Linux 系统编程笔记

郑华

2018 年 6 月 7 日

第一章　基础查漏补缺

1.1　系统编程简介

用程序完成 Linux 命令等干的事情。

1.2　进程组、会话

参考：https://www.cnblogs.com/zengyiwen/p/5755191.html

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 ID 组成　这样说来，一个进程会有如下ID：

- PID ：进程的唯一标识。对于多线程的进程而言，所有线程调用 getpid 函数会返回相同的值。
- PGID ：进程组 ID。每个进程都会有进程组 ID，表示该进程所属的进程组。默认情况下新创建的进程会继承父进程的进程组 ID。
- SID ：会话 ID。每个进程也都有会话 ID。默认情况下，新创建的进程会继承父进程的会话 ID。

查看 ID　->

可以调用如下命令来查看所有进程的层次关系：`ps -ejH` 或 `ps axjf`

可以调用以下**函数**获取进程组 ID 跟会话 ID.`pid_t getpgrp(void);` 或 `pid_t getsid(pid_t pid);`

新进程**默认继承**父进程的*进程组 ID 和会话 ID*，如果都是默认情况的话，那么追根溯源可知，所有的进程应该有共同的进程组 ID 和会话 ID。实际情况并非如此，系统中存在很多不同的会话，每个会话下也有不同的进程组。为何会如此呢？

就像家族企业一样，如果从创业之初，所有家族成员都墨守成规，循规蹈矩，默认情况下，就只会有一个公司、一个部门。但是也有些“叛逆”的子弟，愿意为家族公司开疆拓土，愿意成立新的部门。这些新的部门就是新创建的进程组。如果有子弟“离经叛道”，甚至不愿意呆在家族公司里，他别开天地，另创了一个公司，那这个新公司就是新创建的会话组。由此可见，系统必须要有改变和设置进程组 ID 和会话 ID 的函数接口，否则，系统中只会存在一个会话、一个进程组。

进程组、会话　进程组和会话是为了支持 shell 作业控制而引入的概念。当有新的用户登录 Linux 时，登录进程会为这个用户创建一个会话。用户的登录 shell 就是会话的首进程。会话的首进程 ID 会作为整个会话的 ID。会话是一个或多个进程组的集合，囊括了登录用户的所有活动。

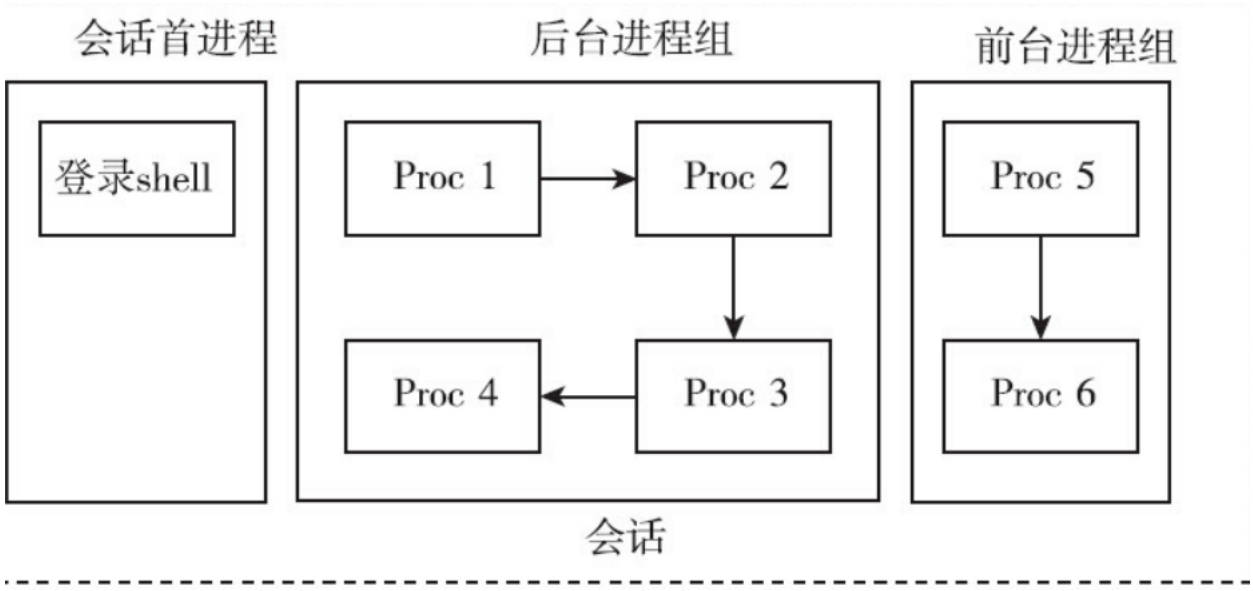


图 1.1: 会话结构

在登录 shell 时，用户可能会使用管道，让多个进程互相配合完成一项工作，这一组进程属于同一个进程组。

最常见的创建进程组的场景就是在 shell 中执行管道命令，代码如下：`cmd1  cmd2 | cmd3|`

下面用一个最简单的命令来说明，其进程之间的关系如图1.2所示。`ps  axgrep nfsd|`

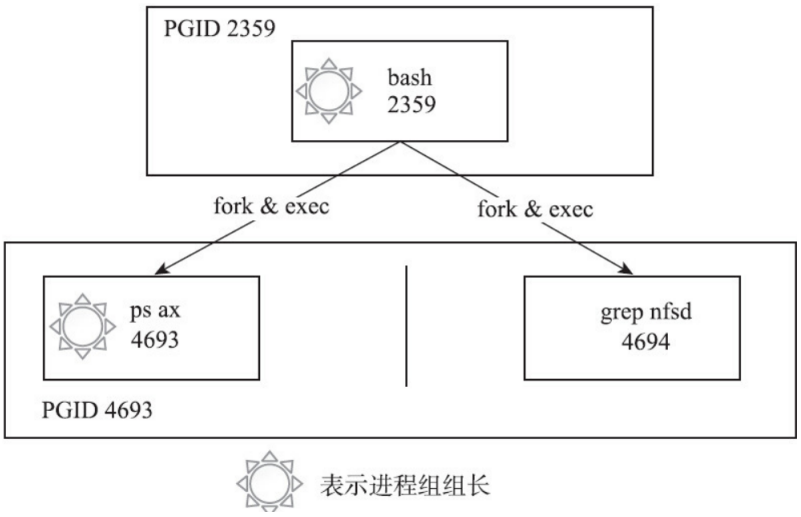


图 1.2: 进程组结构演示

ps 进程和 grep 进程都是 bash 创建的子进程，两者通过管道协同完成一项工作，它们隶属于同一个进程组，其中 ps 进程是进程组的组长。

进程组的概念并不难理解，可以将人与人之间的关系做类比。一起工作的同事，自然比毫不相干的路人更加亲近。shell 中协同工作的进程属于同一个进程组，就如同协同工作的人属于同一个部门一样。

引入了进程组的概念，可以更方便地管理这一组进程了。比如这项工作放弃了，不必向每个进程一一发送信号，可以直接将信号发送给进程组，进程组内的所有进程都会收到该信号。

1.3 errno 与 perror()

