# 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

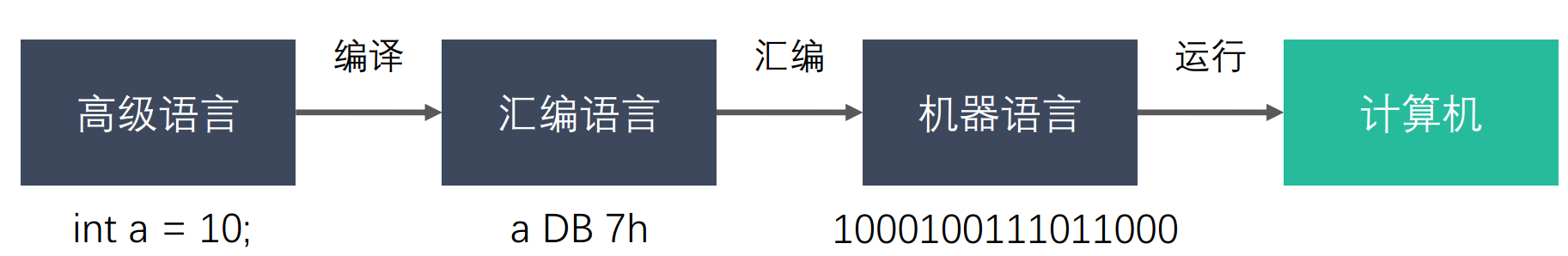
* GCC 原名为 GNU C语言编译器（GNU C Compiler），后来拓展为GCC（GNU Compiler Collection，GNU编译器套件）是由 GNU 开发的编程语言编译器。GNU 编译器套件包括 C、C++、Objective-C、Java、Ada 和 Go 语言前端，也包括了这些语言的库（如 libstdc++，libgcj等）
* GCC 不仅支持 C 的许多“方言”，也可以区别不同的 C 语言标准；可以使用命令行

