# Linux高性能服务器

## Linux系统编程入门

### 搭建环境

Linux系统安装: <https://releases.ubuntu.com/bionic/>，在虚拟机上模拟Linux系统开发，也可移动硬盘装双系统进行开发

安装Visual Studio Code: <https://code.visualstudio.com>，建立ssh远程连接，在Windows端进行代码开发

### 2. GCC编译器

#### 2.1什么是GCC

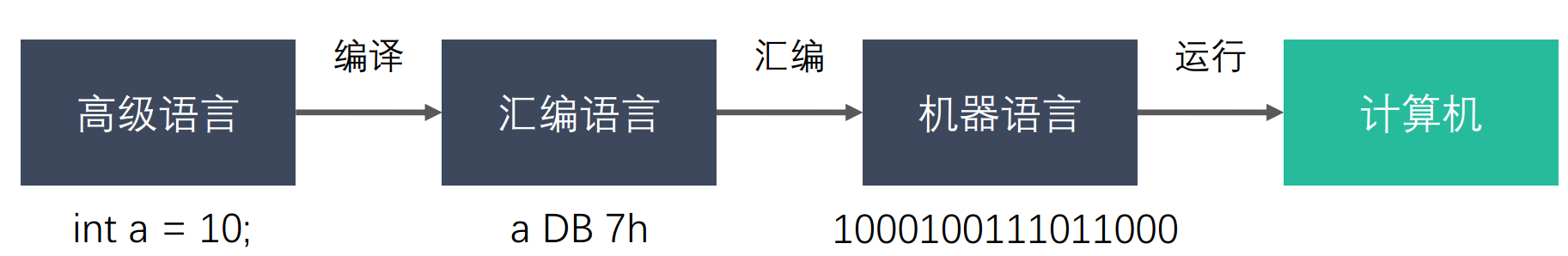
* GCC 原名为 GNU C语言编译器（GNU C Compiler），后来拓展为GCC（GNU Compiler Collection，GNU编译器套件）是由 GNU 开发的编程语言编译器。GNU 编译器套件包括 C、C++、Objective-C、Java、Ada 和 Go 语言前端，也包括了这些语言的库（如 libstdc++，libgcj等）
* GCC 不仅支持 C 的许多“方言”，也可以区别不同的 C 语言标准；可以使用命令行

选项来控制编译器在翻译源代码时应该遵循哪个 C 标准。例如，当使用命令行参数

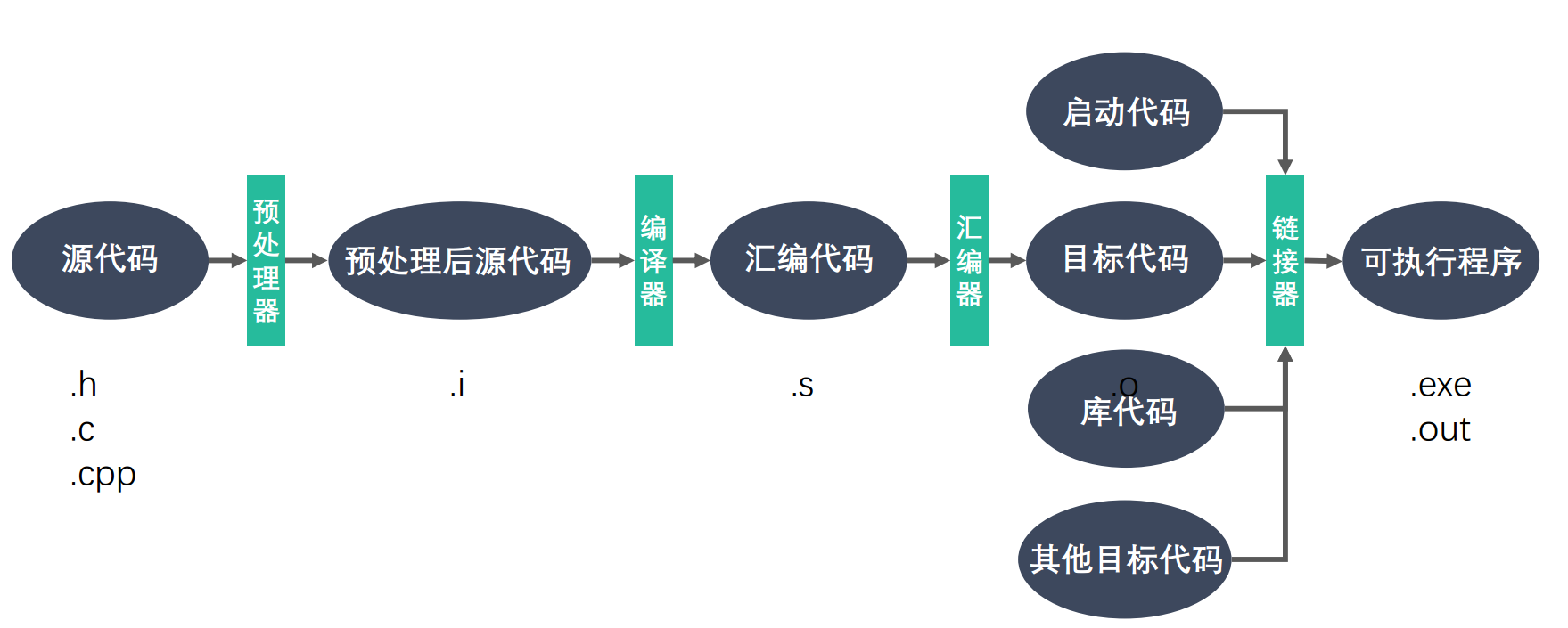
`-std=c99` 启动 GCC 时，编译器支持 C99 标准。

* 安装命令 sudo apt install gcc g++ （版本 > 4.8.5）
* 查看版本 gcc/g++ -v/--version

#### 2.2 编程语言的发展



#### 2.3 GCC工作流程



#### 2.4 GCC和G++的区别

* gcc 和 g++都是GNU(组织)的一个编译器。
* 误区一：gcc 只能编译 c 代码，g++ 只能编译 c++ 代码。两者都可以，请注意：
  + 后缀为.c的，gcc 把它当作是C程序，而g++当作是c++程序
  + 后缀为.cpp的，两者都会认为是C++程序，C++的语法规则更加严谨一些
  + 编译阶段，g++会调用gcc，对于C++代码，两者是等价的，但是因为gcc

命令不能自动和C++程序使用的库联接，所以通常用g++来完成链接，为了统

一起见，干脆编译/链接统统用g++了，这就给人一种错觉，好像cpp程序只能用 g++似的

* 误区二：gcc不会定义\_\_cplusplus宏，而g++会
* 实际上，这个宏只是标志着编译器将会把代码按 C 还是 C++ 语法来解释
* 如上所述，如果后缀为 .c，并且采用 gcc 编译器，则该宏就是未定义的，否则，

就是已定义

* 误区三：编译只能用gcc，链接只能用g++
  + 严格来说，这句话不算错误，但是它混淆了概念，应该这样说：编译可以用

gcc/g++，而链接可以用g++或者gcc -lstdc++。

* gcc命令不能自动和C++程序使用的库联接，所以通常使用g++来完成联接。

但在编译阶段，g++会自动调用gcc，二者等价

#### 2.5 GCC常用参数选项

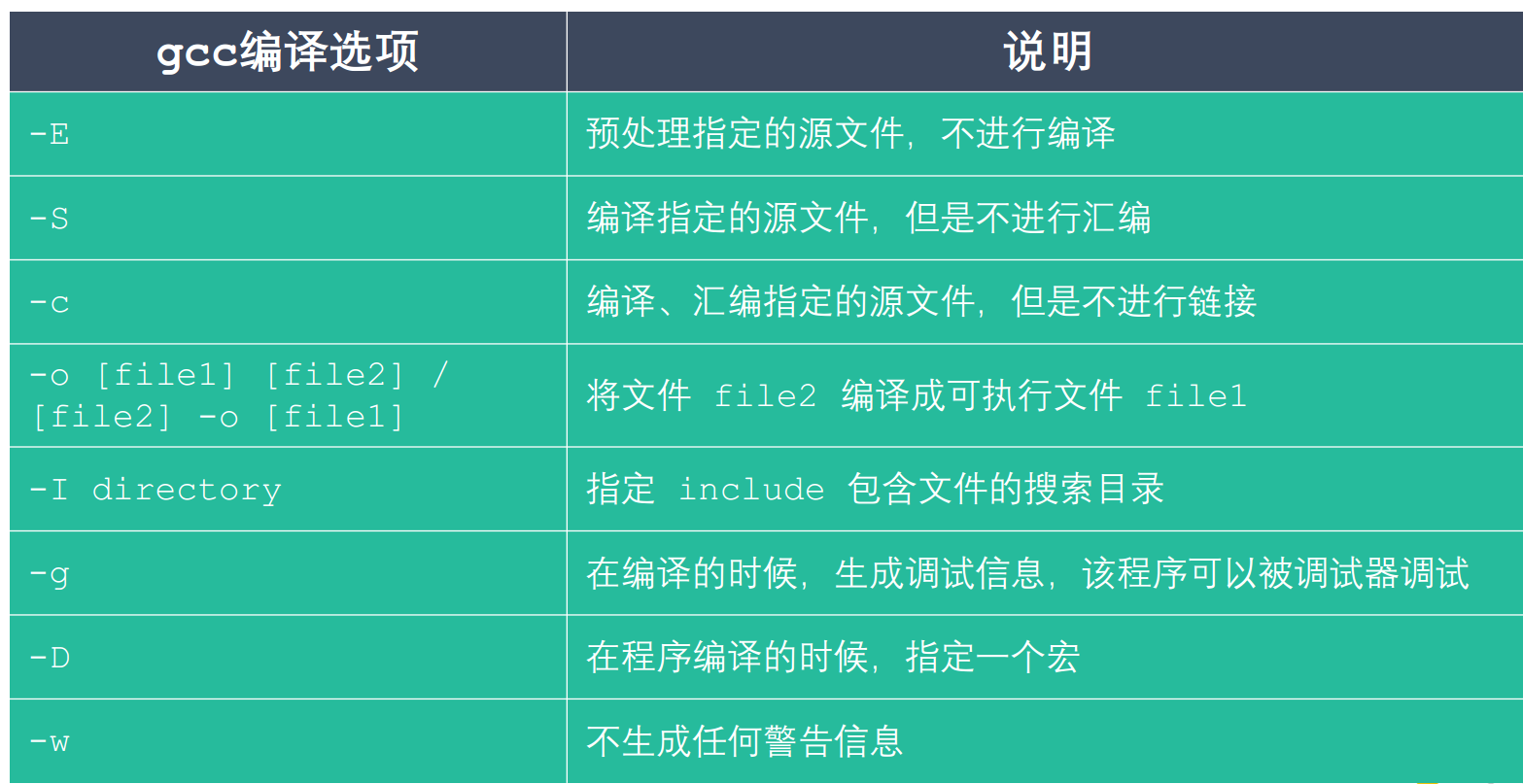
gcc -E test.c => test.i

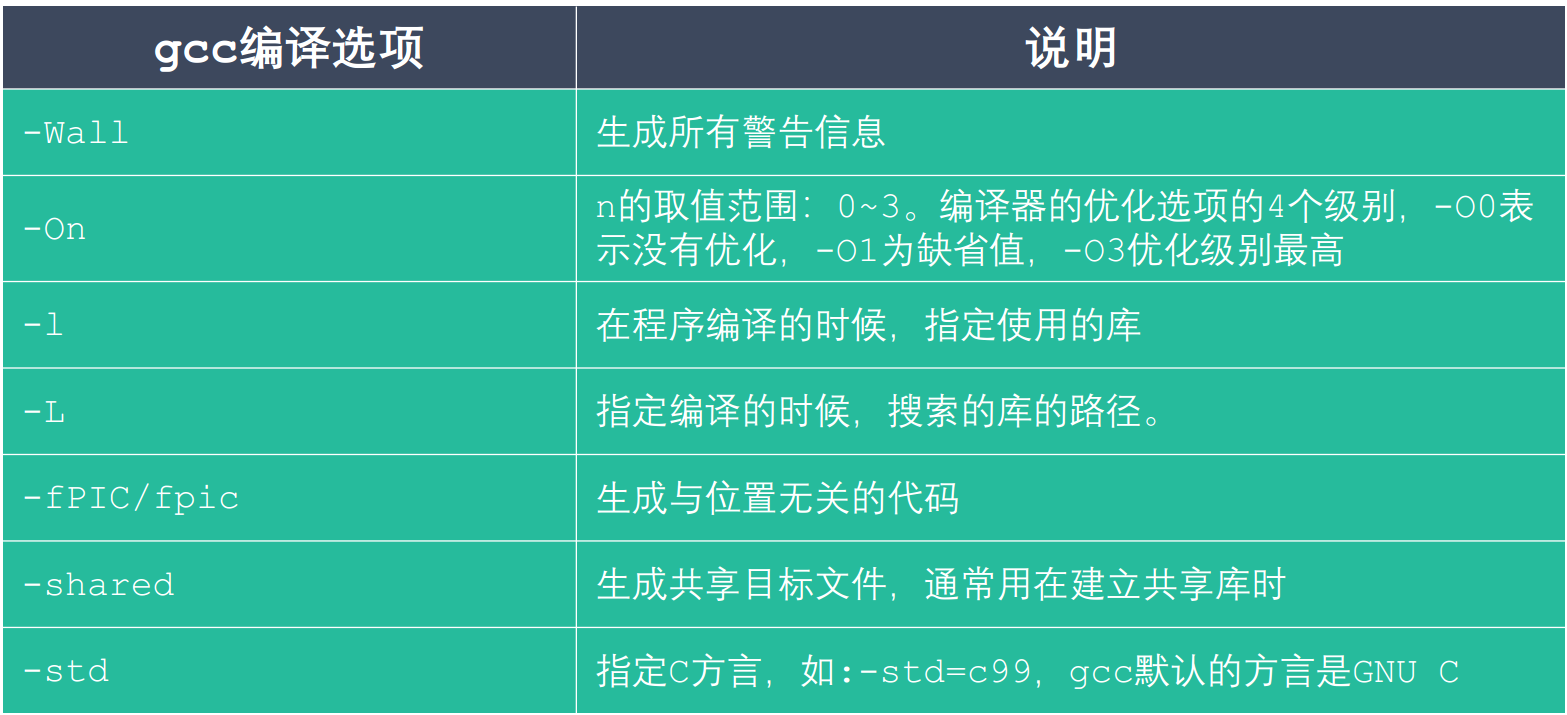
gcc -S test.c => test.s

gcc -c test.c => test.o

gcc -o test.c test => test

gcc -O3 test.c => a.out（防止反编译）





### 静态库的制作

#### 3.1什么是库

* 库文件是计算机上的一类文件，可以简单的把库文件看成一种代码仓库，它提供给使用者一些可以直接拿来用的变量、函数或类。
* 库是特殊的一种程序，编写库的程序和编写一般的程序区别不大，只是库不能单独运行。
* 库文件有两种，静态库和动态库（共享库），区别是：静态库在程序的链接阶段被复制到了程序中；动态库在链接阶段没有被复制到程序中，而是程序在运行时由系统动态加载到内存中供程序调用。
* 库的好处：1.代码保密 2.方便部署和分发

#### 3.2 静态库的制作

* 命名规则：Linux : libxxx.a

lib : 前缀（固定）

xxx : 库的名字，自己起 .a : 后缀（固定）

Windows : libxxx.lib

* 静态库的制作：

gcc 获得 .o 文件

将 .o 文件打包，使用 ar 工具（archive）

ar rcs libxxx.a xxx.o xxx.o

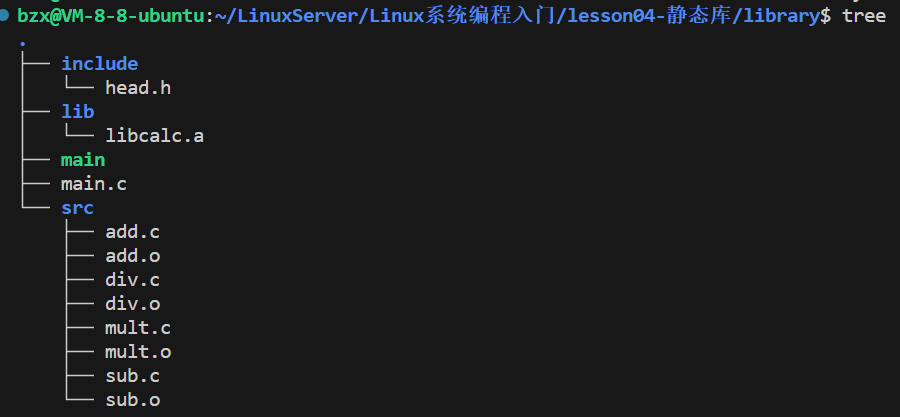
r – 将文件插入备存文件中

c – 建立备存文件

#### 3.3 使用静态库

gcc main.c -o main -I ./include/ -l calc -L ./lib/

-o 起另外一个名 -I 指定头文件搜索文件夹 -l 指定使用的静态库 -L 指定使用的静态库的路径



### 动态库的制作和使用

#### 动态库的制作

命名规则：

Linux : libxxx.so lib : 前缀（固定） xxx : 库的名字，自己起

.so : 后缀（固定），在Linux下是一个可执行文件

Windows : libxxx.dll

动态库的制作：

gcc 得到 .o 文件，得到和位置无关的代码

gcc -c –fpic/-fPIC a.c b.cgcc 得到动态库gcc -shared a.o b.o -o libcalc.so

#### 工作原理

静态库：GCC 进行链接时，会把静态库中代码打包到可执行程序中动态库：GCC 进行链接时，动态库的代码不会被打包到可执行程序中，程序启动之后，动态库会被动态加载到内存中，通过 ldd （list dynamic dependencies）命令检查动态库依赖关系如何定位共享库文件呢：

当系统加载可执行代码时候，能够知道其所依赖的库的名字，但是还需要知道绝对路径。此时就需要系统的动态载入器来获取该绝对路径。对于elf格式的可执行程序，是由ld-linux.so来完成的，它先后搜索elf文件的 DT\_RPATH段 ——> 环境变量LD\_LIBRARY\_PATH ——> /etc/ld.so.cach文件列表 ——> /lib/，/usr/lib目录找到库文件后将其载入内存。

#### 使用动态库

使用动态库 gcc main.c -o main -I ./include/ -l calc -L ./lib/ 会报错

动态库依赖list dynamic dependencies检查动态库依赖 ldd main

当系统加载可执行代码时候，能够知道其所依赖的库的名字，但是还需要知道绝对路径。此时就需要系统的动态载入器来获取该绝对路径。

对于elf格式的可执行程序，是由ld-linux.so来完成的，它先后搜索elf文件的 DT\_RPATH段(虚拟地址空间用户不能改～)

——> 环境变量LD\_LIBRARY\_PATH(用户级即.bashrc source .bashrc 系统级即sudo vim etc/profile 然后 source etc/profile)

在bashrc里面加入下面这一段，然后. bashrc即可

export LD\_LIBRARY\_PATH=/home/bzx/LinuxServer/Linux系统编程入门/lesson06-动态库/library/lib:$LD\_LiBRARY\_PATH

或者在sudo vim etc/profile

export LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:/home/williamwhites/LinuxServer/lesson06/library/lib

执行source etc/profile

——> /etc/ld.so.cache文件列表(sudo vim /etc/ld.so.conf然后sudo ldconfig)

——> /lib/，/usr/lib目录找到库文件后将其载入内存(这种不推荐用)

