1. 开发环境搭建

安装Linux系统（虚拟机安装、云服务器）

https://releases.ubuntu.com/bionic/

安装XSHELL、XFTP

https://www.netsarang.com/zh/free-for-home-school/

安装Visual Studio Code

https://code.visualstudio.com/

安装MySQL数据库

https://segmentfault.com/a/1190000023081074

阿里镜像

https://developer.aliyun.com/mirror/

安装sshd服务

sudo apt install openssh-server

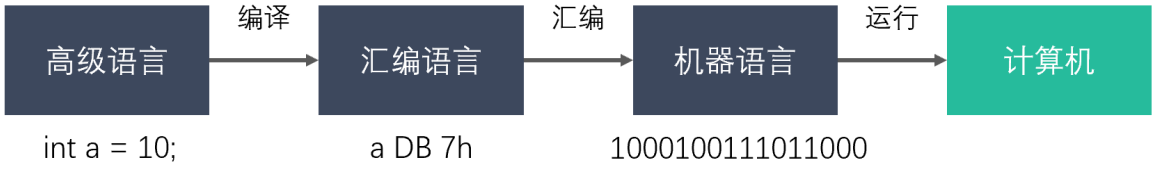
安装gcc/g++/make等 工具

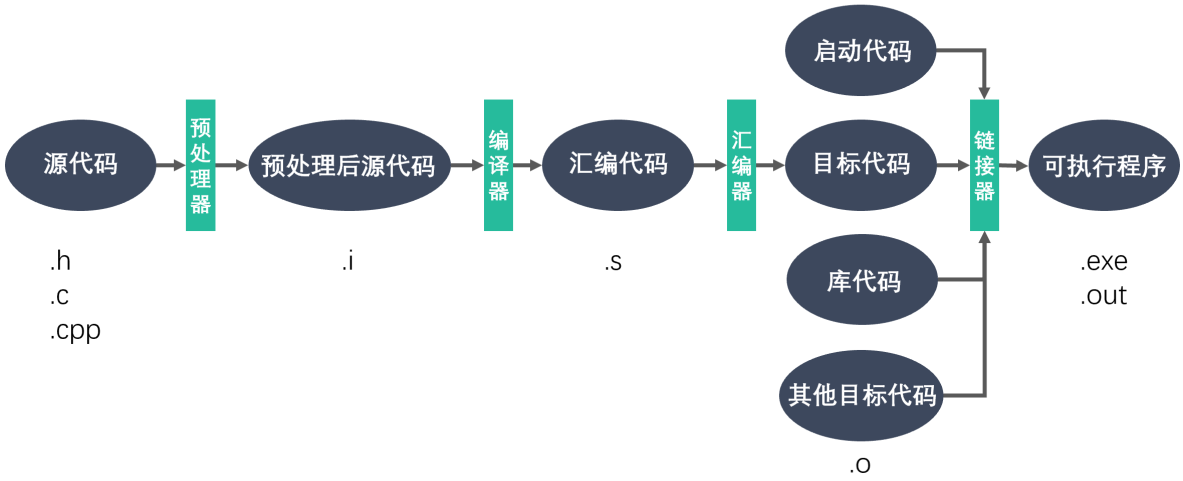
sudo apt install build-essential

2. GCC

- GCC 原名为 GNU C语言编译器（GNU C Compiler）
- GCC（GNU Compiler Collection，GNU编译器套件）是由 GNU 开发的编程语言译器。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 build-essential
- 查看版本 gcc/g++ -v/--version