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

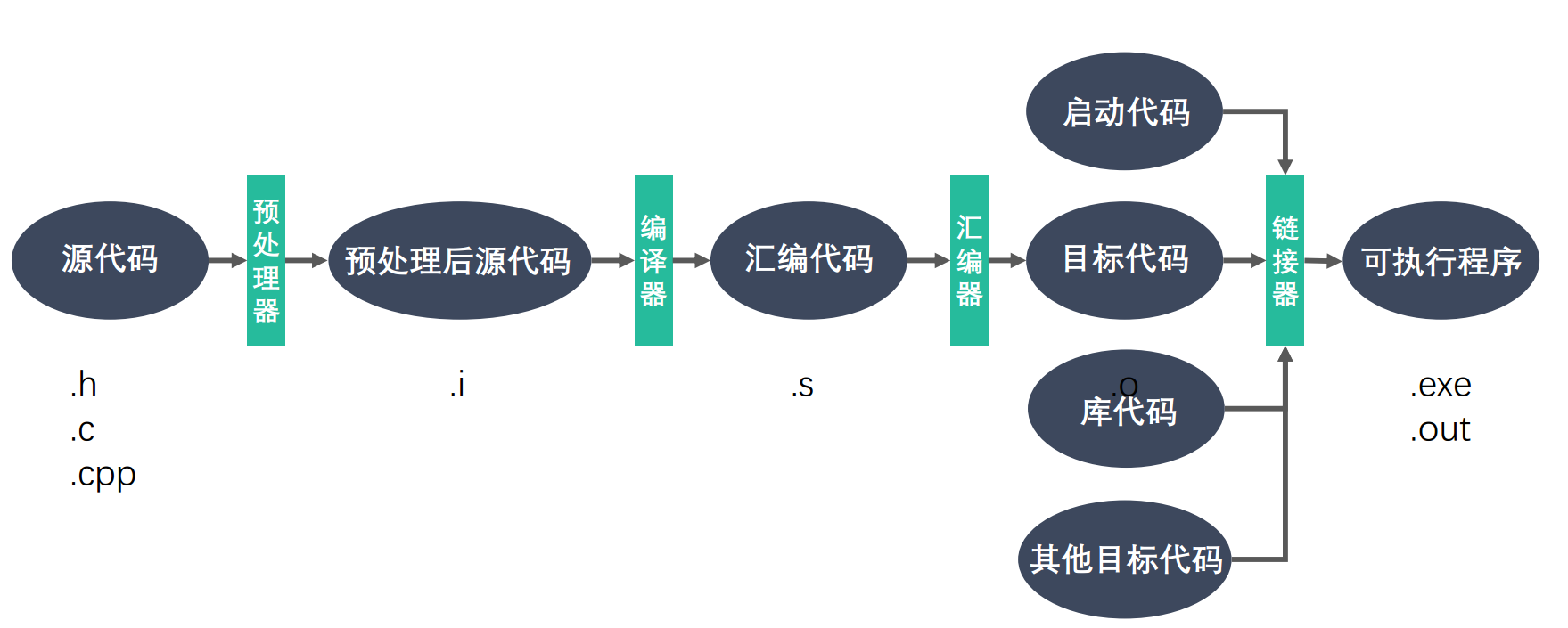
`-std=c99` 启动 GCC 时，编译器支持 C99 标准。

* 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

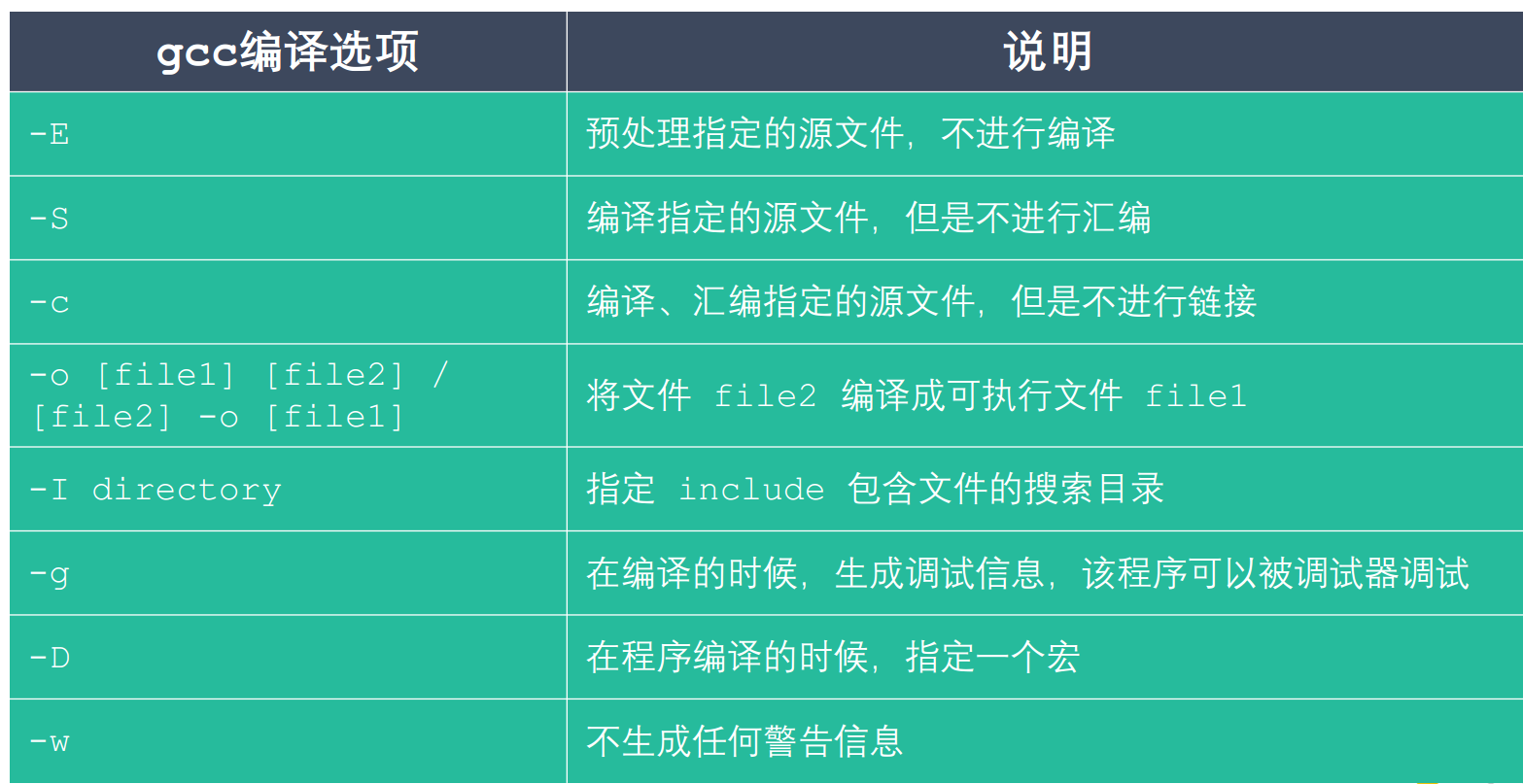
gcc -E test.c => test.i

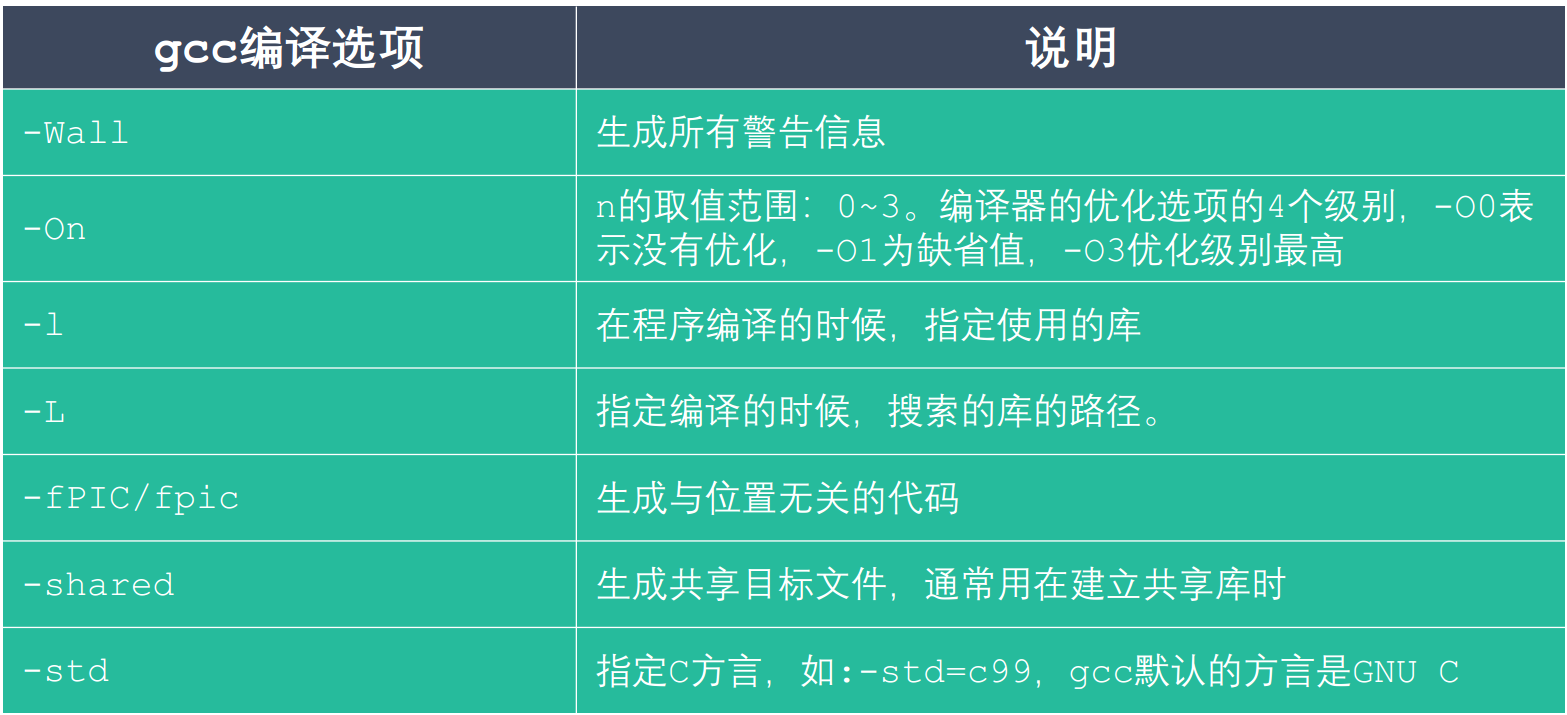