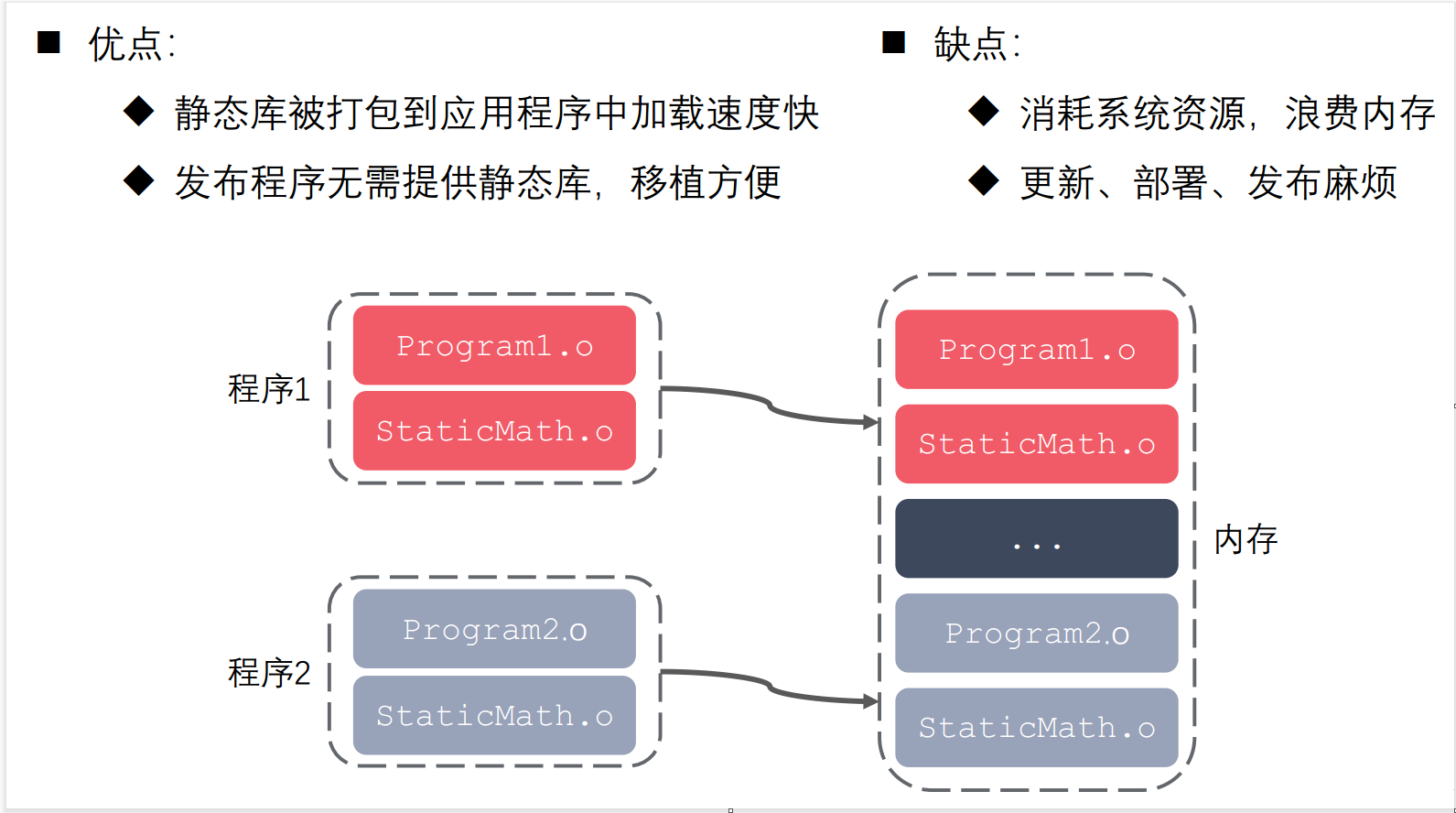
### 动态库和静态库的对比

#### 程序编译成可执行程序的过程

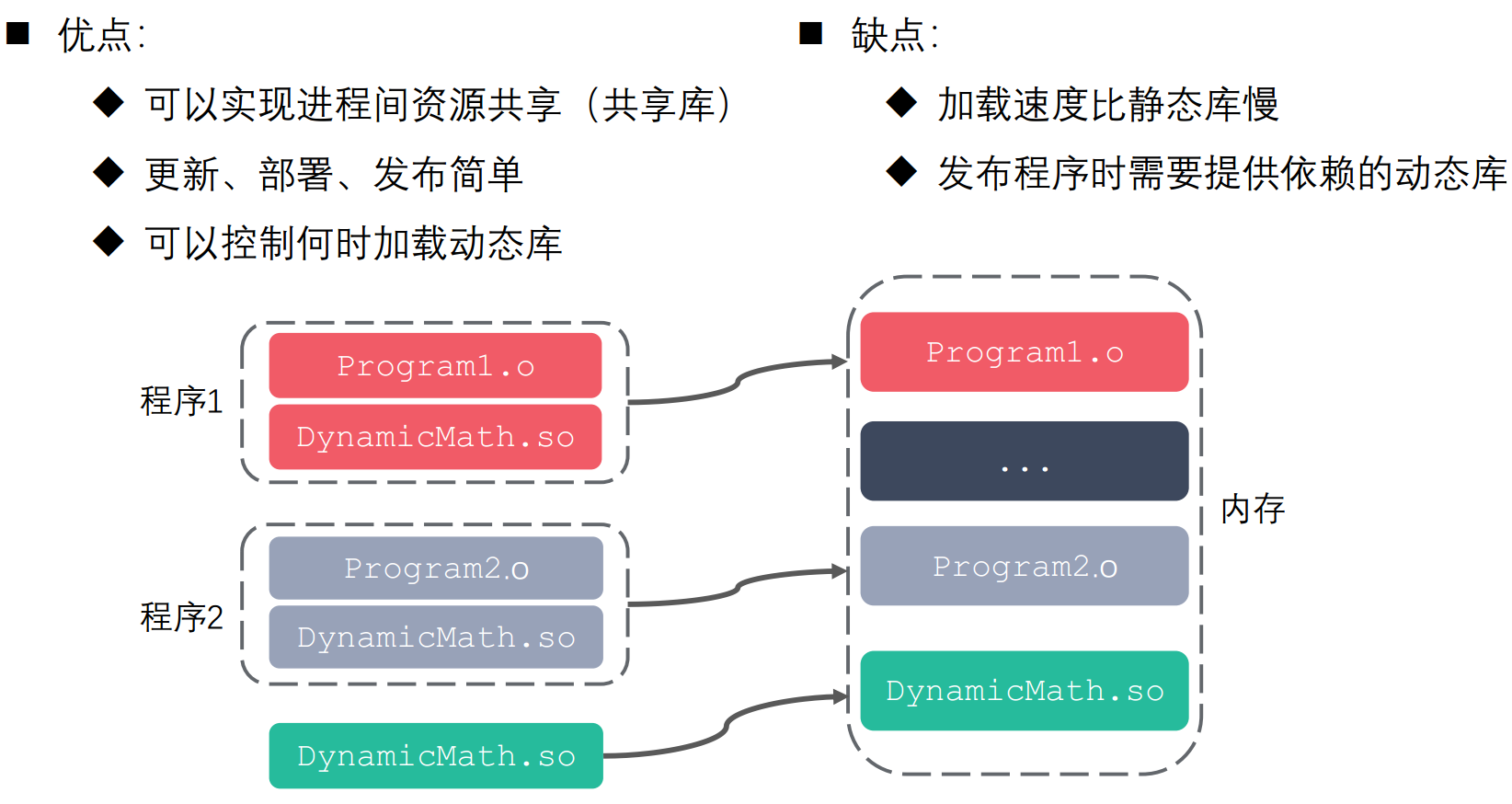
#### 5.2 静态库制作过程

#### 5.3 动态库制作过程

#### 5.4静态库的优缺点



#### 5.5 动态库的优缺点



### Makefile

#### 6.1 什么是Makefile

* 一个工程中的源文件不计其数，其按类型、功能、模块分别放在若干个目录中，Makefile文件定义了一系列的规则来指定哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重新编译，甚至于进行更复杂的功能操作，因为 Makefile文件就像一个Shell脚本一样，也可以执行操作系统的命令。
* Makefile 带来的好处就是“自动化编译”，一旦写好，只需要一个 make 命令，整个工程完全自动编译，极大的提高了软件开发的效率。make 是一个命令工具，是一个解释 Makefile 文件中指令的命令工具，一般来说，大多数的 IDE 都有这个命令，比如 Delphi 的 make，Visual C++ 的 nmake，Linux 下 GNU 的 make。

#### 6.2 Makefile 文件命名和规则

* 文件命名: makefile 或者 Makefile
* Makefile 规则
  + 一个 Makefile 文件中可以有一个或者多个规则

目标 ...: 依赖 ...

命令（Shell 命令）

...

* 目标：最终要生成的文件（伪目标除外）
* 依赖：生成目标所需要的文件或是目标
* 命令：通过执行命令对依赖操作生成目标（命令前必须 Tab 缩进）
* Makefile 中的其它规则一般都是为第一条规则服务的。

#### 6.3 工作原理

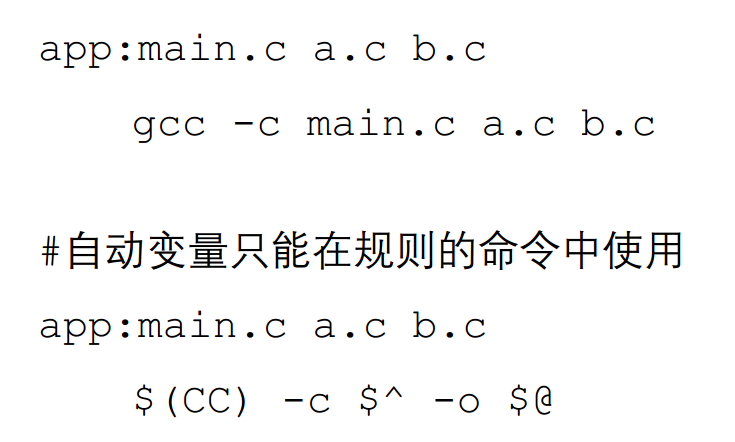
* 命令在执行之前，需要先检查规则中的依赖是否存在
  + 如果存在，执行命令
  + 如果不存在，向下检查其它的规则，检查有没有一个规则是用来生成这个依赖的，如果找到了，则执行该规则中的命令
* 检测更新，在执行规则中的命令时，会比较目标和依赖文件的更新时间
  + 如果依赖的时间比目标的时间晚，需要重新生成目标
  + 如果依赖的时间比目标的时间早，目标不需要更新，对应规则中的命令不需要被执行

#### 6.4 变量

* 自定义变量

变量名=变量值 var=hello $var取变量值

* 预定义变量

AR : 归档维护程序的名称，默认值为 ar

CC : C 编译器的名称，默认值为 cc

CXX : C++ 编译器的名称，默认值为 g++

$@ : 目标的完整名称

$< : 第一个依赖文件的名称

$^ : 所有的依赖文件

* 获取变量的值$(变量名)

#### 6.5 模式匹配

add.o:add.c

gcc -c add.c

div.o:div.c

gcc -c div.c

sub.o:sub.c

gcc -c sub.c

mult.o:mult.c

gcc -c mult.c

main.o:main.c

gcc -c main.c

%.o:%.c

- %: 通配符，匹配一个字符串

- 两个%匹配的是同一个字符串

%.o:%.c

gcc -c $< -o $@

#### 6.6 函数

$(wildcard PATTERN...)

 功能：获取指定目录下指定类型的文件列表

 参数：PATTERN 指的是某个或多个目录下的对应的某种类型的文件，如果有多个目录，一般使用空格间隔返回：得到的若干个文件的文件列表，文件名之间使用空格间隔

 示例：$(wildcard \*.c ./sub/\*.c)

返回值格式: a.c b.c c.c d.c e.c f.c

$(patsubst <pattern>,<replacement>,<text>)

 功能：查找<text>中的单词(单词以“空格”、“Tab”或“回车”“换行”分隔)是否符合模式<pattern>，如果匹配的话，则以<replacement>替换。

 注意： <pattern>可以包括通配符`%`，表示任意长度的字串。如果<replacement>

中也包含`%`，那么，<replacement>中的这个`%`将是<pattern>中的那个%所代表的字串。(可以用`\`来转义，以`\%`来表示真实含义的`%`字符)

 返回：函数返回被替换过后的字符串

 示例：$(patsubst %.c, %.o, x.c bar.c)

返回值格式: x.o bar.o

举例：

#定义变量

# add.c mult.c div.c sub.c main.c

src=$(wildcard ./\*.c )

#功能：获取指定目录下指定类型的文件列表

objs=$(patsubst %.c,%.o,$(src))

#功能：查找 <pattern> 中的单词(单词以“空格”、“ Tab ”或“回车”“换行”分隔)是否符合模式 <pattern>，如果匹配的话，则以 <text> 替换。

target = app

$(target):$(objs)

$(CC) $^ -o $(target)

%.o:%.c

$(CC) -c $< -o $@

.PHONY:clean

#.PHONY 来显示的指名一个目标是伪目标，有两个作用一个是改善性能，另外一个是来避免冲突。如果没有.PHONY且有一个叫clean的文件 make clean将一直是最新 无法删除目标

clean:

rm $(objs)

### GDB调试

#### 7.1什么是GDB

* GDB 是由 GNU 软件系统社区提供的调试工具，同 GCC 配套组成了一套完整的开发环境，GDB 是 Linux 和许多类 Unix 系统中的标准开发环境。
* 一般来说，GDB 主要帮助你完成下面四个方面的功能：

1. 启动程序，可以按照自定义的要求随心所欲的运行程序
2. 可让被调试的程序在所指定的调置的断点处停住（断点可以是条件表达式）
3. 当程序被停住时，可以检查此时程序中所发生的事
4. 可以改变程序，将一个 BUG 产生的影响修正从而测试其他 BUG

#### 7.2 准备工作

* 通常，在为调试而编译时，我们会关掉编译器的优化选项（`-O`）， 并打开调试选项（`-g`）。另外，`-Wall`在尽量不影响程序行为的情况下选项打开所有warning，也可以发现许多问题，避免一些不必要的 BUG。
* gcc -g -Wall program.c -o program
* `-g` 选项的作用是在可执行文件中加入源代码的信息，比如可执行文件中第几条机器指令对应源代码的第几行，但并不是把整个源文件嵌入到可执行文件中，所以在调试时必须保证 gdb 能找到源文件。

#### 7.3 GDB 命令 – 启动、退出、查看代码

1.启动和退出：gdb 可执行程序 quit 退出

2.给程序设置参数/获取设置参数：set args 10 20 show args

3.GDB使用帮助：help

4.查看当前文件代码：

list/l(从默认位置显示)

list/l 行号（从指定的行显示）

list/l 函数名（从指定的函数显示）

1. 查看非当前文件的代码

list/l 文件名:行号

list/l 文件名：函数名

1. 设置显示的行数

show list/listsize

set list/listsize

#### 7.4 GDB 命令 – 断点操作

1. 设置断点

b/break 行号

b/break 函数名

b/break 文件名：行号

b/break 文件名: 函数

1. 查看断点

i/info b/break

1. 删除断点

d/del/delete 断点编号

1. 设置断点无效

dis/disable 断点编号

1. 设置断点生效

ena/enable 断点编号

1. 设置条件断点（一般用在循环的位置）

b/break 10 if i==5

#### 7.5 GDB命令-调试命令

1.运行GDB程序

start(程序停在第一行)

run（遇到断点才停）

1. 继续运行到下一个断点停

c/continue

1. 向下执行一行代码

n/next（不会进入函数体）

1. 向下单步调试

s/step （会进入函数体）

finish(跳出函数体)

1. 变量操作

p/print 变量名（打印变量值）

ptype 变量名（打印变量类型）

1. 自动变量操作

display 变量名（自动打印指定变量的值）

i/info display 查看所有自动打印的变量

undisplay 编号

1. 其他操作

set var 变量名=变量值（循环中用的较多）

until 跳出循环

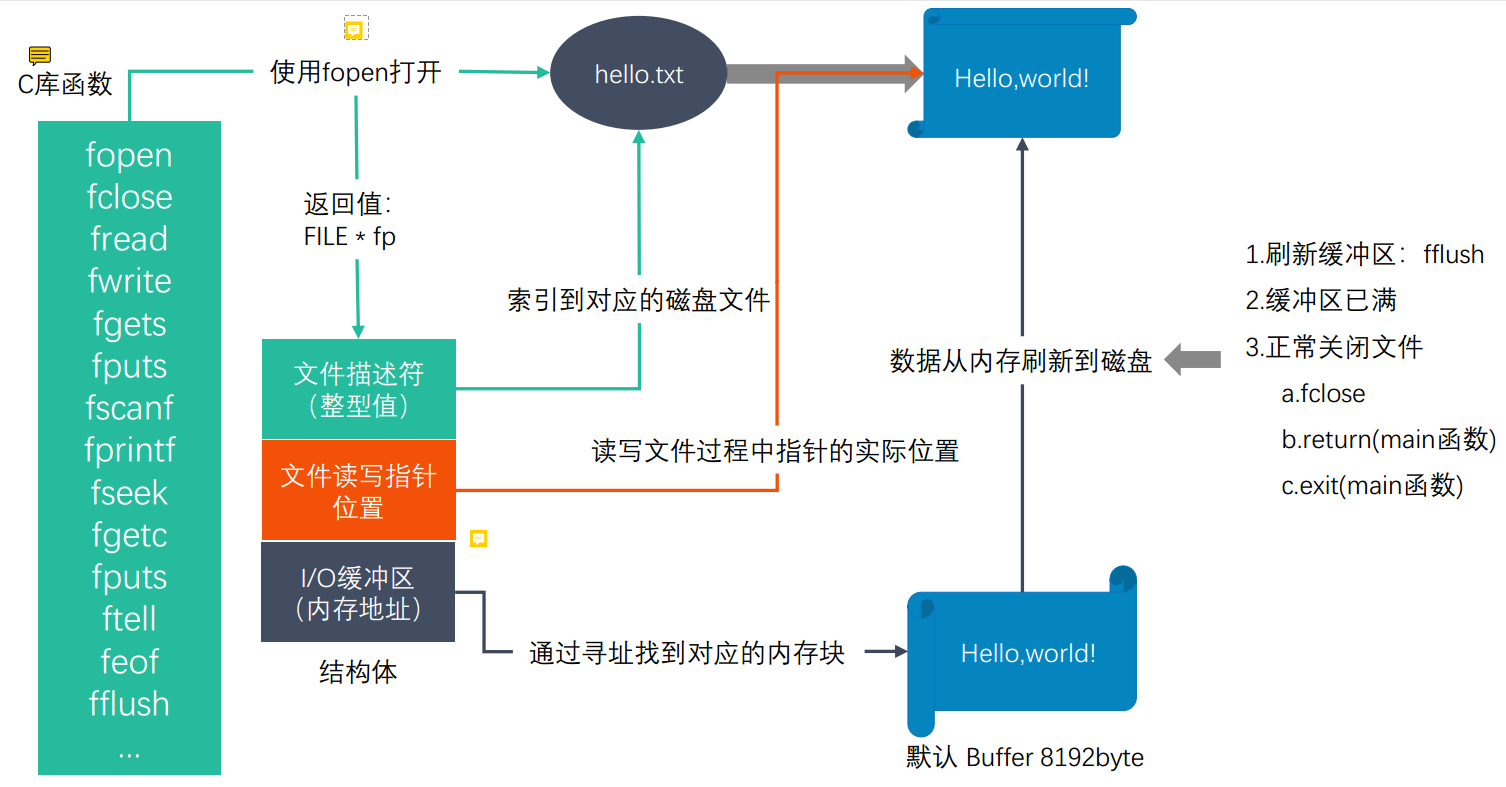
### 文件IO

#### 标准 C 库 IO 函数

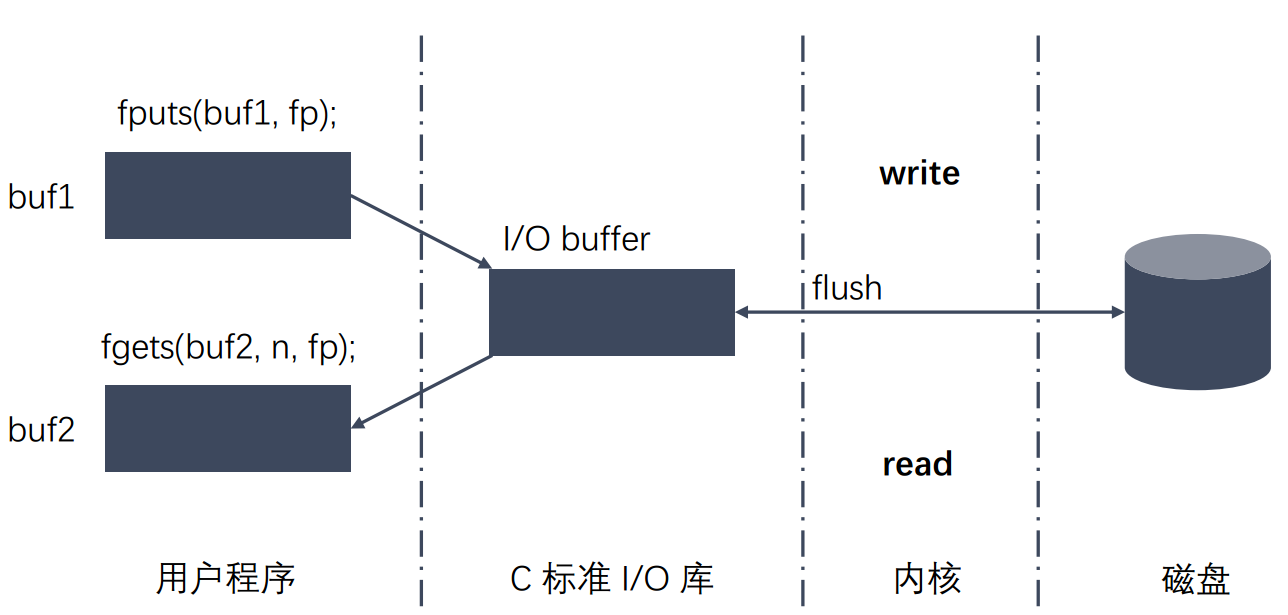
C库函数：C库函数可以通过调用不同的操作系统IO函数实现跨平台，Java不同平台的虚拟机实现跨平台，一般采用C库函数，但是网络通讯为了避免防止缓冲区的延时性，写代码时一般直接调用Linux系统的IO函数

man 3 fopen可以查看C库的一些标准函数说明

缓冲区（实际上在内存里）提高写入磁盘的效率，降低写磁盘的次数，源码中文件描述符是int接受Linux IO函数的返回值，文件读写指针和缓冲区指针都是char指针进行描述