2.1 GCC工作流程





2.2 GCC常用参数选项

gcc编译选项	说明
-E	预处理指定的源文件，不进行编译
-S	编译指定的源文件，但是不进行汇编
-c	编译、汇编指定的源文件，但是不进行链接
-o [file1] [file2] / [file2] -o [file1]	将文件 file2 编译成可执行文件 file1
-I directory	指定 include 包含文件的搜索目录
-g	在编译的时候，生成调试信息，该程序可以被调试器调试
-D	在程序编译的时候，指定一个宏
-w	不生成任何警告信息
-Wall	生成所有警告信息
-On	n的取值范围：0~3。编译器的优化选项的4个级别，-O0表示没有优化，-O1为缺省值，-O3优化级别最高
-l	在程序编译的时候，指定使用的库
-L	指定编译的时候，搜索的库的路径。
-fPIC/fpic	生成与位置无关的代码
-shared	生成共享目标文件，通常用在建立共享库时
-std	指定C方言，如:-std=c99，gcc默认的方言是GNU C

3. Makefile

3.1 简介

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。

3.2 Makefile 文件命名和规则

1.文件命名 : makefile 或者 Makefile

2.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。

3.3 基本原理

1.命令在执行之前，需要先检查规则中的依赖是否存在

a.如果存在，执行命令

b.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

2.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时间

a.如果依赖的时间比目标的时间晚，需要重新生成目标

b.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

3.4 变量

1.自定义变量

变量名=变量值

```
var=hello
#获取变量的值 $(变量名)
$(var)
```

2.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

```
app:main.c a.c b.c
    gcc -c main.c a.c b.c -o app

#自动变量只能在规则的命令中使用
app:main.c a.c b.c
    $(CC) -c $^ -o $@
```

3.5 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 $@

3.6 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 ,,)

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

- 可以包括通配符`%`，表示任意长度的字串。如果中也包含`%`，那么，中的这个`%`将是中的那个%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

4. GDB

1. 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。
2. 一般来说，GDB 主要帮助你完成下面四个方面的功能：
   - 启动程序，可以按照自定义的要求随心所欲的运行程序
   - 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）
   - 当程序被停住时，可以检查此时程序中所发生的事
   - 可以改变程序，将一个 BUG 产生的影响修正从而测试其他 BUG
3. 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
   ```

4. `-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。
5. GDB命令

功能	命令
启动和退出	gdb 可执行程序 quit/q
给程序设置参数/获取设置参数	set args 10 20 show args
GDB 使用帮助	help
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）
查看非当前文件代码	list/l 文件名:行号 list/l 文件名:函数名
设置显示的行数	show list/listsize set list/listsize 行数
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
查看断点	i/info b/break
删除断点	d/del/delete 断点编号
设置断点无效	dis/disable 断点编号
设置断点生效	ena/enable 断点编号
设置条件断点（一般用在循环的位置）	b/break 10 if i==5
运行GDB程序	start（程序停在第一行） run（遇到断点才停）
继续运行，到下一个断点停	c/continue
向下执行一行代码（不会进入函数体）	n/next
变量操作	p/print 变量名（打印变量值） ptype 变量名（打印变量类型）
向下单步调试（遇到函数进入函数体）	s/step finish（跳出函数体）
自动变量操作	display 变量名（自动打印指定变量的值） i/info display undisplay 编号
其它操作	set var 变量名=变量值 （循环中用的较多） until （跳出循环）

5. 静态库和动态库

5.1 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

5.2 静态库

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

2.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

s – 索引

5.3 动态库

命名规则

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

动态库的制作：

1.gcc 得到 .o 文件，得到和位置无关的代码

