# 300、Linux的Hello world

## 一、快速入门Linux

用两天的时间学习Linux的基础知识，视频在《系统化学习C++》中。

https://www.bilibili.com/cheese/play/ss4503

## 二、安装软件包

### 1）安装C和C++的编译器

yum -y install gcc\*

### 2）升级编译器

升级软件包：

yum -y install centos-release-scl devtoolset-8-gcc\*

启用软件包：

echo "source /opt/rh/devtoolset-8/enable" >>/etc/profile

*# 每次启动shell的时候，会执行/etc/profile脚本。*

或

mv /usr/bin/gcc /usr/bin/gcc-4.8.5

ln -s /opt/rh/devtoolset-8/root/bin/gcc /usr/bin/gcc

mv /usr/bin/g++ /usr/bin/g++-4.8.5

ln -s /opt/rh/devtoolset-8/root/bin/g++ /usr/bin/g++

### 3）安装库函数的帮助文档

yum -y install man-pages

帮助文档的使用

man 级别 命令或函数

显示帮助的界面可以用vi的命令，q退出。

**man的级别：**

**1-用户命令**；2-系统接口；**3-库函数**；4-特殊文件，比如设备文件；5-文件；

6-游戏；7-系统的软件包；8-系统管理命令；9-内核。

## 三、编译

gcc/g++ 选项 源代码文件1 源代码文件2 源代码文件n

常用选项：

-o 指定输出的文件名，这个名称不能和源文件同名。如果不给出这个选项，则生成可执行文件a.out。

-g 如果想对源代码进行调试，必须加入这个选项。

-On 在编译、链接过程中进行优化处理，生成的[可执行程序](https://baike.baidu.com/item/%E5%8F%AF%E6%89%A7%E8%A1%8C%E6%96%87%E4%BB%B6?fromModule=lemma_inlink)效率将更高。

-c 只[编译](https://baike.baidu.com/item/%E7%BC%96%E8%AF%91?fromModule=lemma_inlink)，不链接成为[可执行文件](https://baike.baidu.com/item/%E5%8F%AF%E6%89%A7%E8%A1%8C%E6%96%87%E4%BB%B6?fromModule=lemma_inlink)，通常用于把源文件编译成静态库或动态库。

-std=c++11 支持C++11标准。

优化选项：

-O0： 不做任何优化，这是默认的编译选项。

-O或-O1： 对程序做部分编译优化，对于大函数，优化编译占用稍微多的时间和相当大的内存。使用本项优化，编译器会尝试减小生成代码的尺寸，以及缩短执行时间，但并不执行需要占用大量编译时间的优化。

-O2： **这是推荐的优化等级**。与O1比较而言，O2优化增加了编译时间的基础上，提高了生成代码的执行效率。

-O3： 这是最高最危险的优化等级。用这个选项会延长编译代码的时间，并且在使用gcc4.x的系统里不应全局启用。自从3.x版本以来gcc的行为已经有了极大地改变。在3.x，-O3生成的代码也只是比-O2快一点点而已，而gcc4.x中还未必更快。用-O3来编译所有的软件包将产生更大体积更耗[内存](https://so.csdn.net/so/search?q=%E5%86%85%E5%AD%98&spm=1001.2101.3001.7020)的二进制文件，大大增加编译失败的机会或不可预知的程序行为（包括错误）。这样做将得不偿失，记住过犹不及。在gcc 4.x.中使用-O3是不推荐的。

如果使用了优化选项：1）编译的时间将更长；2）目标程序不可调试；3）有效果，但是不可能显著提升程序的性能。

# 301、静态库和动态库

在实际开发中，我们把通用的函数和类分文件编写，称之为库。在其它的程序中，可以使用库中的函数和类。

一般来说，通用的函数和类不提供源代码文件（安全性、商业机密），而是编译成二进制文件。

库的二进制文件有两种：静态库和动态库。

## 一、静态库

### 1）制作静态库

g++ -c -o lib库名.a 源代码文件清单

### 2）使用静态库

不规范的做法：

g++ 选项 源代码文件名清单 静态库文件名

规范的做法：

g++ 选项 源代码文件名清单 -l库名 -L库文件所在的目录名

### 3）静态库的概念

程序在编译时会把库文件的二进制代码链接到目标程序中，这种方式称为静态链接。

如果多个程序中用到了同一静态库中的函数或类，就会存在多份拷贝。

### 4）静态库的特点

* 静态库的链接是在编译时期完成的，执行的时候代码加载速度快。
* 目标程序的可执行文件比较大，浪费空间。
* 程序的更新和发布不方便，如果某一个静态库更新了，所有使用它的程序都需要重新编译。

## 二、动态库

### 1）制作动态库

g++ -fPIC -shared -o lib库名.so 源代码文件清单

### 2）使用动态库

不规范的做法：

g++ 选项 源代码文件名清单 动态库文件名

规范的做法：

g++ 选项 源代码文件名清单 -l库名 -L库文件所在的目录名

运行可执行程序的时候，需要提前设置LD\_LIBRARY\_PATH环境变量。

### 3）动态库的概念

程序在编译时不会把库文件的二进制代码链接到目标程序中，而是在运行时候才被载入。

如果多个进程中用到了同一动态库中的函数或类，那么在内存中只有一份，避免了空间浪费问题。

### 4）动态库的特点

* 程序在运行的过程中，需要用到动态库的时候才把动态库的二进制代码载入内存。
* 可以实现进程之间的代码共享，因此动态库也称为共享库。
* 程序升级比较简单，不需要重新编译程序，只需要更新动态库就行了。

## 三、优先使用动态库

如果动态库和静态库同时存在，编译器将优先使用动态库。

## 四、源代码

### 1）tools/public.h

// 通用函数和类的头文件。

#include <iostream>

// 声明一个通用的函数。

void func();

// 声明一个通用的类。

class AA

{

public:

void show();

};

### 2）tools/public.cpp

// 通用函数和类的代码实现文件。

#include "public.h"

using namespace std;

// 通用函数的代码实现。

void func()

{

cout << "升级了调用了func()函数。\n";

}

// 通用类的代码实现。

void AA::show()

{

cout << "升级了我是一只傻傻鸟。\n";

}

### 3）app/demo01.cpp

#include "/home/wucz/tools/public.h" // 包含通用函数和类的头文件。

using namespace std;

int main()

{

func(); // 调用通用的函数。

AA a; // 用通用类声明对象。

a.show(); // 调用对象的方法。

}

# 302、makefile

在实际开发中，项目的源代码文件比较多，按类型、功能、模块分别存放在不同的目录和文件中，哪些文件需要先编译，那些文件后编译，那些文件需要重新编译，还有更多更复杂的操作。

make是一个强大的实用工具，用于管理项目的编译和链接。make需要一个编译规则文件makefile，可实现自动化编译。

## 一、app/demo01.cpp

#include "public.h" // 包含通用函数和类的头文件。

#include "myapi.h" // 包含另一个通用函数和类的头文件。

using namespace std;

int main()

{

func(); // 调用通用的函数。

func1(); // 调用另一个通用的函数。

AA a; // 用通用类声明对象。

a.show(); // 调用对象的方法。

BB b; // 用另一个通用类声明对象。

b.show(); // 调用对象的方法。

}

## 二、app/makefile

INCLUDEDIR=-I/home/wucz/tools -I/home/wucz/api

LIBDIR=-L/home/wucz/tools -L/home/wucz/api

all:demo01 demo02 demo03

demo01:demo01.cpp

g++ -o demo01 demo01.cpp $(INCLUDEDIR) $(LIBDIR) -lpublic -lmyapi

cp demo01 /tmp/.

demo02:demo02.cpp

g++ -o demo02 demo02.cpp $(INCLUDEDIR) $(LIBDIR) -lpublic -lmyapi

demo03:demo03.cpp

g++ -o demo03 demo03.cpp $(INCLUDEDIR) $(LIBDIR) -lpublic -lmyapi

clean:

rm -f demo01 demo02 demo03

## 三、tools/public.h

// 通用函数和类的头文件。

#include <iostream>

// 声明一个通用的函数。

void func();