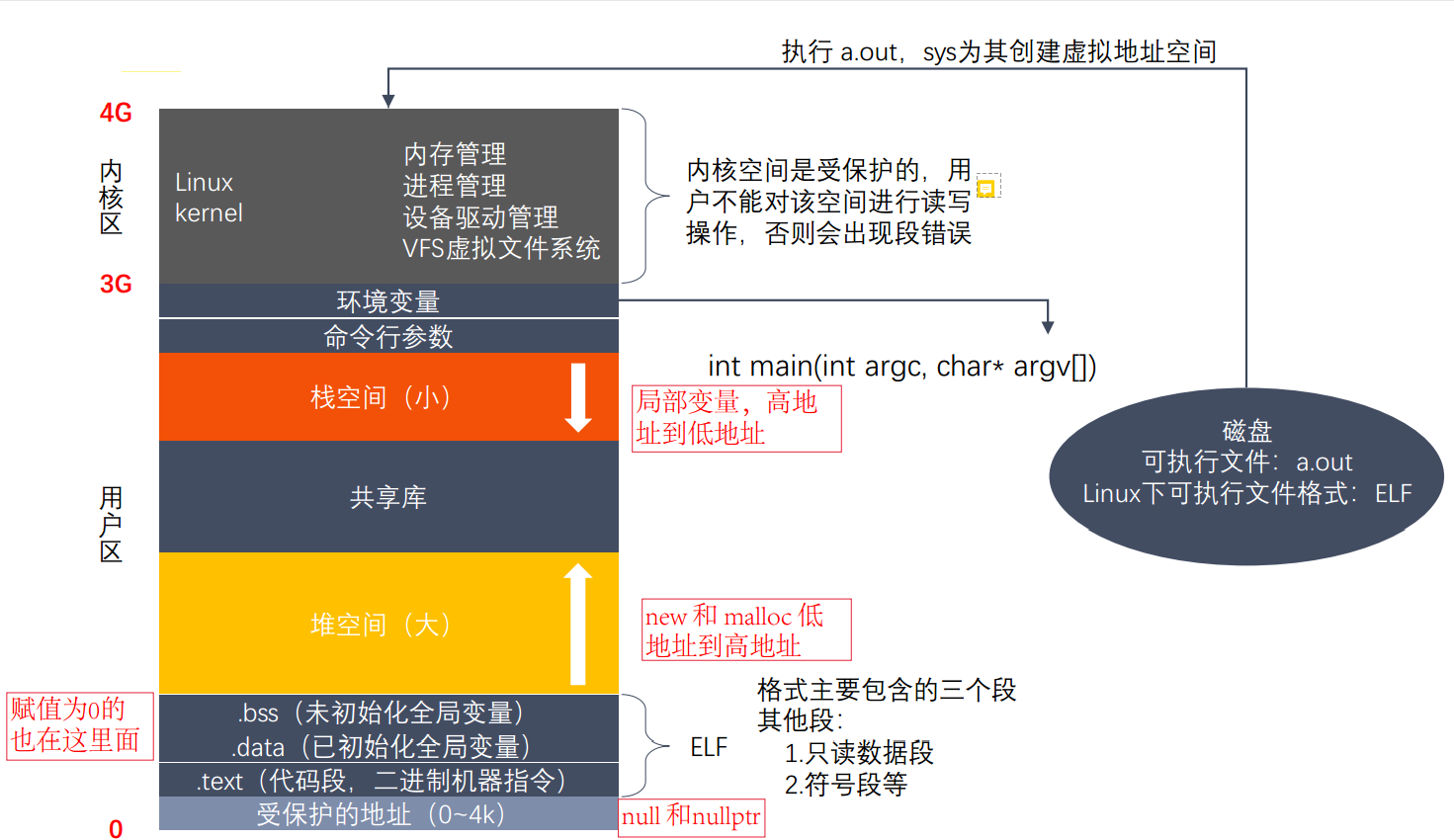
#### 8.2 标准C库IO和Linux系统IO的关系（调用与被调用的关系）



#### 8.3 虚拟地址空间

虚拟地址空间：事实上不存在的地址空间，可执行程序运行期间对应存在的虚拟地址空间，理解应用程序内存的变化 32位机器（2^32B=4G）64位机器（2^48B）

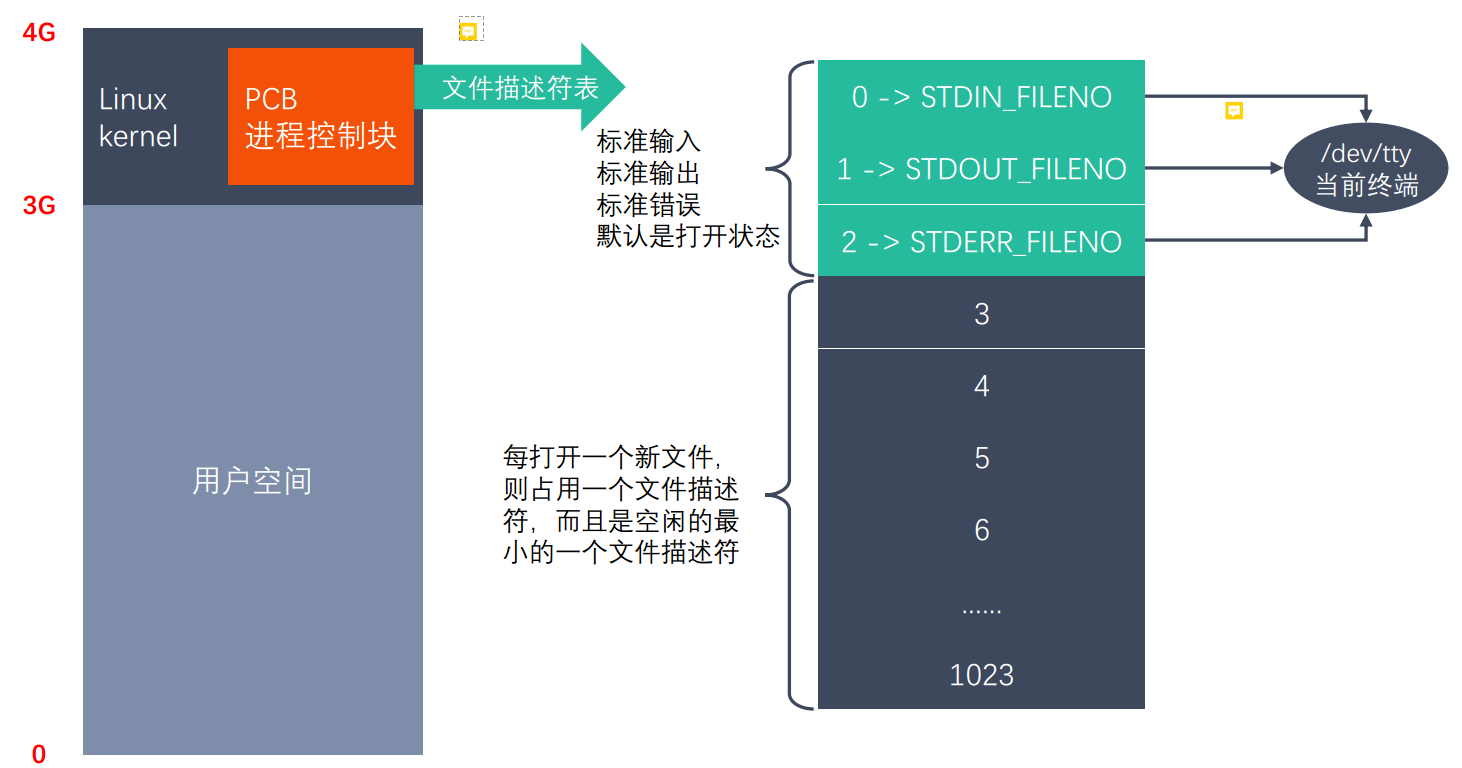
由MMU映射到物理内存中



#### 8.4 文件描述符

文件描述符表是数组，大小默认1024，因为一个进程会同时打开许多个（1024）文件，在内核区，PCB进程控制块实际上就是结构体。

Linux中万物皆文件，0/1/2默认打开对应当前终端，一个文件可以被一个进程同时打开多次，close时释放文件描述符；一般找最小的文件描述符来打开某个新的文件，系统IO传入的就不是FILE\*而是该文件描述符。



#### 8.5 Linux系统IO函数

Linux系统IO与C库IO对应，查询Linux IO可用man 2 open

##### 8.5.1 open函数

#include <sys/types.h> 包含flags宏定义

#include <sys/stat.h> 包含flags宏定义

#include <fcntl.h> open函数名在此

//创建一个新的文件

int open(const char \*pathname, int flags, mode\_t mode); //通过C语言的可变参数实现的而不是函数重载

参数：

-pathname： 要创建的文件的路径

-flags：对文件的操作权限和其他的设置

-必选项 O\_RDONLY, O\_WRONLY, or O\_RDWR 这三个设置是互斥的

-可选项 O\_CREATE 文件不存在创建新文件 O\_APPEND文件后面追加

-mode：八进制的数，表示创建出的新的文件的操作权限

比如：要使得最终文件的权限为0775那么mode应该取0777

最终的文件权限是： mode & ~umask rwx（读/写/可执行）

umask可在终端输入查看 111->7

~umask = 0777-0002(umask=0002)=0775

0777&0775=0775

umask的作用就是抹去某些权限使得权限合理 可以在linux终端改变（umask 022只在当前终端有作用） 也可以在程序里面设置

//打开一个已经存在的文件

int open(const char \*pathname, int flags);

参数：

- pathname：要打开的文件路径

- flags：对文件的操作权限设置还有其他的设置

O\_RDONLY, O\_WRONLY, or O\_RDWR 这三个设置是互斥的

返回值：返回一个新的文件描述符，如果调用失败返回-1

##### 8.5.2 perror函数

#include <stdio.h>

void perror(const char \*s);

s参数：用户描述，比如hello，输出内容hello：xxx(实际的错误描述)

作用：打印errno对应的错误描述

errno:属于Linux系统函数库，库里面的一个全局变量，记录的是最近的错误号

##### 8.5.3 close函数

int close(int fd); 关闭某个文件描述符的函数

##### 8.5.4 read函数

unixstd的意思这个头文件

#include <unistd.h>

ssize\_t read(int fd, void \*buf, size\_t count);

参数：

- fd：文件描述符，open得到的，通过这个文件描述符操作某个文件

- buf: 需要读取数据存放的地方，数组的地址（传出参数）

- count: 指定的数组的大小

返回值：

- 成功：

>0: 返回实际的读取到的字节数

=0: 文件已经读取完了

- 失败： -1，设置errno

##### 8.5.5 write函数

#include <unistd.h>

ssize\_t write(int fd, const void \*buf, size\_t count);

参数：

- fd：文件描述符，open得到的，通过这个文件描述符操作某个文件

- buf: 要往磁盘写入的数据，数据

- count: 要写的数据的实际的大小

返回值：

-成功：实际写入的字节数

-失败： -1，设置errno

##### 8.5.6 lseek函数

Linux系统函数

#include <sys/types.h>

#include <unistd.h>

off\_t lseek(int fd, off\_t offset, int whence);

参数：

- fd：文件描述符，通过open得到，通过fd操作某个文件

- offset：偏移量

- whence:

SEEK\_SET

设置文件指针偏移量：文件开始+第二个参数offset的值

SEEK\_CUR

设置偏移量：当前位置+第二个参数offset的值

SEEK\_END

设置偏移量：文件大小+第二个参数offset的值

返回值：返回文件指针的位置(偏移量)

作用:

1.移动文件指针到文件头

lseek(fd,0,SEEK\_SET);

2.获取当前文件指针的位置

lseek(fd,0,SEEK\_CUR);

3.获取文件的长度

lseek(fd,0,SEEK\_END);

4.拓展文件的长度（要写入一个数据），当前文件10b，增加100字节,变为110b

下载的时候会利用该函数占用这么大的内存，然后会拓展然后在慢慢写入相关文件lseek(fd,100,SEEK\_END); 这个必须写一个字符进去才会生效write(fd, " ", 1);

##### 8.5.7 stat函数

建立软连接的方式：ln -s 源文件 目标文件

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

stat本身也可以用作linux系统命令查看文件信息

最近更改是指内容，最近改动是指属性

int stat(const char \*pathname, struct stat \*statbuf);

作用：获取一个文件(不管软硬链接的源文件)相关的一些信息

参数:

-pathname：操作的文件的路径

-statbuf：结构体变量，传出参数，用于保存获取到的文件信息

返回值：

成功：返回0

失败：返回-1，设置errno

##### 8.5.8 lstat函数

int lstat(const char \*pathname, struct stat \*statbuf);

作用：获取一个软链接指向的文件相关的一些信息

参数:

-pathname：操作的文件的路径

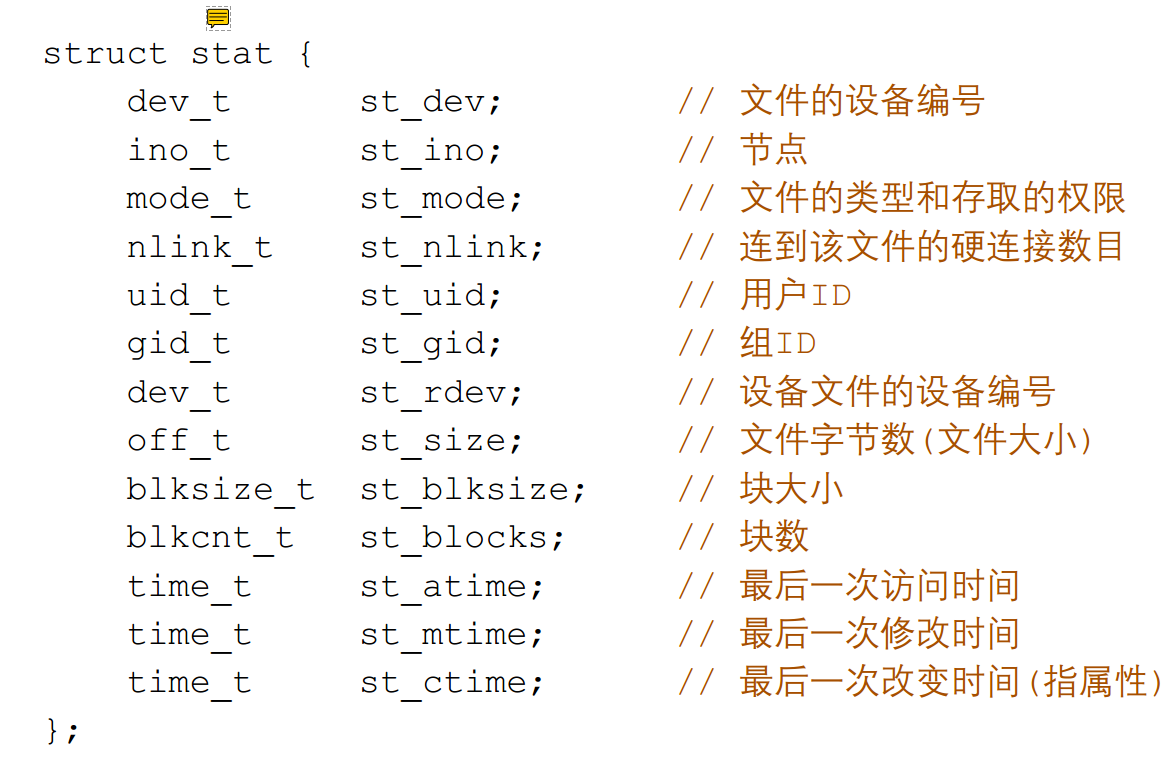
-statbuf：结构体变量，传出参数，用于保存获取到的文件信息

返回值：

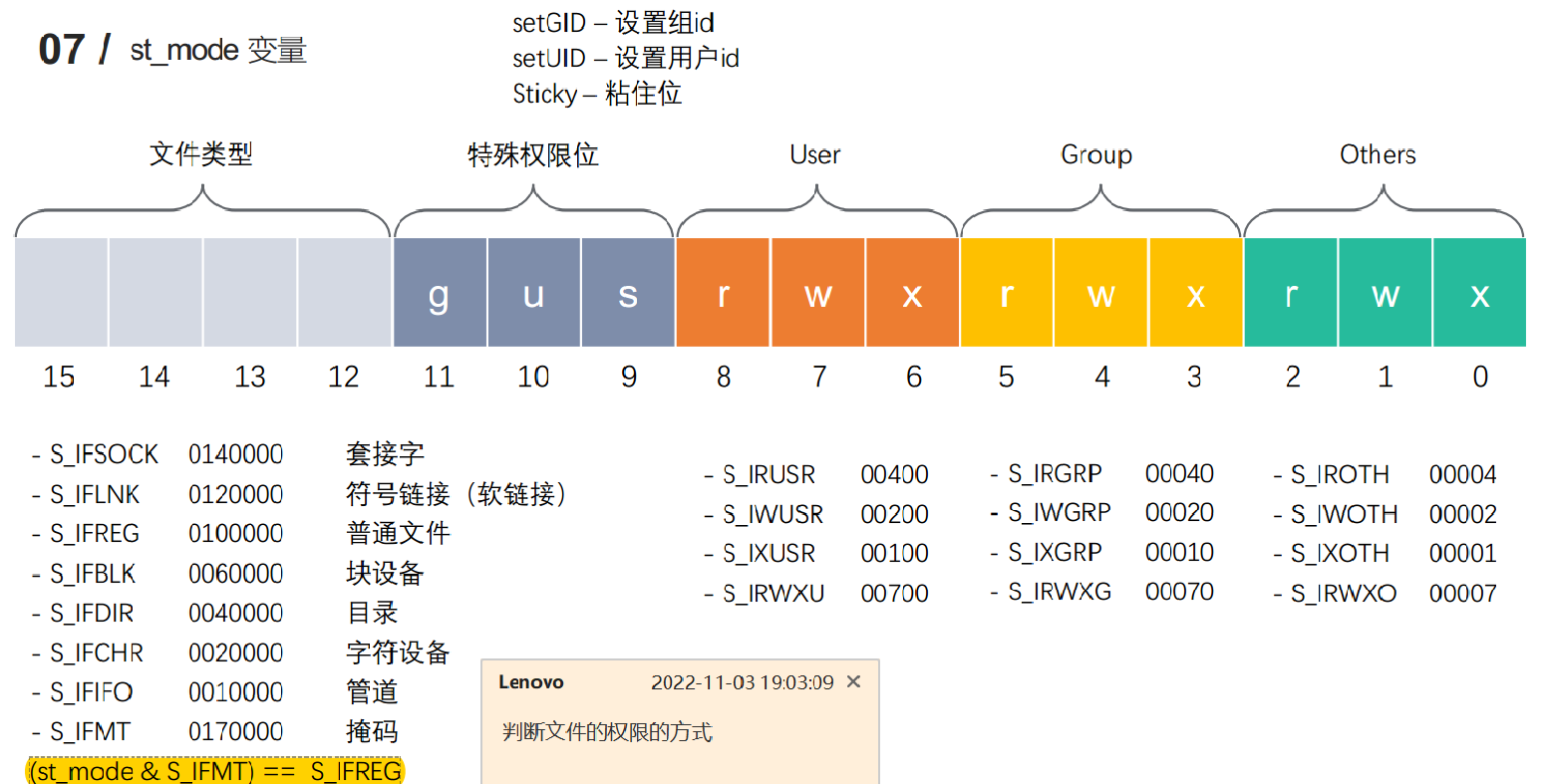
成功：返回0

失败：返回-1，设置errno

#### 8.6 stat结构



#### 8.7 st\_mode变量



#### 8.8 文件属性操作函数

##### 8.8.1 access函数

#include <unistd.h>

int access(const char \*pathname, int mode);

作用:判断某个文件是否有某个权限，或者判断文件是否存在

参数：

-pathname：判断的文件路径

-mode

R\_OK:判断是否有读权限

W\_OK:判断是否有写权限

X\_OK:判断是否有执行权限

F\_OK:判断文件是否存在

返回值: 成功返回0， 失败返回-1

##### 8.8.2 chmod函数

vim/etc/passwd 可以看到所有用户的uid和gid

vim/etc/group 可以看到所有组的gid

sudo useradd 用户名 可以添加用户,id 用户名可以查看uid和gid

#include <sys/stat.h>

int chmod(const char \*pathname, mode\_t mode);

作用：修改文件的权限

参数：

- pathname: 需要修改的文件的路径

- mode:需要修改的权限值，八进制的数

返回：

成功返回0 失败返回-1

##### 8.8.3 truncate函数

#include <unistd.h>

#include <sys/types.h>

int truncate(const char \*path, off\_t length);

作用:缩减或者扩展文件的尺寸到指定大小

参数:

- path: 需要修改的文件路径

-length: 需要最终文件变成的大小 缩减砍掉最后一部分 扩展用空字符替换

#### 8.9 目录操作函数

##### 8.9.1 rename函数

int rename(const char \*oldpath, const char \*newpath);

##### 8.9.2 chdir函数

#include <unistd.h>

int chdir(const char \*path);

作用：修改进程的工作目录

比如在home/williamwhites 启动一个可执行程序a.out 进程的工作目录/home/williamwhites

参数：

path: 需要修改的工作目录

##### 8.9.3 getcwd函数

#include <unistd.h>

char \*getcwd(char \*buf, size\_t size);

作用：获取当前的工作目录

参数：

- buf: 存储的路径，指向的是一个数组(传出参数)

- size: 数组的大小

返回值:

返回值指向一块内存，这个数据就是第一个参数

##### 8.9.4 mkdir函数

#include <sys/stat.h>

#include <sys/types.h>

int mkdir(const char \*pathname, mode\_t mode); 有x权限才能够进入该目录

作用：创建一个目录

参数:

pathname:创建的目录的路径

mode:权限，8进制数,会与掩码相与遮掉某些权限

返回值：

成功返回0，失败返回-1

#### 8.10 目录遍历函数

##### 8.10.1 opendir函数

//打开一个目录

#include <sys/types.h>

#include <dirent.h>

DIR \*opendir(const char \*name);

参数：

- name: 需要打开的目录的名称

返回值：

DIR\*类型， 目录流信息

错误返回NULL

##### 8.10.2 readdir函数

//读取目录中的数据

#include <dirent.h>

struct dirent \*readdir(DIR \*dirp);