gcc -S test.c => test.s

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

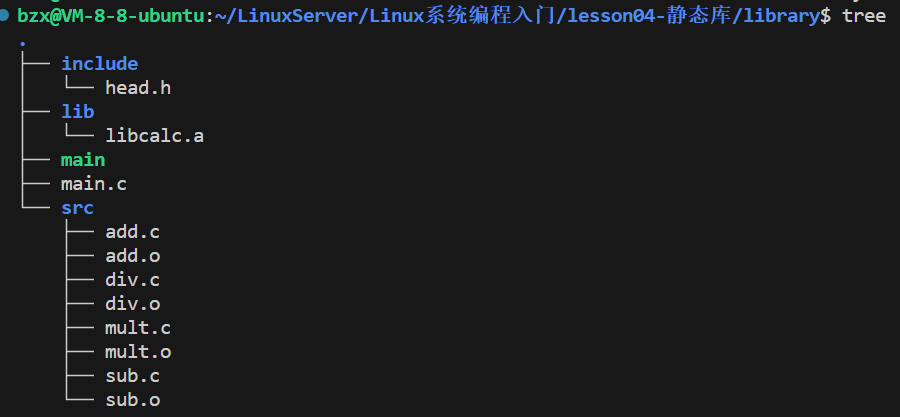
r – 将文件插入备存文件中

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

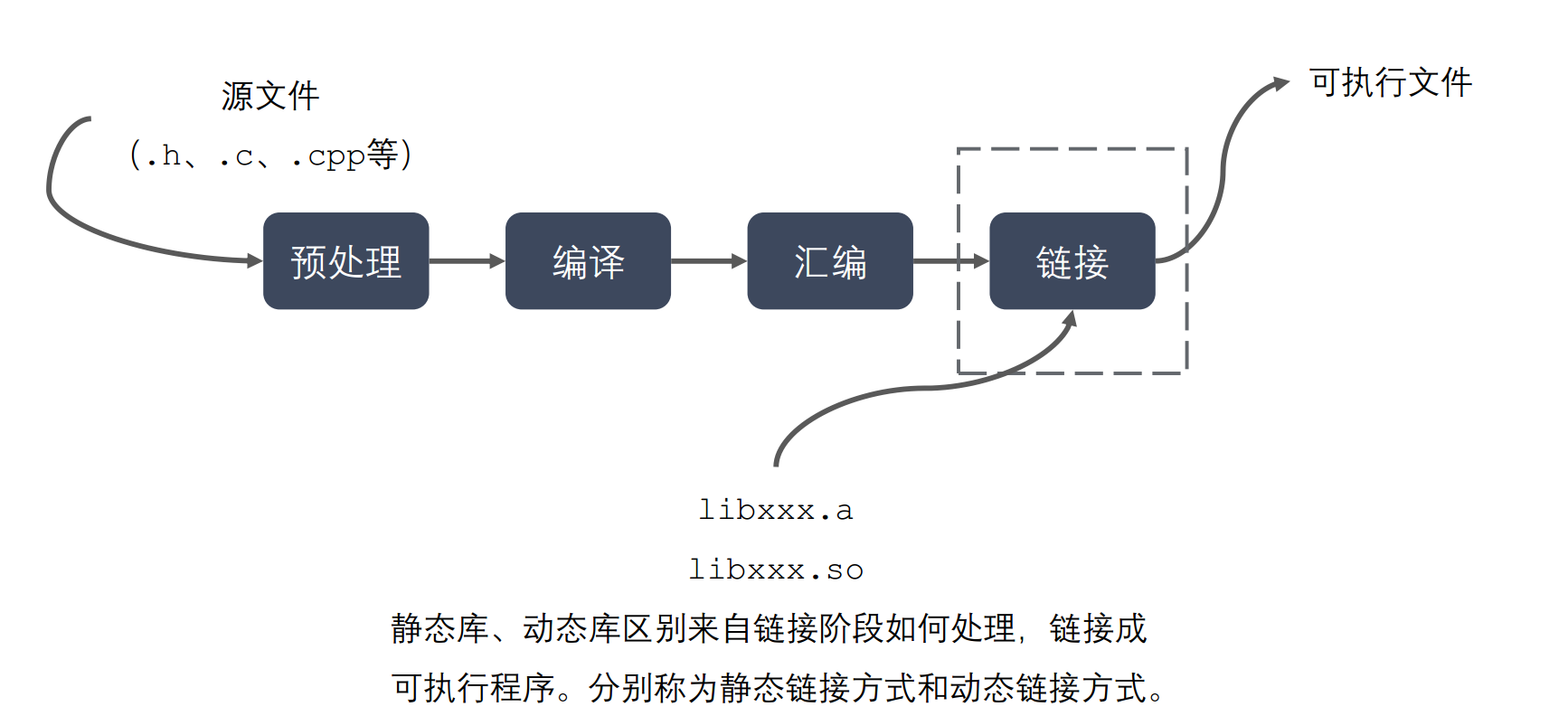
执行source etc/profile

——> /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)