// 声明一个通用的类。

class AA

{

public:

void show();

};

## 四、tools/public.cpp

// 通用函数和类的代码实现文件。

#include "public.h"

using namespace std;

// 通用函数的代码实现。

void func()

{

cout << "升级了调用了func()函数。\n";

}

// 通用类的代码实现。

void AA::show()

{

cout << "升级了我是一只傻傻鸟。\n";

}

## 五、tools/makefile

# 指定编译的目标文件是libpublic.a和libpublic.so

all:libpublic.a \

libpublic.so

# 编译libpublic.a需要依赖public.h和public.cpp

# 如果被依赖文件内容发生了变化，将重新编译libpublic.a

libpublic.a:public.h public.cpp

g++ -c -o libpublic.a public.cpp

libpublic.so:public.h public.cpp

g++ -fPIC -shared -o libpublic.so public.cpp

# clean用于清理编译目标文件，仅在make clean才会执行。

clean:

rm -f libpublic.a libpublic.so

## 六、api/myapi.h

// 另一个通用函数和类的头文件。

#include <iostream>

// 声明一个通用的函数。

void func1();

// 声明一个通用的类。

class BB

{

public:

void show();

};

## 七、api/myapi.cpp

// 另一个通用函数和类的代码实现文件。

#include "myapi.h"

using namespace std;

// 通用函数的代码实现。

void func1()

{

cout << "调用了func1()函数。\n";

}

// 通用类的代码实现。

void BB::show()

{

cout << "你是一只傻傻鸟。\n";

}

## 八、api/makefile

# 指定编译的目标文件是libmyapi.a和libmyapi.so

all:libmyapi.a libmyapi.so

# 编译libmyapi.a需要依赖myapi.h和myapi.cpp

# 如果被依赖文件内容发生了变化，将重新编译libmyapi.a

libmyapi.a:myapi.h myapi.cpp

g++ -c -o libmyapi.a myapi.cpp

libmyapi.so:myapi.h myapi.cpp

g++ -fPIC -shared -o libmyapi.so myapi.cpp

# clean用于清理编译目标文件，仅在make clean才会执行。

clean:

rm -f libmyapi.a libmyapi.so

# 303、main函数的参数

## 一、main函数的参数

main函数有三个参数，argc、argv和envp，它的标准写法如下：

int main(int argc,char \*argv[],char \*envp[])

{

return 0;

}

argc 存放了程序参数的个数，包括程序本身。

argv 字符串的数组，存放了每个参数的值，包括程序本身。

envp 字符串的数组，存放了环境变量，数组的最后一个元素是空。

在程序中，如果不关心main()函数的参数，可以省略不写。

## 二、操作环境变量

### 1）设置环境变量

int setenv(const char \*name, const char \*value, int overwrite);

name 环境变量名。

value 环境变量的值。

overwrite 0-如果环境不存在，增加新的环境变量，如果环境变量已存在，不替换其值；非0-如果环境不存在，增加新的环境变量，如果环境变量已存在，替换其值。

返回值：0-成功；-1-失败（失败的情况极少见）。

注意：此函数设置的环境变量只对本进程有效，不会影响shell的[环境变量](https://baike.baidu.com/item/%E7%8E%AF%E5%A2%83%E5%8F%98%E9%87%8F?fromModule=lemma_inlink)。如果在运行程序时执行了setenv()函数，进程终止后再次运行该程序，上次的设置是无效的。

### 2）获取环境变量的值

char \*getenv(const char \*name);

## 三、示例

#include <iostream>

using namespace std;

int main(int argc,char \*argv[],char \*envp[])

{

if (argc!=4)

{

cout << "表白神器程序的使用方法：./demo 追求者姓名 被追求者姓名 表白内容\n";

return -1;

}

cout << argv[1] << "开始向" << argv[2] << "表白。\n";

cout << argv[3] << endl;

cout << argv[1] << "表白完成。\n";

return 0;

cout << "一共有" << argc << "个参数。\n";

// 显示全部的参数。

for (int ii=0;ii<argc;ii++)

{

cout << "第" << ii << "个参数：" << argv[ii] << endl;

}

// 显示全部的环境变量。

for (int ii=0;envp[ii]!=0;ii++) // 环境变量数组最后一个元素是0。

{

cout << envp[ii] << endl;

}

// 设置环境变量AA。

setenv("AA","aaaa",0);

// 显示环境变量AA的值。

cout << "AA=" << getenv("AA") << endl;

return 0;

}

# 304、gdb的常用命令

如果程序有问题，不要问别人为什么会这样，而是立即动手调试。

## 一、安装gdb

yum -y install gdb

## 二、gdb常用命令

如果希望程序可调试，编译时需要加-g选项，并且，不能使用-O的优化选项。

gdb 目标程序

|  |  |  |
| --- | --- | --- |
| **命令** | **简写** | **命令说明** |
| set args |  | 设置程序运行的参数。  例如：./demo 张三 西施 我是一只傻傻鸟  设置参数的方法是：  set args 张三 西施 我是一只傻傻鸟 |
| break | b | 设置断点，b 20 表示在第20行设置断点，可以设置多个断点。 |
| run | r | 开始运行程序, 程序运行到断点的位置会停下来，如果没有遇到断点，程序一直运行下去。 |
| next | n | 执行当前行语句，如果该语句为函数调用，不会进入函数内部。 VS的F10 |
| step | s | 执行当前行语句，如果该语句为函数调用，则进入函数内部。VS的F11  注意了，如果函数是库函数或第三方提供的函数，用s也是进不去的，因为没有源代码，如果是自定义的函数，只要有源码就可以进去。 |
| print | p | 显示变量或表达式的值，如果p后面是表达式，会执行这个表达式。 |
| continue | c | 继续运行程序，遇到下一个断点停止，如果没有遇到断点，程序将一直运行。  VS的F5 |
| set var |  | 设置变量的值。  假设程序中定义了两个变量：  int ii;  char name[21];  set var ii=10 把ii的值设置为10；  set var name="西施"。 |
| quit | q | 退出gdb。 |

注意：在gdb中，用上下光标键可以选择执行过的gdb命令。

## 三、示例

#include <iostream>

using namespace std;

void show(const char \*name1,const char \*name2,const char \*message)

{

cout << name1 << "开始表白。\n";

cout << name2 << "：" << message << endl;

}

int main(int argc,char \*argv[],char \*envp[])

{

if (argc!=4)

{

cout << "表白神器程序的使用方法：./demo 追求者姓名 被追求者姓名 表白内容\n"; return -1;

}

cout << "表白前的准备工作一。\n";

cout << "表白前的准备工作二。\n";

cout << "表白前的准备工作三。\n";

cout << "表白前的准备工作四。\n";

cout << "表白前的准备工作五。\n";

show(argv[1],argv[2],argv[3]);

cout << "表白完成。\n";

for (int ii=0;ii<10;ii++)

{

string str="这是第"+to\_string(ii)+"个超级女生。";

cout << str << endl;

}

return 0;

}

# 305、gdb调试core文件

如果程序在运行的过程中发生了内存泄漏，会被内核强行终止，提示“段错误（吐核）”，内存的状态将保存在core文件中，方便程序员进一步分析。

Linux缺省不会生成core文件，需要修改系统参数。

调试core文件的步骤如下：

1）用ulimit -a查看当前用户的资源限制参数；

2）用ulimit -c unlimited把core file size改为unlimited；

3）运行程序，产生core文件；

4）运行gdb 程序名 core文件名；

5）在gdb中，用bt查看函数调用栈。

示例：

#include <cstring>

#include <iostream>

using namespace std;

void bb(const int bh,const string xm)

{

char \*ptr=nullptr;

\*ptr=3;

//strcpy(ptr,xm.c\_str());