```
gcc -c -fpic/-fPIC a.c b.c
```

2.gcc 得到动态库

```
gcc -shared a.o b.o -o libcalc.so
```

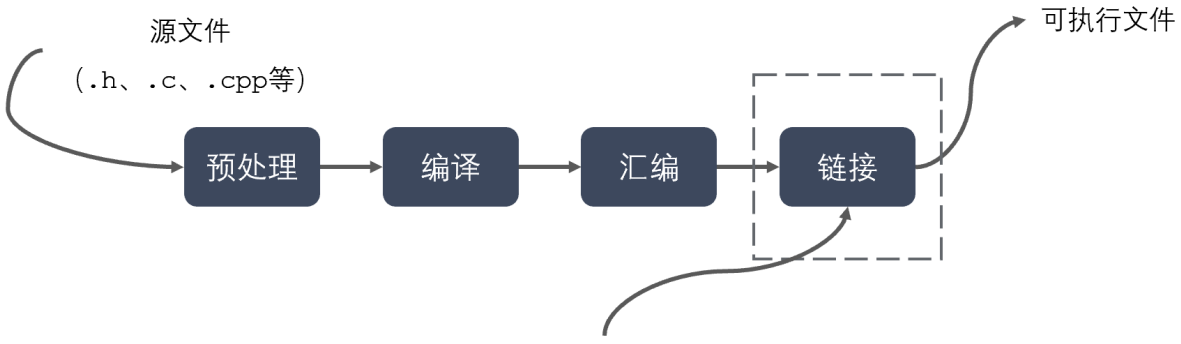
5.4 工作原理

- 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.cache 文件列表 ——> /lib/，/usr/lib 目录找到库文件后将其载入内存。

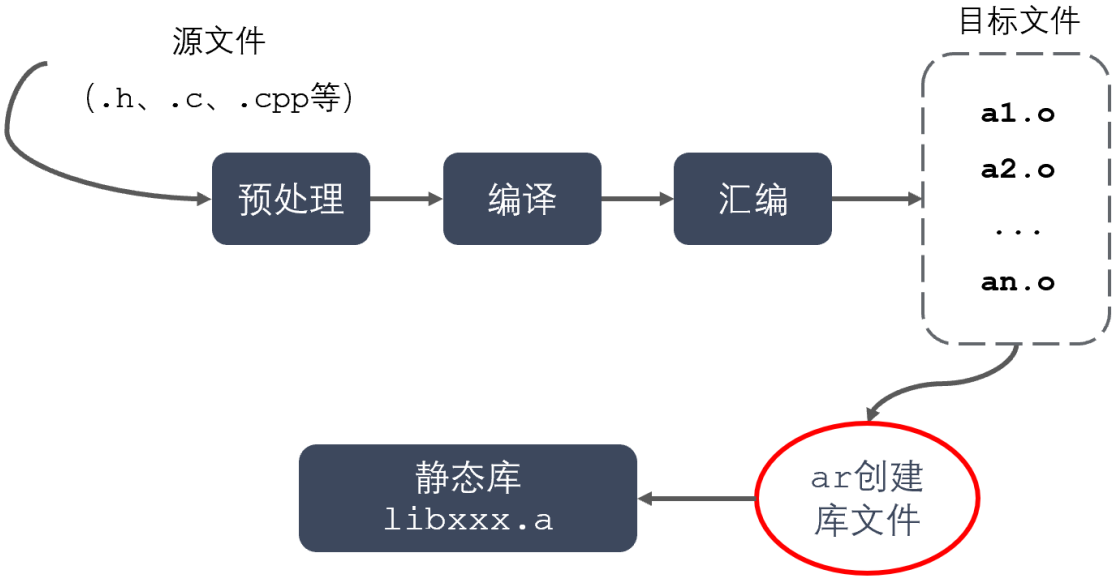