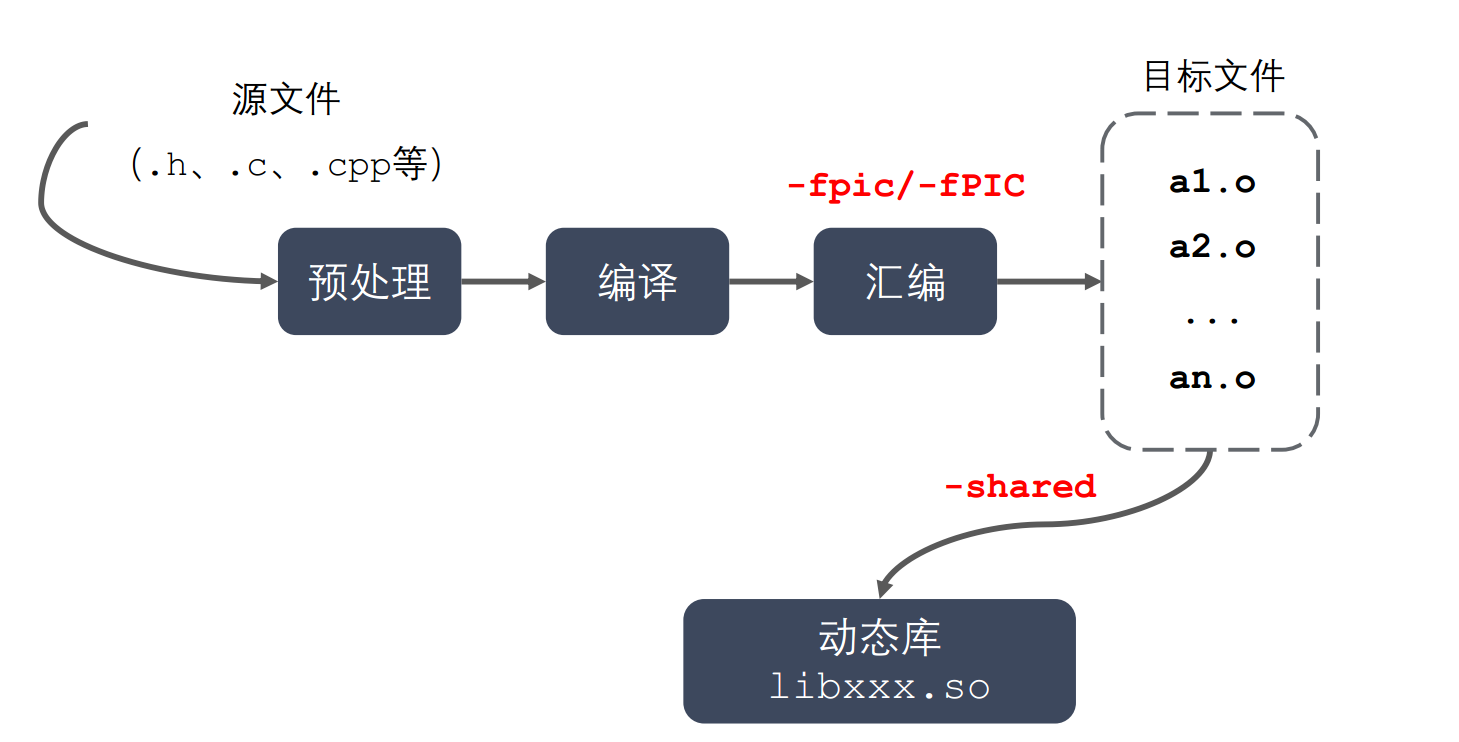
### 动态库和静态库的对比

#### 程序编译成可执行程序的过程

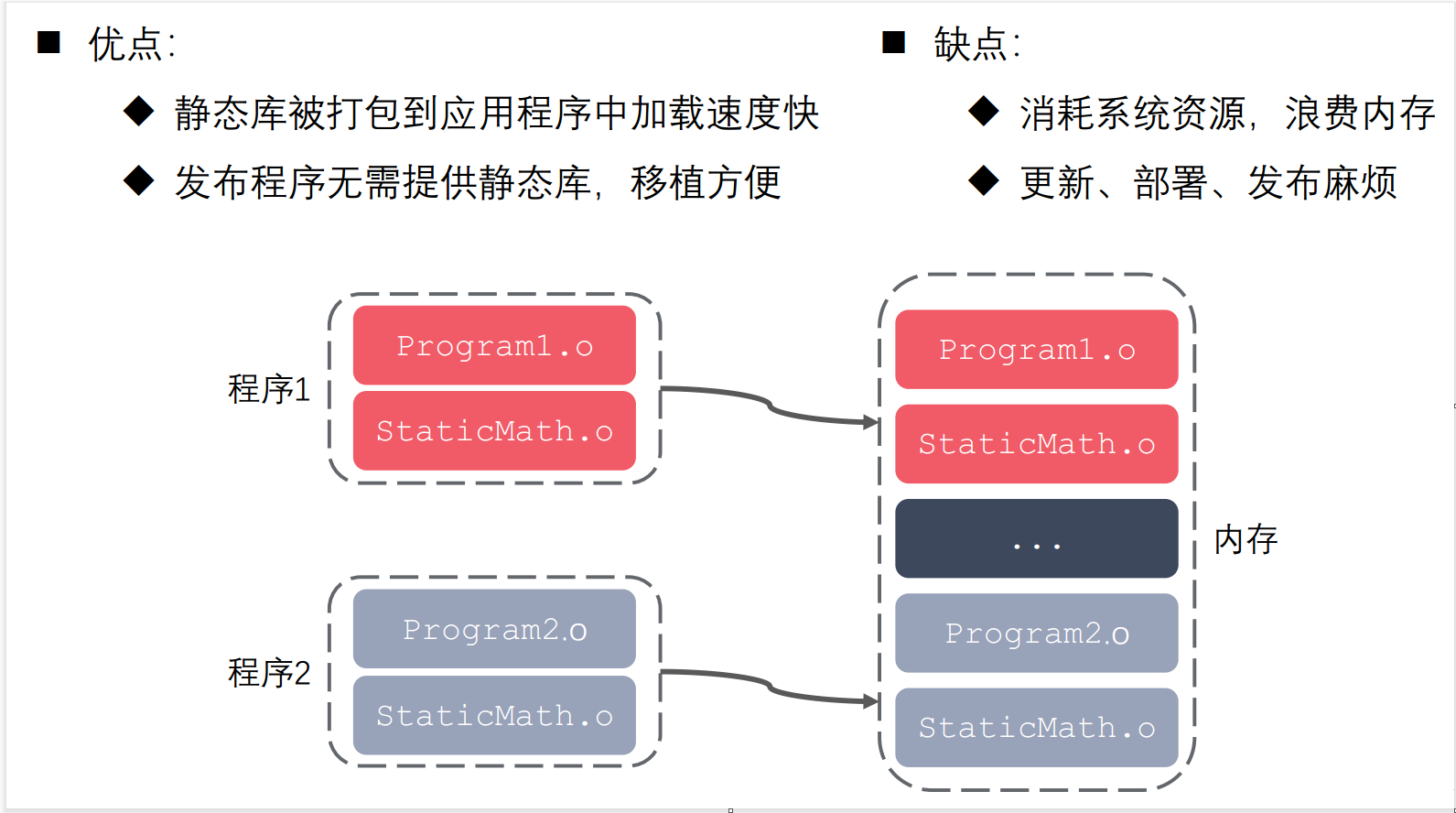


#### 5.2 静态库制作过程

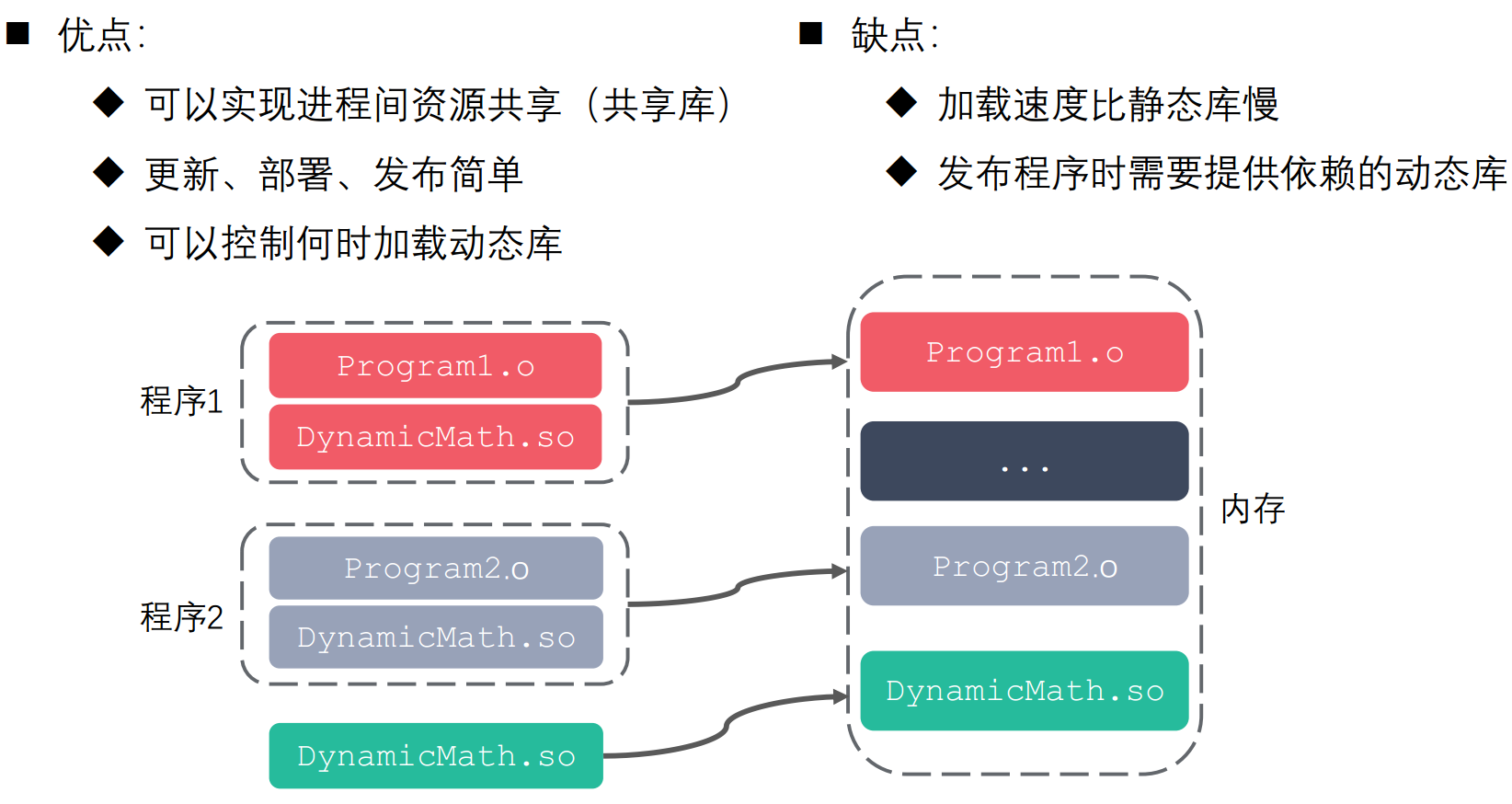
#### 5.3 动态库制作过程



#### 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

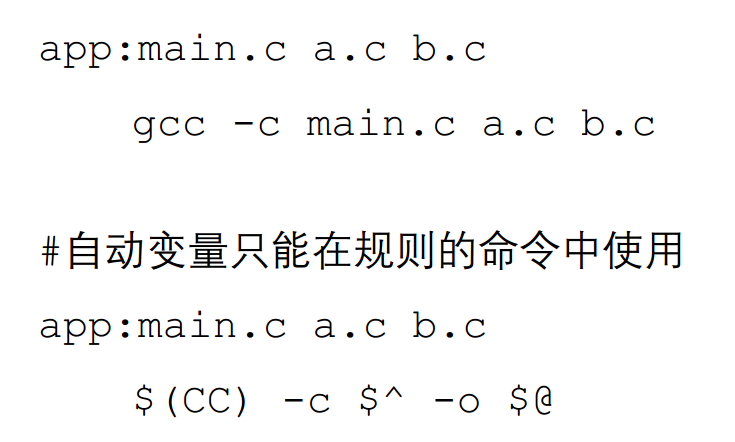
* 命令在执行之前，需要先检查规则中的依赖是否存在
  + 如果存在，执行命令
  + 如果不存在，向下检查其它的规则，检查有没有一个规则是用来生成这个依赖的，如果找到了，则执行该规则中的命令
* 检测更新，在执行规则中的命令时，会比较目标和依赖文件的更新时间
  + 如果依赖的时间比目标的时间晚，需要重新生成目标
  + 如果依赖的时间比目标的时间早，目标不需要更新，对应规则中的命令不需要被执行