}

void aa(const int no,const string name)

{

bb(3,"冰冰");

}

int main()

{

aa(8,"西施");

return 0;

}

# 306、gdb调试正在运行中的程序

gdb 程序名 -p 进程编号

示例：

#include <unistd.h>

#include <iostream>

using namespace std;

void bb(const int bh,const string xm)

{

for (int ii=0;ii<1000000;ii++)

{

sleep(1);

cout << "ii=" << ii << endl;

}

}

void aa(const int no,const string name)

{

bb(3,"冰冰");

}

int main()

{

aa(8,"西施");

return 0;

}

# 307、Linux的时间操作

UNIX操作系统根据计算机产生的年代把1970年1月1日作为UNIX的纪元时间，1970年1月1日是时间的中间点，将从1970年1月1日起经过的秒数用一个整数存放。

## 一、time\_t别名

time\_t用于表示时间类型，它是一个long类型的别名，在<time.h>文件中定义，表示从1970年1月1日0时0分0秒到现在的秒数。

typedef long time\_t;

## 二、time()库函数

time()库函数用于获取操作系统的当前时间。

包含头文件：<time.h>

声明：

time\_t time(time\_t \*tloc);

有两种调用方法：

time\_t now=time(0);     // 将空地址传递给time()函数，并将time()返回值赋给变量now。

或

time\_t now; time(&now);   // 将变量now的地址作为参数传递给time()函数。

## 三、tm结构体

time\_t是一个长整数，不符合人类的使用习惯，需要转换成tm结构体，tm结构体在<time.h>中声明，如下： 2022-10-01 15:30:25 Oct 1,2022 15:30:25

struct tm

{

  int tm\_year; // 年份：其值等于实际年份减去1900

  int tm\_mon; // 月份：取值区间为[0,11]，其中0代表一月，11代表12月

  int tm\_mday; // 日期：一个月中的日期，取值区间为[1,31]

  int tm\_hour;  // 时：取值区间为[0,23]

  int tm\_min; // 分：取值区间为[0,59]

  int tm\_sec;      // 秒：取值区间为[0,59]

  int tm\_wday; // 星期：取值区间为[0,6]，其中0代表星期天，6代表星期六

  int tm\_yday; // 从每年的1月1日开始算起的天数：取值区间为[0,365]

  int tm\_isdst;   // 夏令时标识符，该字段意义不大

};

## 四、localtime()库函数

localtime()函数用于把time\_t表示的时间转换为tm结构体表示的时间。

localtime()函数不是线程安全的，localtime\_r()是线程安全的。

包含头文件：<time.h>

函数声明：

struct tm \*localtime(const time\_t \*timep);

struct tm \*localtime\_r(const time\_t \*timep, struct tm \*result);

示例：

#include <iostream>

#include <time.h> // 时间操作的头文件。

using namespace std;

int main()

{

time\_t now=time(0); // 获取当前时间，存放在now中。

cout << "now=" << now << endl; // 显示当前时间，1970年1月1日到现在的秒数。

tm tmnow;

localtime\_r(&now,&tmnow); // 把整数的时间转换成tm结构体。

// 根据tm结构体拼接成中国人习惯的字符串格式。

string stime = to\_string(tmnow.tm\_year+1900)+"-"

+ to\_string(tmnow.tm\_mon+1)+"-"

+ to\_string(tmnow.tm\_mday)+" "

+ to\_string(tmnow.tm\_hour)+":"

+ to\_string(tmnow.tm\_min)+":"

+ to\_string(tmnow.tm\_sec);

cout << "stime=" << stime << endl;

}

## 五、mktime()库函数

mktime()函数的功能与localtime()函数相反，用于把tm结构体时间转换为time\_t时间。

包含头文件：<time.h>

函数声明：

time\_t mktime(struct tm \*tm);

该函数主要用于时间的运算，例如：把2022-03-01 00:00:25加30分钟。

思路：1）解析字符串格式的时间，转换成tm结构体；2）用mktime()函数把tm结构体转换成time\_t时间；3）把time\_t时间加30\*60秒；4）用localtime\_r()函数把time\_t时间转换成tm结构体；5）把tm结构体转换成字符串。

## 六、gettimeofday()库函数

用于获取1970年1月1日到现在的秒和当前秒中已逝去的微秒数，可以用于程序的计时。

包含头文件：<sys/time.h>

函数声明：

int gettimeofday(struct timeval \*tv, struct timezone \*tz);

struct timeval {

time\_t tv\_sec; /\* 1970-1-1到现在的秒数 \*/

suseconds\_t tv\_usec; /\* 当前秒中，已逝去的微秒数 \*/

};

struct timezone { /\* 在实际开发中，派不上用场 \*/

int tz\_minuteswest; /\* minutes west of Greenwich \*/

int tz\_dsttime; /\* type of DST correction \*/

};

示例：

#include <iostream>

#include <sys/time.h> // gettimeofday()需要的头文件。

using namespace std;

int main()

{

timeval start,end;

gettimeofday(&start, 0 ); // 计时开始。

for (int ii=0;ii<1000000000;ii++)

;

gettimeofday(&end, 0 ); // 计时结束。

// 计算消耗的时长。

timeval tv;

tv.tv\_usec=end.tv\_usec-start.tv\_usec;

tv.tv\_sec=end.tv\_sec-start.tv\_sec;

if (tv.tv\_usec<0)

{

tv.tv\_usec=1000000-tv.tv\_usec;

tv.tv\_sec--;

}

cout << "耗时：" << tv.tv\_sec << "秒和" << tv.tv\_usec << "微秒。\n";

}

## 七、程序睡眠

如果需要把程序挂起一段时间，可以使用sleep()和usleep()两个库函数。

包含头文件：<unistd.h>

函数声明：

unsigned int sleep(unsigned int seconds);

int usleep(useconds\_t usec);

# 308、Linux的目录操作

## 一、几个简单的目录操作函数

### 1）获取当前工作目录

包含头文件：<unistd.h>

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

char \*get\_current\_dir\_name(void);

示例：

#include <iostream>

#include <unistd.h>

using namespace std;

int main()

{

char path1[256]; // linux系统目录的最大长度是255。

getcwd(path1,256);

cout << "path1=" << path1 << endl;

char \*path2=get\_current\_dir\_name();

cout << "path2=" << path2 << endl;

free(path2); // 注意释放内存。malloc() new delete

}

### 2）切换工作目录

包含头文件：<unistd.h>

int chdir(const char \*path);

返回值：0-成功；其它-失败（目录不存在或没有权限）。

### 3）创建目录

包含头文件：<sys/stat.h>

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

pathname-目录名。

mode-访问权限，如0755，不要省略前置的0。

返回值：0-成功；其它-失败（上级目录不存在或没有权限）。 /tmp/aaa /tmp/aaa/bbb

### 4）删除目录

包含头文件： <unistd.h>

int rmdir(const char \*path);

path-目录名。

返回值：0-成功；其它-失败（目录不存在或没有权限）。

## 二、获取目录中文件的列表

文件存放在目录中，在处理文件之前，必须先知道目录中有哪些文件，所以要获取目录中文件的列表。

### 1）包含头文件

#include <dirent.h>

### 2）相关的库函数

**步骤一：用opendir()函数打开目录。**

DIR \*opendir(const char \*pathname);

成功-返回目录的地址，失败-返回空地址。

**步骤二：用readdir()函数循环的读取目录。**

struct dirent \*readdir(DIR \*dirp);

成功-返回struct dirent结构体的地址，失败-返回空地址。

**步骤三：用closedir()关闭目录。**

int closedir(DIR \*dirp);

### 3）数据结构

目录指针：

DIR \*目录指针变量名;

每次调用readdir()，函数返回struct dirent的地址，存放了本次读取到的内容。

struct dirent

{

   long d\_ino;                     // inode number 索引节点号。

   off\_t d\_off;                    // offset to this dirent 在目录文件中的偏移。

   unsigned short d\_reclen;      // length of this d\_name 文件名长度。

   unsigned char d\_type;          // the type of d\_name 文件类型。

   char d\_name [NAME\_MAX+1];    // file name文件名，最长255字符。

};