  export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/nowcoder/Linux/lesson03/04_lib/library/lib

5.5 区别

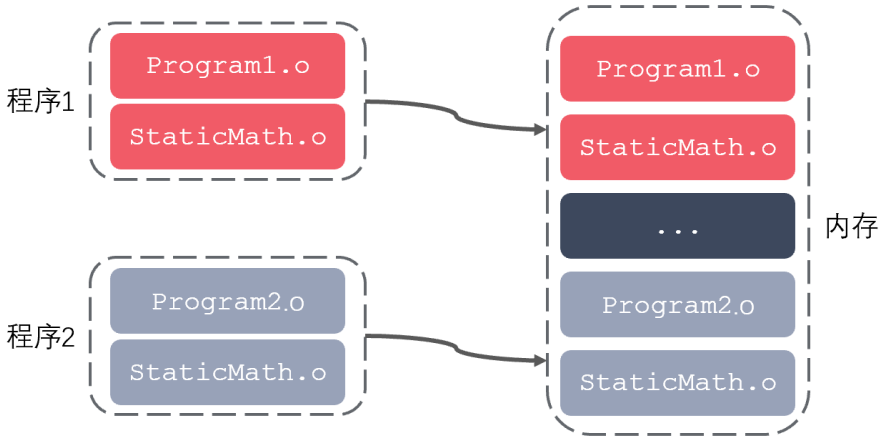


静态库、动态库区别来自链接阶段如何处理，链接成可执行程序。分别称为静态链接方式和动态链接方式。

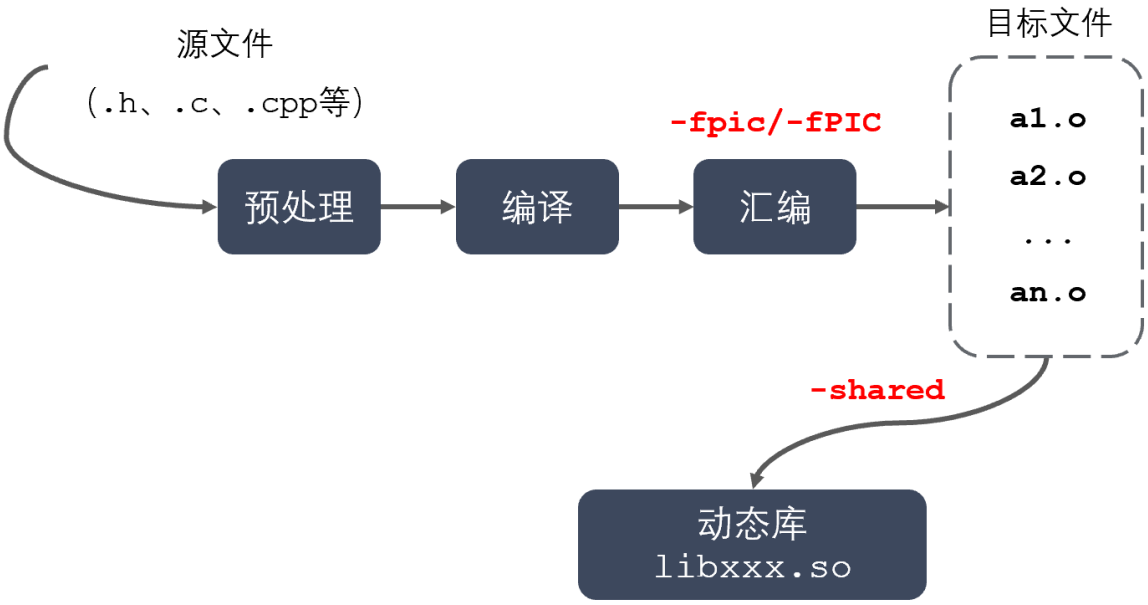
静态库



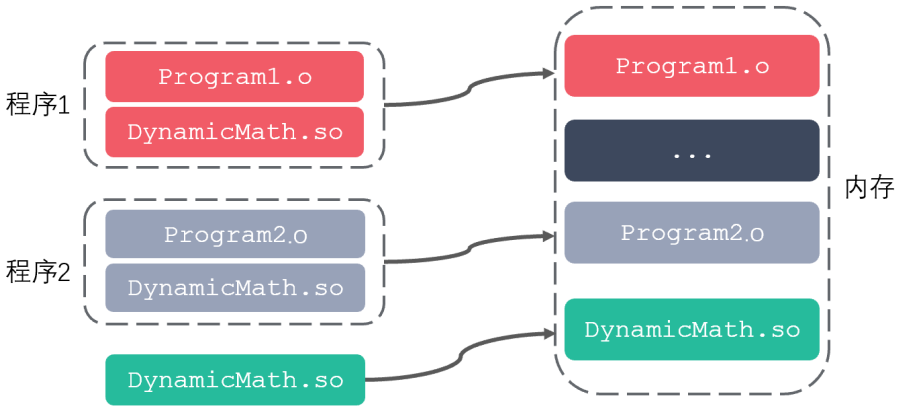