使用 perror 函数打印对应 errno 对应的错误信息。

```
int fd = open("english",O_RDWR);
if(fd == -1)
{
    perror("Message");
}

//--> Message: 错误信息
```

char* strerror(int errno) 打印对应的错误信息。

1.4 逻辑地址与物理地址

从逻辑地址到物理地址的映射称为地址重定向。分为：

静态重定向–在程序装入主存时已经完成了逻辑地址到物理地址和变换，在程序执行期间不会再发生改变。

动态重定向–程序执行期间完成，其实现依赖于硬件地址变换机构，如基址寄存器。

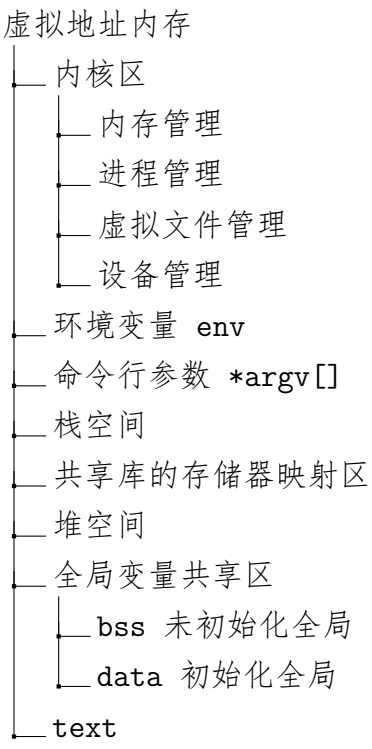
逻辑地址：CPU 所生成的地址。CPU 产生的逻辑地址被分为:p （页号）它包含每个页在物理内存中的基址，用来作为页表的索引；d（页偏移），同基址相结合，用来确定送入内存设备的物理内存地址。

物理地址：内存单元所看到的地址。

用户程序看不见真正的物理地址。用户只生成逻辑地址，且认为进程的地址空间为 0 到 max。物理地址范围从 R+0 到 R+max，R 为基地址, 地址映射－将程序地址空间中使用的逻辑地址变换成内存中的物理地址的过程。由内存管理单元（MMU）来完成。

1.5 UID、EUID

第二章　虚拟地址空间


虚拟地址内存
内核区
内存管理
进程管理
虚拟文件管理
设备管理
环境变量 env
命令行参数 *argv[]
栈空间
共享库的存储器映射区
堆空间
全局变量共享区
bss 未初始化全局
data 初始化全局
text

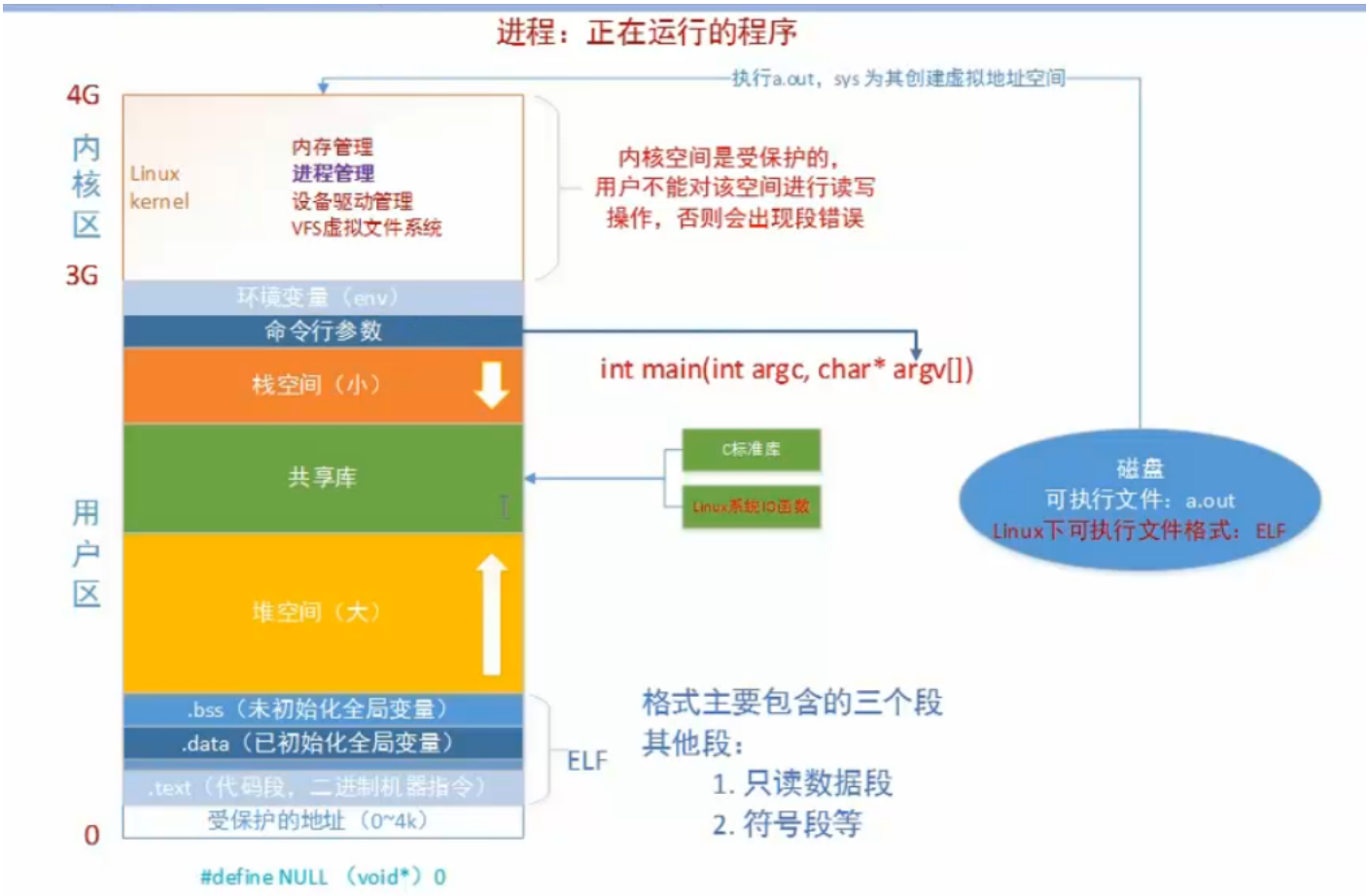

进程：正在运行的程序
执行a.out，sys 为其创建虚拟地址空间
4G
内核区
3G
用户区
0
Linux kernel
内存管理
进程管理
设备驱动管理
VFS虚拟文件系统
内核空间是受保护的，
用户不能对该空间进行读写
操作，否则会出现段错误
环境变量（env）
命令行参数
栈空间（小）
int main(int argc, char* argv[])
共享库
C标准库
Linux系统IO函数
磁盘
可执行文件：a.out
Linux下可执行文件格式：ELF
堆空间（大）
.bss（未初始化全局变量）
.data（已初始化全局变量）
.text（代码段，二进制机器指令）
受保护的地址（0~4k）
ELF
格式主要包含的三个段
其他段：
1. 只读数据段
2. 符号段等
#define NULL （void*）0

图 2.1: 内存地址空间

第三章　文件操作

- 读写操作
  - 读取
    - Open
    - ReadWrite
    - Lseek
  - 文件状态
    - Stat
    - 属性
  - 内容复制
    - dup
    - dup2