#### 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

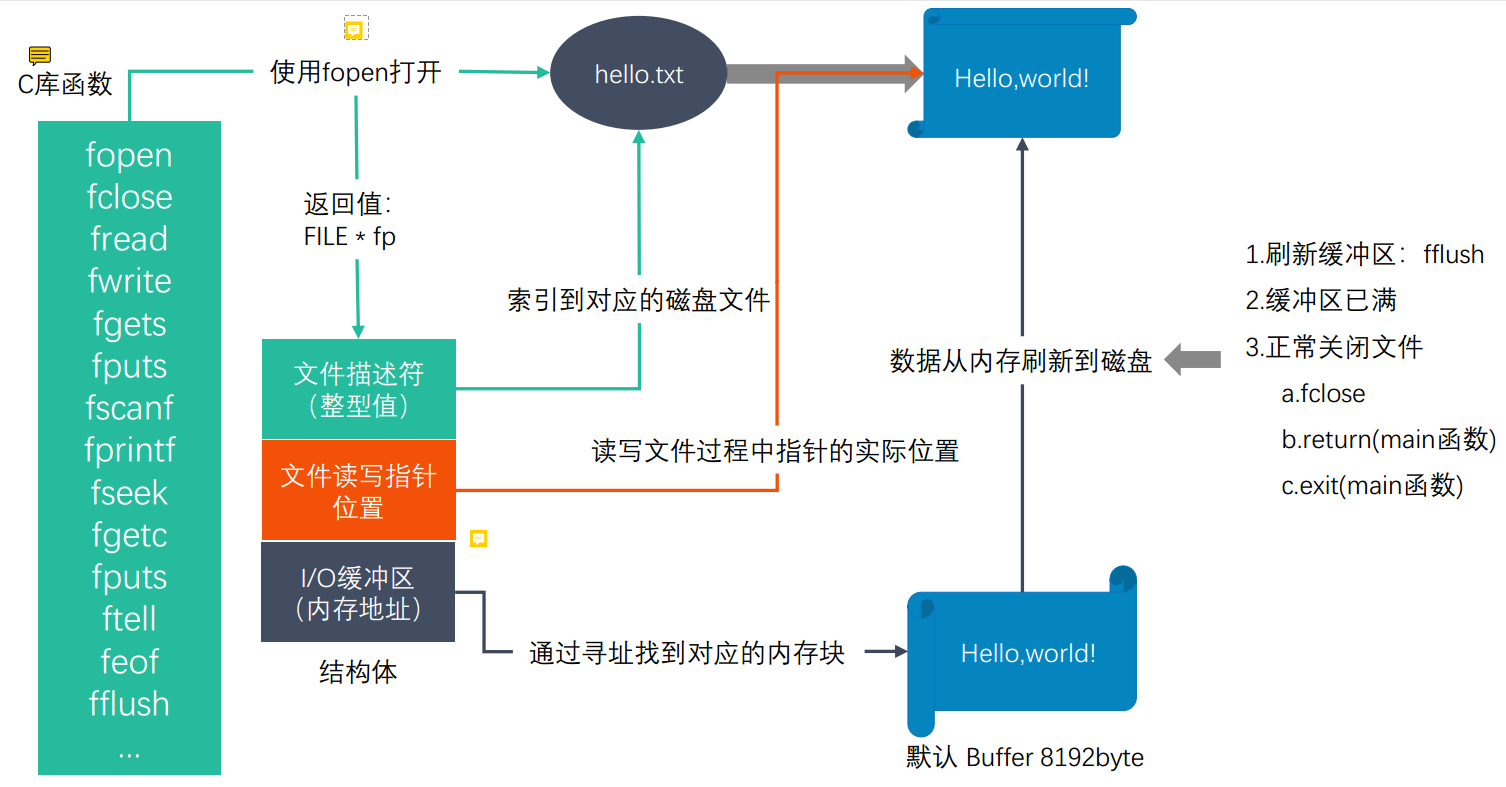
### 文件IO

#### 标准 C 库 IO 函数

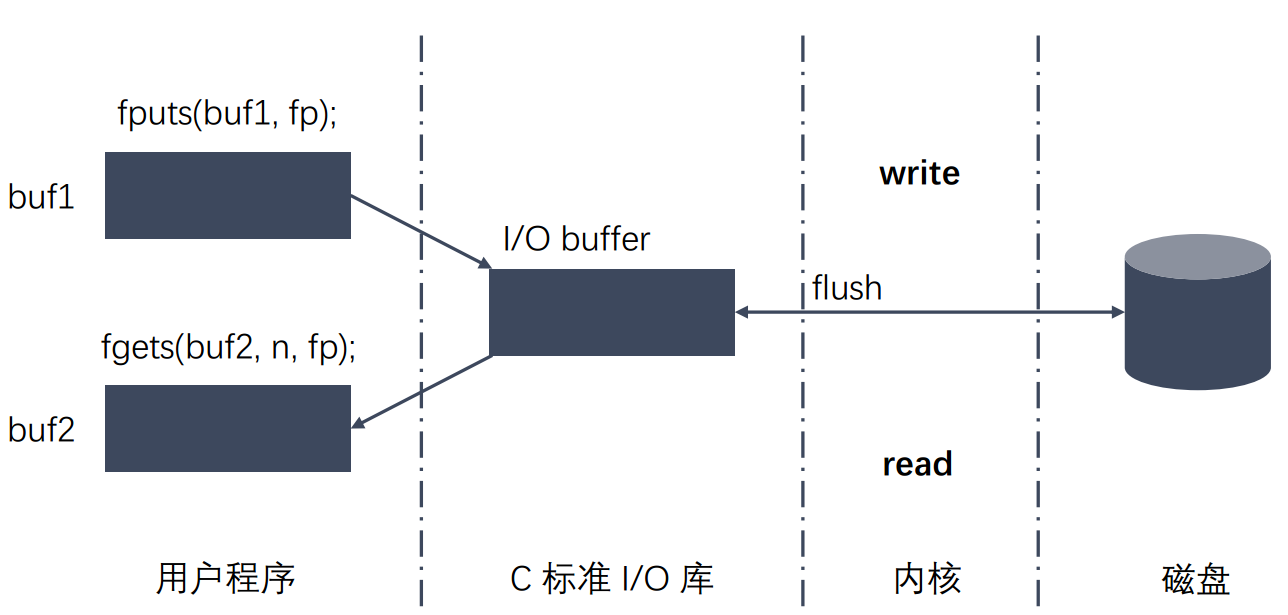
C库函数：C库函数可以通过调用不同的操作系统IO函数实现跨平台，Java不同平台的虚拟机实现跨平台，一般采用C库函数，但是网络通讯为了避免防止缓冲区的延时性，写代码时一般直接调用Linux系统的IO函数