- 优点：
  - 静态库被打包到应用程序中加载速度快
  - 发布程序无需提供静态库，移植方便
- 缺点：
  - 消耗系统资源，浪费内存
  - 更新、部署、发布麻烦



动态库

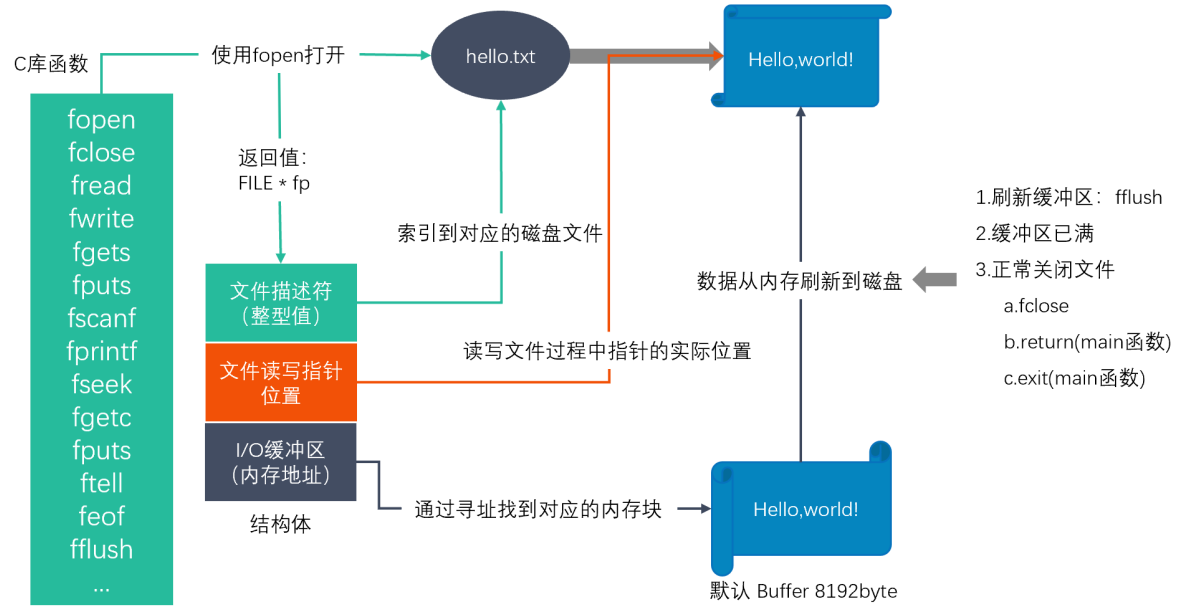


- 优点：
  - 可以实现进程间资源共享（共享库）
  - 更新、部署、发布简单
  - 可以控制何时加载动态库
- 缺点：
  - 加载速度比静态库慢
  - 发布程序时需要提供依赖的动态库

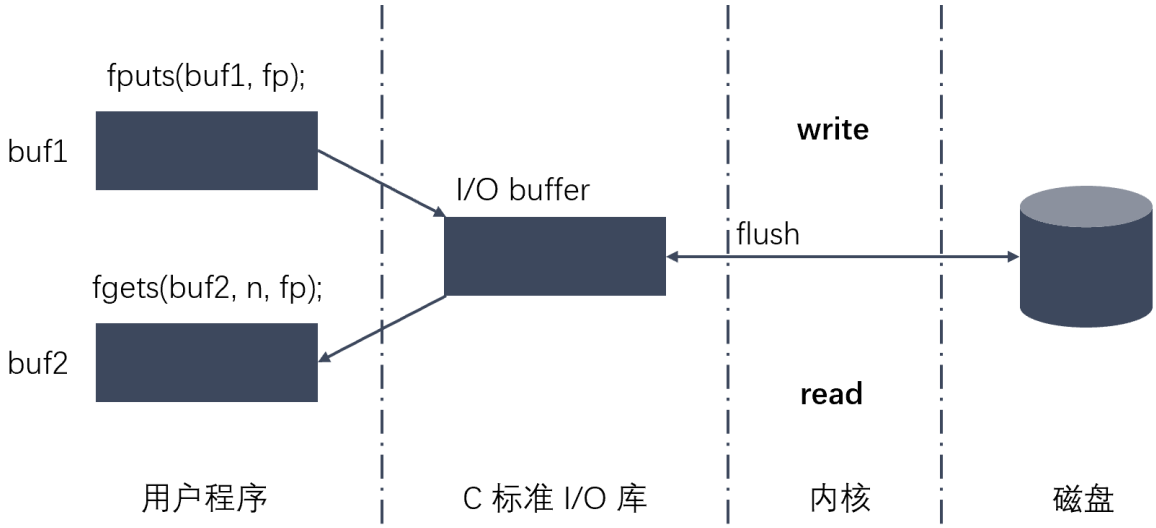