重点关注结构体的d\_name和d\_type成员。

d\_name-文件名或目录名。

d\_type-文件的类型，有多种取值，最重要的是8和4，8-常规文件（A regular file）；4-子目录（A directory），其它的暂时不关心。注意，d\_name的数据类型是字符，不可直接显示。

示例：

#include <iostream>

#include <dirent.h>

using namespace std;

int main(int argc,char \*argv[])

{

if (argc != 2) { cout << "Using ./demo 目录名\n"; return -1; }

DIR \*dir; // 定义目录指针。

// 打开目录。

if ( (dir=opendir(argv[1])) == nullptr ) return -1;

// 用于存放从目录中读取到的内容。

struct dirent \*stdinfo=nullptr;

while (1)

{

// 读取一项内容并显示出来。

if ((stdinfo=readdir(dir)) == nullptr) break;

cout << "文件名=" << stdinfo->d\_name << "，文件类型=" << (int)stdinfo->d\_type << endl;

}

closedir(dir); // 关闭目录指针。

}

# 310、Linux的系统错误

在C++程序中，如果调用了库函数，可以通过函数的返回值判断调用是否成功。其实，还有一个整型的全局变量errno，存放了函数调用过程中产生的错误代码。

如果调用库函数失败，可以通过errno的值来查找原因，这也是调试程序的一个重要方法。

errno在<errno.h>中声明。

配合 strerror()和perror()两个库函数，可以查看出错的详细信息。

## 一、strerror()库函数

strerror() 在<string.h>中声明，用于获取错误代码对应的详细信息。

char \*strerror(int errnum); // 非线程安全。

int strerror\_r(int errnum, char \*buf, size\_t buflen); // 线程安全。

gcc8.3.1一共有133个错误代码。

示例一：

#include <iostream>

#include <cstring>

using namespace std;

int main()

{

int ii;

for(ii=0;ii<150;ii++) // gcc8.3.1一共有133个错误代码。

{

cout << ii << ":" << strerror(ii) << endl;

}

}

示例二：

#include <iostream>

#include <cstring>

#include <cerrno>

#include <sys/stat.h>

using namespace std;

int main()

{

int iret=mkdir("/tmp/aaa",0755);

cout << "iret=" << iret << endl;

cout << errno << ":" << strerror(errno) << endl;

}

## 二、perror()库函数

perror() 在<stdio.h>中声明，用于在控制台显示最近一次系统错误的详细信息，在实际开发中，服务程序在后台运行，通过控制台显示错误信息意义不大。（对调试程序略有帮助）

void perror(const char \*s);

## 三、注意事项

### 1）调用库函数失败不一定会设置errno

并不是全部的库函数在调用失败时都会设置errno的值，以man手册为准（一般来说，不属于系统调用的函数不会设置errno，属于系统调用的函数才会设置errno）。什么是系统调用？百度“库函数和系统调用的区别”。

### 2）errno不能作为调用库函数失败的标志

errno的值只有在库函数调用发生错误时才会被设置，当库函数调用成功时，errno的值不会被修改，不会主动的置为 0。

在实际开发中，判断函数执行是否成功还得靠函数的返回值，只有在返回值是失败的情况下，才需要关注errno的值。

示例：

#include <iostream>

#include <cstring> // strerror()函数需要的头文件。

#include <cerrno> // errno全局变量的头文件。

#include <sys/stat.h> // mkdir()函数需要的头文件。

using namespace std;

int main()

{

int iret=mkdir("/tmp/aaa/bb/cc/dd",0755);

if (iret!=0)

{

cout << "iret=" << iret << endl;

cout << errno << ":" << strerror(errno) << endl;

perror("调用mkdir(/tmp/aaa/bb/cc/dd)失败");

}

iret=mkdir("/tmp/dd",0755);

if (ireet!=0)

{

cout << "iret=" << iret << endl;

cout << errno << ":" << strerror(errno) << endl;

perror("调用mkdir(/tmp/dd)失败");

}

}

# 311、目录和文件的更多操作

## 一、access()库函数

access()函数用于判断当前用户对目录或文件的存取权限。

包含头文件：

#include <unistd.h>

函数声明：

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

参数说明：

pathname 目录或文件名。

mode  需要判断的存取权限。在头文件<unistd.h>中的预定义如下：

#define R\_OK  4  // 判断是否有读权限。

#define W\_OK 2  // 判断是否有写权限。

#define X\_OK  1  // 判断是否有执行权限。

#define F\_OK  0   // 判断是否存在。

返回值：

当pathname满足mode权限返回0，不满足返回-1，errno被设置。

在实际开发中，access()函数主要用于判断目录或文件是否存在。

## 二、stat()库函数

### 1）stat结构体

struct stat结构体用于存放目录或文件的详细信息，如下：

struct stat

{

  dev\_t st\_dev;    // 文件的设备编号。

  ino\_t st\_ino;    // 文件的i-node。

  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，刚建立的文件值为1。

  uid\_t st\_uid;    // 文件所有者的用户识别码。

  gid\_t st\_gid;    // 文件所有者的组识别码。

  dev\_t st\_rdev;   // 若此文件为设备文件，则为其设备编号。

  off\_t st\_size;   // 文件的大小，以字节计算。

  size\_t st\_blksize; // I/O 文件系统的I/O 缓冲区大小。

  size\_t st\_blocks;   // 占用文件区块的个数。

  time\_t st\_atime;   // 文件最近一次被存取或被执行的时间，

// 在用mknod、 utime、read、write 与tructate 时改变。

  time\_t st\_mtime;   // 文件最后一次被修改的时间，

// 在用mknod、 utime 和write 时才会改变。

  time\_t st\_ctime;   // 最近一次被更改的时间，在文件所有者、组、 权限被更改时更新。

};

struct stat结构体的成员变量比较多，重点关注st\_mode、st\_size和st\_mtime成员。注意：st\_mtime是一个整数表示的时间，需要程序员自己写代码转换格式。

st\_mode成员的取值很多，用以下两个宏来判断：

S\_ISREG(st\_mode)  // 是否为普通文件，如果是，返回真。

S\_ISDIR(st\_mode)  // 是否为目录，如果是，返回真。

### 2）stat()库函数

包含头文件：

#include <sys/stat.h>

函数声明：

int stat(const char \*path, struct stat \*buf);

stat()函数获取path参数指定目录或文件的详细信息，保存到buf结构体中。

返回值：0-成功，-1-失败，errno被设置。

示例：

#include <stdio.h>

#include <iostream>

#include <cstdio>

#include <sys/stat.h>

#include <unistd.h>

using namespace std;

int main(int argc,char \*argv[])

{

if (argc != 2) { cout << "Using:./demo 文件或目录名\n"; return -1; }

struct stat st; // 存放目录或文件详细信息的结构体。

// 获取目录或文件的详细信息

if (stat(argv[1],&st) != 0)

{

cout << "stat(" << argv[1] << "):" << strerror(errno) << endl; return -1;

}

if (S\_ISREG(st.st\_mode))

cout << argv[1] << "是一个文件(" << "mtime=" << st.st\_mtime << ",size=" << st.st\_size << ")\n";

if (S\_ISDIR(st.st\_mode))

cout << argv[1] << "是一个目录(" << "mtime=" << st.st\_mtime << ",size=" << st.st\_size << ")\n";

}

## 三、utime()库函数

utime()函数用于修改目录或文件的时间。

包含头文件：

#include <sys/types.h>

#include <utime.h>

函数声明：

int utime(const char \*filename, const struct utimbuf \*times);

utime()函数用来修改参数filename的st\_atime和st\_mtime。如果参数times为空地址，则设置为当前时间。结构utimbuf 声明如下：

struct utimbuf

{

  time\_t actime;

  time\_t modtime;