-参数： dirp是opendir返回的结果

-返回值：struct dirent 代表读取到的文件的信息

读取到了末尾或者失败了，返回NULL

##### 8.10.3 closedir函数

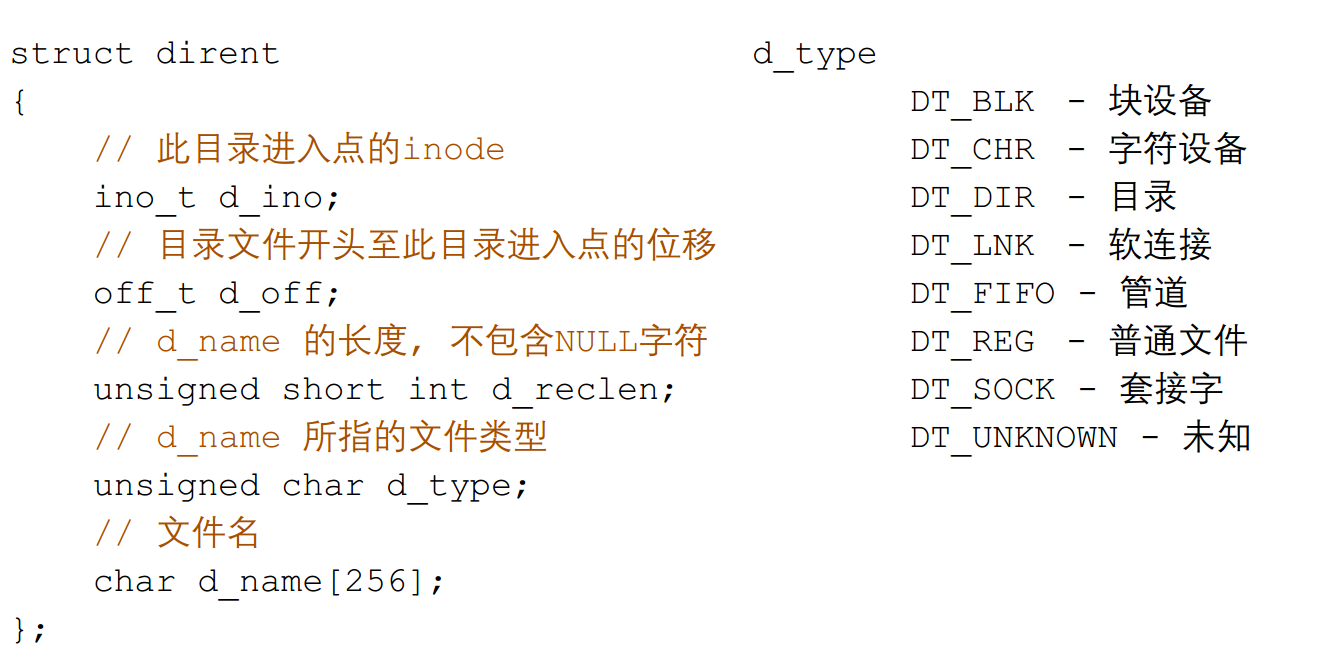
//关闭目录

#include <sys/types.h>

#include <dirent.h>

int closedir(DIR \*drip);

#### 8.11 dirent结构体和d\_type



#### 8.12 dup和dup函数

##### 8.12.1 dup函数

#include <unistd.h>

int dup(int oldfd);

作用：复制一个新的文件描述符

fd = 3, int fd1 = dup(fd),

fd指向a.txt,fd1也指向a.txt

从空闲的文件描述符表中找一个最小的，作为新的拷贝的文件描述符

##### 8.12.2 dup2函数

#include <unistd.h>

int dup2(int oldfd, int newfd);

作用：重定向文件描述符

oldfd 指向 a.txt, newfd 指向 b.txt

调用函数后：

newfd 和 b.txt 做close，newfd 指向了 a.txt

oldfd必须是一个有效的文件描述符

oldfd和newfd值相同，相当于什么都没做

返回值是newfd的值

#### 8.13 fcntl函数

#include <unistd.h>

#include <fcntl.h>

int fcntl(int fd, int cmd, ...);

参数:

fd:表示需要操作的文件描述符

cmd：表示对文件描述符进行如何操作

- F\_DUPFD : 复制文件描述符,复制的是第一个参数fd，得到一个新的文件描述符（返回值）

int ret = fcntl(fd,F\_DUPFD);

- F\_GETFL: 获取指定的文件描述符文件状态的flag

获取的flag和我们通过open函数传递的flag是一个东西

- F\_SETFL: 设置文件描述符文件状态flag

必选项:O\_RDONLY,O\_WRONLY,O\_RDWR 不可以被修改

可选项:O\_APPEND, O\_NONBLOCK 可以被修改

O\_APPEND 表示追加数据,但不能更改是否有写权限

O\_NONBLOCK 设置成非阻塞

阻塞和非阻塞：描述的是函数调用的行为。

## Linux多进程开发

### 进程概述

#### 程序和进程

程序是包含一系列信息的文件，这些信息描述了如何在运行时创建一个进程组：

* 二进制格式标识：每个程序文件都包含用于描述可执行文件格式的元信息。内核利用此信息来解释文件中的其他信息。（ELF可执行连接格式）（如何连接起来）
* 机器语言指令：对程序算法进行编码。
* 程序入口地址：标识程序开始执行时的起始指令位置。
* 数据：程序文件包含的变量初始值和程序使用的字面量值（比如字符串）。
* 符号表及重定位表：描述程序中函数和变量的位置及名称。这些表格有多重用途，其中包括调试和运行时的符号解析（动态链接）。
* 共享库和动态链接信息：程序文件所包含的一些字段，列出了程序运行时需要使用的共享库，以及加载共享库的动态连接器的路径名
* 其他信息：程序文件还包含许多其他信息，用以描述如何创建进程。

进程是正在运行的程序的实例。是一个具有一定独立功能的程序关于某个数据集合的一次运行活动。它是操作系统动态执行的基本单元，在传统的操作系统中，进程既是基本的分配单元，也是基本的执行单元。

可以用一个程序来创建多个进程，进程是由内核定义的抽象实体，并为该实体分配用以执行程序的各项系统资源。从内核的角度看，进程由用户内存空间和一系列内核数据结构组成，其中用户内存空间包含了程序代码及代码所使用的变量，而内核数据结构则用于维护进程状态信息。记录在内核数据结构中的信息包括许多与进程相关的标识号（IDs）、虚拟内存表、打开文件的描述符表、信号传递及处理的有关信息、进程资源使用及限制、当前工作目录和大量的其他信息。

#### 单道程序设计和多道程序设计

* 单道程序，即在计算机内存中只允许一个的程序运行。
* 多道程序设计技术是在计算机内存中同时存放几道相互独立的程序，使它们在管理程序控制下，相互穿插运行，两个或两个以上程序在计算机系统中同处于开始到结束之间的状态, 这些程序共享计算机系统资源。引入多道程序设计技术的根本目的是为了提高 CPU 的利用率。
* 对于一个单 CPU 系统来说，程序同时处于运行状态只是一种宏观上的概念，他们虽然都已经开始运行，但就微观而言，任意时刻，CPU 上运行的程序只有一个。
* 在多道程序设计模型中，多个进程轮流使用 CPU。而当下常见 CPU 为纳秒级，1秒可以执行大约 10 亿条指令。由于人眼的反应速度是毫秒级，所以看似同时在运行。

#### 时间片

* 时间片（timeslice）又称为“量子（quantum）”或“处理器片（processor slice）”

是操作系统分配给每个正在运行的进程微观上的一段 CPU 时间。事实上，虽然一台计算机通常可能有多个 CPU，但是同一个 CPU 永远不可能真正地同时运行多个任务。在只考虑一个 CPU 的情况下，这些进程“看起来像”同时运行的，实则是轮番穿插地运行，由于时间片通常很短（在 Linux 上为 5ms－800ms），用户不会感觉到。

* 时间片由操作系统内核的调度程序分配给每个进程。首先，内核会给每个进程分配相等的初始时间片，然后每个进程轮番地执行相应的时间，当所有进程都处于时间片耗尽的状态时，内核会重新为每个进程计算并分配时间片，如此往复。

#### 并行和并发

并行(parallel)：指在同一时刻，有多条指令在多个处理器上同时执行。并行是两个队列同时使用两台咖啡机。

并发(concurrency)：指在同一时刻只能有一条指令执行，但多个进程指令被快速的轮换执行，使得在宏观上具有多个进程同时执行的效果，但在微观上并不是同时执行的，只是把时间分成若干段，使多个进程快速交替的执行。并发是两个队列交替使用一台咖啡机。

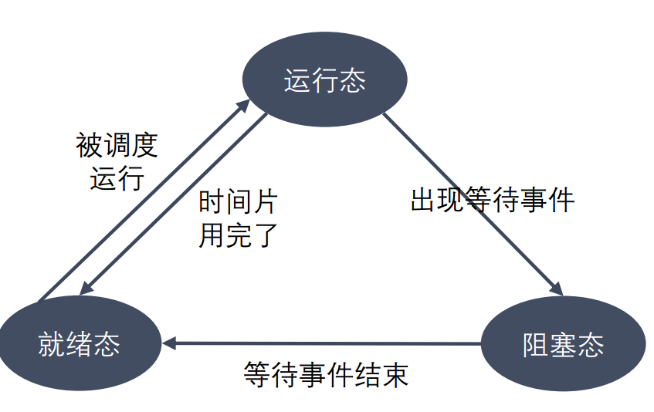
#### 进程控制块

* 为了管理进程，内核必须对每个进程所做的事情进行清楚的描述。内核为每个进程分配一个 PCB(Processing Control Block)进程控制块，维护进程相关的信息，Linux 内核的进程控制块是 task\_struct 结构体。
* 在 /usr/src/linux-headers-xxx/include/linux/sched.h 文件中可以查看 struct task\_struct 结构体定义。其内部成员有很多，我们只需要掌握以下部分即可：
* 进程id：系统中每个进程有唯一的 id，用 pid\_t 类型表示，其实就是一个非负整数进程的状态：有就绪、运行、挂起、停止等状态
* 进程切换时需要保存和恢复的一些CPU寄存器
* 描述虚拟地址空间的信息
* 描述控制终端的信息
* 当前工作目录（Current Working Directory）umask 掩码文件描述符表，包含很多指向 file 结构体的指针和信号相关的信息
* 用户 id 和组 id
* 会话（Session）和进程组
* 进程可以使用的资源上限（Resource Limit）（Iinux命令 ulimit -a显示资源上限 ulimit -n open files可以改默认上限）

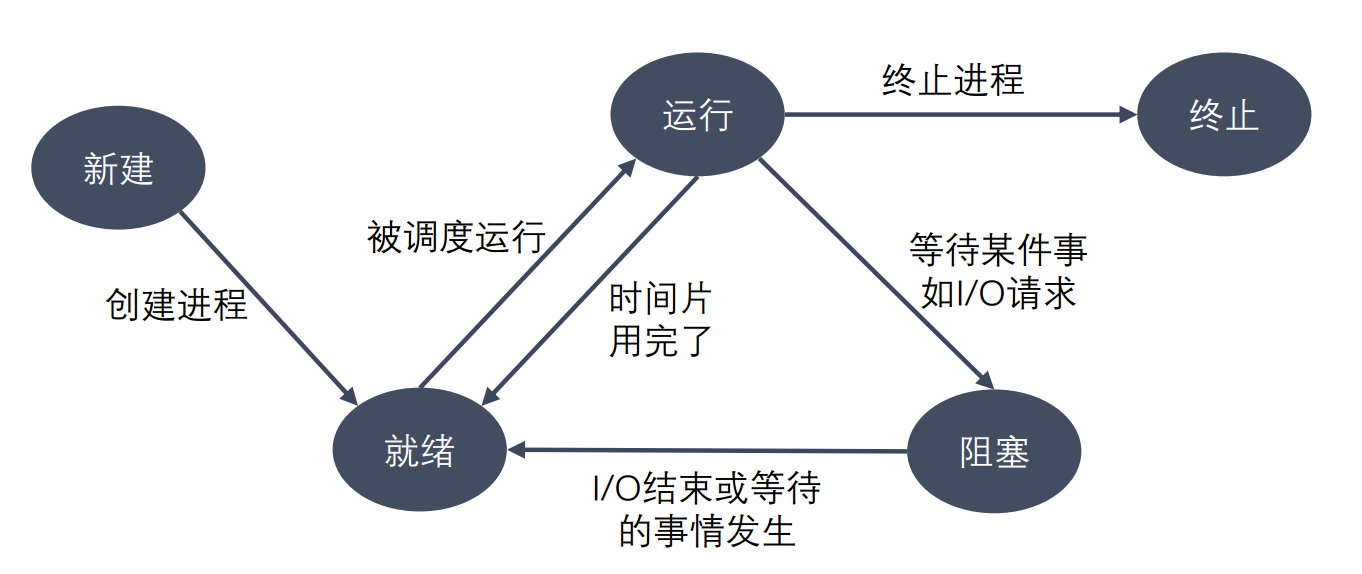
### 进程状态转换

#### 进程的状态

进程状态反映进程执行过程的变化。这些状态随着进程的执行和外界条件的变化而转换。在三态模型中，进程状态分为三个基本状态，即就绪态，运行态，阻塞态。在五态模型中，进程分为新建态、就绪态，运行态，阻塞态，终止态。

运行态：进程占有处理器正在运行

就绪态：进程具备运行条件，等待系统分配处理器以便运行。当进程已分配到除CPU以外的所有必要资源后，只要再获得CPU，便可立即执行。在一个系统中处于就绪状态的进程可能有多个，通常将它们排成一个队列，称为就绪队列

阻塞态：又称为等待(wait)态或睡眠(sleep)态，指进程不具备运行条件，正在等待某个事件的完成

新建态：进程刚被创建时的状态，尚未进入就绪队列

终止态：进程完成任务到达正常结束点，或出现无法克服的错误而异常终止，或被操作系统及有终止权的进程所终止时所处的状态。进入终止态的进程以后不再执行，但依然保留在操作系统中等待善后。一旦其他进程完成了对终止态进程的信息抽取之后，操作系统将删除该进程。

#### 进程相关命令

##### 2.2.1 查看进程：

tty查看当前对应的终端是什么

PPID 表示父进程

PID表示该进程

PGID表示组进程的ID

ps aux / ajx

a：显示终端上的所有进程，包括其他用户的进程u：显示进程的详细信息

j：列出与作业控制相关的信息

x：显示没有控制终端的进程

##### 2.2.2 STAT参数意义：

D 不可中断 Uninterruptible（usually IO）

R 正在运行，或在队列中的进程

S(大写) 处于休眠状态

T 停止或被追踪

Z 僵尸进程W 进入内存交换（从内核2.6开始无效）

X 死掉的进程

< 高优先级

N 低优先级

s 包含子进程

+ 位于前台的进程组

##### 2.2.3 实时显示进程动态

top

可以在使用 top 命令时加上 -d 来指定显示信息更新的时间间隔，在 top 命令

执行后，可以按以下按键对显示的结果进行排序：

M 根据内存使用量排序

P 根据 CPU 占有率排序

T 根据进程运行时间长短排序

U 根据用户名来筛选进程

K 输入指定的 PID 杀死进程

##### 2.2.4 杀死进程

./a.out & 可以实现后台运行，仍然可以输入命令

kill [-signal] pid

kill –l 列出所有信号

kill –SIGKILL 进程ID

kill -9 进程ID 强制杀死某个进程，包括当前终端也可以

killall name 根据进程名杀死进程

#### 进程号和相关函数

* 每个进程都由进程号来标识，其类型为 pid\_t（整型），进程号的范围：0～32767。

进程号总是唯一的，但可以重用。当一个进程终止后，其进程号就可以再次使用。

* 任何进程（除 init 进程）都是由另一个进程创建，该进程称为被创建进程的父进程，

对应的进程号称为父进程号（PPID）。

* 进程组是一个或多个进程的集合。他们之间相互关联，进程组可以接收同一终端的各

种信号，关联的进程有一个进程组号（PGID）。默认情况下，当前的进程号会当做当

前的进程组号

进程号和进程组相关函数：

pid\_t getpid(void); pid\_t getppid(void); pid\_t getpgid(pid\_t pid);

#### 进程创建

系统允许一个进程创建新进程，新进程即为子进程，子进程还可以创建新的子进程，形成

进程树结构模型。

##### 2.4.1 fork函数

#include <sys/types.h>

#include <unistd.h>

pid\_t fork(void);

作用：用于创建子进程

返回值：

fork()的返回值返回2次，一次是在父进程，一次是在子进程

在父进程中返回创建的子进程的ID，

在子进程中返回0

如何区分父进程和子进程：通过fork的返回值。

在父进程中返回-1，表示创建子进程失败，并且设置errno

失败的两个主要原因：

1. 当前系统的进程数已经达到了系统规定的上限，这时 errno 的值被设置为 EAGAIN

2. 系统内存不足，这时 errno 的值被设置为 ENOMEM

##### 2.4.2 父子进程虚拟地址空间

父子进程之间的关系：

区别：

1.fork()函数的返回值不同

父进程中: >0 返回的是子进程的ID

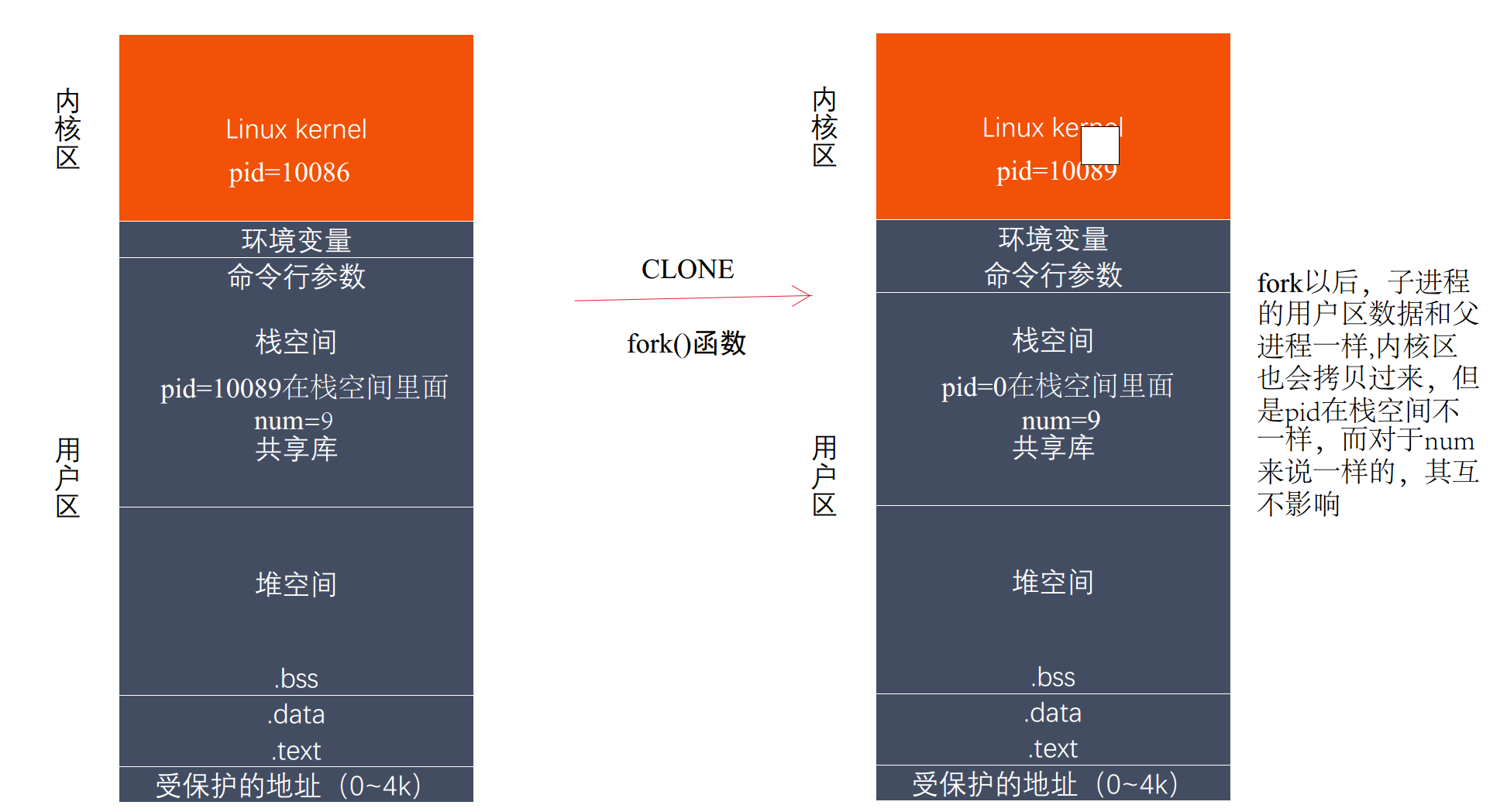
子进程中: =0

2.pcb中的一些数据

当前进程的id pid

当前进程的父进程的id ppid

信号集

 共同点:

某些状态下：子进程刚被创建出来，还没有执行任何的写数据的操作

- 用户区的数据

- 文件描述符表

父子进程对变量是不是共享的？

- 刚开始的时候，是一样的，共享的。如果修改了数据，不共享了。

 - 读时共享（子进程被创建，两个进程没有进行任何写的操作），写时拷贝。

##### 2.4.3 GDB多进程调试

使用 GDB 调试的时候，GDB 默认只能跟踪一个进程，可以在 fork 函数调用之前，通

过指令设置 GDB 调试工具跟踪父进程或者是跟踪子进程，默认跟踪父进程。设置调试父进程或者子进程：set follow-fork-mode [parent（默认）| child]

设置调试模式：set detach-on-fork [on | off]

默认为 on，表示调试当前进程的时候，其它的进程继续运行，如果为 off，调试当前进

程的时候，其它进程被 GDB 挂起。

查看调试的进程：info inferiors切换当前调试的进程：inferior id

使进程脱离 GDB 调试：detach inferiors id

#### exce函数族（实现相同或者相似功能的一些函数组成的集体）

##### 2.5.1 exec函数族介绍

exec 函数族的作用是根据指定的文件名找到可执行文件，并用它来取代调用进程的

内容，换句话说，就是在调用进程内部执行一个可执行文件。

exec 函数族的函数执行成功后不会返回，因为调用进程的实体，包括代码段，数据

段和堆栈等都已经被新的内容取代，只留下进程 ID 等一些表面上的信息仍保持原样，

颇有些神似“三十六计”中的“金蝉脱壳”。看上去还是旧的躯壳，却已经注入了新的灵

魂。只有调用失败了，它们才会返回 -1，从原程序的调用点接着往下执行。

##### 2.5.2 execl函数族作用图解



##### 2.5.3 exec函数族的使用

l(list) 参数地址列表，以空指针结尾 （execl("/bin/ps","ps","aux",NULL);）

v(vector) 存有各参数地址的指针数组的地址

（char \*argv[] = {"ps","aux",NULL};execv("/bin/ps",argv);）

p(path) 按PATH(未指定envp就是系统环境变量)环境变量指定的目录搜索可执行文件

e(environment)存有环境变量字符串地址的指针数组的地址

char\* envp[] = {"/home/williamwhites/","/home/bbb","/home/aaa"};

path指要输入路径，而name表示指明文件名即可

#include <unistd.h>

int execl(const char \*path, const char \*arg, ...);

- 参数:

- path:需要执行的文件的路径

a.out /home/williamwhites/a.out 推荐使用绝对路径

./a.out hello world

- arg:是执行可执行文件所需要的参数列表

第一个参数一般没有什么作用，为了方便，一般写的是可执行程序的名称

从第二个参数开始 就是程序执行所需要的列表。

参数最后需要以NULL结束(哨兵)