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



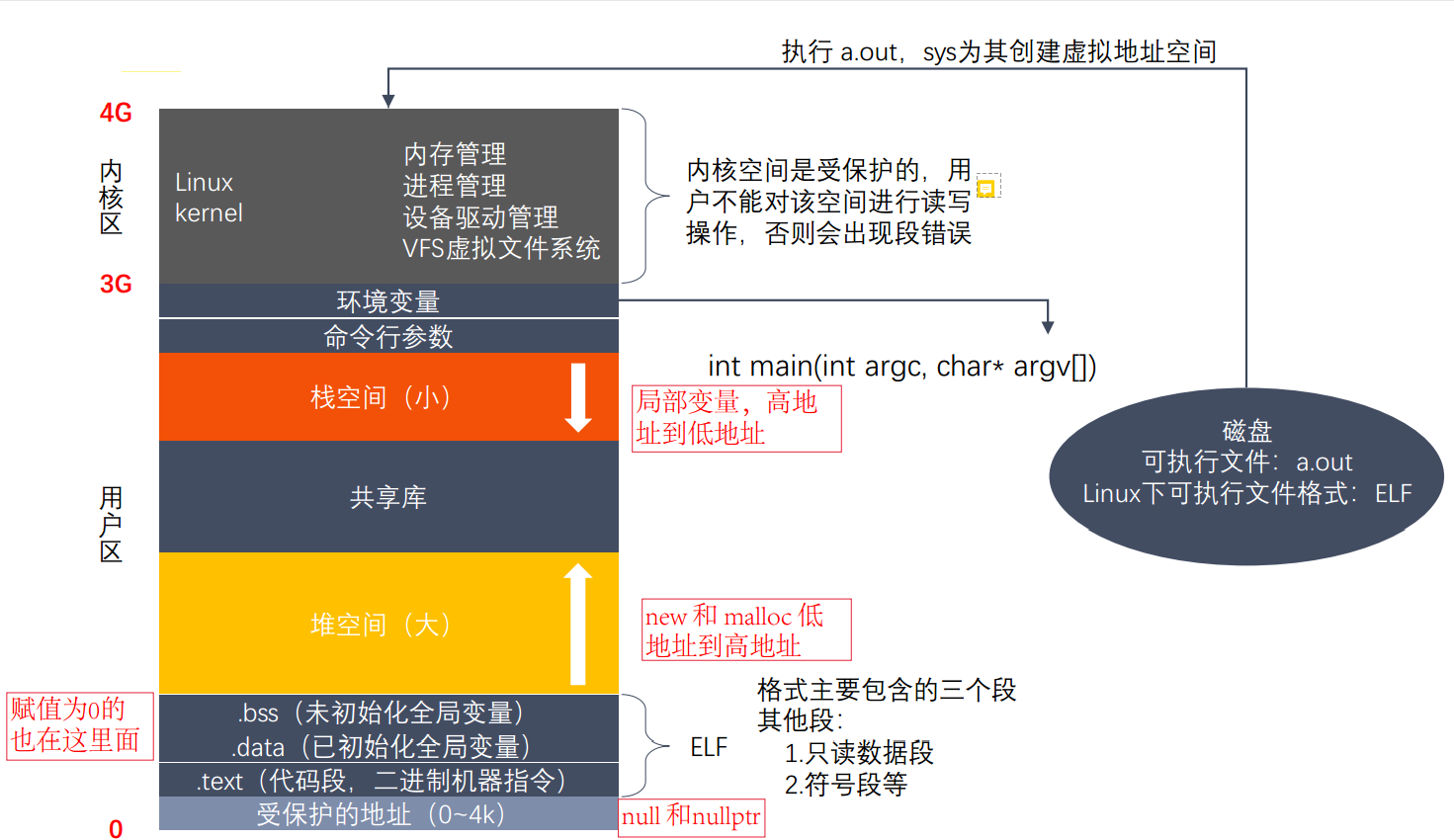
#### 8.2 标准C库IO和Linux系统IO的关系（调用与被调用的关系）



#### 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）

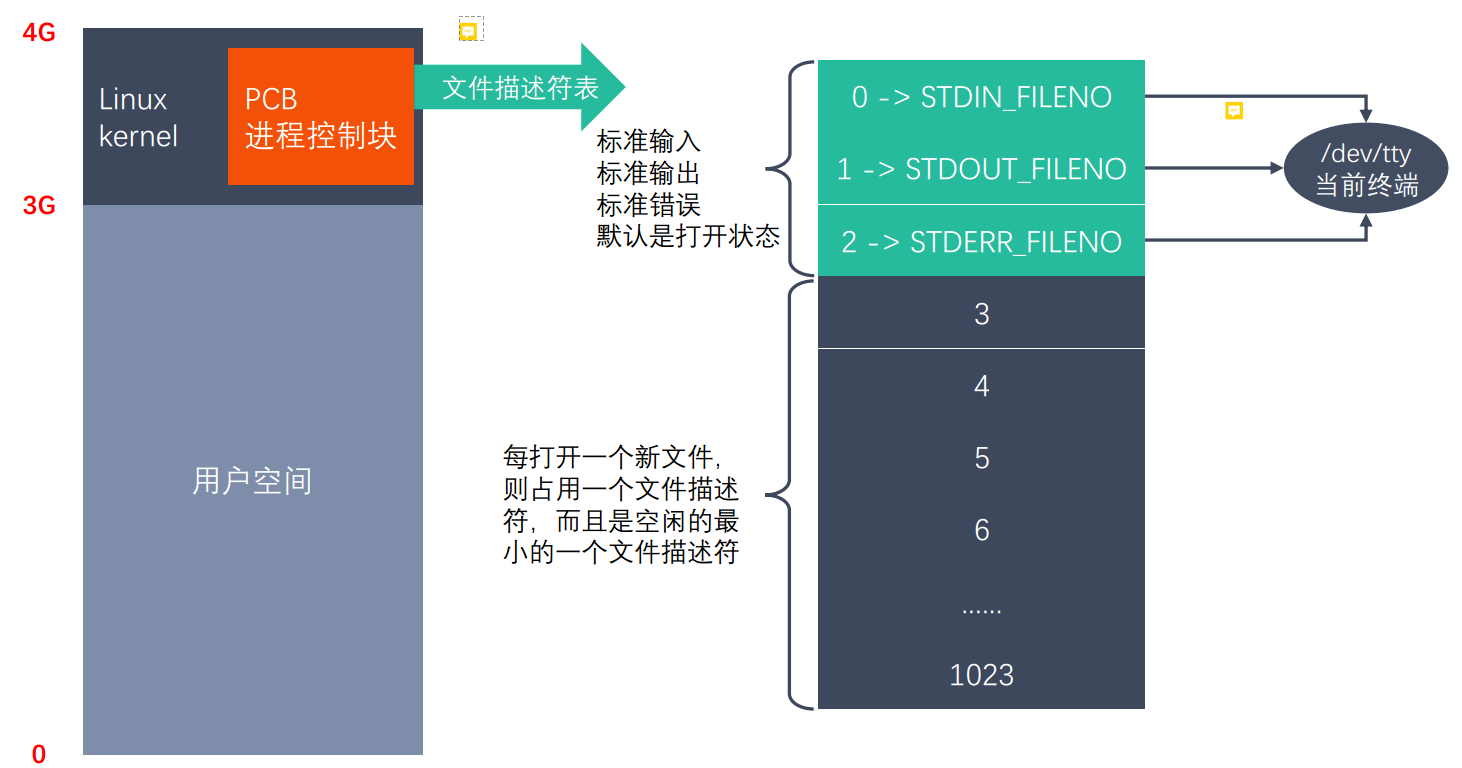
由MMU映射到物理内存中



#### 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

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

1. 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

1. close函数

int close(int fd); 关闭某个文件描述符的函数

1. 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

1. 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

1. 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);