};

返回值：0-成功，-1-失败，errno被设置。

## 四、rename()库函数

rename()函数用于重命名目录或文件，相当于操作系统的mv命令。

包含头文件：

#include <stdio.h>

函数声明：

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

参数说明：

oldpath  原目录或文件名。

newpath  目标目录或文件名。

返回值：0-成功，-1-失败，errno被设置。

## 五、remove()库函数

remove()函数用于删除目录或文件，相当于操作系统的rm命令。

包含头文件：

#include <stdio.h>

函数声明：

int remove(const char \*pathname);

参数说明：

pathname 待删除的目录或文件名。

返回值：0-成功，-1-失败，errno被设置。

# 312、Linux的信号

## 一、信号的基本概念

信号（signal）是软件中断，是进程之间相互传递消息的一种方法，用于通知进程发生了事件，但是，不能给进程传递任何数据。

信号产生的原因有很多，在Shell中，可以用kill和killall命令发送信号：

kill -信号的类型 进程编号

killall -信号的类型 进程名

## 二、信号的类型

|  |  |  |  |
| --- | --- | --- | --- |
| 信号名 | 信号值 | 默认处理动作 | 发出信号的原因 |
| SIGHUP | 1 | A | 终端挂起或者控制进程终止 |
| **SIGINT** | **2** | **A** | **键盘中断Ctrl+c** |
| SIGQUIT | 3 | C | 键盘的退出键被按下 |
| SIGILL | 4 | C | 非法指令 |
| SIGABRT | 6 | C | 由abort(3)发出的退出指令 |
| SIGFPE | 8 | C | 浮点异常 |
| **SIGKILL** | **9** | **AEF** | **采用kill -9 进程编号 强制杀死程序。** |
| **SIGSEGV** | **11** | **CEF** | **无效的内存引用（数组越界、操作空指针和野指针等）。** |
| SIGPIPE | 13 | A | 管道破裂，写一个没有读端口的管道。 |
| **SIGALRM** | **14** | **A** | **由闹钟alarm()函数发出的信号。** |
| **SIGTERM** | **15** | **A** | **采用“kill 进程编号”或“killall 程序名”通知程序。** |
| SIGUSR1 | 10 | A | 用户自定义信号1 |
| SIGUSR2 | 12 | A | 用户自定义信号2 |
| **SIGCHLD** | **17** | **B** | **子进程结束信号** |
| SIGCONT | 18 |  | 进程继续（曾被停止的进程） |
| SIGSTOP | 19 | DEF | 终止进程 |
| SIGTSTP | 20 | D | 控制终端（tty）上按下停止键 |
| SIGTTIN | 21 | D | 后台进程企图从控制终端读 |
| SIGTTOU | 22 | D | 后台进程企图从控制终端写 |
| 其它 | <=64 | A | 自定义信号 |

处理动作一项中的字母含义如下：

A 缺省的动作是终止进程。

B 缺省的动作是忽略此信号，将该信号丢弃，不做处理。

C 缺省的动作是终止进程并进行内核映像转储（core dump）。

D 缺省的动作是停止进程，进入停止状态的程序还能重新继续，一般是在调试的过程中。

E 信号不能被捕获。

F 信号不能被忽略。

## 三、信号的处理

进程对信号的处理方法有三种：

1）对该信号的处理采用系统的默认操作，大部分的信号的默认操作是终止进程。

2）设置信号的处理函数，收到信号后，由该函数来处理。

3）忽略某个信号，对该信号不做任何处理，就像未发生过一样。

signal()函数可以设置程序对信号的处理方式。

函数声明：

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

参数signum表示信号的编号（信号的值）。

参数handler表示信号的处理方式，有三种情况：

1）SIG\_DFL：恢复参数signum信号的处理方法为默认行为。

2）一个自定义的处理信号的函数，函数的形参是信号的编号。

3）SIG\_IGN：忽略参数signum所指的信号。

## 四、信号有什么用

服务程序运行在后台，如果想让中止它，杀掉不是个好办法，因为进程被杀的时候，是突然死亡，没有安排善后工作。

如果向服务程序发送一个信号，服务程序收到信号后，调用一个函数，在函数中编写善后的代码，程序就可以有计划的退出。

如果向服务程序发送0的信号，可以检测程序是否存活。

## 五、信号应用示例

#include <iostream>

#include <unistd.h>

#include <signal.h>

using namespace std;

void EXIT(int sig)

{

cout << "收到了信号：" << sig << endl;

cout << "正在释放资源，程序将退出......\n";

// 以下是释放资源的代码。

cout << "程序退出。\n";

exit(0); // 进程退出。

}

int main(int argc,char \*argv[])

{

// 忽略全部的信号，防止程序被信号异常中止。

for (int ii=1;ii<=64;ii++) signal(ii,SIG\_IGN);

// 如果收到2和15的信号（Ctrl+c和kill、killall），本程序将主动退出。

signal(2,EXIT); signal(15,EXIT);

while (true)

{

cout << "执行了一次任务。\n";

sleep(1);

}

}

## 六、发送信号

Linux操作系统提供了kill和killall命令向进程发送信号，在程序中，可以用kill()函数向其它进程发送信号。

函数声明：

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

kill()函数将参数sig指定的信号给参数pid 指定的进程。

参数pid 有几种情况：

1）pid>0 将信号传给进程号为pid 的进程。

2）pid=0 将信号传给和当前进程相同进程组的所有进程，常用于父进程给子进程发送信号，注意，发送信号者进程也会收到自己发出的信号。

3）pid=-1 将信号广播传送给系统内所有的进程，例如系统关机时，会向所有的登录窗口广播关机信息。

sig：准备发送的信号代码，假如其值为0则没有任何信号送出，但是系统会执行错误检查，通常会利用sig值为零来检验某个进程是否仍在运行。

返回值说明： 成功执行时，返回0；失败返回-1，errno被设置。

# 313、进程终止

有8种方式可以中止进程，其中5种为正常终止，它们是：

1）在main()函数用return返回；

2）在任意函数中调用exit()函数；

3）在任意函数中调用\_exit()或\_Exit()函数；

4）最后一个线程从其启动例程（线程主函数）用return返回；

5）在最后一个线程中调用pthread\_exit()返回；

异常终止有3种方式，它们是：

6）调用abort()函数中止；

7）接收到一个信号；

8）最后一个线程对取消请求做出响应。

## 一、进程终止的状态

在main()函数中，return的返回值即终止状态，如果没有return语句或调用exit()，那么该进程的终止状态是0。

在Shell中，查看进程终止的状态：echo $?

正常终止进程的3个函数（exit()和\_Exit()是由ISO C说明的，\_exit()是由POSIX说明的）。

void exit(int status);

void \_exit(int status);

void \_Exit(int status);

status也是进程终止的状态。

如果进程被异常终止，终止状态为非0。 服务程序的调度、日志和监控

## 二、资源释放的问题

retun表示函数返回，会调用局部对象的析构函数，main()函数中的return还会调用全局对象的析构函数。

**exit()表示终止进程，不会调用局部对象的**[**析构函数**](https://so.csdn.net/so/search?q=%E6%9E%90%E6%9E%84%E5%87%BD%E6%95%B0&spm=1001.2101.3001.7020)**，只调用全局对象的析构函数。**

exit()会执行清理工作，然后退出，\_exit()和\_Exit()直接退出，不会执行任何清理工作。

## 三、进程的终止函数

进程可以用atexit()函数登记终止函数（最多32个），这些函数将由exit()自动调用。

int atexit(void (\*function)(void));

exit()调用终止函数的顺序与登记时相反。 进程退出前的收尾工作

# 314、调用可执行程序

Linux提供了system()函数和exec函数族，在C++程序中，可以执行其它的程序（二进制文件、操作系统命令或Shell脚本）。

## 一、system()函数

system()函数提供了一种简单的执行程序的方法，把需要执行的程序和参数用一个字符串传给system()函数就行了。

函数的声明：

int system(const char \* string);

system()函数的返回值比较麻烦。

1）如果执行的程序不存在，system()函数返回非0；

2）如果执行程序成功，并且被执行的程序终止状态是0，system()函数返回0；

3）如果执行程序成功，并且被执行的程序终止状态不是0，system()函数返回非0。

## 二、exec函数族

exec函数族提供了另一种在进程中调用程序（二进制文件或Shell脚本）的方法。

exec函数族的声明如下：

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

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

int execle(const char \*path, const char \*arg,...,char \* const envp[]);