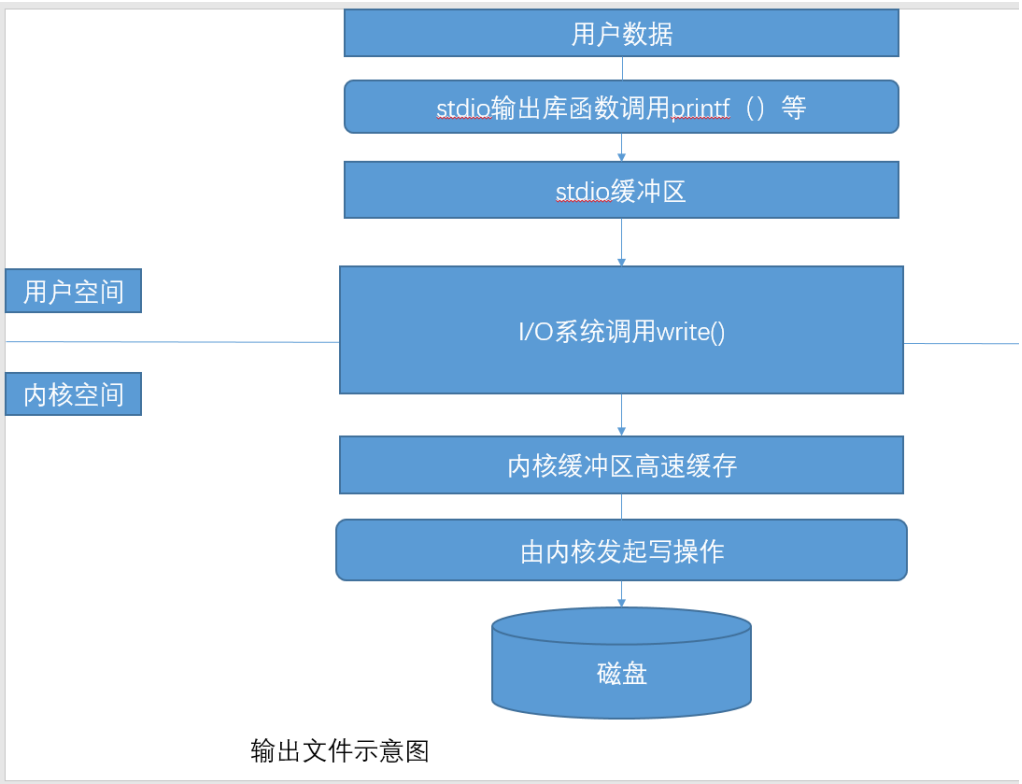


图 3.1: I/O 系统结构

3.1 文件描述符表

每打开一个文件就占用一个文件描述符，系统通过文件描述符找到对应的磁盘进行文件操作。

并且文件描述符的分配遵循：在空闲的文件描述符集合中，选取最小的分配给当前文件

系统占用前三个描述符,并默认打开,分别为stdin（0）:STDIN_FILENO,stdout（1）STDOUT_FILENO，stderr（2）STDERR_FILENO

3.2 阻塞非阻塞

阻塞：到达该函数后，必须执行完才能继续向下执行

非阻塞：到达后，基础处理后面代码，等待消息再处理。

/dev/tty 类似与 c++ 中的 this, 指向当前终端。

要点

- 属于**文件的属性**
- 普通文件**默认不阻塞**
- 终端设备/dev/tty 或 STDOUT_FILENO 或 管道 或 套接字等**默认阻塞**

3.3 读写操作

open() 打开文件 (文件路径，读写选项，默认权限)

函数原型 int open(const char* path, int flags, mode_t mode)

参数说明

- 返回 文件描述符
- path 文件路径
- flag 文件打开方式，读、写、追加、创建等

- `mode` 针对当 flags 中具有创建文件权限时，指定文件权限的参数

示例 ->

```
int fd = open("startUp",O_RDONLY);
if(fd == -1)
    errExit("open");

fd = open("w.log",O_WRONLY | O_CREAT | O_TRUNC | O_APPEN, S_IRUSR | S_IWUSER);
```

close()

函数原型 int close(int fd)

参数说明 fd 文件描述符

lseek() 修改文件的偏移量，文件偏移量是指执行下一个 read() 或 write() 操作的文件起始位置。

函数原型 off_t lseek(int fd, off_t offset, int whence)

参数说明

- `offset` 指定一个以字节为单位的数值，定义偏移量。
- `whence` 表明参照哪个基点来解释 offset 参数[SEEK_SET, SEEK_END)，

  SEEK_SET 文件开始位置

  SEEK_CUR 当前位置

  SEEK_END 文件结束位置的后一个位置

read() 将文件内容读至缓存

函数原型 ssize_t read(int fd, void *buffer, size_t count)

参数说明

- 返回值 返回实际读取的字节数，如果遇到 EOF 返回 0，出现错误返回-1.
- buffer 用来存放输入数据的内存地址
- count 指定最多能读取的字节数

readv()

函数原型

参数说明

write() 将缓冲数据写入文件

函数原型 ssize_t write(int fd, void* buffer, size_t count)

参数说明

- 返回值 返回实际写入文件的字节数
- buffer 数据内存地址
- count 预从 buffer 写入文件的数据字节数

writev() 将缓冲区数据写入文件

函数原型

参数说明

truncate()

函数原型

参数说明

ftruncate()

函数原型

参数说明

ioctl()

函数原型

参数说明

3.4 文件属性

3.4.1 属性操作

图 3.2: 文件属性表示图

文件类型包括如下 7 种：

- S_IFSOCK 套接字
- S_IFLNK 符号链接（软链接）
- S_IFREG 普通文件
- S_IFBLK 块设备
- S_IFDIR 目录
- S_IFCHR 字符设备
- S_IFIFO 管道
- *S_IFMT 类型掩码

stat() 查看文件大小、时间戳、权限权限等基本属性，类似`ls -l fileName`

函数原型 int stat(const char* path, struct stat* buf)

参数说明 输入输出参数：`struct stat*` ，主要包括

- 文件的设备编号
- i 节点
- 文件的类型和存取权限
- 用户 ID
- 组 ID
- 文件大小
- 硬链接计数
- 修改时间等

判断文件的类型示例 st 为 struct stat 结构体，st_mode 表示其中文件的类型和存取权限，如图3.2所示。`if(st.st_mode & S_IFMT == S_IFREG)`

lstat()

函数原型 `int lstat(const char* path, struct stat* buf)`

与 stat 的区别 读取软链接文件的方式不同。

- `stat()` 函数会对软链接文件进行解析，读取的是软链接指向的文件
- `lstat()` 则只是对传进去的文件进行分析，如果是软链接则不继续解析、追踪。

fstat() 操作同 stat(), 区别是第一个参数不同

函数原型 `int fstat(int fd, struct stat* buf)`

3.4.2 权限操作

access() 测试当前用户指定的文件是否具有某种属性。

chmod()

chown()

trucncate() 修改文件大小，传进去的大小比原来小，删掉后面的内容，比原来大，则向后拓展。

函数原型 `int truncate(const char *path, off_t length)`

ftruncate() 同 truncate.

函数原型 `int ftruncate(int fd, off_t length)`

3.4.3 目录操作

rename()

chdir() 修改当前进程的路径，cd

getcwd() 获取当前工作目录路径，pwd

mkdir() 创建目录

rmdir() 删除目录

opendir() 打开一个目录，遍历

readdir()

closedir()

3.5 文件重定向

dup() 复制文件描述符

函数原型 int dup(int oldFd)

参数说明

- `oldFd` 要复制的文件描述符
- `返回值` 返回新的文件描述符，同文件描述符分配过程，取最小的
- `dup 调用成功` 则两个文件描述符指向同一个文件（oldFd）

dup2() 复制文件描述符

函数原型 int dup2(int oldFd, int newFd)

参数说明

- oldFd = dup 中的 oldFd
- newFd = dup 中的返回值
- dup2 调用成功 则两个文件描述符都指向同一个文件（oldFd）

fcntl() 变更文件属性

函数原型 int fcntl(int fd, int cmd, ...)