1. 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

1. 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

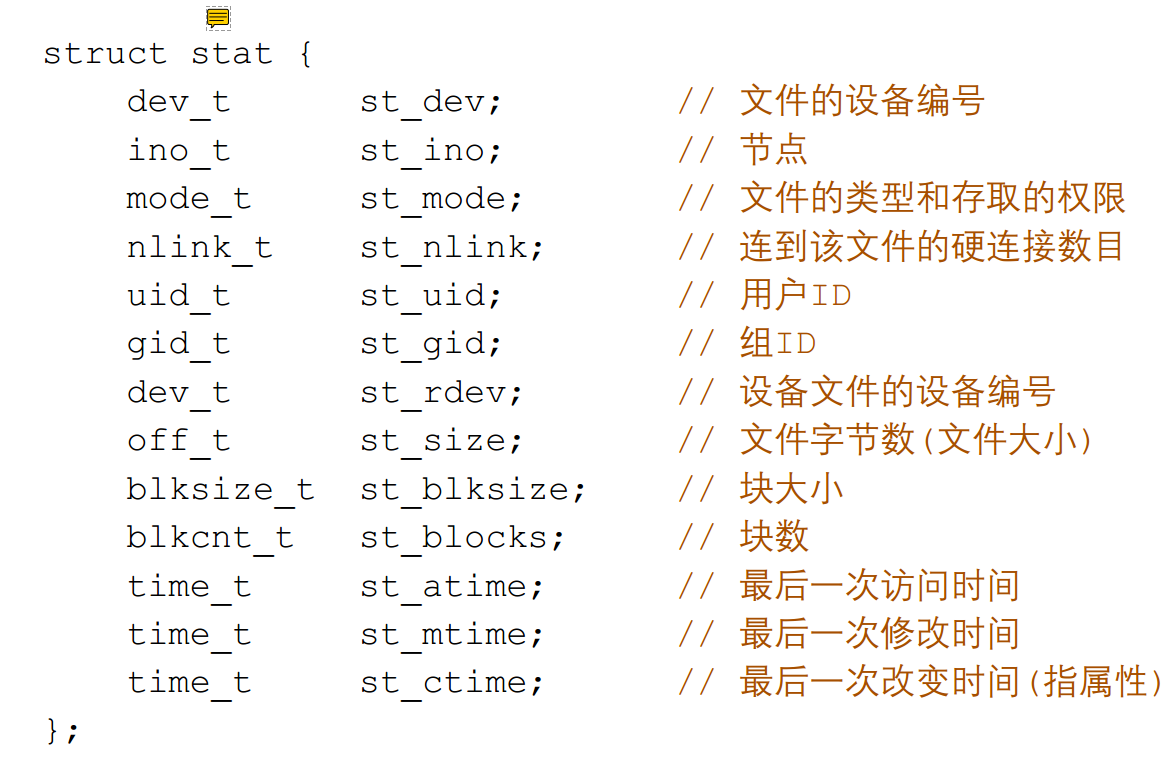
-statbuf：结构体变量，传出参数，用于保存获取到的文件信息

返回值：

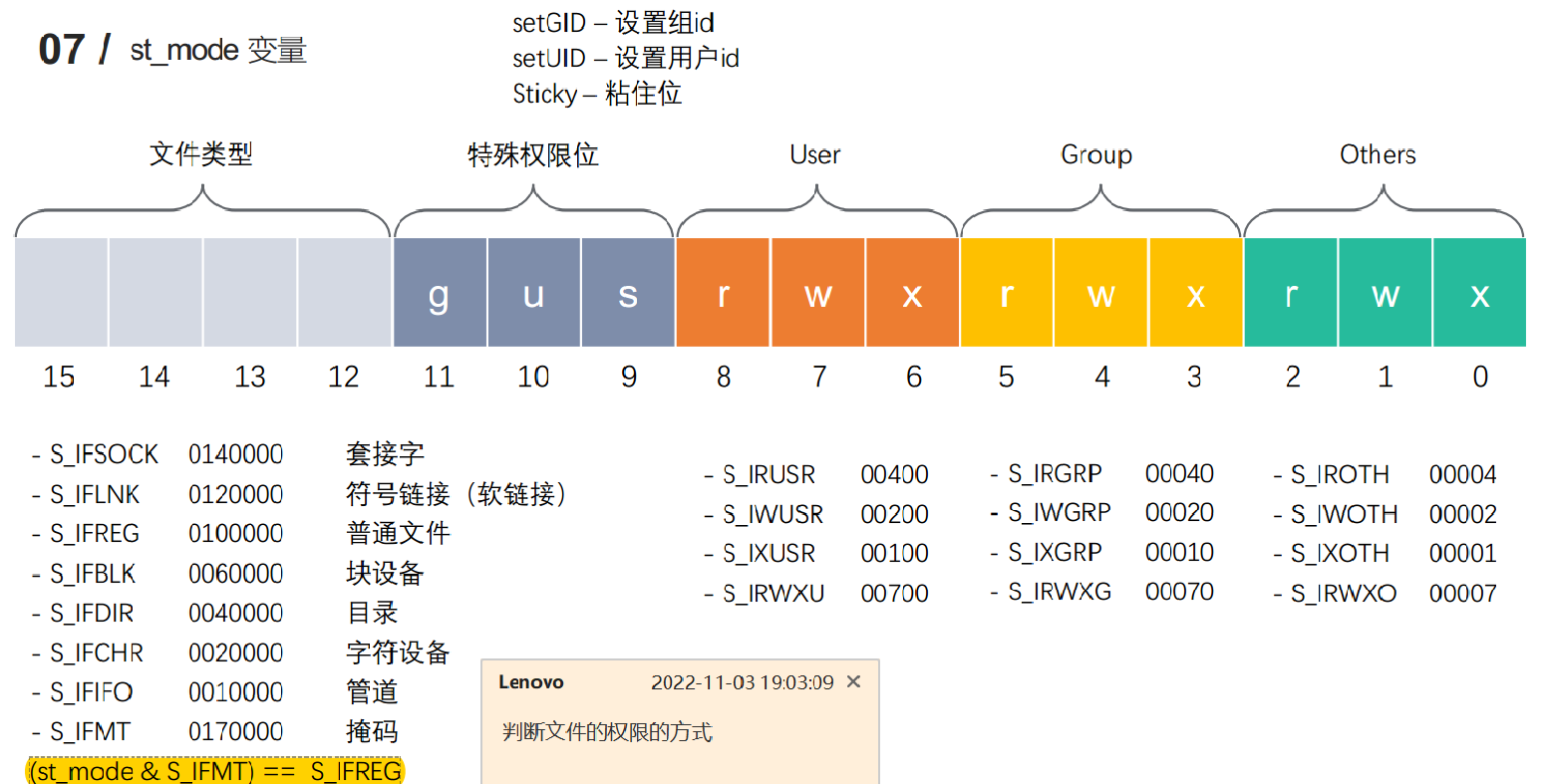
成功：返回0

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

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

1. 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

1. 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

1. rename函数

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

1. 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

1. 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

1. 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

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

1. 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

1. closedir函数

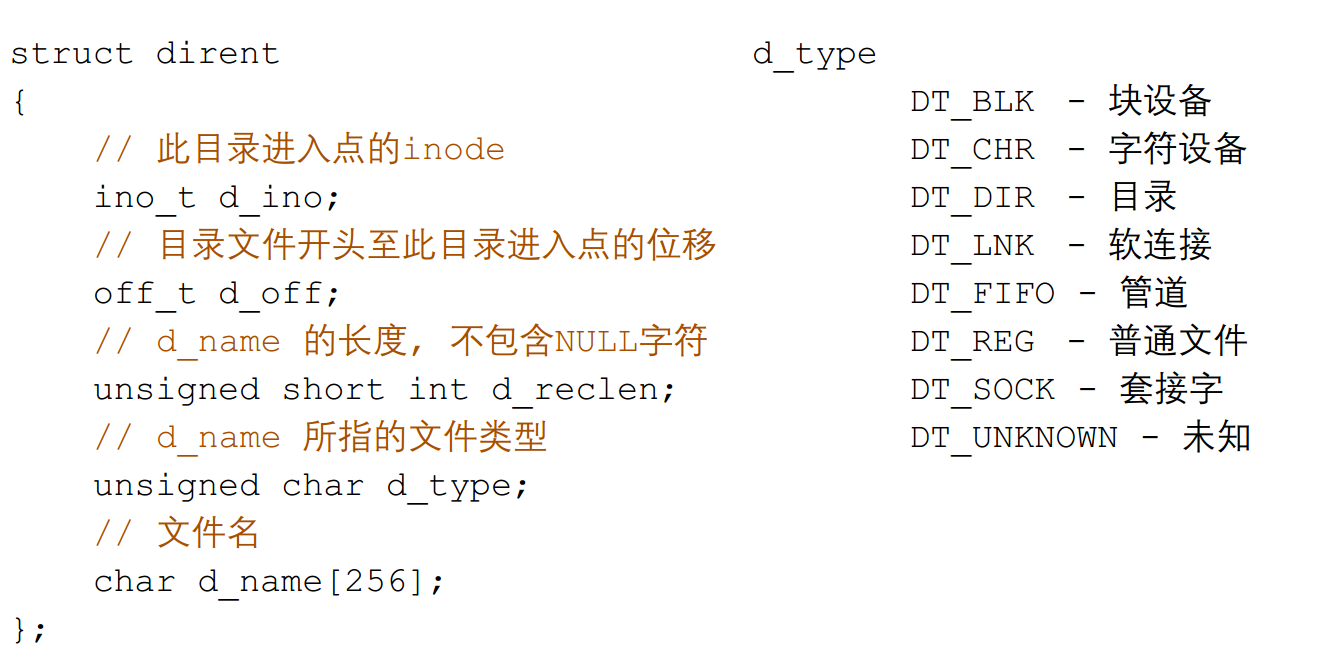
//关闭目录

#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

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

1. 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。