- 返回值：

只有调用失败时候，才会有返回值，返回-1并设置errno

#include <unistd.h>

int execlp(const char \*file, const char \*arg, ...);

会到环境变量查找指定的可执行文件，如果找到了就执行，找不到就执行不成功

- 参数:

- file:需要执行的文件的文件名(不写路径)

a.out ps

./a.out hello world

- arg:是执行可执行文件所需要的参数列表

第一个参数一般没有什么作用，为了方便，一般写的是可执行程序的名称

从第二个参数开始 就是程序执行所需要的列表。

参数最后需要以NULL结束(哨兵)

- 返回值：

只有调用失败时候，才会有返回值，返回-1并设置errno

int execv(const char\* path,char \*const argv[]);

argv是需要的参数的一个字符串数组

char \*argv[] = {"ps","aux",NULL};

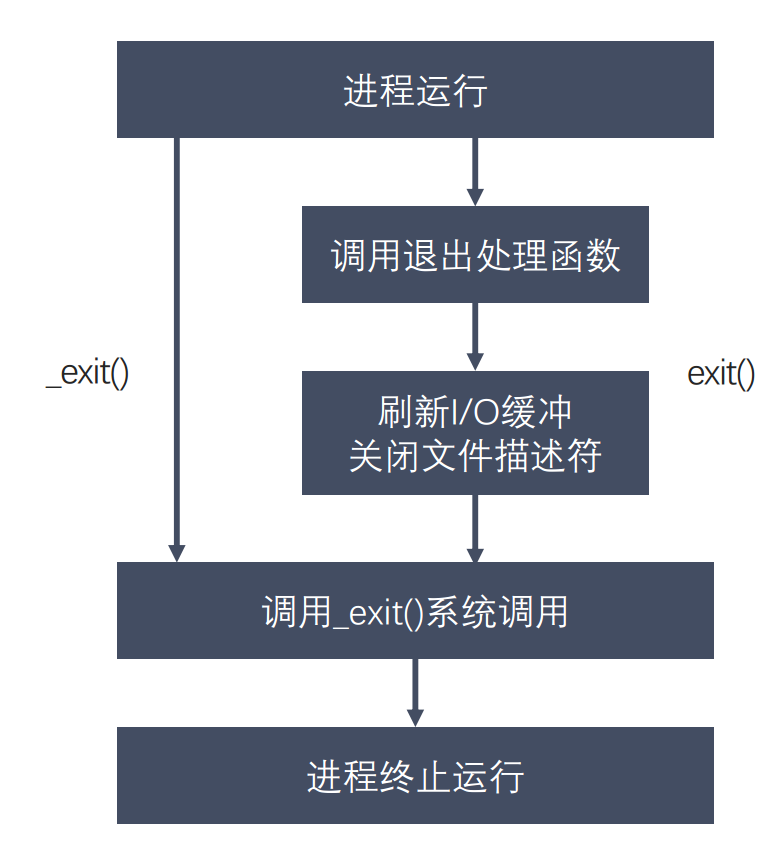
execv("/bin/ps",argv);

int execve(const char\* filename, char\* const argv[],char\* const envp[]);

char\* envp[] = {"/home/williamwhites/","/home/bbb","/home/aaa"};

#### 进程控制

##### 2.6.1 进程退出



#include <stdlib.h>

void exit(int status);

#include <unistd.h>

void \_exit(int status);

status参数:是进程退出的一个状态信息，父进程回收子进程资源的时候可以获取到

##### 2.6.2 孤儿进程

* 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 ，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。

##### 2.6.3僵尸进程

* 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。

* 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。

##### 2.6.4 进程回收

* 在每个进程退出的时候，内核释放该进程所有的资源、包括打开的文件、占用的内

存等。但是仍然为其保留一定的信息，这些信息主要主要指进程控制块PCB的信息

（包括进程号、退出状态、运行时间等）。

* 父进程可以通过调用wait或waitpid得到它的退出状态同时彻底清除掉这个进程。
* wait() 和 waitpid() 函数的功能一样，区别在于，wait() 函数会阻塞，waitpid() 可以设置不阻塞，waitpid() 还可以指定等待哪个子进程结束。
* 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。

#include <sys/types.h>

#include <sys/wait.h>

pid\_t wait(int \*wstatus);

功能:等待任意一个子进程结束，如果任意一个子进程结束了，此函数会回收子进程的资源。

参数:int\* wstatus

进程退出时的状态信息，传入的是一个int类型的地址，传出参数。

返回值：

- 成功: 返回被回收的子进程的id

- 失败: -1(所有的子进程都结束/调用函数失败)

调用wait函数的进程会被挂起(阻塞)，直到它的一个子进程退出或者收到一个不能忽略的信号时才被唤醒(继续往下进行)

如果没有子进程，函数会立刻被返回，返回-1；如果子进程都已经结束了，也会立即返回，返回-1

#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

参数:

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，回收指定进程组的子进程

options :

0 : 阻塞

WNOHANG: 非阻塞

返回值:

>0: 返回子进程的id

=0： options = WNOHANG，表示还有子进程

=-1：调用函数失败，或者没有子进程了

##### 2.6.5 退出信息相关宏函数

WIFEXITED(status) 非0，进程正常退出

WEXITSTATUS(status) 如果上宏为真，获取进程退出的状态（exit的参数）

WIFSIGNALED(status) 非0，进程异常终止

WTERMSIG(status) 如果上宏为真，获取使进程终止的进程编号

WIFSTOPPED(status) 非0，进程处于暂停状态

WSTOPSIG(status) 如果上宏为真，获取使进程暂停的信号的编号WIFCONTINUED(status) 非0，进程暂停后已经继续运行

### 3.进程间的通信

#### 3.1 进程间通信的概念

* 进程是一个独立的资源分配单元，不同进程（这里所说的进程通常指的是用户进程）之间

的资源是独立的，没有关联，不能在一个进程中直接访问另一个进程的资源。

* 但是，进程不是孤立的，不同的进程需要进行信息的交互和状态的传递等，因此需要进程

间通信( IPC：Inter Processes Communication )。

* 进程间通信的目的：
  + 数据传输：一个进程需要将它的数据发送给另一个进程。
  + 通知事件：一个进程需要向另一个或一组进程发送消息，通知它（它们）发生了某种

事件（如进程终止时要通知父进程）。

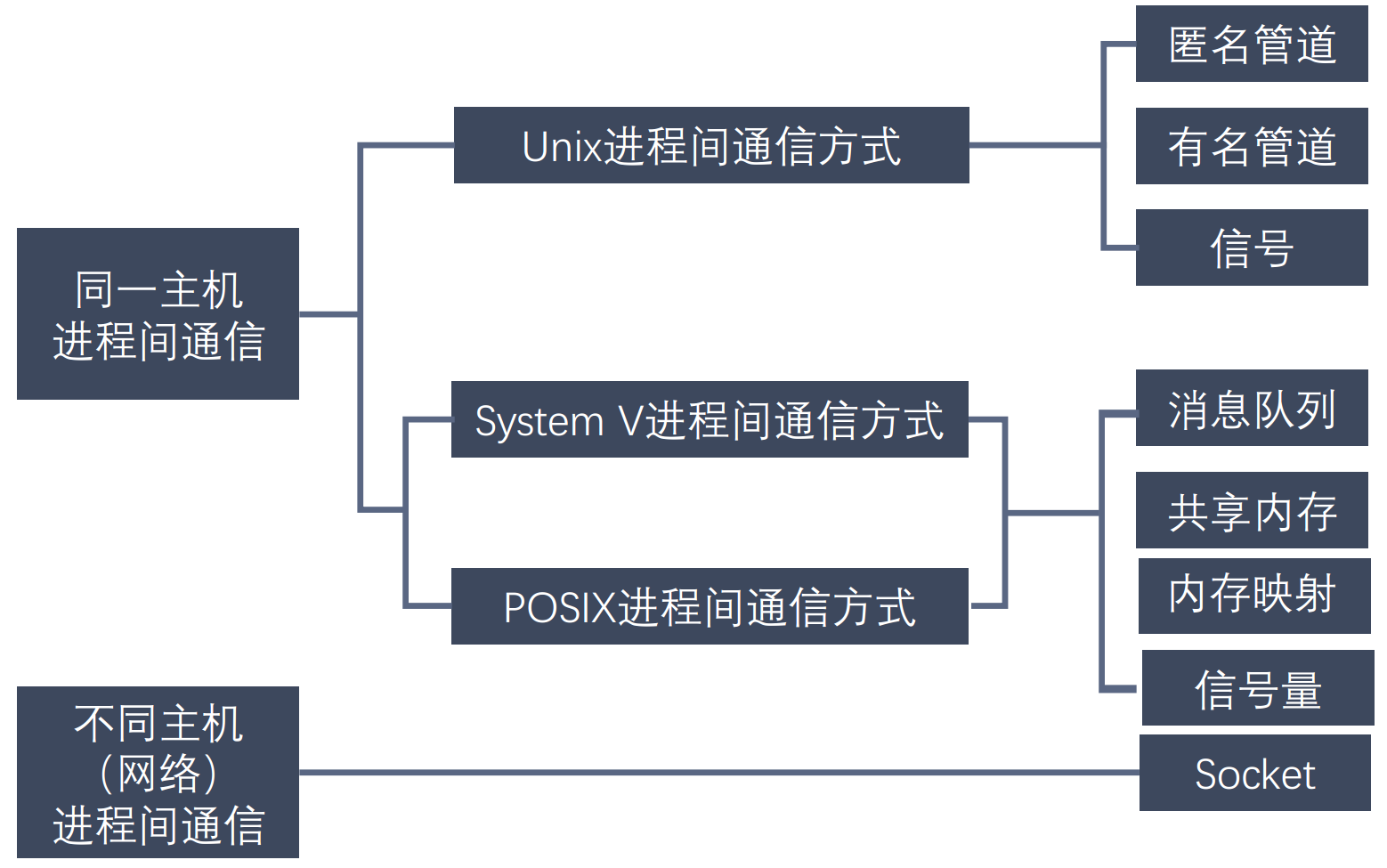
* + 资源共享：多个进程之间共享同样的资源。为了做到这一点，需要内核提供互斥和同

步机制。

* + 进程控制：有些进程希望完全控制另一个进程的执行（如 Debug 进程），此时控制

进程希望能够拦截另一个进程的所有陷入和异常，并能够及时知道它的状态改变。

#### 3.2 Linux进程间通信的方式



#### 3.3 匿名管道

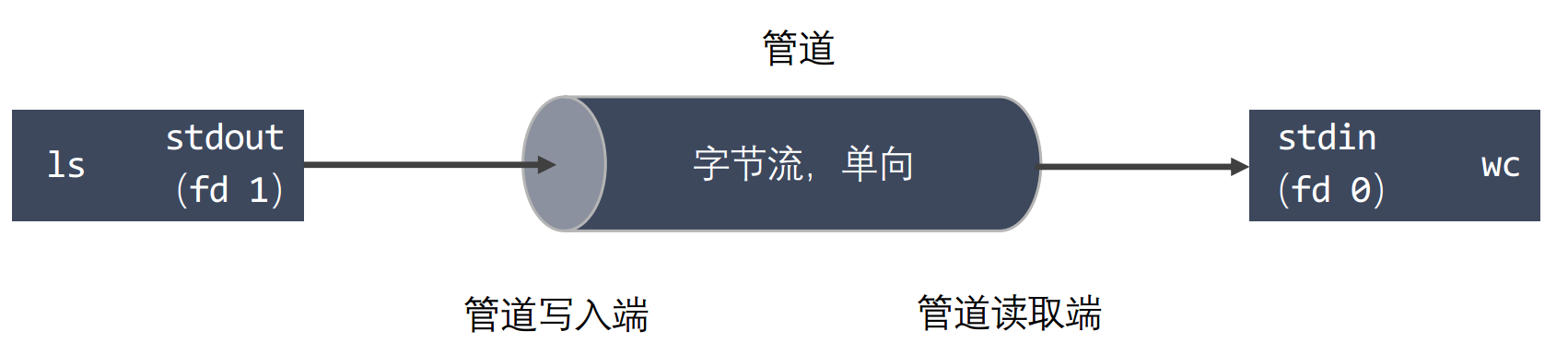
##### 3.3.1 (匿名)管道概念

* 管道也叫无名（匿名）管道，它是是 UNIX 系统 IPC（进程间通信）的最古老形式，

所有的 UNIX 系统都支持这种通信机制。

* 统计一个目录中文件的数目命令：ls | wc –l，为了执行该命令，shell 创建了两

个进程来分别执行 ls 和 wc



##### 3.3.2 匿名管道的特点

* 管道其实是一个在内核内存中维护的缓冲器，这个缓冲器的存储能力是有限的，不同的操作系统大小不一定相同。
* 管道拥有文件的特质：读操作、写操作，匿名管道没有文件实体，有名管道有文件实体，

但不存储数据。可以按照操作文件的方式对管道进行操作。

* 一个管道是一个字节流，使用管道时不存在消息或者消息边界的概念，从管道读取数据

的进程可以读取任意大小的数据块，而不管写入进程写入管道的数据块的大小是多少。

* 通过管道传递的数据是顺序的，从管道中读取出来的字节的顺序和它们被写入管道的顺

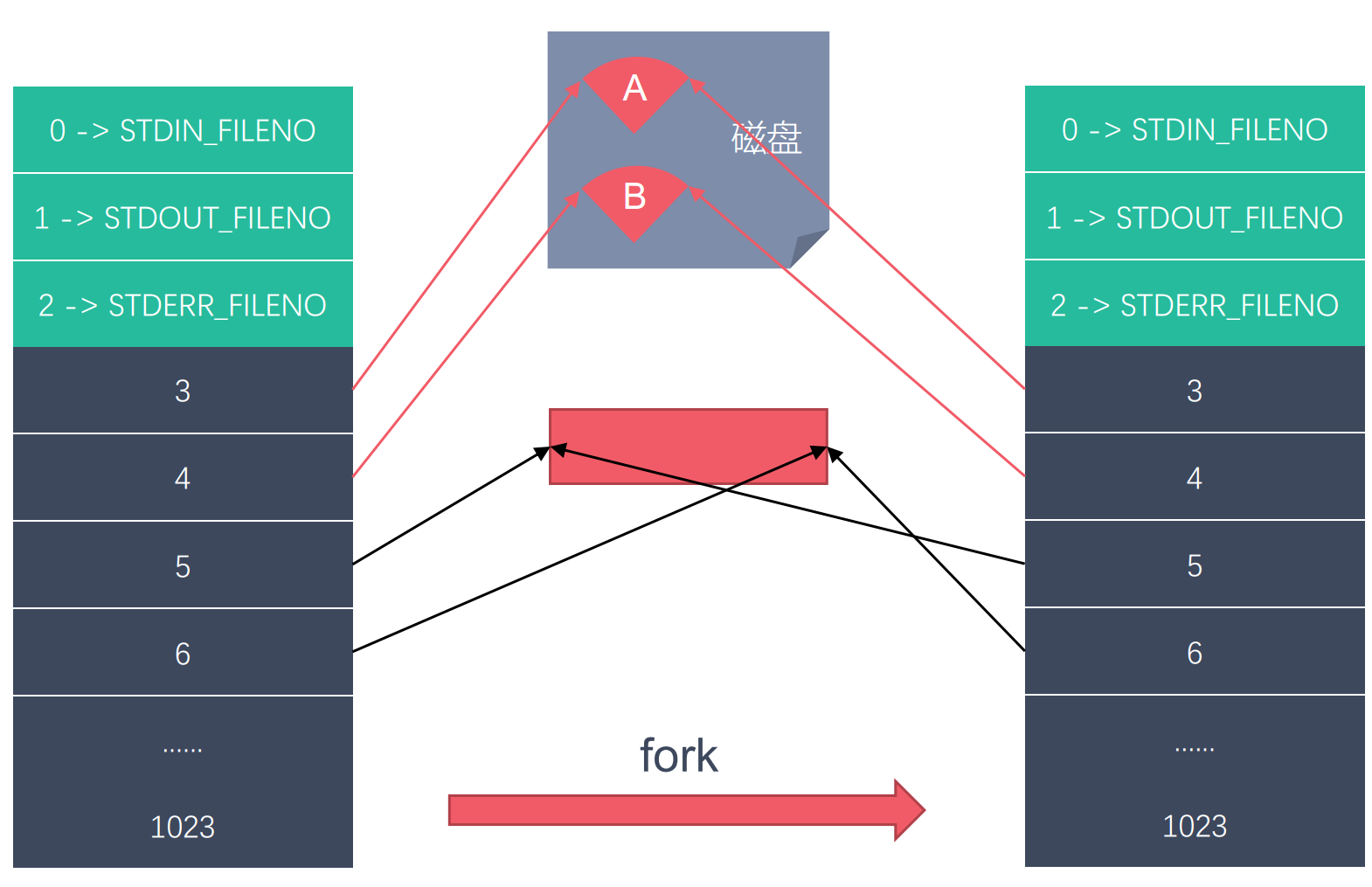
序是完全一样的。

* 在管道中的数据的传递方向是单向的，一端用于写入，一端用于读取，管道是半双工的。
* 从管道读数据是一次性操作，数据一旦被读走，它就从管道中被抛弃，释放空间以便写

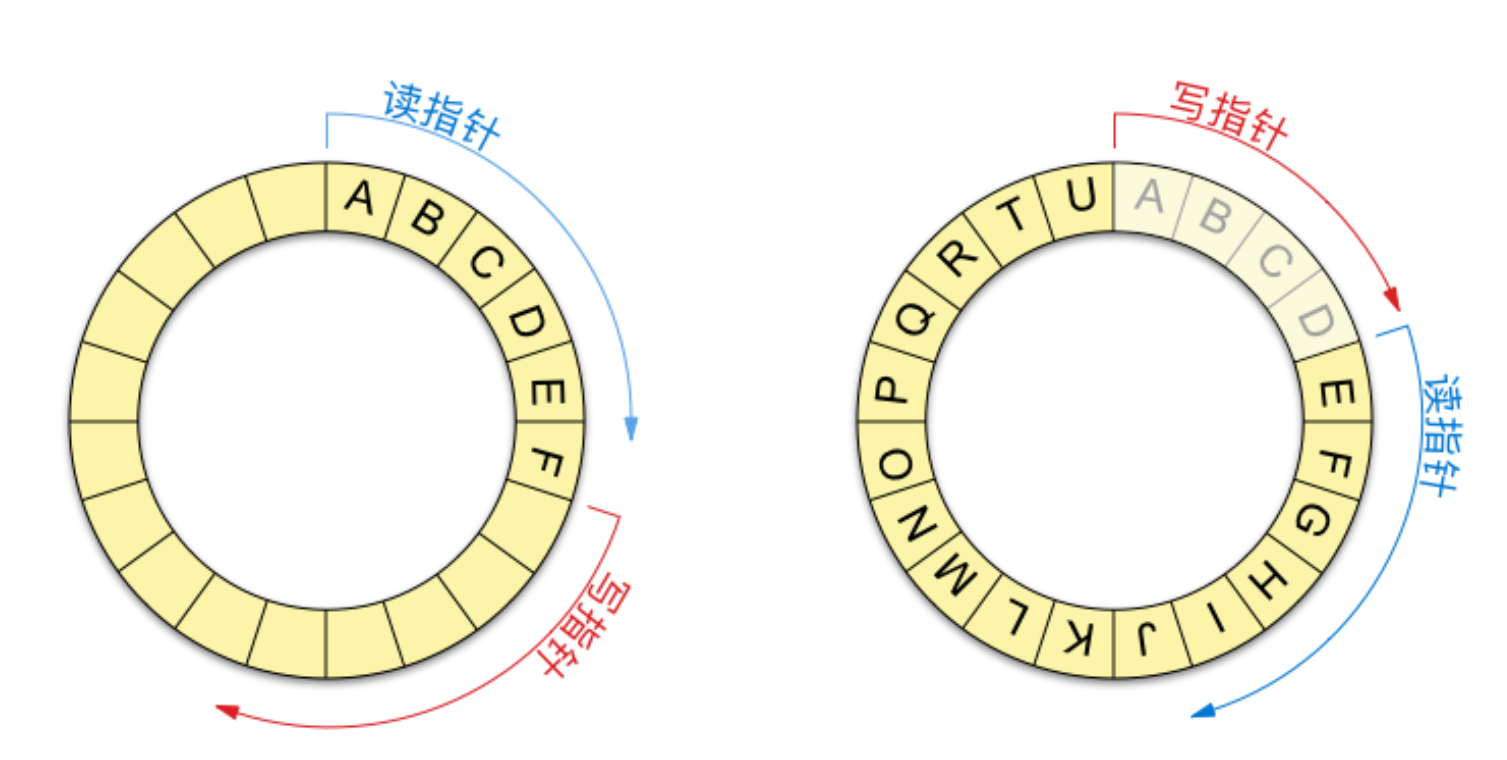
更多的数据，在管道中无法使用 lseek() 来随机的访问数据。

* 匿名管道只能在具有公共祖先的进程（父进程与子进程，或者两个兄弟进程，具有亲缘关系）之间使用。

##### 3.3.3 为什么可以使用匿名管道通信



##### 3.3.4 匿名管道的数据结构



##### 3.3.5匿名管道的使用

#include <unistd.h>

int pipe(int pipefd[2]);

功能:创建一个匿名管道，用来进行进程间通信。

参数:int pipefd[2] 这个数组是一个传出参数

pipefd[0]对应的是管道的读端

pipefd[1]对应的是管道的写端

返回值:

成功 0

失败 -1

管道默认是阻塞的，如果管道中没有数据:read阻塞(pipefd[0]),如果管道满了，Write阻塞(pipefd[1])

注意:匿名管道只能用于具有关系的进程之间的通信(父子进程，兄弟进程)

查看管道缓存的大小Linux命令：ulimit –a

获取管道的大小：long size = fpathconf(pipefd[0],\_PC\_PIPE\_BUF);

