Linux 系统编程笔记

郑华

2018 年 5 月 13 日

第一章　Linux 系统编程

1.1　虚拟地址空间

1.2　线程

pthread_create　`pthread_create(pthread_t* id, pthread_attr_t* attr, void*(*start_rout`

- `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。

1.3　进程

1.3.1　进程空间

http://www.cnblogs.com/xzzzh/p/6596982.html

http://www.cnblogs.com/smile267/archive/2012/10/21/2732099.html

1.3.2　uid、euid

1.3.3　fork

fork 之后子进程到底复制了父进程什么? http://blog.csdn.net/xy010902100449/article/details/44851453

这里就涉及到物理地址和逻辑地址（或称虚拟地址）的概念。

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。

fork （）会产生一个和父进程完全相同的子进程，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。

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（当然是虚拟地址 (相对位置)）也是一样的。

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））

1.4 共享内存区

1.5 消息队列

1.6 信号量

1.7 库的使用

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

1.7.1 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：

`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)。

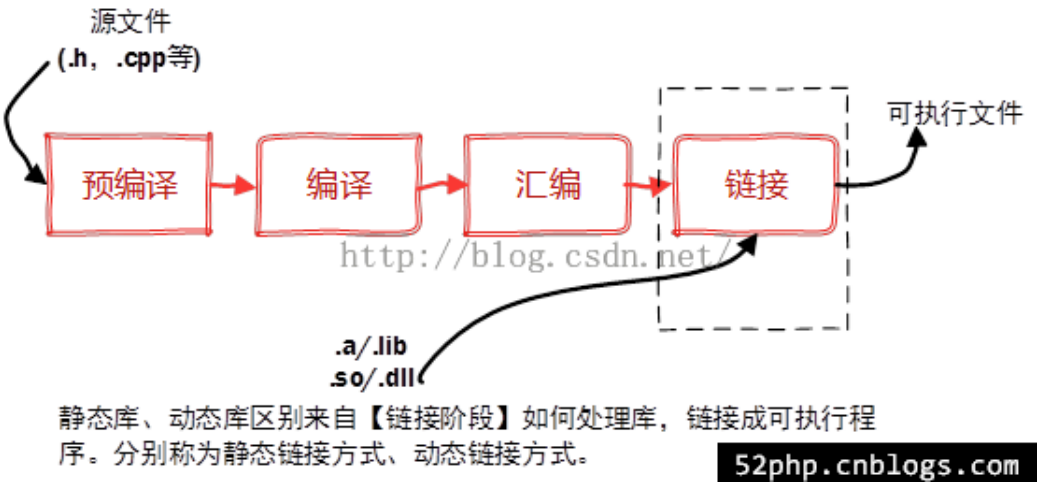

图 1.1: 编译过程

1.7.2 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

1.7.3 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：

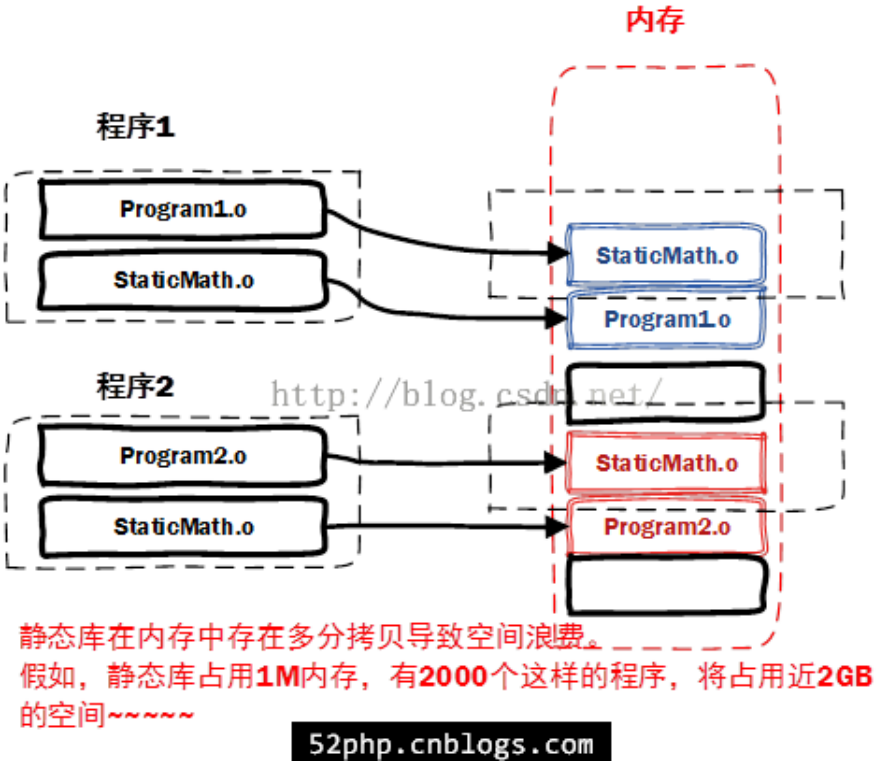

图 1.2: 静态库冗余空间

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 M ath 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：

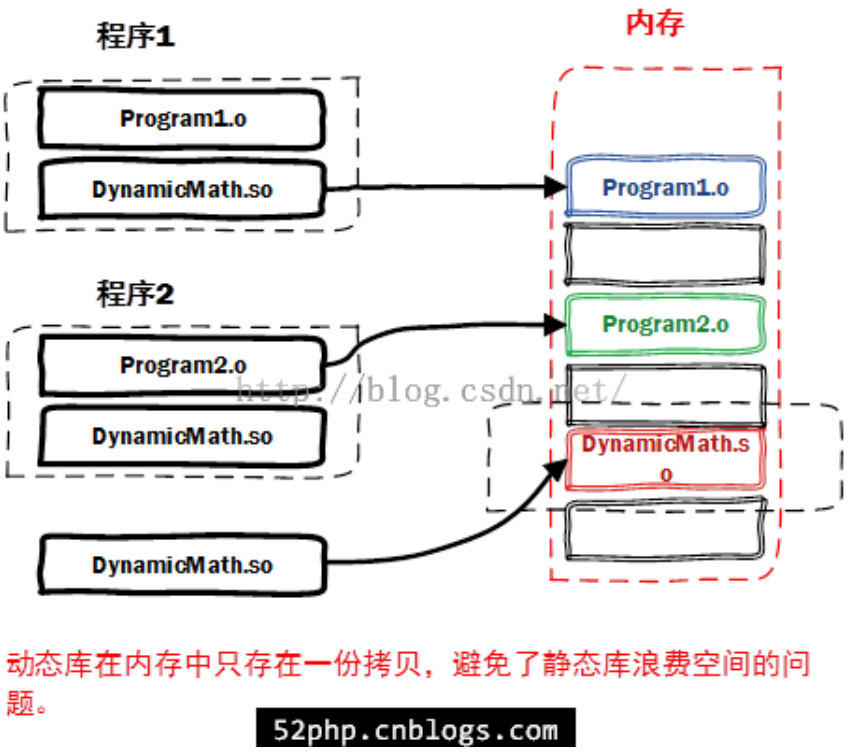

图 1.3: 动态库解决静态库冗余空间问题

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