题目六解答：

- a) 文件管理：Linux 提供“查找文件”的系统程序——find

功能：

find 是 Linux 中强大的搜索命令，不仅可以按照文件名搜索文件，还可以按照权限、大小、时间、inode 号等来搜索文件。

特点：

find 命令是完全匹配的，必须和搜索关键字一模一样才会列出。

注意：

但是 find 命令是直接硬盘中进行搜索的，如果指定的搜索范围过大，find 命令就会消耗较大的系统资源，导致服务器压力过大。所以，在使用 find 命令搜索时，不要指定过大的搜索范围。

实例：

```
napping@napping-virtual-machine:~$ find -name copy
./Vscode/Test/copy
napping@napping-virtual-machine:~$ find -iname CoPY.C
./Vscode/Test/copy.c
```

- b) 状态信息：Linux 中查看“进程状态信息”的系统程序——ps

功能：

ps 命令用于显示当前进程的状态，类似于 windows 的任务管理器，但是 ps 只是捕获当前进程的快照，需要实时更新的话可使用指令 top

选项：

"ps aux" 可以查看系统中所有的进程；

"ps -le" 可以查看系统中所有的进程，而且还能看到进程的父进程的 PID 和进程优先级；

"ps -l" 只能看到当前 Shell 产生的进程；

实例：

```
napping@napping-virtual-machine:~$ ps
  PID TTY          TIME CMD
 2840 pts/0        00:00:00 bash
 4089 pts/0        00:00:00 watch
 4117 pts/0        00:00:00 watch
 4216 pts/0        00:00:00 ps
```

- c) 文件修改：Linux 中“文件权限修改”的系统程序——chmod 功能：

chmod 命令用来变更文件或目录的权限。在 UNIX 系统家族里，文件或目录权限的控制分别以读取、写入、执行 3 种一般权限来区分，另有 3 种特殊权限可供运用。用户可以使用 chmod 指令去变更文件与目录的权限，设置方式采用文字或数字代号皆可。符号连接的权限无法变更，如果用户对符号连接修改权限，其改变会作用在被连接的原始文件。

选项：

- c 或 —changes：效果类似“-v”参数，但仅回报更改的部分；
- f 或 --quiet 或 —silent：不显示错误信息；
- R 或 —recursive：递归处理，将指令目录下的所有文件及子目录一并处理；
- v 或 —verbose：显示指令执行过程；
- reference=<参考文件或目录>：把指定文件或目录的所属群组全部设成和参考文件或目录的所属群组相同；
- <权限范围>+<权限设置>：开启权限范围的文件或目录的该选项权限设置；
- <权限范围>-<权限设置>：关闭权限范围的文件或目录的该选项权限设置；
- <权限范围>=<权限设置>：指定权限范围的文件或目录的该选项权限设置；

实例：

当我们把原有的 c 文件权限修改后，再尝试编译发现权限不够

```
napping@napping-virtual-machine:~/Vscode/Test$ chmod 111 copy.c
napping@napping-virtual-machine:~/Vscode/Test$ vim copy.c
napping@napping-virtual-machine:~/Vscode/Test$ gcc copy.c -o copy
cc1: fatal error: copy.c: 权限不够
compilation terminated.
```

- d) 程序语言支持：

Linux 中自带 python3 解释器 (Ubuntu20.01.04)、GNU Binutils, GNU Binutils 主要包含汇编器 as, 链接器 ld; as 使用的 AT&T 汇编语法, 如果要使用 Intel 语法需要安装 nasm。c 语言编译器: gcc 和 cc, cc 其实是 Unix 下的 c 语言编译器, 收费的, 不遵循 GPL 许可协议; 而 gcc 来自 Linux 世界, 是 GNU compiler collection 的缩写。其实在 Linux 下的 cc 是指向 gcc 的链接而已, 而 gcc 也是使用的 as 汇编器。

```
napping@napping-virtual-machine:~/Vscode/Test$ python --version
Python 3.8.5
napping@napping-virtual-machine:~/Vscode/Test$ gcc --version
gcc (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

napping@napping-virtual-machine:~/Vscode/Test$ as --version
GNU 汇编器 (GNU Binutils for Ubuntu) 2.34
Copyright (C) 2020 Free Software Foundation, Inc.
本软件是自由软件; 您可以使用 GNU General Public License 版本 3 或更新版本重新发
放。
本软件完全不带有任何保证。
为目标"x86_64-linux-gnu"配置汇编程序。
napping@napping-virtual-machine:~/Vscode/Test$ ld --version
GNU ld (GNU Binutils for Ubuntu) 2.34
Copyright (C) 2020 Free Software Foundation, Inc.
这个程序是自由软件; 您可以遵循GNU 通用公共授权版本 3 或
(您自行选择的) 稍后版本以再次散布它。
这个程序完全没有任何担保。
```

e) 程序加载与执行：Linux 中加载执行程序的——exec 系列程序

功能：

exec 替换进程映像，由于 Unix 中进程的创建和加载新进程是分离的，所以在新进程产生后用 exec 替换当前进程。

函数：

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execlxe(const char *path, const char *arg, ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