设置管道非阻塞

int flags = fcntl(fd[0],F\_GETFL);//获取原来的flag

flags |= O\_NONBLOCK; //修改flag的值

fcntl(fd[0],F\_SETFL,flags);//设置新的flag

管道的读写特点:

使用管道时，需要注意以下几种特殊情况(假设都是阻塞I/O操作)

1.所有的指向管道写端的文件描述符都关闭了(管道写端引用计数为0)，进程从管道读数据，

那么管道中剩余的数据被读取以后，再次read会返回0，就像读到文件末尾以后。

2.如果有指向管道写端的文件描述符没有关闭(管道的写端引用计数大于0)，而持有管道写端的进程也没有往管道中写数据，这个时候有进程从管道中读取数据，那么管道中剩余的数据读取后再次read会被阻塞，直到管道中有数据可以读了才读数据并返回。

3.如果所有指向管道读读端的文件描述符都关闭了(管道的读端引用计数等于0)，这个时候有进程向管道中写数据，那么该进程会受到一个信号SIGPIPE，通常会导致进程异常终止。

4.如果有指向管道读端的文件描述符没有关闭(管道的读端引用计数大于0)，而持有管道读端的进程也没有从管道中读取数据，这时有进程向管道中写数据，那么在管道被写满的时候再次write会阻塞直到管道中有空位置才能再次写入数据并返回。

总结（阻塞IO）:

读管道:

管道中有数据,read返回实际读到的字节数

管道中无数据：

写端被全部关闭，read返回0，相当于读到文件末尾

写端没有完全关闭，read阻塞等待

写管道：

管道读端全部关闭，进程异常终止(进程收到SIGPIPE信号)

管道读端没有全部关闭:

管道已满,write阻塞

管道没有满,write将数据写入,并返回实际写入的字节数

#### 3.4 有名管道

##### 3.4.1 有名管道的概念

* 匿名管道，由于没有名字，只能用于亲缘关系的进程间通信。为了克服这个缺点，提

出了有名管道（FIFO），也叫命名管道、FIFO文件。

* 有名管道（FIFO）不同于匿名管道之处在于它提供了一个路径名与之关联，以 FIFO

的文件形式存在于文件系统中，并且其打开方式与打开一个普通文件是一样的，这样

即使与 FIFO 的创建进程不存在亲缘关系的进程，只要可以访问该路径，就能够彼此

通过 FIFO 相互通信，因此，通过 FIFO 不相关的进程也能交换数据。

* 一旦打开了 FIFO，就能在它上面使用与操作匿名管道和其他文件的系统调用一样的I/O系统调用了（如read()、write()和close()）。与管道一样，FIFO 也有一个写入端和读取端，并且从管道中读取数据的顺序与写入的顺序是一样的。FIFO 的名称也由此而来：先入先出。
* 有名管道（FIFO)和匿名管道（pipe）有一些特点是相同的，不一样的地方在于：

1. FIFO 在文件系统中作为一个特殊文件存在，但 FIFO 中的内容却存放在内存中。

2. 当使用 FIFO 的进程退出后，FIFO 文件将继续保存在文件系统中以便以后使用。

3. FIFO 有名字，不相关的进程可以通过打开有名管道进行通信。

##### 3.4.2 有名管道的使用

* 通过命令创建fifo文件（有名管道）：mkfifo 名字
* 通过函数创建有名管道

通过函数: int mkfifio(const char\* pathname, mode\_t mode)

#include <sys/types.h>

#include <sys/stat.h>

int mkfifo(const char \*pathname, mode\_t mode);

参数:

-pathname:管道名称的路径

- mode: 文件的权限和open的mode是一样的

返回值:

成功返回0,失败返回-1，设置errno

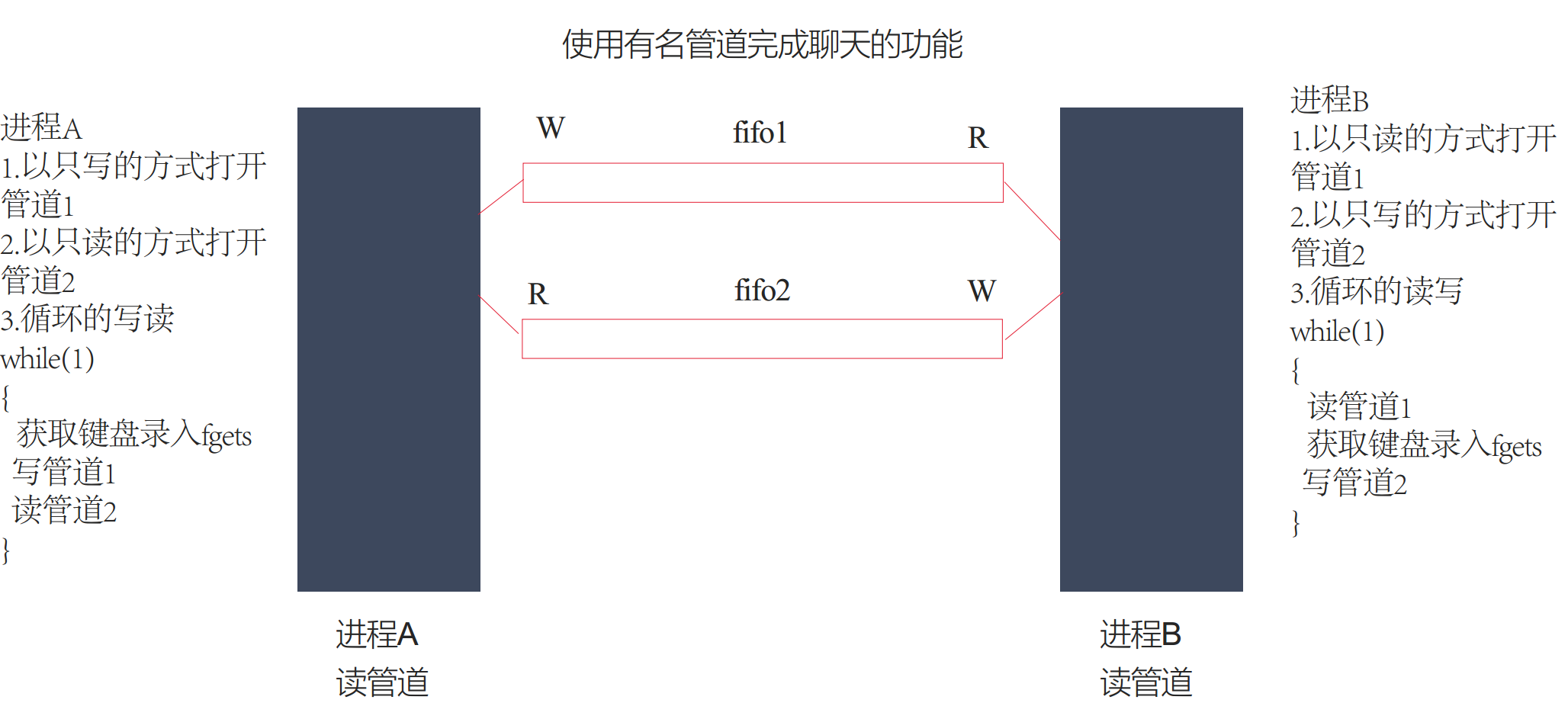
* 一旦使用 mkfifo 创建了一个 FIFO，就可以使用 open 打开它，常见的文件

I/O 函数都可用于 fifo。如：close、read、write、unlink 等。

* FIFO 严格遵循先进先出（First in First out），对管道及 FIFO 的读总是

从开始处返回数据，对它们的写则把数据添加到末尾。它们不支持诸如 lseek()

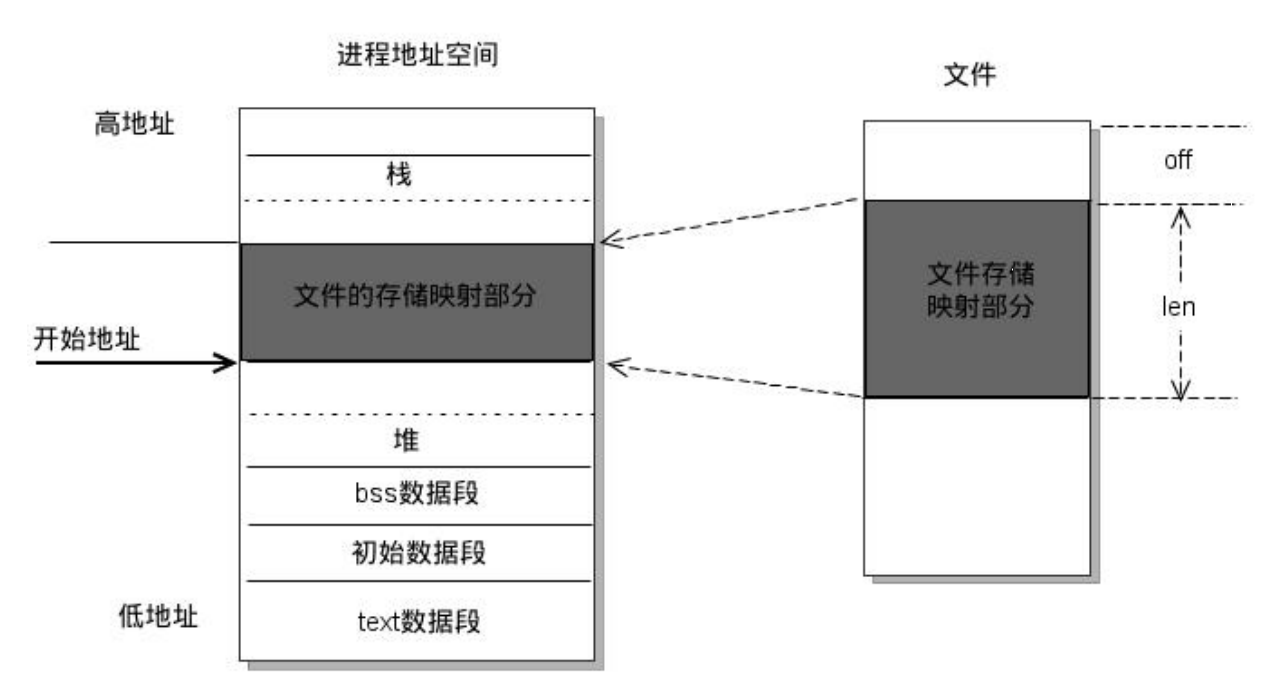
等文件定位操作。



#### 3.5 内存映射

##### 3.5.1 内存映射基本概念

内存映射（Memory-mapped I/O）是将磁盘文件的数据映射到内存，用户通过修改

内存就能修改磁盘文件

##### 3.5.2 内存映射的使用

#include <sys/mman.h>

void \*mmap(void \*addr, size\_t length, int prot, int flags,int fd, off\_t offset);

功能:将一个文件或者设备的数据映射到内存当中

参数：

void \*addr:NULL,由内核指定

length:要映射的数据的长度，这个值不能为0，建议使用文件的长度,默认为分页大小的整数倍

获取文件的长度:stat lseek

prot:对申请的内存映射区的操作权限

PROT\_EXEC Pages may be executed. 可执行

PROT\_READ Pages may be read. 读权限

PROT\_WRITE Pages may be written. 写权限

PROT\_NONE Pages may not be accessed. 没有权限

要操作映射内存，必须要有读的权限。

PROT\_ READ、PROT\_READ|PROT\_WRITE

flags:

MAP\_ANONYMOUS：匿名映射，不需要文件实体进程进行内存映射

MAP\_SHARED:映射区的数据会自动和磁盘文件进行同步，进程间通信，必须要设置这个选项

MAP\_\_PRIVATE:不同步，内存映射区的数据改变了，对原来的文件不会修改，会重新创建一个新的文件。(copy on write)

fd:需要映射的文件的文件描述符 -1为不用文件的匿名映射，常用于有关系进程之间的通信，通过oepn得到，open的是一个磁盘文件

注意：文件的大小不能为0，open指定的权限不能与prot参数有冲突

prot:PROT\_READ open:只读/读写

prot:PROT\_READ | PROT\_WRITE open:读写

offset:偏移量，一般不用。必须指定的是4k的整数倍，0表示不偏移

返回值:返回创建的内存的首地址

失败返回MAP\_FAILED

int munmap(void \*addr, size\_t length);

功能:释放内存映射

参数:

addr : 要释放的内存的首地址

length: 要释放的内存的大小,要和mmap函数中的length参数的值一样

使用内存映射实现进程间通信:

1.有关系的进程(父子进程)

- 还没有子进程的时候

-通过唯一的父进程，先创建内存映射区

- 有了内存映射区以后，创建子进程

- 父子共享创建的内存映射区

2.没有关系的进程间通信

- 准备一个大小不是0的磁盘文件

- 进程1 通过磁盘文件创建内存映射区

- 得到一个操作这块内存的指针

- 进程2 通过磁盘文件创建内存映射区

- 得到一个操作这块内存的指针

- 使用内存映射区通信

注意:内存映射区通信，是非阻塞。

##### 3.5.3 内存映射的注意事项

1.如果对 mmap 的返回值 (ptr) 做++ 操作 (ptr++), munmap 是否能够成功?

void \*ptr = map(...);

ptr++; 可以对其进行++操作

munmap(ptr,len); //错误，要保存地址

2.如果 open 时O\_RDONLY, mmap 时prot 参数指定 PROT\_READ | PROT\_WRITE 会怎样?

错误，返回MAP\_FAILED

open()函数中的权限建议和prot参数的权限保持一致。

3.如果文件偏移量为 1000 会怎样?

偏移量必须是4K的整数倍，返回MAP\_FAILED

4.mmap 什么情况下会调用失败?

- 第二个参数: length = 0

- 第三个参数: prot

- 只指定了写权限

- PROT\_READ | PROT\_WRITE

第五个参数fd通过open函数时指定的O\_RDONLY / O\_WRONLY

5.可以 open 的时候O\_CREAT 一个新文件来创建映射区吗?

- 可以的，但是创建的文件的大小如果为0的话，肯定不行

- 可以对新的文件进行扩展

- lseek()

- truncate()

6.mmap 后关闭文件描述符，对 mmap 映射有没有影响？

int fd = open("XXX");

mmap(,,,fd,0);

close(fd);

映射区还存在，创建映射区的fd被关闭，没有任何影响

7.对ptr 越界操作会怎样？

void \*ptr = mmap(NULL,100,,,,);

一般只有4K容量

越界操作操作的内存的是非法的内存 -> 段错误

#### 3.6 信号

##### 3.6.1信号的概念

* 信号是 Linux 进程间通信的最古老的方式之一，是事件发生时对进程的通知机制，有时也称之为软件中断，它是在软件层次上对中断机制的一种模拟，是一种异步通信的方式。信号可以导致一个正在运行的进程被另一个正在运行的异步进程中断，转而处理某一个突发事件。
* 发往进程的诸多信号，通常都是源于内核。引发内核为进程产生信号的各类事件如下：
  + 对于前台进程，用户可以通过输入特殊的终端字符来给它发送信号。比如输入Ctrl+C

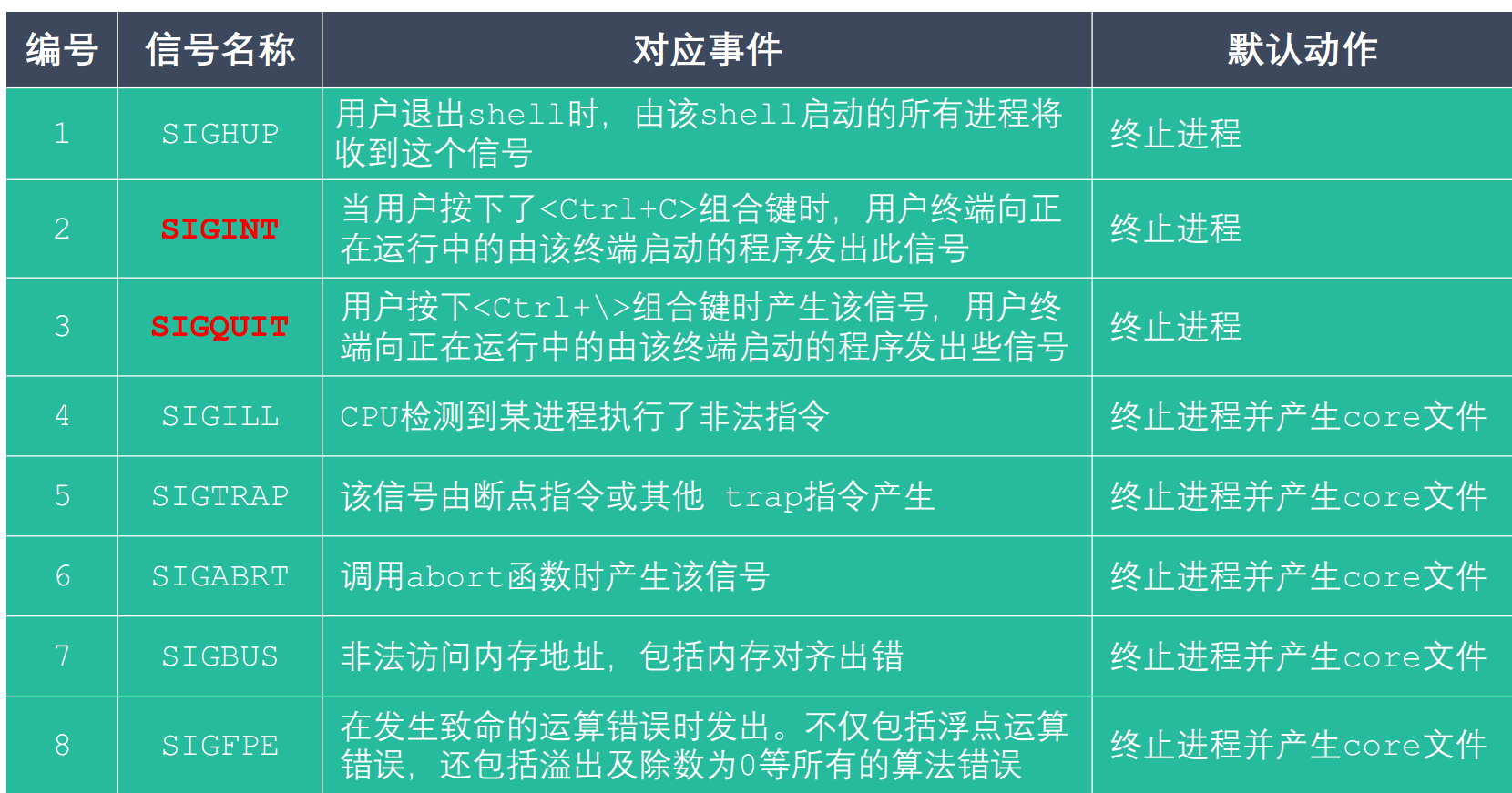
通常会给进程发送一个中断信号。

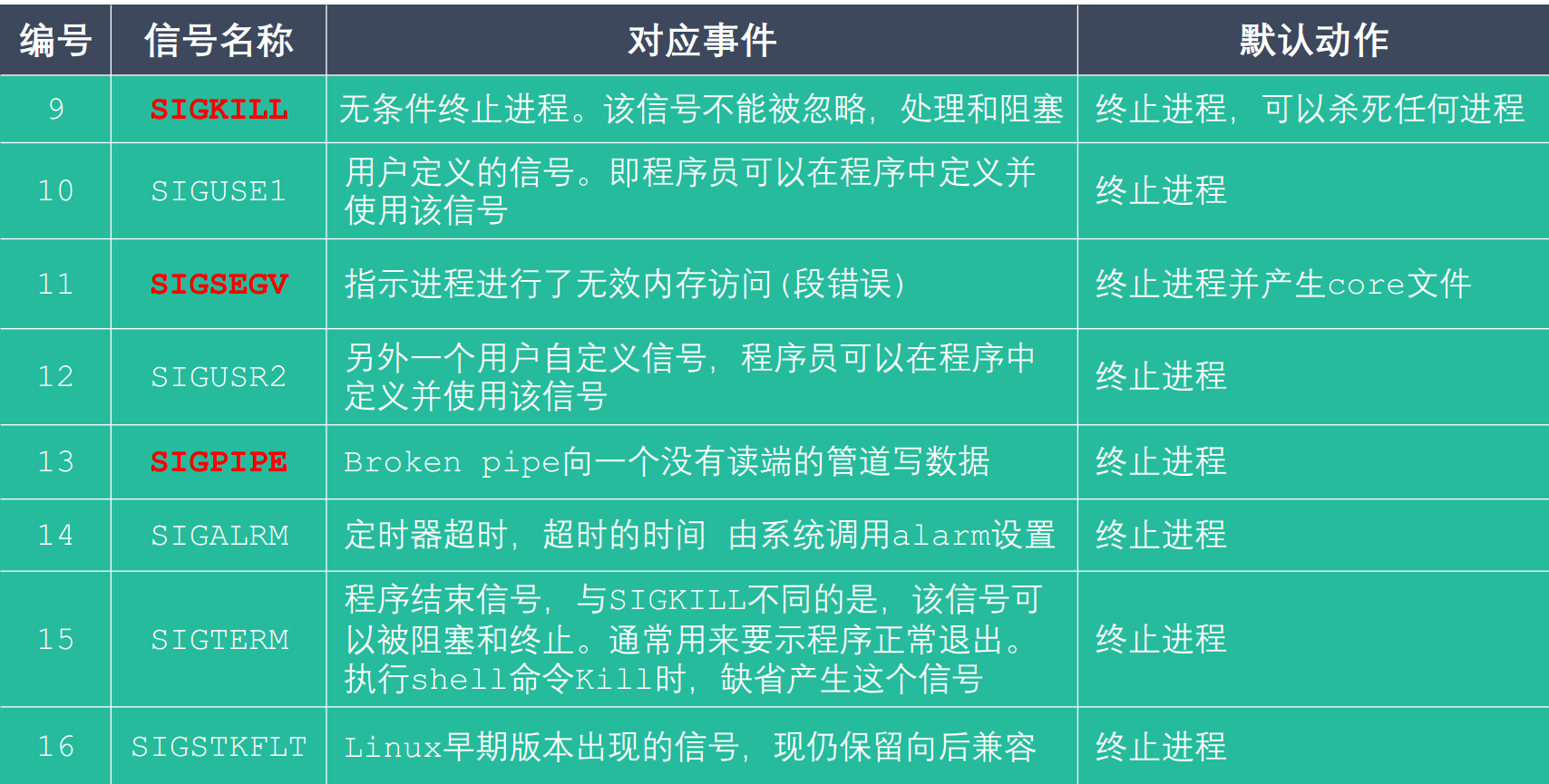
* 硬件发生异常，即硬件检测到一个错误条件并通知内核，随即再由内核发送相应信号给相关进程。比如执行一条异常的机器语言指令，诸如被 0 除，或者引用了无法访问的内存区域。
* 系统状态变化，比如 alarm 定时器到期将引起 SIGALRM 信号，进程执行的 CPU