参数说明 变参主要针对不同的 cmd，体现出不同，cmd 具有如下几种：

- F_DUPFD 复制一个已有文件描述符，同 dup
- F_GETFL 获取文件状态标识`int flag = fcntl(fd, F_GETFL);`
- F_SETFL 设置文件状态`fcntl(fd, F_SETFL, flag);`, 文件已经打开，然后需要修改文件属性。

第四章　线程

4.1　线程创建

pthread_create()

函数原型　pthread_create(pthread_t* id, pthread_attr_t* attr, void*(*start_routine)(void*), void* arg)

参数说明

- `void* (*start_routine) (void*)`: 这里是个陷阱，如果使用成员函数会存在一个参数不匹配问题，因为成员函数有一个隐含的第一参数 this, 而 this 为类指针类型，与 void* 不匹配，造成了 notMatch 错误。
- `*id` 输出参数
- `*attr` 用于设置父子线程分离，完成自我回收

创建时设置属性分离　使得子线程完成自我回收。

- 线程属性类型othread_attr_t attr;
- 对线程属性变量初始化 ptread_attr_init(pthread_attr_t* attr)
- 设置线程分离属性 pthread_attr_setdetachstate(pthread_attr_t* attr, int detachstate)
  - `attr` 线程属性
  - `detachstate` 分离类型:PTHREAD_CREATE_DETACH 非分离类型:..CREATE_JOINABLE
- 释放线程资源函数 pthread_attr_destroy(pthread_attr_t *attr)

创建线程后与原进程的关系 ->

- 创建线程后，地址空间没有变化
- **进程退化成了线程**-主线程
- 创建出的子线程和主线程 **共用地址空间**
- 主线程和子线程 **在内核区有了各自的 PCB**，而子线程的 PCB 是从主线程拷贝过来的，**OS 进程调度的时候是根据 PCB 进行的**

共享、私有内容 除了栈区都共享，因此**线程间通信**可以**使用其他区域的任何变量**来实现。**有几个线程，栈会平均分为几份。**

进程 ID、线程号、线程 ID ->

- **进程 ID** PID- 给用户看的
- **线程号** LWP- 给内核看的ps -Lf _PID, 一般情况 PID 相同，LWP 不同
- **线程 ID** TID- 给用户看的

pthread_self() 查看当前线程的线程 id。

4.2 线程运行

pthread_detach() 线程分离, 调用该函数后，主线程不需要调用othread_join() 完成 pcb 的回收，子线程负责自我回收。

函数原型 int pthread_detach(pthread_t thread)

4.3 线程终止

pthread_exit() 单个线程退出，对子线程无影响，但是 PCB 没有回收

函数原型 void pthread_exit(void *retval) 输出参数：退出信息，此参数必须是全局的

pthread_join() 主线程阻塞等待给定子线程结束，并回收资源（PCB）

函数原型 `void pthread_join(pthread_t thread, void **retval)` 输出内存：退出信息。

4.4 线程同步

第五章　进程

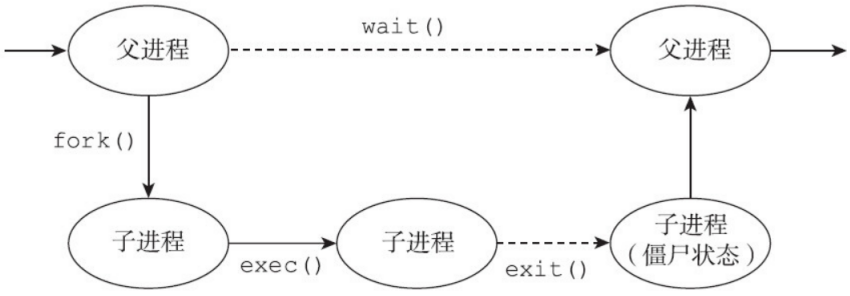


图4-1　进程的生命周期

图 5.1: 进程生存演示

5.1　进程空间

- 进程空间
  - 内核区
    - PCB 进程控制块
  - 用户区
    - 栈
    - 堆
    - bss
    - data
    - text
    - 动态库加载区
    - 环境变量
    - 命令行参数

5.2　进程状态

进程的五种状态，如图5.2所示

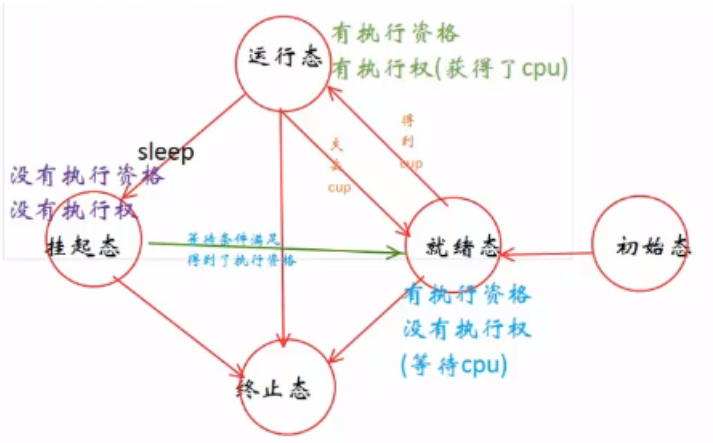


图 5.2: 进程状态

5.3 PCB 进程控制块

位于内核区，名为 struct `task_struct`, 内容主要包括：

- 进程 ID `pid_t`，`_t` 是 `typedef` 使用的简写暗指形式。
- 进程状态 就绪 运行 挂起 终止
- 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
- 描述控制终端信息
- 当前工作目录
- umask 掩码
- 文件描述符表
- 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）`ulimit -a`

5.4 进程创建-父子进程

5.4.1 fork

父子进程执行的操作有一定关系时使用 fork 函数完成子进程的创建。

创建时的动作 ->

fork() 会**产生一个和父进程完全相同的子进程，（进程空间中的用户区 完全复制，但是内核区部分不同，如 PCB 中的进程 ID，但是文件描述符表相同。）**，但子进程在此后多会 exec 系统调用，出于效率考虑，linux 中引入了“写时复制“技术，也就是只有进程空间的各段的内容要发生变化时，才会将父进程的内容复制一份给子进程。在 fork 之后 exec 之前两个进程用的是相同的物理空间（内存区），子进程的代码段、数据段、堆栈都是指向父进程的物理空间，也就是说，**两者的虚拟空间不同，但其对应的物理空间是同一个。当父子进程中有更改相应段的行为发生时，再为子进程相应的段分配物理空间**，如果不是因为 exec，内核会给子进程的数据段、堆栈段分配相应的物理空间（至此两者有各自的进程空间，互不影响），**而代码段继续共享父进程的物理空间**（两者的代码完全相同）。而如果是因为 exec，由于两者执行的代码不同，子进程的代码段也会分配单独的物理空间。

子进程创建成功后代码的执行位置 ->

fork 后，分别进入父子进程中的 fork 函数，然后执行各自的返回操作，但是用户区的内容是相同的。

也可以这样理解，父进程执行到哪，子进程就执行到哪。

因为栈中保存着函数调用栈，因此函数的调用位置也会存在，而复制的是逻辑空间，因此地址的意义也是一样的。即 **fork 时子进程获得父进程数据空间、堆和栈的复制，所以变量的地址、函数的地址（当然是虚拟地址 (相对位置)）也是一样的**。

fork 后父子进程区分 ->

- 父亲进程`pid > 0`
- 孩子进程`pid == 0`

读共享、写复制机制 ->

每个进程都有自己的虚拟地址空间，不同进程的相同的虚拟地址显然可以对应不同的物理地址。因此地址相同（虚拟地址）而值不同没什么奇怪。具体过程是这样的：

fork 子进程完全复制父进程的栈空间，也复制了页表，但没有复制物理页面，所以这时虚拟地址相同，物理地址也相同，但是会把父子共享的页面标记为“只读”（类似 mmap 的 private 的方式），如果父子进程一直对这个页面是同一个页面，知道其中任何一个进程要对共享的页面“写操作”，这时内核会复制一个物理页面给这个进程使用，同时修改页表。而把原来的只读页面标记为“可写”，留给另外一个进程使用。

这就是所谓的“写时复制”。正因为 fork 采用了这种写时复制的机制，所以 fork 出来子进程之后，父子进程哪个先调度呢？内核一般会先调度子进程，因为很多情况下子进程是要马上执行 exec，会清空栈、堆。。这些和父进程共享的空间，加载新的代码段。。。，这就避免了“写时复制”拷贝共享页面的机会。如果父进程先调度很可能写共享页面，会产生“写时复制”的无用功。所以，一般是子进程先调度滴。