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

int execvp(const char \*file, char \*const argv[]);

int execvpe(const char \*file, char \*const argv[],char \*const envp[]);

注意：

1）如果执行程序失败则直接返回-1，失败原因存于errno中。

2）新进程的进程编号与原进程相同，但是，新进程取代了原进程的代码段、数据段和堆栈。

3）如果执行成功则函数不会返回，当在主程序中成功调用exec后，被调用的程序将取代调用者程序，也就是说，exec函数之后的代码都不会被执行。

4）在实际开发中，最常用的是execl()和execv()，其它的极少使用。

示例：

#include <iostream>

#include <string.h>

#include <unistd.h>

using namespace std;

int main(int argc,char \*argv[])

{

int ret=execl("/bin/ls","/bin/ls","-lt","/tmp",0); // 最后一个参数0不能省略。

cout << "ret=" << ret << endl;

perror("execl");

/\*

char \*args[10];

args[0]="/bin/ls";

args[1]="-lt";

args[2]="/tmp";

args[3]=0; // 这行代码不能省略。

int ret=execv("/bin/ls",args);

cout << "ret=" << ret << endl;

perror("execv");

\*/

}

# 315、创建进程

## 一、Linux的0、1和2号进程

整个linux系统全部的进程是一个树形结构。

0号进程（系统进程）是所有进程的祖先，它创建了1号和2号进程。

1号进程（systemd）负责执行内核的初始化工作和进行系统配置。

2号进程（kthreadd）负责所有内核线程的调度和管理。

用pstree命令可以查看进程树（yum -y install psmisc）。

pstree -p 进程编号

## 二、进程标识

每个进程都有一个非负整数表示的唯一的进程ID。虽然是唯一的，但是进程ID可以复用。当一个进程终止后，其进程ID就成了复用的候选者。Linux采用延迟复用算法，让新建进程的ID不同于最近终止的进程所使用的ID。这样防止了新进程被误认为是使用了同一ID的某个已终止的进程。

pid\_t getpid(void); // 获取当前进程的ID。

pid\_t getppid(void); // 获取父进程的ID。

## 三、fork()函数

一个现有的进程可以调用fork()函数创建一个新的进程。

pid\_t fork(void);

由fork()创建的新进程被称为子进程。子进是父进程的副本，父进程和子进程都从调用fork()之后的代码开始执行。

fork()函数被调用一次，但返回两次。两次返回的区别是子进程的返回值是0，而父进程的返回值则是子进程的进程ID。

子进程获得了父进程数据空间、堆和栈的副本（**注意：子进程拥有的是副本，不是和父进程共享**）。

fork()之后，父进程和子进程的执行顺序是不确定的。

#include <iostream>

#include <unistd.h>

using namespace std;

int main()

{

int bh=8;

string message="我是一只傻傻鸟。";

pid\_t pid=fork();

if (pid>0)

{ // 父进程将执行这段代码。

sleep(1);

cout << "父：pid=" << pid << endl;

cout << "父：亲爱的" << bh << "号：" << message << endl;

}

else

{ // 子进程将执行这段代码。

bh=3; message="你是一只傻傻鸟。";

cout << "子：pid=" << pid << endl;

cout << "子：亲爱的" << bh << "号：" << message << endl;

}

## 四、fork()的两种用法

1）父进程复制自己，然后，父进程和子进程分别执行不同的代码。这种用法在网络服务程序中很常见，父进程等待客户端的连接请求，当请求到达时，父进程调用fork()，让子进程处理些请求，而父进程则继续等待下一个连接请求。

2）进程要执行另一个程序。这种用法在Shell中很常见，子进程从fork()返回后立即调用exec。

示例：

#include <iostream>

#include <unistd.h>

using namespace std;

int main()

{

if (fork()>0)

{ // 父进程将执行这段代码。

while (true)

{

sleep(1);

cout << "父进程运行中...\n";

}

}

else

{ // 子进程将执行这段代码。

sleep(10);

cout << "子进程开始执行任务...\n";

execl("/bin/ls","/bin/ls","-lt","/tmp",0);

cout << "子进程执行任务结束，退出。\n";

}

}

## 五、共享文件

fork()的一个特性是在父进程中打开的文件描述符都会被复制到子进程中，父进程和子进程共享同一个文件偏移量。

如果父进程和子进程写同一描述符指向的文件，但又没有任何形式的同步，那么它们的输出可能会相互混合。

示例：

#include <iostream>

#include <fstream>

#include <unistd.h>

using namespace std;

int main()

{

ofstream fout;

fout.open("/tmp/tmp.txt"); // 打开文件。

fork();

for (int ii=0;ii<10000000;ii++) // 向文件中写入一千万行数据。

{

fout << "进程" << getpid() << "西施" << ii << "极漂亮" << "\n"; // 写入的

内容无所谓。

}

fout.close(); // 关闭文件。

}

## 六、vfork()函数

vfork()函数的调用和返回值与fork()相同，但两者的语义不同。

vfork()函数用于创建一个新进程，而该新进程的目的是exec一个新程序，它不复制父进程的地址空间，因为子进程会立即调用exec，于是也就不会使用父进程的地址空间。如果子进程使用了父进程的地址空间，可能会带来未知的结果。

vfork()和fork()的另一个区别是：vfork()保证子进程先运行，在子进程调用exec或exit()之后父进程才恢复运行。

# 316、僵尸进程

如果父进程比子进程先退出，子进程将被1号进程托管（这也是一种让程序在后台运行的方法）。

如果子进程比父进程先退出，而父进程没有处理子进程退出的信息，那么，子进程将成为僵尸进程。

僵尸进程有什么危害？内核为每个子进程保留了一个数据结构，包括进程编号、终止状态、使用CPU时间等。父进程如果处理了子进程退出的信息，内核就会释放这个数据结构，父进程如果没有处理子进程退出的信息，内核就不会释放这个数据结构，子进程的进程编号将一直被占用。系统可用的进程编号是有限的，如果产生了大量的僵尸进程，将因为没有可用的进程编号而导致系统不能产生新的进程。

僵尸进程的避免：

1）子进程退出的时候，内核会向父进程发头SIGCHLD信号，如果父进程用signal(SIGCHLD,SIG\_IGN)通知[内核](https://baike.baidu.com/item/%E5%86%85%E6%A0%B8?fromModule=lemma_inlink)，表示自己对子进程的退出不感兴趣，那么子进程退出后会立即释放数据结构。

2）父进程通过wait()/waitpid()等函数等待子进程结束，在子进程退出之前，父进程将被阻塞待。

pid\_t wait(int \*stat\_loc);

pid\_t waitpid(pid\_t pid, int \*stat\_loc, int options);

pid\_t wait3(int \*status, int options, struct rusage \*rusage);

pid\_t wait4(pid\_t pid, int \*status, int options, struct rusage \*rusage);

返回值是子进程的编号。

stat\_loc是子进程终止的信息：a）如果是正常终止，宏WIFEXITED(stat\_loc)返回真，宏WEXITSTATUS(stat\_loc)可获取终止状态；b）如果是异常终止，宏WTERMSIG(stat\_loc)可获取终止进程的信号。

3）如果父进程很忙，可以捕获SIGCHLD信号，在信号处理函数中调用wait()/waitpid()。

示例一：

#include <iostream>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

using namespace std;

int main()

{

if (fork()>0)