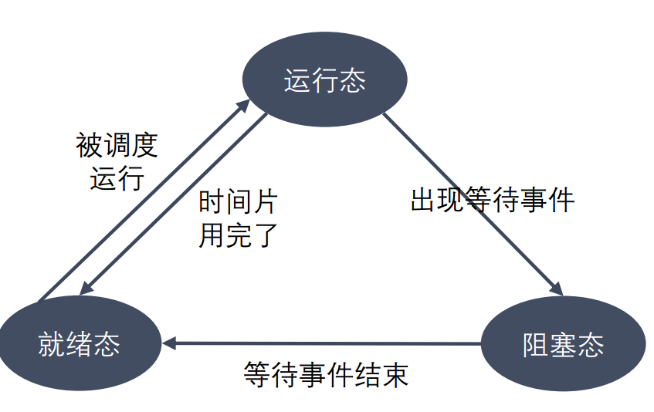
#### 进程控制块

* 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）

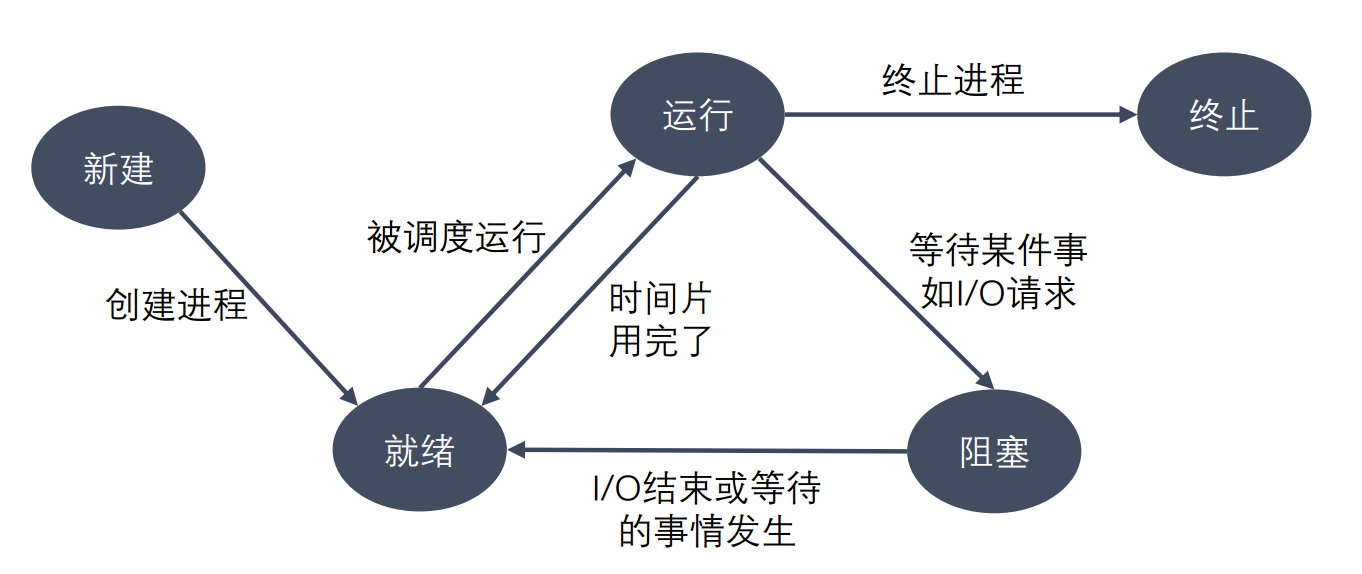
### 进程状态转换

#### 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

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

1. 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

1. 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

1. 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。

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

1. 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

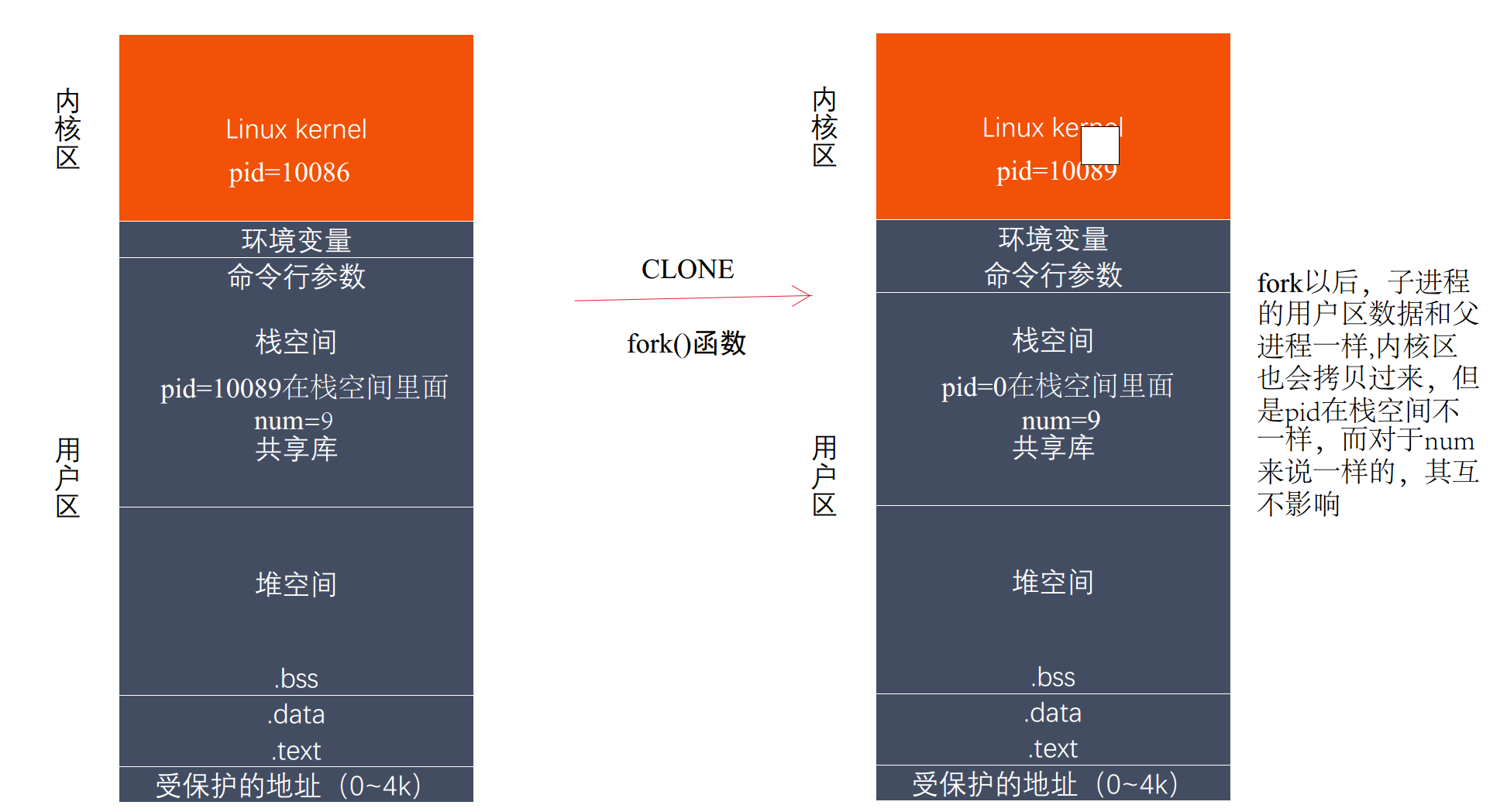
子进程中: =0

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:

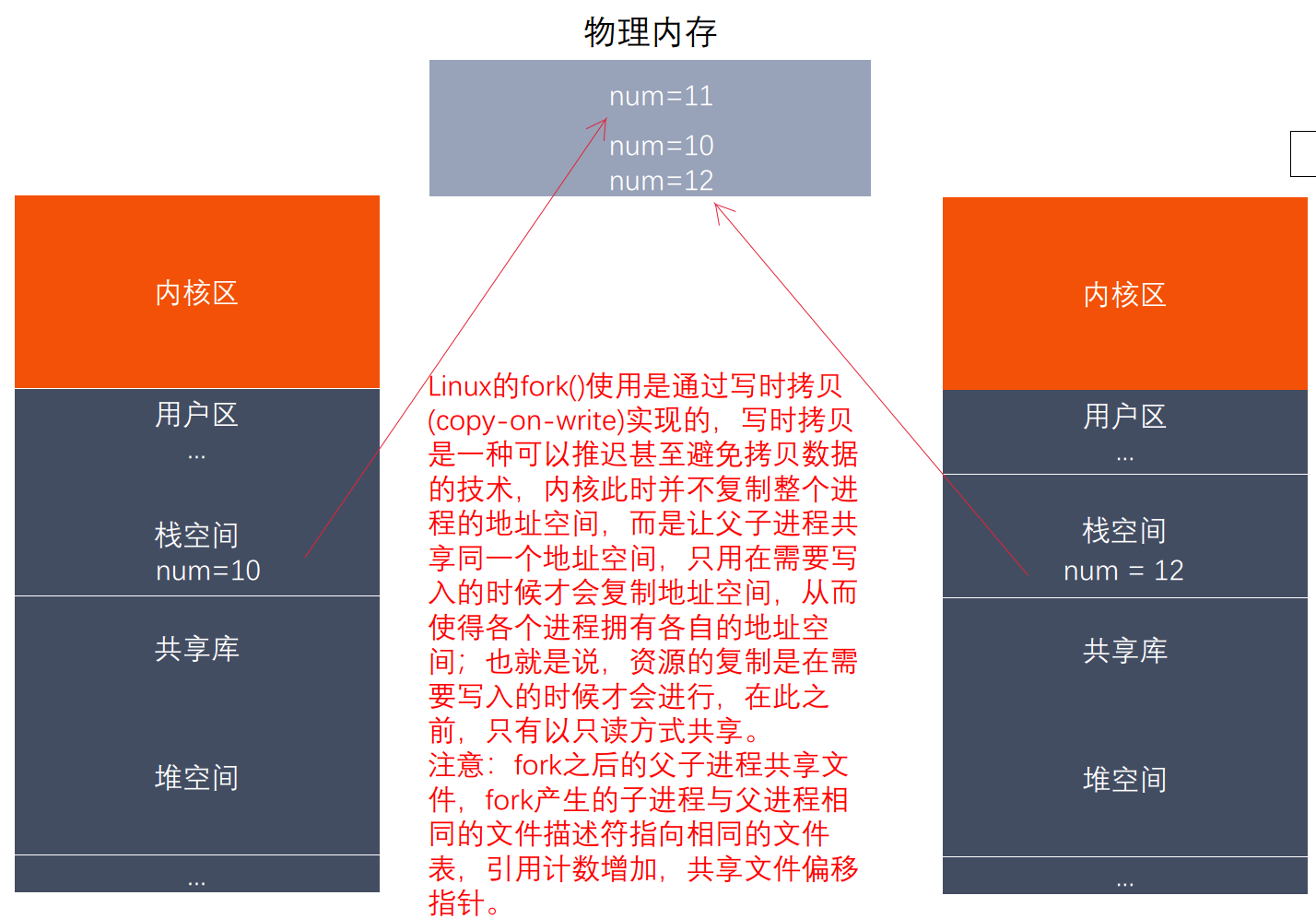
某些状态下：子进程刚被创建出来，还没有执行任何的写数据的操作

- 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。

1. 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）

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。

1. execl函数族作用图解



1. exec函数族

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

1. 进程退出

## Linux多线程开发

## Linux网络编程

## Linux高性能服务器