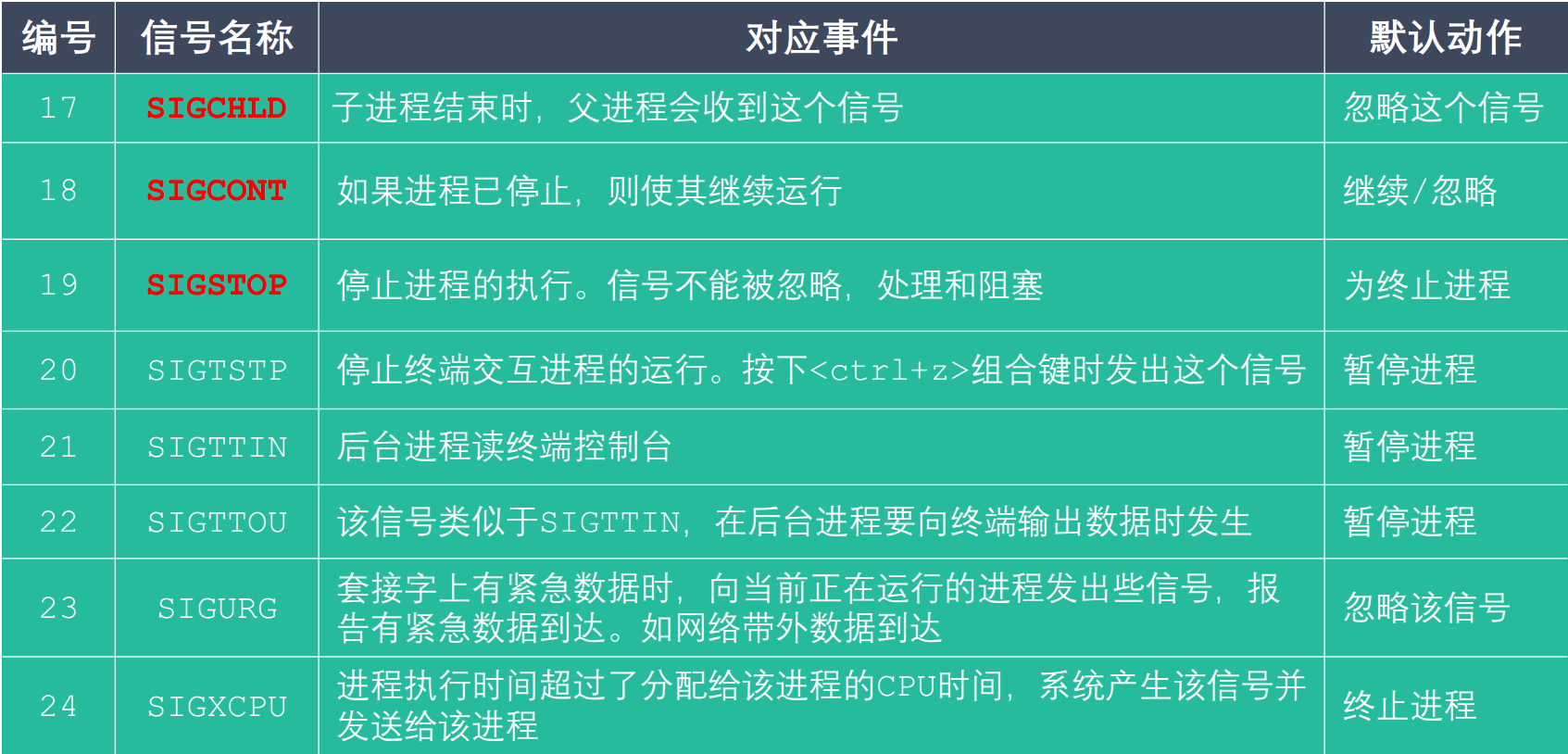
时间超限，或者该进程的某个子进程退出。

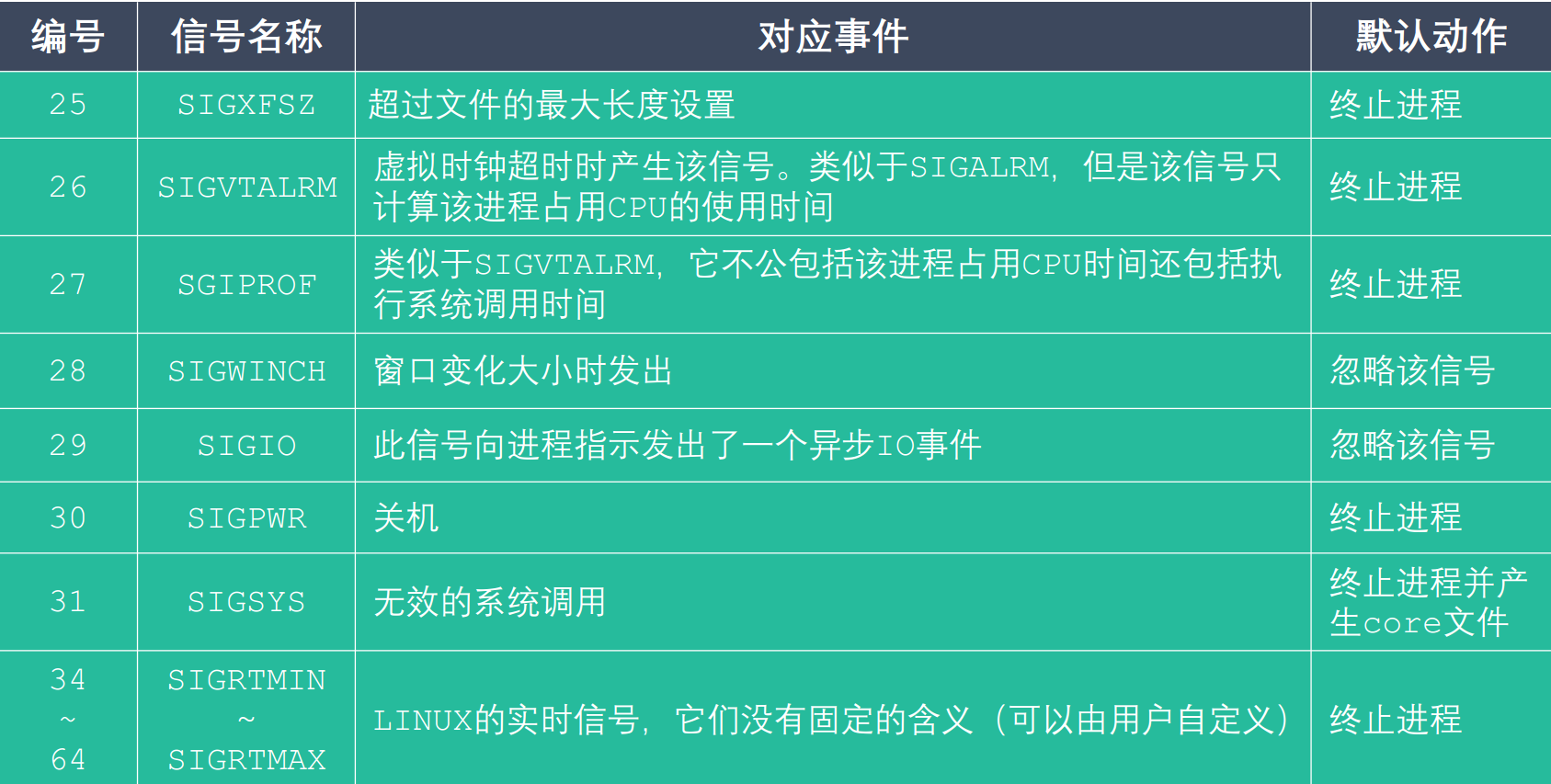
* 运行 kill 命令或调用 kill 函数。
* 使用信号的两个主要目的是：
  + 让进程知道已经发生了一个特定的事情
  + 强迫进程执行它自己代码中的信号处理程序。
* 信号的特点：
  + 简单
  + 不能携带大量信息
  + 满足某个特定条件才发送
  + 优先级比较高
* 查看系统定义的信号列表：kill –l
* 前 31 个信号为常规信号，其余为实时信号。

##### 3.6.2 Linux信号一览表









##### 3.6.3 信号的 5 种默认处理动作

* 查看信号的详细信息：man 7 signal
* 信号的 5 种默认处理动作
  + Term 终止进程
  + Ign 当前进程忽略掉这个信号
  + Core 终止进程，并生成一个core文件（为了处理异常）

/\*不生成core文件时候可用该命令sudo service apport stop\*/

/\*core-file core可以调试core文件信息\*/

* + Stop 暂停当前进程
  + Cont 继续执行当前被暂停的进程
* 信号的几种状态：产生、未决、递达
* SIGKILL 和 SIGSTOP 信号不能被捕捉、阻塞或者忽略，只能执行默认动作。

##### 3.6.4 信号相关的函数

###### 3.6.4.1 kill函数

#include <sys/types.h>

#include <signal.h>

int kill(pid\_t pid, int sig);

- 功能:给任何的进程或者进程组pid,发送任何的信号sig

- 参数:

- pid :

> 0 : 将信号发送给指定的进程

= 0 : 将信号发送给当前的进程组

= -1: 将信号发送给每一个有权限接收这个信号的进程

< -1: 这个pid=某个进程组的ID取反(-12345)

- sig : 需要发送的信号的编号或者是宏值(推荐宏值),0表示不发送任何信号

kill(getppid(),9);

kill(getpid(),9);

###### 3.6.4.2 raise函数

int raise(int sig)

- 功能:给当前进程发送信号

- 参数:

- sig : 要发送的信号

- 返回值:

- 成功 0

- 失败 非0

kill(getpid(),sig);

###### 3.6.4.3 abort函数

void abort(void);

- 功能 : 发送SIGABRT信号给当前的进程,杀死当前进程

kill(getpid(),SIGABRT);

###### 3.6.4.4 alarm函数

#include <unistd.h>

unsigned int alarm(unsigned int seconds);

- 功能: 设置定时器(闹钟),函数调用开始倒计时，倒计时为0时候，函数会给当前的进程发送一个信号:SIGALARM；再次调用时候会刷新定时器的时间

- 参数:

seconds:倒计时的时长,单位:秒。如果参数为0,定时器无效(不进行倒计时，不发信号)。

取消一个定时器，通过alarm(0)。

- 返回值:

之前没有定时器，返回0

之前有定时器，返回之前定时器倒计时剩余的时间

-SIGALARM:默认终止当前的进程，每一个进程都有且只有唯一的一个定时器

alarm(10);

过了1秒

alarm(5);

alarm(100)->该函数是不阻塞的

###### 3.6.4.5 setitimer函数

#include <sys/time.h>

int setitimer(int which, const struct itimerval \*new\_value,struct itimerval \*old\_value);

-功能:设置定时器(闹钟),可以替代alarm函数，精度微秒us,可以实现周期性定时

-参数:

-which : 定时器以什么时间计时

ITIMER\_REAL:真实时间，时间到达，发送SIGALARM信号 常用

ITIMER\_VIRTUAL:用户时间,执行用户自己写的代码

RING3级别下，0~3G虚拟内存空间时间,时间到达,发送SIGVTALARM

内核时间表示在虚拟空间3G~4G最高RING0优先级系统调用的时间

ITIMER\_PROF:以该进程在用户态和内核态下所消耗的时间来计算,时间到达,发送SIGPROF

-new\_value:设置定时器的属性

struct itimerval { //定时器的结构体

struct timeval it\_interval; //每个阶段的定时时间(间隔时间)

struct timeval it\_value; //延迟多长时间开始执行定时器

};

struct timeval { //时间的结构体

time\_t tv\_sec; //秒数

suseconds\_t tv\_usec; //微妙

};

过10秒后,每隔两秒定时一次

- old\_value : 记录上一次的定时的时间参数，一般不使用，指定NULL

-返回值：

成功 0

失败 -1，设置错误号

##### 3.6.5 信号捕捉函数

###### 3.6.5.1 sighandler函数

#include <signal.h>

typedef void (\*sighandler\_t)(int);

sighandler\_t signal(int signum, sighandler\_t handler);

-功能: 设置某个信号的捕捉行为

-参数:

-signum:要捕捉的信号

-handler:捕捉到信号要如何处理

-SIG\_IGN:忽略信号

-SIG\_DFL:使用信号默认的行为

-回调函数: 这个函数是内核调用，程序员只负责写，捕捉到信号如何处理信号。

回调函数:

-需要程序员实现，提前准备好的，函数的类型根据实际需求，看函数指针的定义

-不是程序员调用，而是当信号产生，由内核调用

-函数指针是实现回调的手段，函数实现之后，将函数名放到函数指针的位置就可以了

-返回值:

成功：返回上一次注册的信号处理函数的地址

失败: 返回SIG\_ERR,设置错误号

SIGKILL,SIGSTOP不能被捕捉，不能被忽略

###### 3.6.5.2 sigaction函数

#include <signal.h>

int sigaction(int signum, const struct sigaction \*act,

struct sigaction \*oldact);

功能: 检查或者改变信号的处理,信号捕捉

参数:

- signum : 需要捕捉的信号的编号或者宏值(信号的名称)

- act : 捕捉到信号之后的处理动作

- oldact:上一次对信号捕捉相关的设置，一般不使用，传递NULL

返回值:

成功 0

失败 -1

struct sigaction {

// 函数指针,指向的函数就是信号捕捉到之后的处理函数

void (\*sa\_handler)(int);

// 不常用

void (\*sa\_sigaction)(int, siginfo\_t \*, void \*);

// 临时阻塞信号集,在信号捕捉函数执行过程中,临时阻塞某些信号。

sigset\_t sa\_mask;

//使用哪一个信号处理(前两个函数指针之一) 对捕捉到的信号处理

//这个值可以是0,表示使用sa\_handler,也可以是SA\_SIGINFO表示sa\_sigaction

int sa\_flags;

//被废弃掉了

void (\*sa\_restorer)(void);

##### 3.6.6 信号集

###### 3.6.6.1 基本概念

* 许多信号相关的系统调用都需要能表示一组不同的信号，多个信号可使用一个称之为

信号集的数据结构来表示，其系统数据类型为 sigset\_t。

* 在 PCB 中有两个非常重要的信号集。一个称之为 “阻塞信号集” ，另一个称之为“未决信号集”。这两个信号集都是内核使用位图机制来实现的。但操作系统不允许我们直接对这两个信号集进行位操作。而需自定义另外一个集合，借助信号集操作函数来对 PCB 中的这两个信号集进行修改
* 信号的 “未决” 是一种状态，指的是从信号的产生到信号被处理前的这一段时间。
* 信号的 “阻塞” 是一个开关动作，指的是阻止信号被处理，但不是阻止信号产生。
* 信号的阻塞就是让系统暂时保留信号留待以后发送。由于另外有办法让系统忽略信号，所以一般情况下信号的阻塞只是暂时的，只是为了防止信号打断敏感的操作。

###### 3.6.6.2 阻塞信号集和未决信号集



1.用户通过键盘 ctrl+c,产生2号信号SIGINT(信号被创建)

2.信号产生但是没有被处理(未决)

- 在内核中将所有的没有被处理的信号存储在一个集合中(未决信号集)

- SIGINT信号状态被存储在第二个标志位上

- 这个标志位为0,说明信号不是未决状态

- 这个标志位为1，说明信号处于未决状态

3.这个未决状态的信号,需要被处理,处理之前需要和另一个信号集(阻塞信号集)，进行比较

- 阻塞信号集默认不阻塞任何信号

- 如果想要阻塞某些信号需要用户调用系统的API

4.在处理的时候和阻塞信号集中的标志位进行查询,看是不是对该信号设置阻塞了

- 如果没有阻塞，这个信号就被处理

- 如果阻塞了，这个信号就继续处于未决状态，直到阻塞解除，这个信号就被处理

###### 3.6.6.3 信号集相关的函数(信号集相关的函数都是对自定义的信号集进行操作，一般是后续是对阻塞信号集操作)

1. sigemptyset函数

#include <signal.h>

int sigemptyset(sigset\_t \*set);

- 功能: 清空信号集中的数据,将信号集中的所有的标志位置为0

- 参数: set,传出参数,需要操作的信号集

- 返回值: 成功返回0, 失败返回-1

1. sigfillset函数

int sigfillset(sigset\_t \*set);

- 功能: 将信号集中的所有标志位置为1

- 参数: set,传出参数，需要操作的信号集

- 返回值: 成功返回0, 失败返回 -1

1. sigaddset函数

int sigaddset(sigset\_t \*set, int signum);

- 功能: 设置信号集中的某一个信号对应的标志位为1,表示阻塞这个信号

- 参数:

- set:传出参数，需要操作的信号集

- 返回值: 成功返回0, 失败返回 -1

1. sigdelset函数

int sigdelset(sigset\_t \*set, int signum);

- 功能: 设置信号集中的某一个信号对应的标志位为0,表示不阻塞这个信号

- 参数:

- set: 传出参数,需要操作的信号集

- signum: 需要设置不阻塞的那个信号

- 返回值: 成功返回0, 失败返回-1

1. sigismember函数

int sigismember(const sigset\_t \*set, int signum);

- 功能: 判断某个信号是否阻塞

- 参数:

- set: 需要操作的信号集

- signum: 需要判断的那个信号

- 返回值:

1 : signum被阻塞

0 : signum不阻塞

-1 : 失败

1. sigprocmask函数

int sigprocmask(int how, const sigset\_t\* set, sigset\_t\* oldset);

- 功能: 将自定义信号集中的数据设置到内核中阻塞信号集中(设置阻塞,解除阻塞,替换)

- 参数:

- how : 如何对内核阻塞信号集进行处理

SIG\_BLOCK: 将用户设置的阻塞信号集添加到内核中,内核中原来的数据不变

假设内核中默认的阻塞信号集是mask, mask | set

SIG\_UNBLOCK: 根据用户设置的数据,对内核中的数据进行解除阻塞

假设内核中默认的阻塞信号集是mask, mask &= ~set

SIG\_SETMASK: 覆盖内核中原来的值

- set : 已经初始化好的用户自定义的信号集

- oldset : 保存设置之前的内核中的阻塞信号集的状态,可以是NULL

- 返回值:

成功: 0

失败: -1

设置错误号: EFAULT、EINVAL

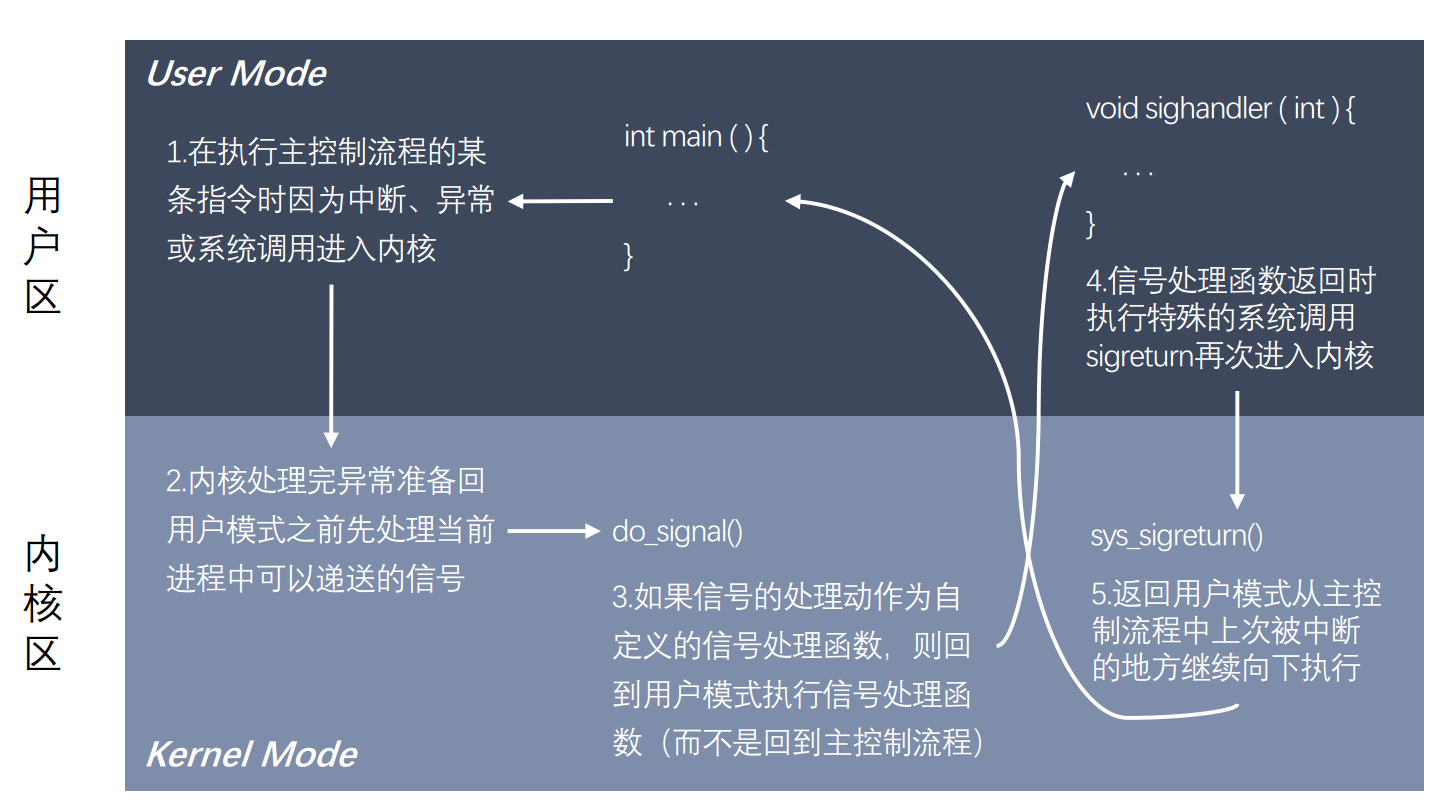
1. sigpending函数

int sigpending(sigset\_t \*set)