假定父进程 malloc 的指针指向 0x12345678, fork 后，子进程中的指针也是指向 0x12345678，但是这两个地址都是虚拟内存地址（virtual memory)，经过内存地址转换后所对应的物理地址是不一样的。所以两个进程中的这两个地址相互之间没有任何关系。

（注 1：在理解时，你可以认为 fork 后，这两个相同的虚拟地址指向的是不同的物理地址，这样方便理解父子进程之间的独立性）（注 2：但实际上，Linux 为了提高 fork 的效率，采用了 copy-on-write 技术，fork 后，这两个虚拟地址实际上指向相同的物理地址（内存页），只有任何一个进程试图修改这个虚拟地址里的内容前，两个虚拟地址才会指向不同的物理地址（新的物理地址的内容从原物理地址中复制得到））

fork Loop ->

当在循环中调用 fork 时，那么在子进程中也会执行相同的操作。但是不同的是，子进程中的初始变量不同，即循环 index 不同。

```
for(int i = 0; i < 3; ++i)
{
    fork();
}
```

执行上面代码是效果如下：

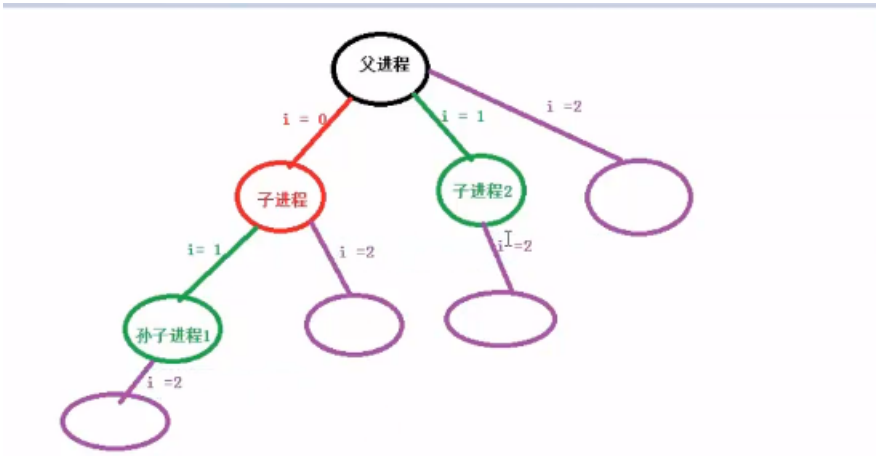


图 5.3: forkLoop

过程如下：执行第一次循环时，`i = 0`, 产生一个子进程，并且复制父进程的用户区，则此时的子进程的`i = 0` 且 fork 执行完成，进入`++i` , 即子进程产生后的`i = 1`.

在执行第 2 次 fork 时，那么父亲进程与第一次产生的子进程都进入相同的循环`i = 1`, 并执行 fork, 此时则分别产生各自的子进程，并拷贝用户区，此时的`i = 1`, 但是 fork 执行完进入`++i`, 即相当于父子进程都同时到达`i = 2`.

以此类推，则产生如上图所示效果。

getpid()、getppid() ->

获取当前进程 ID：`getpid()`

获取当前进程的父进程 ID：`getppid()`

vfork() ->

相同 vfork() 函数的调用序列和返回值与 fork 相同

不同的是 vfork 创建的子进程与父进程共享地址空间，即子进程完全运行在父进程的地址空间上，子进程对虚拟地址空间的修改同样为父进程所见，用 vfork 创建子进程后，父进程会被阻塞到子进程调用 exec 或 exit。

vfork 避免了（fork 函数子进程被创建后，仅仅为调用 exec 执行另一个程序，它对地址空间的复制是多余的）这个问题，减少了不必要的开销。

vfork 保证子进程先运行，它调用 exec 或 exit 后父进程才能调度运行，fork 的父子进程运行顺序不定，取决于内核的调度算法。

5.4.2 exec 函数族

让父子进程执行不相干的操作时使用 exec 函数族**对 fork 创建出的子进程进行更换.text 区**，即更换执行代码、重新填充新的代码，`exec("ls")`, 实现**换核不换壳**的功能。以完成当前子进程调用另外一个应用程序的目的。

执行另外的程序不需要创建额外的地址空间。

execl() 一般执行自己写的程序，让子进程执行指定程序

函数原型 int execl(const char* path, const char *arg, ...)

参数

- `path` 要执行的程序的绝对路径
- `arg` 起到占位的作用，写什么都行
- `...` 要执行的程序需要的参数（变参）
- `NULL` 以 NULL 告诉程序参数输完了，哨兵

execv() 在子进程中更换执行程序，简化`execl`

函数原型 int execv(const char* path, char* const argv[])

参数

- `path = /bin/ps`
- `char* args[] = {"ps","aux",NULL}`
- 示例：`execv("/bin/ps",args);`

execlp() 执行`PATH` 环境变量能够搜索到的程序

函数原型 int execlp(const char* file, const char *arg, ...)

参数

- `file` 要执行的程序名字，如果要执行自定义的程序，使用绝对路径。
- `arg` 起到占位的作用，写什么都行
- `...` 要执行的程序需要的参数（变参）
- `NULL` 以 NULL 告诉程序参数输完了，哨兵

execle() 执行指定路径、指定环境变量下的程序（了解即可）

5.5 进程创建-守护进程

守护进程 管家、仆人、不重要的角色，只做某种单一的事务。

- 后台服务进程
- 独立于控制终端
- 周期性执行某任务
- 不受用户登陆注销影响
- 一般采用以 d 结尾的名字（服务）

进程组组长

- 组里面的**第一个**进程
- 进程组的ID = 进程组的组长的ID

会话 创建一个会话需要注意以下几个方面

- 不能是进程组组长
- 创建会话的进程成为新进程组的组长
- 有些 linux 需要 root 权限执行此操作 (ubuntu 不需要)

- 创建出的新会话会丢弃原有的控制终端
- 一般步骤：先fork()，父进程死，儿子进程执行创建会话操作 (setsid())

守护进程创建步骤

1. fork()
2. 父进程退出 (kill(getpid(),SIGKILL))
3. 子进程创建新会话 (setsid())
4. 改变当前工作目录 (chdir())（不是必须的）
5. 重设文件掩码 (umask())（不是必须的）
6. 关闭文件描述符（close() 0,1,2),释放终端占用STDIN STDOUT STDERR -> /dev/tty (不是必须的)
7. 执行核心工作（必须的）

setsid() 使得普通进程变为会话组会长，完成脱离终端。

5.6 进程回收-父子进程

孤儿进程

- 父亲进程产生子进程
- 父亲进程死亡，孩子进程还活着（活进程，仍能运行），此时孩子进程被称为孤儿进程。
- 孤儿进程被 init 进程领养，init 进程变为孤儿进程的父亲。目的有以下几个方面
    1. 进程结束后，能够释放用户区空间
    2. 释放不了 PCB，必须由父亲进程释放

僵尸进程

- 父亲产生子进程
- 子进程死亡（一个死进程，不能运行），父亲进程存活

- 父亲进程不去（或处于工作中无法）释放子进程的 PCB，子进程变为僵尸进程`stat = Z+、defunct`

如何处理僵尸进程 杀死该僵尸进程的父亲进程即可。因为此时该进程会被 init 进程领养，释放 PCB 工作会有 init 来完成。

wait() 阻塞等待

函数原型 `pid_t wait(int* status)`

参数

- **返回值**

  1. `-1` 回收失败，已经没有子进程了
  2. `>0` 清理成功的子进程**对应的 pid**

- **输入输出参数：status** 用于判断子进程是如何死的

  1. 正常退出：通过`WIFEXITED(status)` 为非0判断，如果此状态为真，则使用`WEXITSTATUS(status)` 获取进程的退出状态。
  2. 被某个信号杀死：通过`WIFSIGNALED(status)` 为非0 判断，如果此状态为真，则使用`WTERMSIG(status)` 获取进程终止的信号的编号。