区别：

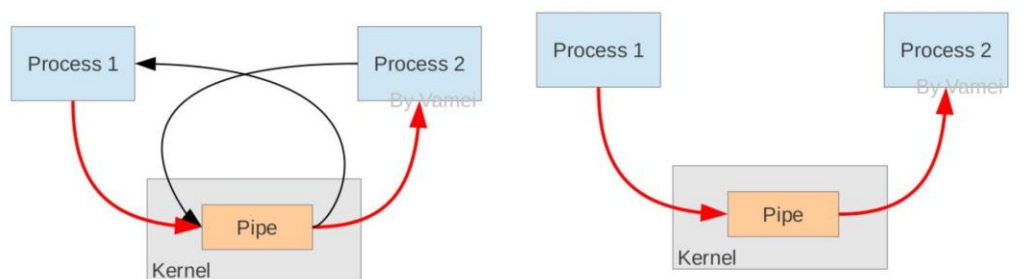
- 1、带 l 的 exec 函数：execl,execlp,execlxe，表示后边的参数以可变参数的形式给出且都以一个空指针结束。
- 2、带 p 的 exec 函数：execlp,execvp，表示第一个参数 path 不用输入完整路径，只有给出命令名即可，它会在环境变量 PATH 当中查找命令
- 3、不带 l 的 exec 函数：execv,execvp 表示命令所需的参数以 char *arg[]形式给出且 arg 最后一个元素必须是 NULL
- 4、带 e 的 exec 函数：execlxe 表示，将环境变量传递给需要替换的进程

f) 通信：Linux 进程通信的方法之一——pipe

管道是如何创建和通信的：

从原理上，管道利用 fork 机制建立，从而让两个进程可以连接到同一个 PIPE 上。最开始的时候，上面的两个箭头都连接在同一个进程 Process 1 上(连接在 Process 1 上的两个箭头)。当 fork 复制进程的时候，会将这两个连接也复制到新的进程(Process 2)。随后，每个进程关闭自己不需要的一个连接 (两个黑色的箭头被关闭; Process 1 关闭从 PIPE 来的输入连接，Process 2 关闭输出到 PIPE 的连接)，这样，剩下的红色连接就构成了如上图的 PIPE。

管道被设计成为环形的数据结构，以便管道可以被循环利用。当管道中没有信息的话，从管道中读取的进程会等待，直到另一端的进程放入信息。当管道被放满信息的时候，尝试放入信息的进程会等待，直到另一端的进程取出信息。当两个进程都终结的时候，管道也自动消失。



g) 后台服务：后台执行指令——&、bg

功能：后台执行程序，或者将后台被暂停的程序继续转为后台执行

实例：

```
napping@napping-virtual-machine:~/Vscode/Test$ ./loop &
[4] 4594
napping@napping-virtual-machine:~/Vscode/Test$ jobs
[1] 已停止                  watch -n 1 'ps -aux --sort -pmem, -pcpu' (工作目录:
-)
[2]- 已停止                  watch -n 1 'ps -aux --sort -pmem, -pcpu' (工作目录:
-)
[3]+ 已停止                  ./loop
[4] 运行中                  ./loop &
```

(后台执行)

```
napping@napping-virtual-machine:~/Vscode/Test$ ./loop
^Z
[1]+ 已停止                  ./loop
napping@napping-virtual-machine:~/Vscode/Test$ jobs
[1]+ 已停止                  ./loop
napping@napping-virtual-machine:~/Vscode/Test$ bg 1
[1]+ ./loop &
napping@napping-virtual-machine:~/Vscode/Test$ ps
  PID TTY          TIME CMD
 2840 pts/0        00:00:00 bash
 4619 pts/0        00:00:03 loop
 4620 pts/0        00:00:00 ps
napping@napping-virtual-machine:~/Vscode/Test$
```

(恢复后台停止进程)