- 功能: 获取内核中的未决信号集

- 参数: set,传出参数,保存的是内核中的未决信号集中的信息

##### 3.6.7 内核实现信号捕捉的过程



##### 3.6.8 SIGCHILD信号

* SIGCHLD信号产生的条件
  + 子进程终止时
  + 子进程接收到 SIGSTOP 信号停止时
  + 子进程处在停止态，接受到SIGCONT后唤醒时
* 以上三种条件都会给父进程发送 SIGCHLD 信号，父进程默认会忽略该信号

#### 3.7 共享内存

##### 3.7.1 共享内存的概念

* 共享内存允许两个或者多个进程共享物理内存的同一块区域（通常被称为段）。由于一个共享内存段会称为一个进程用户空间的一部分，因此这种 IPC 机制无需内核介入。所以需要做的就是让一个进程将数据复制进共享内存中，并且这部分数据会对其他所有共享同一个段的进程可用。
* 与管道等要求发送进程将数据从用户空间的缓冲区复制进内核内存和接收进程将数据从内核内存复制进用户空间的缓冲区的做法相比，这种 IPC 技术的速度更快。

##### 3.7.2 共享内存使用步骤

* 调用 shmget() 创建一个新共享内存段或取得一个既有共享内存段的标识符（即由其他进程创建的共享内存段）。这个调用将返回后续调用中需要用到的共享内存标识符。
* 使用 shmat() 来附上共享内存段，即使该段成为调用进程的虚拟内存的一部分。
* 此刻在程序中可以像对待其他可用内存那样对待这个共享内存段。为引用这块共享内存，程序需要使用由 shmat() 调用返回的 addr 值，它是一个指向进程的虚拟地址空间中该共享内存段的起点的指针
* 调用 shmdt() 来分离共享内存段。在这个调用之后，进程就无法再引用这块共享内存了。这一步是可选的，并且在进程终止时会自动完成这一步。
* 调用 shmctl() 来删除共享内存段。只有当当前所有附加内存段的进程都与之分离之后内存段才会销毁。只有一个进程需要执行这一步。

##### 3.7.3 共享内存操作函数

###### 3.7.3.1 ftok函数

key\_t ftok(const char \*pathname, int proj\_id);

- 功能: 根据指定的路径名,和int值,生成一个共享内存的key

- 参数:

- pathname:指定一个存在的路径

/home/williamwhites/Linux/a.txt

/

- proj\_id: int类型的值,但是这个系统调用只会使用其中的1个字节

范围: 0~255 一般指定一个字符 'a'

###### 3.7.3.2 shmget函数

#include<sys/ipc.h>

#include<sys/shm.h>

int shmget(key\_t key, size\_t size, int shmflg);

- 功能:创建一个新的共享内存段，或者获取一个既有的共享内存段的标识。新创建的内存段中的数据都会被初始化为 0

- 参数:

- key : key\_t类型是一个整型，通过这个找到或者创建一个共享内存。

一般使用16进制表示，非0值

- size: 共享内存的大小(分页)

- shmflg:属性

- 访问权限

- 附加属性: 创建/判断共享内存是不是存在

-创建:IPC\_CREATE,必须要有

-判断共享内存是否存在: IPC\_EXCL, 需要和IPC\_CREAT一起使用

IPC\_CREATE | IPC\_EXCL | 0664

- 返回值

失败: -1 设置错误号

成功: >0 返回共享内存引用的ID,后面操作共享内存都是通过这个值

###### 3.7.3.3 shmat函数

void \*shmat(int shmid, const void \*shmaddr, int shmflg);

- 功能: 和当前的进程进行关联

- 参数:

- shmid: 共享内存的标识(ID),由shmget返回值获取

- shmaddr: 申请的共享内存的起始地址,指定NULL(由内核指定)

- shmflg: 对共享内存的操作

- 读: SHM\_RDONLY,必须要有读权限

- 读写: 0

- 返回值:

成功:返回共享内存的首(起始)地址

失败: (void \*)-1

###### 3.7.3.4 shmdt函数

int shmdt(const void \*shmaddr);

- 功能: 解除当前进程和共享内存的关联

- 参数:

shmaddr:共享内存的首地址

- 返回值: 成功0, 失败-1

###### 3.7.3.5 shmctl函数

int shmctl(int shmid, int cmd, struct shmid\_ds \*buf);

- 功能: 对共享内存进行操作。删除共享内存，共享内存要删除才会消失，创建共享内存的进程被销毁了对共享内存没有影响

- 参数:

- shmid:共享内存的ID

- cmd: 要做的操作

- IPC\_STAT: 获取共享内存的当前的状态

- IPC\_SET: 设置共享内存的状态

- IPC\_RMID: 标记共享内存被销毁

- buf: 需要设置或者获取的共享内存的属性信息

- IPC\_STAT : buf存储获取的共享内存的属性信息

- IPC\_SET: buf中需要初始化数据,设置到内核中

- IPC\_RMID: 没有用,NULL

##### 3.7.4 共享内存操作命令

* ipcs 用法
  + ipcs -a // 打印当前系统中所有的进程间通信方式的信息
  + ipcs -m // 打印出使用共享内存进行进程间通信的信息（可以显示标记删除后键值变为0 并且断开连接（只能在程序里调用detach分离）时候连接会减1）
  + ipcs -q // 打印出使用消息队列进行进程间通信的信息
  + ipcs -s // 打印出使用信号进行进程间通信的信息
* ipcrm 用法ipcrm -M shmkey // 移除用shmkey创建的共享内存段ipcrm -m shmid // 移除用shmid标识的共享内存段（标记删除）

ipcrm -Q msgkey // 移除用msqkey创建的消息队列ipcrm -q msqid // 移除用msqid标识的消息队列

ipcrm -S semkey // 移除用semkey创建的信号

ipcrm -s semid // 移除用semid标识的信号

问题1: 操作系统如何知道一块共享内存被多少个进程关联？

- 共享内存维护一个结构体struct shmid\_ds这个结构体中有一个成员shm\_nattach

- shm\_nattach记录了关联的进程个数

问题2:可不可以对共享内存进行多次删除 shmctl

- 可以的

- 因为shmctl 标记删除共享内存,不是直接删除

当和共享内存关联的进程数为0的时候,就真正被删除

- 当共享内存的key为0的时候，表示共享内存被标记删除了

如果一个进程和共享内存取消关联,那么这个进程就不能继续操作这个共享内存。也不能进行关联

共享内存和内存映射的区别:

1.共享内存可以直接创建,内存映射需要磁盘文件(匿名映射除外)

2.共享内存效率更高

3.内存

所有的进程操作的是同一块共享内存

内存映射，每个进程在自己的虚拟地址空间中有一个独立的内存。

4.数据安全

-进程突然退出

共享内存还存在

内存映射会消失

-运行进程的电脑宕机了

数据存储在共享内存中，没有了

内存映射区的数据，由于磁盘文件中的数据还在，所以内存映射区的数据还存在

5.生命周期

- 内存映射区: 进程退出,内存映射区销毁

- 共享内存: 进程退出,共享内存还在,标记删除(所有的关联进程数为0),或者关机

如果一个进程退出，会自动和共享内存进行一个取消关联。

#### 守护进程

##### 终端

* 在 UNIX 系统中，用户通过终端登录系统后得到一个 shell 进程，这个终端成为 shell 进程的控制终端（Controlling Terminal），进程中，控制终端是保存在 PCB 中的信息，而 fork() 会复制 PCB 中的信息，因此由 shell 进程启动的其它进程的控制终端也是这个终端。（命令行执行程序就是fork和exec）（echo $$可以查看当前控制终端对应进程的PID；ps aux可以查看对应的终端名；tty 也可以查看对应的终端名）
* 默认情况下（没有重定向），每个进程的标准输入、标准输出和标准错误输出都指向控制终端，进程从标准输入读也就是读用户的键盘输入，进程往标准输出或标准错误输出写也就是输出到显示器上。
* 在控制终端输入一些特殊的控制键可以给前台进程发信号，例如 Ctrl + C 会产生 SIGINT 信号，Ctrl + \ 会产生 SIGQUIT 信号。

##### 进程组

* 进程组和会话在进程之间形成了一种两级层次关系：进程组是一组相关进程的集合，会话是一组相关进程组的集合。进程组和会话是为支持 shell 作业控制而定义的抽象概念，用户通过 shell 能够交互式地在前台或后台运行命令。
* 进行组由一个或多个共享同一进程组标识符（PGID）的进程组成。一个进程组拥有一个进程组首进程，该进程是创建该组的进程，其进程 ID 为该进程组的 ID，新进程会继承其父进程所属的进程组 ID。
* 进程组拥有一个生命周期，其开始时间为首进程创建组的时刻，结束时间为最后一个成员进程退出组的时刻。一个进程可能会因为终止而退出进程组，也可能会因为加入了另外一个进程组而退出进程组。进程组首进程无需是最后一个离开进程组的成员。

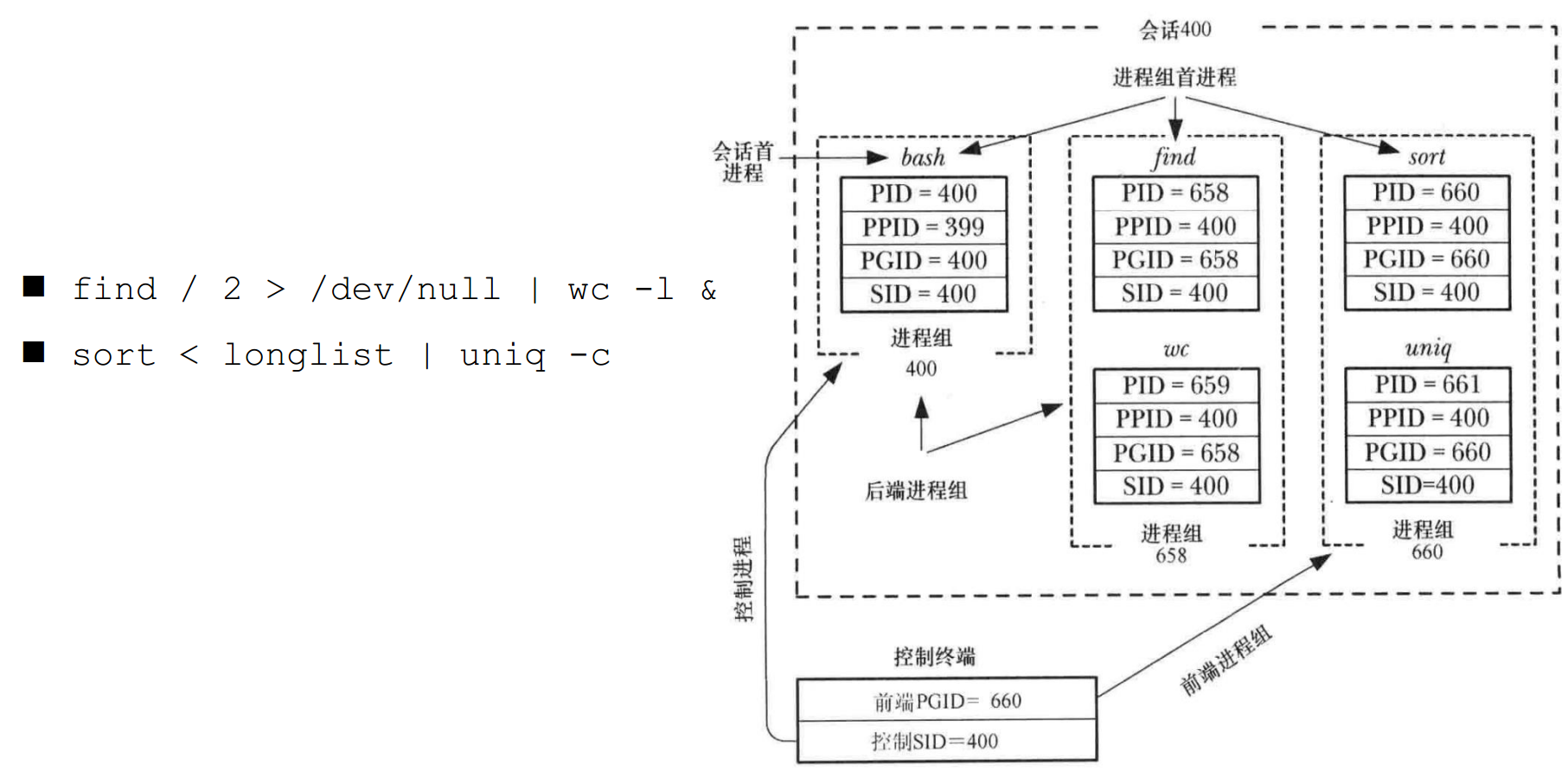
##### 会话

* 会话是一组进程组的集合。会话首进程是创建该新会话的进程，其进程 ID 会成为会话 ID。新进程会继承其父进程的会话 ID。
* 一个会话中的所有进程共享单个控制终端。控制终端会在会话首进程首次打开一个终

端设备时被建立。一个终端最多可能会成为一个会话的控制终端。

* 在任一时刻，会话中的其中一个进程组会成为终端的前台进程组，其他进程组会成为后台进程组。只有前台进程组中的进程才能从控制终端中读取输入。当用户在控制终端中输入终端字符生成信号后，该信号会被发送到前台进程组中的所有成员。
* 当控制终端的连接建立起来之后，会话首进程会成为该终端的控制进程。

##### 进程组、会话、控制终端之间的关系



##### 进程组、会话操作函数（一般不用）

pid\_t getpgrp(void);

pid\_t getpgid(pid\_t pid);

int setpgid(pid\_t pid, pid\_t pgid);

pid\_t getsid(pid\_t pid);

pid\_t setsid(void);

##### 守护进程的基础概念

* 守护进程（Daemon Process），也就是通常说的 Daemon 进程（精灵进程），是Linux 中的后台服务进程。它是一个生存期较长的进程，通常独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。一般采用以 d 结尾的名字。
* 守护进程具备下列特征：
  + 生命周期很长，守护进程会在系统启动的时候被创建并一直运行直至系统被关闭。
  + 它在后台运行并且不拥有控制终端。没有控制终端确保了内核永远不会为守护进程自动生成任何控制信号以及终端相关的信号（如 SIGINT、SIGQUIT）。、
* Linux 的大多数服务器就是用守护进程实现的。比如，Internet 服务器 inetd，Web 服务器 httpd 等。

##### 守护进程的创建步骤

* 执行一个 fork()，之后父进程退出，子进程继续执行。（1.为了不出现用户提示符2.创建会话的进程不能是进程组的组长）
* 子进程调用 setsid() 开启一个新会话。（如果父进程创建会话就会产生不同会话有相同的进程组，产生冲突）
* 清除进程的 umask 以确保当守护进程创建文件和目录时拥有所需的权限。
* 修改进程的当前工作目录，通常会改为根目录（/）。
* 关闭守护进程从其父进程继承而来的所有打开着的文件描述符。
* 在关闭了文件描述符0、1、2之后，守护进程通常会打开/dev/null 并使用dup2() 使所有这些描述符指向这个设备
* 核心业务逻辑

## Linux多线程开发

### 线程

#### 1.1 线程基本概念

* 与进程（process）类似，线程（thread）是允许应用程序并发执行多个任务的一种机制。一个进程可以包含多个线程。同一个程序中的所有线程均会独立执行相同程序，且共享同一份全局内存区域，其中包括初始化数据段、未初始化数据段，以及堆内存段。（传统意义上的 UNIX 进程只是多线程程序的一个特例，该进程只包含一个线程）
* 进程是 CPU 分配资源的最小单位，线程是操作系统调度执行的最小单位。
* 线程是轻量级的进程（LWP：Light Weight Process），在 Linux 环境下线程的本质仍是进程。
* 查看指定进程的 LWP 号：ps -Lf pid

#### 线程和进程的区别

* 进程间的信息难以共享。由于除去只读代码段外，父子进程并未共享内存，因此必须采用

一些进程间通信方式，在进程间进行信息交换。

* 调用 fork() 来创建进程的代价相对较高，即便利用写时复制技术，仍然需要复制诸如内存页表和文件描述符表之类的多种进程属性，这意味着 fork() 调用在时间上的开销依然不菲
* 线程之间能够方便、快速地共享信息。只需将数据复制到共享（全局或堆）变量中即可。
* 创建线程比创建进程通常要快 10 倍甚至更多。线程间是共享虚拟地址空间的，无需采

用写时复制来复制内存，也无需复制页表。

#### 线程和进程的虚拟地址空间



#### 线程之间共享和非共享资源

共享资源：进程 ID 和父进程 ID 非共享资源：线程 ID

进程组 ID 和会话 ID 信号掩码

用户 ID 和 用户组 ID 线程特有数据

文件描述符表 error变量

信号处置 实时调度策略和优先级

文件系统的相关信息：文件权限掩码 栈，本地变量和函数的调用链接信息

（umask）、当前工作目录虚拟地址空间（除栈、.text）

#### Native POSIX Thread Library (NPTL)

* 当 Linux 最初开发时，在内核中并不能真正支持线程。但是它的确可以通过 clone() 系统调用将进程作为可调度的实体。这个调用创建了调用进程（calling process）的一个拷贝，这个拷贝与调用进程共享相同的地址空间。LinuxThreads 项目使用这个调用来完成在用户空间模拟对线程的支持。不幸的是，这种方法有一些缺点，尤其是在信号处理、调度和进程间同步等方面都存在问题。另外，这个线程模型也不符合 POSIX 的要求。
* 要改进 LinuxThreads，需要内核的支持，并且重写线程库。有两个相互竞争的项目开始来满足这些要求。一个包括 IBM 的开发人员的团队开展了 NGPT（Next-Generation POSIX Threads）项目。同时，Red Hat 的一些开发人员开展了 NPTL 项目。NGPT 在 2003 年中期被放弃了，把这个领域完全留给了 NPTL。
* NPTL，或称为 Native POSIX Thread Library，是 Linux 线程的一个新实现，它克服了 LinuxThreads 的缺点，同时也符合 POSIX 的需求。与 LinuxThreads 相比，它在性能和稳定性方面都提供了重大的改进。
* 查看当前 pthread 库版本：getconf GNU\_LIBPTHREAD\_VERSION

#### 线程基本操作

##### pthread\_create函数

一般情况下,main函数所在的线程我们称之为主线程(main线程),其余创建的线程称之为子线程。

程序中默认只有一个进程,fork()函数调用，2个进程

程序中默认只有一个线程,pthread\_create()函数调用,2个线程

#include <pthread.h>

int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr,

void \*(\*start\_routine) (void \*), void \*arg);

功能:创建一个子线程

参数:

- thread:传出参数，线程创建成功后，子线程的线程id被写到该变量中

- attr:设置线程的属性，一般使用默认值,NULL

- start\_routine : 函数指针,这个函数是子线程需要处理的逻辑代码

- arg : 给第三个参数使用,传参

返回值:

成功:0

失败:返回错误号。这个错误号和之前的errno不一样。

获取错误号的信息: char \*strperror(int errnum)

第三方库:gcc pthread\_create.c -o create -pthread

##### pthread\_self函数

pthread\_t pthread\_self(void);

功能:返回当前线程的id

##### pthread\_equal函数

int pthread\_equal(pthread\_t t1, pthread\_t t2);

功能:比较两个线程ID是否相等

不同的操作系统，pthread\_t类型的实现不一样，有的是无符号的长整型，有的是使用结构体去实现的。

##### pthread\_exit函数

void pthread\_exit(void \*retval);

功能: 终止一个线程,在哪个线程中调用，就表示终止哪个线程

参数:

retval:需要传递一个指针,作为一个返回值,可以在pthread\_join()中获取到

##### pthread\_join函数

#include <pthread.h>

int pthread\_join(pthread\_t thread, void \*\*retval);

功能: 和一个已经终止的线程进行连接

回收子线程的资源

这个函数是阻塞函数,调用一次只能回收一个子线程

一般在主线程中使用

参数:

thread:需要回收的子线程的ID

retval:接收子线程退出时的返回值

返回值:

0 :成功

非0 ： 失败，返回的错误号

##### pthread\_detach函数

#include <pthread.h>

int pthread\_detach(pthread\_t thread);

功能:分离一个线程,被分离的线程在终止的时候会自动释放资源返回给系统。

1. 不能多次分离,会产生不可预料的行为。

2. 不能去连接一个已经分离的进程,如果去连接会报错。

参数：需要分离的线程的ID

返回值: 分离失败返回错误号，成功返回0

##### pthread\_cancel函数

#include <pthread.h>

int pthread\_cancel(pthread\_t thread);

功能: 取消线程(让线程终止)

取消某个线程，可以终止某个线程的运行

但是并不是立马终止，而是当子线程执行到一个取消点,线程才会终止

取消点:系统规定好的一些系统调用，我们可以粗略的理解为用户区到内核区的切换，这个位置称为取消点。

##### 线程属性操作函数

#include <pthread.h>

int pthread\_attr\_init(pthread\_attr\_t \*attr);

- 初始化线程属性变量

int pthread\_attr\_destroy(pthread\_attr\_t \*attr);

- 释放线程属性的资源

int pthread\_attr\_setdetachstate(pthread\_attr\_t \*attr, int detachstate);

- 设置线程分离的状态属性（getstacksize取出线程栈的大小）

int pthread\_attr\_getdetachstate(const pthread\_attr\_t \*attr, int \*detachstate);

- 获取线程分离的状态属性

### 2. 线程同步

## Linux网络编程

## Linux高性能服务器