- **调用一次只能回收一个子进程**

waitpid() options 设置等待方式（阻塞、非阻塞）

函数原型 `pid_t waitpid(pid_t pid, int *status, int options)`

参数

- **返回值**

  1. **>0** 返回清理掉的子进程的 ID
  2. **-1** 无子进程

3. **=0** options 为`WNOHANG` ，且子进程正在运行

- **pid**

1. `pid == -1` 等待任一子进程，与 wait 等效。
2. `pid > 0` 等待其进程 ID 与 PID 相等的子进程
3. `pid == 0` 等待其组 ID 等于调用进程的组 ID 的任一子进程
4. `pid < -1` 等待其组 ID 等于 pid 的绝对值的任一子进程

- **status** 同 wait
- **options**

1. 设置为`WNOHANG` 表示函数非阻塞
2. 设置为 0 表示函数阻塞

5.7 进程终止

exit()

第六章　系统通信

6.1　管道

如 bash 中的 | 符号

pipe()

函数原型　int pipe(int pipeFd[2])

说明

- 在**内核缓冲区**（默认 4k, 可以适当的放大，可以通过`fpathconf(fd[0],_PC_PIPE_BUF)` 查看）**中操作**（伪文件），操作管道的进程被销毁后，管道自动被释放。
- 管道默认是**阻塞的**（伪文件）
- 实现方式：**环形队列**
- 数据**只能够读取一次**
- 半双工，**单向的**
- **适用**于*有血缘关系*的进程
- **读端** 对应文件描述符`Fd[0]`，传出参数，函数返回相应的文件描述保存于此，以便操作。
- **写端** 对应文件描述符`Fd[1]`，同上

ps -aux | grep ”bash” 原理分析

- `ps -aux` 作为管道的输入端、写入端

- grep "bash" 作为管道的读取端
- 而标准输入、输出都是终端文件/dev/tty ，因此在操作的过程中，需要重定向输出至管道写段Fd[1]，重定向输入端至管道的读端Fd[0]，最后恢复即可。

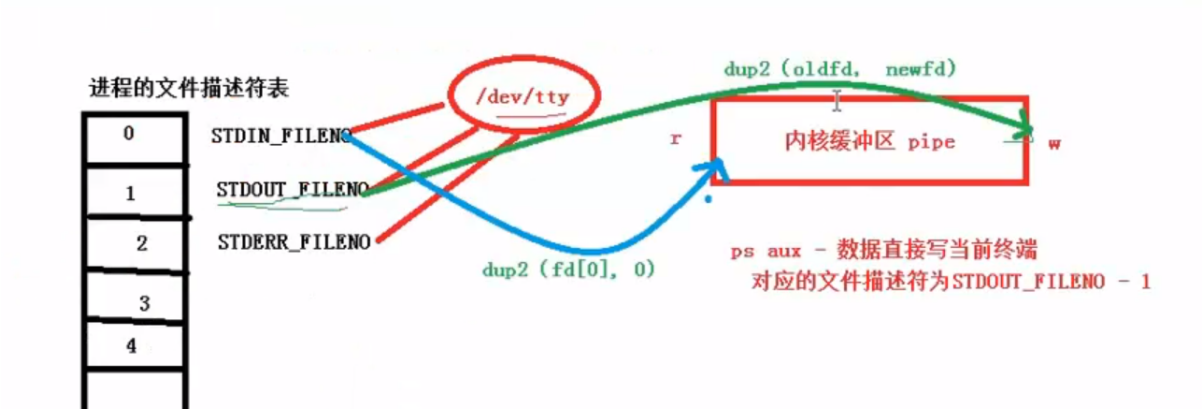


图 6.1: bash 管道原理分析图

fifo() 有名管道

函数原型 | int mkfifo(const char* path, mode_t mode)

说明

- 内核缓冲区、有名管道
- 创建成功后磁盘会多一个文件，但是占用大小为 0.
- 操作管道相当于操作文件
- 半双工、单向
- 没有血缘关系的进程间通信
- path 指定创建出来的伪文件名，用于间接指向管道缓冲区
- 读进程 a 打开path 指定的文件，读取
- 写进程 b 打开path 指定的文件，写

6.2 内存映射

mmap() 创建内存映射，将文件映射到内存区，操作内存以替代操作文件，因此在操作时需要注意同步问题，因为内存不存在阻塞的属性。

函数原型 | void* mmap(void* addr, size_t length, int prot, int flags, int fd, off_t offset)

说明

- 返回值
  1. 成功返回内存指针。
  2. 失败返回MAP_FAILED, 等价于-1。
- *addr 映射区首地址，传NULL
- lenght 映射区大小，4k 倍且不能为 0，一般文件多大，length 就制定多大
- prot 映射区权限，PROT_READ 读权限 PROT_WRITE 写
- flags 标志位参数，
  1. MAP_SHARED 共享 表示修改了内存数据会同步到磁盘
  2. MAP_PRIVATE 私有 修改了内存不会同步到磁盘
- fd 文件描述符, 打开文件时权限必须大于等于 prot
- offset 映射文件的偏移量，必须是 4k(4096) 的整数倍

内存位置与原理：http://kdf5000.com/2017/02/17/mmap%E5%86%85%E5%AD%98%E6%98%A0%E5%B0%84/

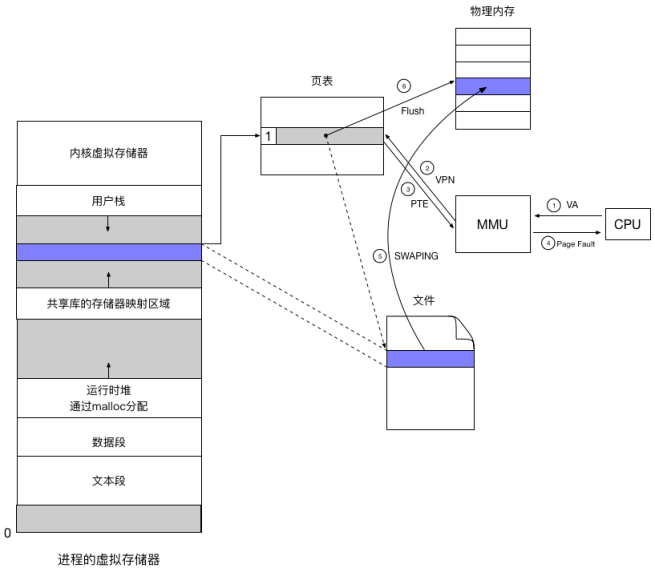


图 6.2: 内存映射原理

munmap() 释放内存映射区

函数原型 int munmap(void* addr, size_t length)

说明

- `*addr` 为`mmap()` 的返回值
- `length` 映射区大小

创建匿名内存映射 确定固定的参数，使得 mmap() 函数创建出匿名的内存映射，具体如下：

```
void *ptr = mmap(NULL, len, PROT_READ|PROT_WRITE, MAP_SHARED | MAP_ANON, -1, 0)
```

主要修改内容有

- `flags` 权限 添加宏`MAP_ANON` 说明是匿名内存映射
- `fd` 此处限定为-1，因为不存在文件了
- `offset` 此处限定为 0，因为不存在文件了

父子进程 A、B 如何使用内存映射通信 首先：**父子进程共享内存映射区**。因为共享映射区已经映射到内存中了，所以该处就是内存地址的复制。

非血缘关系进程 A、B 如何使用内存映射通信 如何通信？

- **不能使用匿名映射**的方式
- **只能借助磁盘文件**创建映射区，假设文件为 hello
- 内存映射区操作是**不阻塞**的

实现 ->

A 进程：

```
int fd = open("hello");
void *ptr = mmap(,,,,fd,0);
// 对映射区操作
strcpy(ptr,"hello␣world!\n");
```