{ // 父进程的流程。

int sts;

pid\_t pid=wait(&sts);

cout << "已终止的子进程编号是：" << pid << endl;

if (WIFEXITED(sts)) { cout << "子进程是正常退出的，退出状态是：" << WEXITSTATUS(sts) << endl; }

else { cout << "子进程是异常退出的，终止它的信号是：" << WTERMSIG(sts) << endl; }

}

else

{ // 子进程的流程。

//sleep(100);

int \*p=0; \*p=10;

exit(1);

}

}

示例二：

#include <iostream>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

using namespace std;

void func(int sig) // 子进程退出的信号处理函数。

{

int sts;

pid\_t pid=wait(&sts);

cout << "已终止的子进程编号是：" << pid << endl;

if (WIFEXITED(sts)) { cout << "子进程是正常退出的，退出状态是：" << WEXITSTATUS(sts) << endl; }

else { cout << "子进程是异常退出的，终止它的信号是：" << WTERMSIG(sts) << endl; }

}

int main()

{

signal(SIGCHLD,func); // 捕获子进程退出的信号。

if (fork()>0)

{ // 父进程的流程。

while (true)

{

cout << "父进程忙着执行任务。\n";

sleep(1);

}

}

else

{ // 子进程的流程。

sleep(5);

// int \*p=0; \*p=10;

exit(1);

}

}

# 317、多进程与信号

在多进程的服务程序中，如果子进程收到退出信号，子进程自行退出，如果父进程收到退出信号，则应该先向全部的子进程发送退出信号，然后自己再退出。

示例：

#include <iostream>

#include <unistd.h>

#include <signal.h>

using namespace std;

void FathEXIT(int sig); // 父进程的信号处理函数。

void ChldEXIT(int sig); // 子进程的信号处理函数。

int main()

{

// 忽略全部的信号，不希望被打扰。

for (int ii=1;ii<=64;ii++) signal(ii,SIG\_IGN);

// 设置信号,在shell状态下可用 "kill 进程号" 或 "Ctrl+c" 正常终止些进程

// 但请不要用 "kill -9 +进程号" 强行终止

signal(SIGTERM,FathEXIT); signal(SIGINT,FathEXIT); // SIGTERM 15 SIGINT 2

while (true)

{

if (fork()>0) // 父进程的流程。

{

sleep(5); continue;

}

else // 子进程的流程。

{

// 子进程需要重新设置信号。

signal(SIGTERM,ChldEXIT); // 子进程的退出函数与父进程不一样。

signal(SIGINT ,SIG\_IGN); // 子进程不需要捕获SIGINT信号。

while (true)

{

cout << "子进程" << getpid() << "正在运行中。\n"; sleep(3); continue;

}

}

}

}

// 父进程的信号处理函数。

void FathEXIT(int sig)

{

// 以下代码是为了防止信号处理函数在执行的过程中再次被信号中断。

signal(SIGINT,SIG\_IGN); signal(SIGTERM,SIG\_IGN);

cout << "父进程退出，sig=" << sig << endl;

kill(0,SIGTERM); // 向全部的子进程发送15的信号，通知它们退出。

// 在这里增加释放资源的代码（全局的资源）。

exit(0);

}

// 子进程的信号处理函数。

void ChldEXIT(int sig)

{

// 以下代码是为了防止信号处理函数在执行的过程中再次被信号中断。

signal(SIGINT,SIG\_IGN); signal(SIGTERM,SIG\_IGN);

cout << "子进程" << getpid() << "退出，sig=" << sig << endl;

// 在这里增加释放资源的代码（只释放子进程的资源）。

exit(0);

}

# 321、共享内存

多线程共享进程的地址空间，如果多个线程需要访问同一块内存，用全局变量就可以了。

在多进程中，每个进程的地址空间是独立的，不共享的，如果多个进程需要访问同一块内存，不能用全局变量，只能用共享内存。

共享内存（Shared Memory）允许多个进程（不要求进程之间有血缘关系）访问同一块内存空间，是多个进程之间共享和传递数据最高效的方式。进程可以将共享内存连接到它们自己的地址空间中，如果某个进程修改了共享内存中的数据，其它的进程读到的数据也会改变。

共享内存没有提供锁机制，也就是说，在某一个进程对共享内存进行读/写的时候，不会阻止其它进程对它的读/写。如果要对共享内存的读/写加锁，可以使用信号量。

Linux中提供了一组函数用于操作共享内存。

## 一、shmget函数

该函数用于创建/获取共享内存。

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

key 共享内存的键值，是一个整数（typedef unsigned int key\_t），一般采用十六进制，例如0x5005，不同共享内存的key不能相同。

size 共享内存的大小，以字节为单位。

shmflg 共享内存的访问权限，与文件的权限一样，例如0666|IPC\_CREAT，0666表示全部用户对它可读写，IPC\_CREAT表示如果共享内存不存在，就创建它。

返回值：成功返回共享内存的id（一个非负的整数），失败返回-1（系统内存不足、没有权限）

用ipcs -m可以查看系统的共享内存，包括：键值（key），共享内存id（shmid），拥有者（owner），权限（perms），大小（bytes）。

黑暗中有钟表

描述已自动生成

用ipcrm -m 共享内存id可以手工删除共享内存，如下：

黑暗中有钟表

描述已自动生成

## 二、shmat函数

该函数用于把共享内存连接到当前进程的地址空间。

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

shmid 由shmget()函数返回的共享内存标识。

shmaddr 指定共享内存连接到当前进程中的地址位置，通常填0，表示让系统来选择共享内存的地址。

shmflg 标志位，通常填0。

调用成功时返回共享内存起始地址，失败返回(void\*)-1。

## 三、shmdt函数

该函数用于将共享内存从当前进程中分离，相当于shmat()函数的反操作。

int shmdt(const void \*shmaddr);

shmaddr shmat()函数返回的地址。

调用成功时返回0，失败时返回-1。

## 四、shmctl函数

该函数用于操作共享内存，最常用的操作是删除共享内存。

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

shmid shmget()函数返回的共享内存id。

command 操作共享内存的指令，如果要删除共享内存，填IPC\_RMID。

buf 操作共享内存的数据结构的地址，如果要删除共享内存，填0。

调用成功时返回0，失败时返回-1。

**注意，用root创建的共享内存，不管创建的权限是什么，普通用户无法删除。**

## 五、示例

#include <iostream>

#include <cstdio>

#include <cstdlib>

#include <cstring>

#include <unistd.h>

#include <sys/ipc.h>

#include <sys/shm.h>

using namespace std;

struct stgirl // 超女结构体。

{

int no; // 编号。

char name[51]; // 姓名，注意，不能用string。

};

int main(int argc,char \*argv[])

{

if (argc!=3) { cout << "Using:./demo no name\n"; return -1; }

// 第1步：创建/获取共享内存，键值key为0x5005，也可以用其它的值。

int shmid=shmget(0x5005, sizeof(stgirl), 0640|IPC\_CREAT);

if ( shmid ==-1 )

{

cout << "shmget(0x5005) failed.\n"; return -1;

}

cout << "shmid=" << shmid << endl;

// 第2步：把共享内存连接到当前进程的地址空间。

stgirl \*ptr=(stgirl \*)shmat(shmid,0,0);

if ( ptr==(void \*)-1 )

{

cout << "shmat() failed\n"; return -1;

}

// 第3步：使用共享内存，对共享内存进行读/写。

cout << "原值：no=" << ptr->no << ",name=" << ptr->name << endl; // 显示共享内存中的原值。

ptr->no=atoi(argv[1]); // 对超女结构体的no成员赋值。

//strcpy(ptr->name,argv[2]); // 对超女结构体的name成员赋值。

ptr->name=argv[2];

cout << "新值：no=" << ptr->no << ",name=" << ptr->name << endl; // 显示共享内存中的当前值。

// 第4步：把共享内存从当前进程中分离。

shmdt(ptr);