6. 文件IO

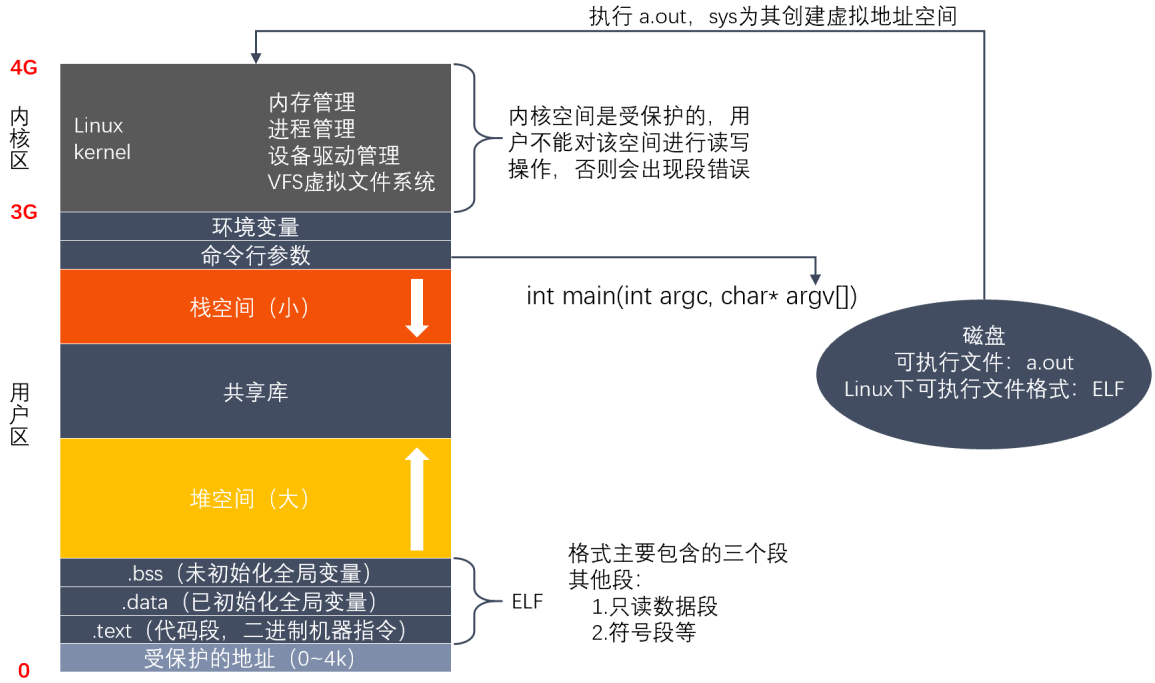
6.1 标准C库IO函数



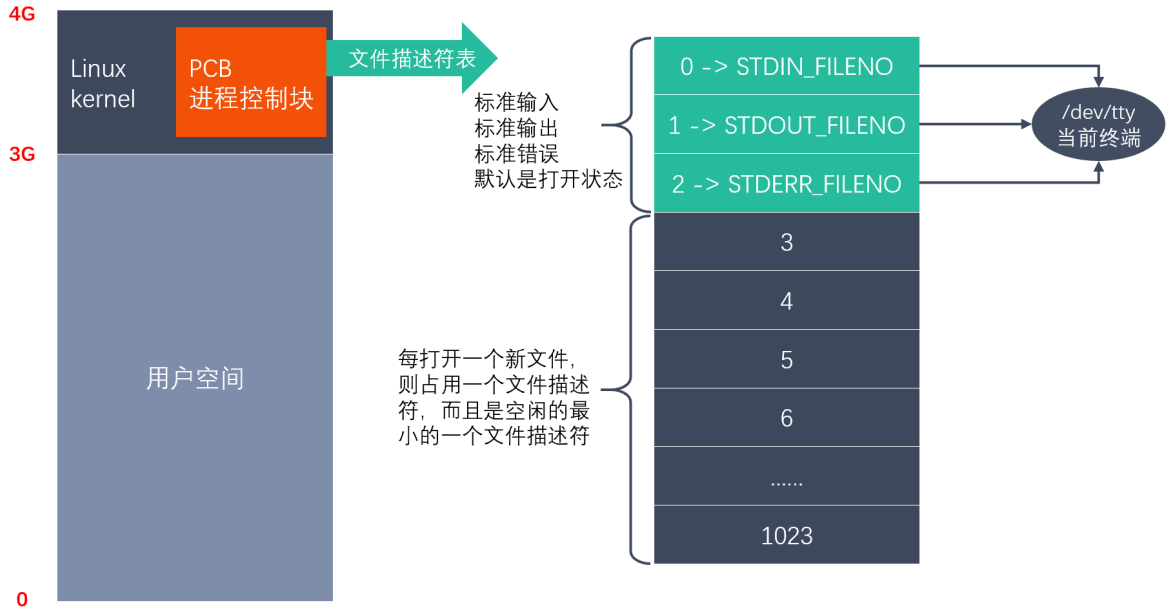
6.2 标准 C 库 IO 和 Linux 系统 IO 的关系



6.3 虚拟地址空间



6.4 文件描述符



6.5 Linux系统IO函数(Linux系统api一般也成为系统调用)

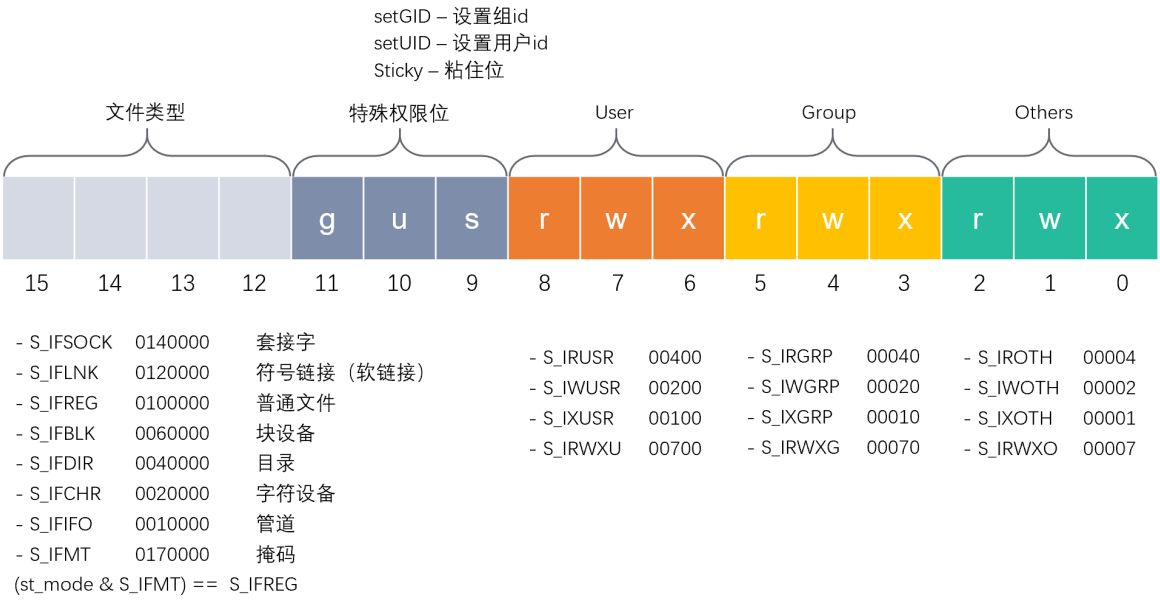
```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int close(int fd);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
off_t lseek(int fd, off_t offset, int whence);
int stat(const char *pathname, struct stat *statbuf);
int lstat(const char *pathname, struct stat *statbuf);
perro("aaa");       "aaa":XXXX
```

stat 结构体:

```
struct stat {
```

```
    dev_t   st_dev;     // 文件的设备编号
    ino_t   st_ino;     // 节点
    mode_t  st_mode;    // 文件的类型和存取的权限
    nlink_t st_nlink;   // 连到该文件的硬连接数目
    uid_t   st_uid;     // 用户ID
    gid_t   st_gid;     // 组ID
    dev_t   st_rdev;    // 设备文件的设备编号
    off_t   st_size;          // 文件字节数(文件大小)
    blksize_t   st_blksize;     // 块大小
    blkcnt_t    st_blocks;      // 块数
    time_t  st_atime;       // 最后一次访问时间
    time_t  st_mtime;       // 最后一次修改时间
    time_t  st_ctime;       // 最后一次改变时间(指属性)
};
```

st_mode 变量:



6.6 文件属性操作函数

```
int access(const char *pathname, int mode);
int chmod(const char *filename, int mode);
int chown(const char *path, uid_t owner, gid_t group);
int truncate(const char *path, off_t length);
```

6.7 目录操作函数

```
int rename(const char *oldpath, const char *newpath);
int chdir(const char *path);
char *getcwd(char *buf, size_t size);
int mkdir(const char *pathname, mode_t mode);
int rmdir(const char *pathname);
```

6.8 目录遍历函数

```
DIR *opendir(const char *name);
struct dirent *readdir(DIR *dirp);
int closedir(DIR *dirp);
```

6.9 dirent 结构体和 d_type

```
struct dirent
{
    // 此目录进入点的inode
    ino_t d_ino;
    // 目录文件开头至此目录进入点的位移
    off_t d_off;
    // d_name 的长度, 不包含NULL字符
    unsigned short int d_reclen;
    // d_name 所指的文件类型
    unsigned char d_type;
    // 文件名
    char d_name[256];
};
```

```
d_type
    DT_BLK  - 块设备
    DT_CHR  - 字符设备
    DT_DIR  - 目录
    DT_LNK  - 软连接
    DT_FIFO - 管道
    DT_REG  - 普通文件
    DT_SOCK - 套接字
    DT_UNKNOWN - 未知
```

6.10 dup、dup2 函数

```
// 复制文件描述符
int dup(int oldfd);
// 重定向文件描述符
int dup2(int oldfd, int newfd);
```

6.11fcntl 函数

```
// 复制文件描述符、设置/获取文件的状态标志
int fcntl(int fd, int cmd, ... /* arg */ );
```