B 进程：

```
int fd2 = open("hello");
void *ptr2 = mmap(,,,,fd2,0);
// 对映射区操作
cout<< *ptr2 <<endl;
```

释放 只有一个进程进行释放内存映射。

6.3 共享内存区

shmget()

6.4 信号

系统开销小

产生、状态、处理方式

产生 主要有 5 种方式，但是需要注意的是，信号是由内核产生发送的：

- **键盘**：ctrl+c 等

- **命令**：`kill`
- **系统函数**：`kill()`
- **软条件**：定时器
- **硬件**：段错误、除 0 错误

状态 主要分为三种方式：

- 产生状态
- 未决状态：信号未被处理
- 递达状态：信号被处理

执行过程 信号的优先级最高，当进程收到信号之后，暂停正在处理的工作，优先处理信号，处理完成之后再继续暂停的工作。

处理方式 在信号的处理方式上，主要有以下 5 种方式：

- `Term` 终止进程
- `Ign` 忽略信号
- `Core` 终止进程，并产生 coreDump 文件用于调试
- `Stop` 暂停进程（挂起）
- `Cont` 恢复进程（恢复执行）

信号帮助 `man 7 signal`

kill() 发送信号给指定进程

函数原型 int kill(pid_t pid, int sig)

参数说明

- `pid` 指定的进程号

- `sig` 指定的信号

raise() 自己给自己发信号

函数原型 int raise(int sig)

参数说明

- `sig` 指定的信号

abort() 给自己发送异常终止信号, 永远不会调用失败

函数原型 void abort()

进程运行时间 ->

`time ./程序` 测试程序运行时间，计算如下。

```
real = 用户 + 内核 + 损耗(文件I/O操作)
```

alarm() 设置定时器（每个进程只有一个定时器），使用的是自然定时法，不受进程状态影响

函数原型 unsigned int alarm(unsigned int seconds)

参数说明

- 当时间到达后，函数发出一个信号`SIGALRM` 给自己的当前进程
- 只能相应一次

setitimer()

函数原型 int setitimer(int which, const struct itimerval *new_value, struct itimerval *old_value)

参数说明

- struct itimerval

```
struct itimerval{
    struct timeval it_interval; // 定时器循环周期
    struct timeval it_value; // 第一次触发定时器的时间
};
```

- struct timeval

```
struct timeval{
    time_t tv_sec; // 秒
    suseconds_t tv_usec; // 微秒
};
```

- which ：定时法则

1. ITIMER_REAL = 用户 + 内核 + 损耗
2. ITIMER_VIRTUAL = 用户
3. ITIMER_PROF = 用户 + 内核

signal() 用于信号捕捉

函数原型 | sighandler_t signal(int signum, sighandler_t handler)

参数说明

- sighandler_t = typedef void(*sighandler_t)(int);
- signum 要处理的信号
- handler 处理函数的函数指针

sigaction() 用于信号捕捉

函数原型 | int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)

参数说明

- signum 要处理的信号
- struct sigaction

```
struct sigaction{
    void (*sa_handler)(int); // 处理函数
    void (*sa_sigaction)(int, siginfo_t*, void*);
    sigset_t sa_mask; // 信号集（内核区），在信号函数执行过程中，临时屏蔽指定的信
        号。函数执行完后，自动解除屏蔽
    int sa_flags; // 处理函数使用sa_handler 时赋值为0，  否则赋值为其他值
    void (*sa_restorer)(void);
}
```

- NULL

信号集相关

未决信号集

阻塞信号集

sigemptyset() 将 set 集合置空

sigaddset() 将信号加入集合

sigfillset() 将所有信号加入 set 集合

sigprocmask() 屏蔽 and 接触信号屏蔽，将自定义信号集设置给阻塞信号集

sigpending() 读取当前进程的未决信号集

6.5 socket

sock()

第七章　图形界面

7.1　GTK-GNOME

第八章　库的使用

http://www.cnblogs.com/52php/p/5681711.html

8.1　介绍

使用 GNU 的工具我们如何在 Linux 下创建自己的程序函数库? 一个“程序函数库”简单的说就是一个文件包含了一些编译好的代码和数据，这些编译好的代码和数据可以在事后供其他的程序使用。程序函数库可以使整个程序更加模块化，更容易重新编译，而且更方便升级。

引用的那些**头文件中的函数**是怎么被执行的呢？这就要牵扯到**链接库**了！

库有两种，一种是 **静态链接库**，一种是 **动态链接库**，不管是哪一种库，要使用它们，都要在程序中包含相应的 *include* 头文件。编译过程如下：

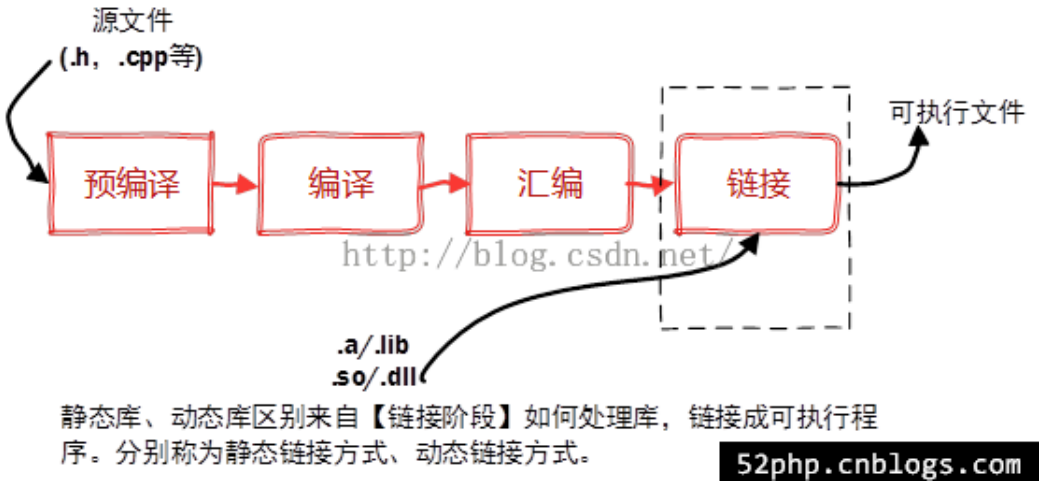


图 8.1: 编译过程

`gcc -o helloWorld helloWorld.c` 生成一个 helloWorld 的执行文件，格式为 ELF(与 widows 的 exe 类似)。

8.2 静态函数库

是在程序执行前就加入到目标程序中去了；

静态函数库实际上就是简单的一个普通的目标文件的集合，一般来说习惯用“.a”作为文件的**后缀**

静态库函数允许程序员把程序 link 起来而不用重新编译代码，节省了重新编译代码的时间。

如你想把自己提供的函数给别人使用，但是又想对函数的源代码进行保密，你就可以给别人提供一个静态函数库文件。理论上说，使用 ELF 格式的静态库函数生成的代码可以比使用共享函数库（或者动态函数库）的程序**运行速度上快一些，大概 1 − 5%，但是占用空间却大了很多。**

Example ->

定义一个加法函数，做成静态库, 首先需要将声明放到头文件以便引用。

```
// add.h
#ifndef _ADD_
#define _ADD_
#include <iostream>

int add(int a, int b);
#endif
```

实现该函数

```
#include "add.h"

int add(int a, int b)
{
   return a+b;
}
```

生成静态库：

1. g++ -c add.cc 生成.o 文件
2. ar -crv libadd.a add.o 生成一个静态库

测试：

```
//目录结构
project
```

```
    |
    +---Main.cc
    |
    +---addlib
           |
           +----add.h
           |
           +----add.cc
```

```
#include <iostream>
#include "./addlib/add.h"

using namespace std;
int main()
{
    int num1 = 10;
    int num2 = 90;
    cout <<"the␣result␣is"<< add(num1,num2)<<endl;
    return 0;
}
```

不管是哪一种库，要使用它们，都要在程序中包含相应的 *include* 头文件, 所以 include 使用相对路径找到头文件add.h

然后我们使用g++ -o Test Main.cc -L ./addlib/ -ladd 进行编译

- L 是指定加载库文件的路径
- l 是指定加载的库文件

静态库搜索路径 ->

1. 编译目标代码时指定的静态库搜索路径。-L ./src/ -lpthread
2. 环境变量LIBRARY_PATH 指定的动态库搜索路径。

   export LIBRARY_PATH=/opt/lib:$LIBRARY_PATH

3. 配置文件/etc/ld.so.conf 中指定的动态库搜索路径, 然后运行/sbin/ldconfig 刷新缓存
4. 默认的动态库搜索路径/lib，/usr/lib

8.3 动态函数库

静态链接的话，文件会很大，往往实现很小的一个功能就需要占用很大的空间，而且每次库文件升级的话，都要重新编译源文件，很不方便。如下：

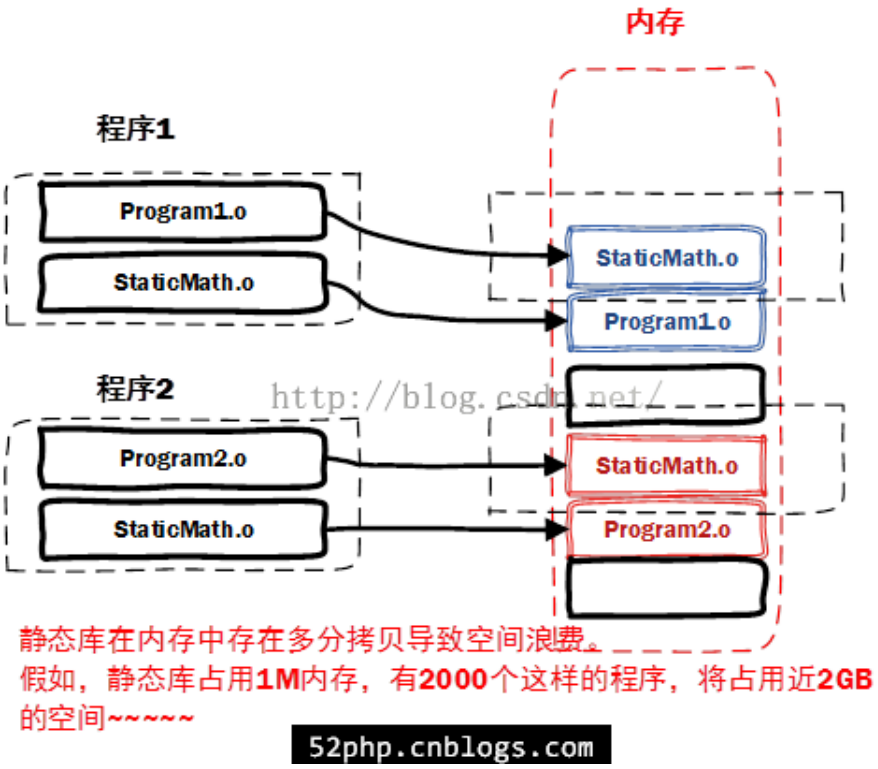


图 8.2: 静态库冗余空间

对于静态编译的程序 1 和程序 2，都应用库 staticMath。在内存中就又**两份相同的 static Math 目标文件，很浪费空间**，一旦程序数量过多就很可能会内存不足。

这么大的内存才只能运行这几个程序，实在不甘心。这样就又了动态库发挥威力的地方了。我们来看看动态链接的结果：

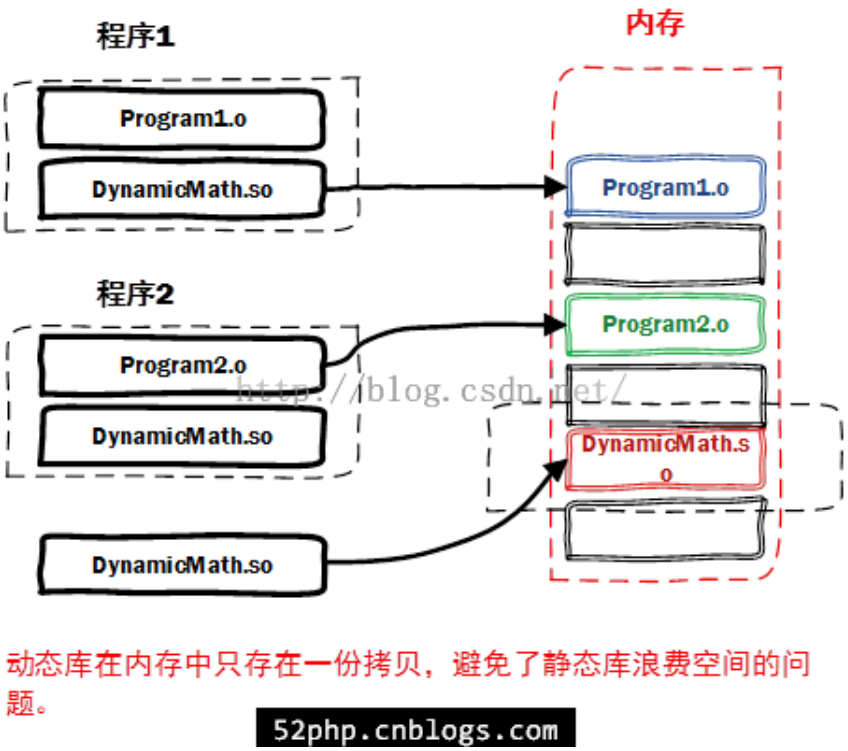


图 8.3: 动态库解决静态库冗余空间问题

在这种模型中，两个程序只应用一个库，这个目标文件在内存中只有一份，供所有程序使用。并且在程序运行过程中动态调用库文件，**很方便，又不占空间**，但是动态链接有一个**缺点就是可移植性太差**，如果两台电脑运行环境不同，动态库存放的位置不一样，很可能导致程序运行失败。

Example 使用静态库代码结构与代码

生成一个`libadd.so` 的动态库 `g++ -fPIC -shared -o libadd.so add.cpp`。这样就生成一个了一个 libadd.so 的动态库。

使用如下命令进行编译:`g++ -o Test Main.cc -L ./addlib/ -ladd -Wl,-rpath=./addlib/`

但是如果不指定运行时搜索路径的话，会出现 cannot open shared obj 错误。如下编译：

```
g++ g++ -o Test Main.cc -L ./addlib/ -ladd
```

此时可以通过`ldd ./Test` 进行查看调用的动态库情况。具体的设置可参考动态库搜索路径的 4 种方法。

动态库搜索路径 ->

http://blog.csdn.net/weicao1990/article/details/51028335 https://www.cnblogs.com/cute/archive/2011/02/24/1963957.html

1. 编译目标代码时指定的动态库搜索路径。-L./src/ -ladd -Wl,-rpath=./src/

前面两个-L -l 分别指明了动态库的位置和库的名字用于编译，只有引用，后面的-Wl,-rpath 则指明运行时到哪里运行, 有内容。

2. 环境变量LD_LIBRARY_PATH 指定的动态库搜索路径。

```
export LD_LIBRARY_PATH=/opt/lib:$LD_LIBRARY_PATH
```

3. 配置文件/etc/ld.so.conf 中指定的动态库搜索路径，然后运行/sbin/ldconfig 刷新缓存

4. 默认的动态库搜索路径/lib，/usr/lib，mv ./src/xx.lib /lib

第九章　工具

9.1　gdb

9.2　CMake