// 第5步：删除共享内存。

//if (shmctl(shmid,IPC\_RMID,0)==-1)

//{

// cout << "shmctl failed\n"; return -1;

//}

}

# 322、循环队列、信号量、生产/消费者模型的源代码

## 一、demo1.cpp

// demo1.cpp，本程序演示循环队列的使用。

#include "\_public.h"

int main()

{

using ElemType=int;

squeue<ElemType,5> QQ;

ElemType ee; // 创建一个数据元素。

cout << "元素（1、2、3）入队。\n";

ee=1; QQ.push(ee);

ee=2; QQ.push(ee);

ee=3; QQ.push(ee);

cout << "队列的长度是" << QQ.size() << endl;

QQ.printqueue();

ee=QQ.front(); QQ.pop(); cout << "出队的元素值为" << ee << endl;

ee=QQ.front(); QQ.pop(); cout << "出队的元素值为" << ee << endl;

cout << "队列的长度是" << QQ.size() << endl;

QQ.printqueue();

cout << "元素（11、12、13、14、15）入队。\n";

ee=11; QQ.push(ee);

ee=12; QQ.push(ee);

ee=13; QQ.push(ee);

ee=14; QQ.push(ee);

ee=15; QQ.push(ee);

cout << "队列的长度是" << QQ.size() << endl;

QQ.printqueue();

}

## 二、demo2.cpp

// demo2.cpp，本程序演示基于共享内存的循环队列。

#include "\_public.h"

int main()

{

using ElemType=int;

// 初始化共享内存。

int shmid=shmget(0x5005, sizeof(squeue<ElemType,5>), 0640|IPC\_CREAT);

if ( shmid ==-1 )

{

cout << "shmget(0x5005) failed.\n"; return -1;

}

// 把共享内存连接到当前进程的地址空间。

squeue<ElemType,5> \*QQ=(squeue<ElemType,5> \*)shmat(shmid,0,0);

if ( QQ==(void \*)-1 )

{

cout << "shmat() failed\n"; return -1;

}

QQ->init(); // 初始化循环队列。

ElemType ee; // 创建一个数据元素。

cout << "元素（1、2、3）入队。\n";

ee=1; QQ->push(ee);

ee=2; QQ->push(ee);

ee=3; QQ->push(ee);

cout << "队列的长度是" << QQ->size() << endl;

QQ->printqueue();

ee=QQ->front(); QQ->pop(); cout << "出队的元素值为" << ee << endl;

ee=QQ->front(); QQ->pop(); cout << "出队的元素值为" << ee << endl;

cout << "队列的长度是" << QQ->size() << endl;

QQ->printqueue();

cout << "元素（11、12、13、14、15）入队。\n";

ee=11; QQ->push(ee);

ee=12; QQ->push(ee);

ee=13; QQ->push(ee);

ee=14; QQ->push(ee);

ee=15; QQ->push(ee);

cout << "队列的长度是" << QQ->size() << endl;

QQ->printqueue();

shmdt(QQ); // 把共享内存从当前进程中分离。

}

## 三、demo3.cpp

// demo3.cpp，本程序演示用信号量给共享内存加锁。

#include "\_public.h"

struct stgirl // 超女结构体。

{

int no; // 编号。

char name[51]; // 姓名，注意，不能用string。

};

int main(int argc,char \*argv[])

{

if (argc!=3) { cout << "Using:./demo no name\n"; return -1; }

// 第1步：创建/获取共享内存，键值key为0x5005，也可以用其它的值。

int shmid=shmget(0x5005, sizeof(stgirl), 0640|IPC\_CREAT);

if ( shmid ==-1 )

{

cout << "shmget(0x5005) failed.\n"; return -1;

}

cout << "shmid=" << shmid << endl;

// 第2步：把共享内存连接到当前进程的地址空间。

stgirl \*ptr=(stgirl \*)shmat(shmid,0,0);

if ( ptr==(void \*)-1 )

{

cout << "shmat() failed\n"; return -1;

}

// 创建、初始化二元信号量。

csemp mutex;

if (mutex.init(0x5005)==false)

{

cout << "mutex.init(0x5005) failed.\n"; return -1;

}

cout << "申请加锁...\n";

mutex.wait(); // 申请加锁。

cout << "申请加锁成功。\n";

// 第3步：使用共享内存，对共享内存进行读/写。

cout << "原值：no=" << ptr->no << ",name=" << ptr->name << endl; // 显示共享内存中的原值。

ptr->no=atoi(argv[1]); // 对超女结构体的no成员赋值。

strcpy(ptr->name,argv[2]); // 对超女结构体的name成员赋值。

cout << "新值：no=" << ptr->no << ",name=" << ptr->name << endl; // 显示共享内存中的当前值。

sleep(10);

mutex.post(); // 解锁。

cout << "解锁。\n";

// 查看信号量 ：ipcs -s // 删除信号量 ：ipcrm sem 信号量id

// 查看共享内存：ipcs -m // 删除共享内存：ipcrm -m 共享内存id

// 第4步：把共享内存从当前进程中分离。

shmdt(ptr);

// 第5步：删除共享内存。

//if (shmctl(shmid,IPC\_RMID,0)==-1)

//{

// cout << "shmctl failed\n"; return -1;

//}

}

## 四、incache.cpp

// 多进程的生产消费者模型的生产者程序

#include "\_public.h"

int main()

{

struct stgirl // 循环队列的数据元素是超女结构体。

{

int no;

char name[51];

};

using ElemType=stgirl;

// 初始化共享内存。

int shmid=shmget(0x5005, sizeof(squeue<ElemType,5>), 0640|IPC\_CREAT);

if ( shmid ==-1 )

{

cout << "shmget(0x5005) failed.\n"; return -1;

}

// 把共享内存连接到当前进程的地址空间。

squeue<ElemType,5> \*QQ=(squeue<ElemType,5> \*)shmat(shmid,0,0);

if ( QQ==(void \*)-1 )

{

cout << "shmat() failed\n"; return -1;

}

QQ->init(); // 初始化循环队列。

ElemType ee; // 创建一个数据元素。

csemp mutex; mutex.init(0x5001); // 用于给共享内存加锁。

csemp cond; cond.init(0x5002,0,0); // 信号量的值用于表示队列中数据元素的个数。

mutex.wait(); // 加锁。

// 生产3个数据。

ee.no=3; strcpy(ee.name,"西施"); QQ->push(ee);

ee.no=7; strcpy(ee.name,"冰冰"); QQ->push(ee);

ee.no=8; strcpy(ee.name,"幂幂"); QQ->push(ee);

mutex.post(); // 解锁。

cond.post(3); // 实参是3，表示生产了3个数据。

shmdt(QQ); // 把共享内存从当前进程中分离。

}

## 五、outcache.cpp

// 多进程的生产消费者模型的消费者程序

#include "\_public.h"

int main()

{

struct stgirl // 循环队列的数据元素是超女结构体。

{

int no;

char name[51];

};

using ElemType=stgirl;

// 初始化共享内存。

int shmid=shmget(0x5005, sizeof(squeue<ElemType,5>), 0640|IPC\_CREAT);

if ( shmid ==-1 )

{

cout << "shmget(0x5005) failed.\n"; return -1;

}

// 把共享内存连接到当前进程的地址空间。

squeue<ElemType,5> \*QQ=(squeue<ElemType,5> \*)shmat(shmid,0,0);

if ( QQ==(void \*)-1 )

{

cout << "shmat() failed\n"; return -1;

}

QQ->init(); // 初始化循环队列。

ElemType ee; // 创建一个数据元素。

csemp mutex; mutex.init(0x5001); // 用于给共享内存加锁。

csemp cond; cond.init(0x5002,0,0); // 信号量的值用于表示队列中数据元素的个数。

while (true)

{

mutex.wait(); // 加锁。

while (QQ->empty()) // 如果队列空，进入循环，否则直接处理数据。必须用循环，不能用if

{

mutex.post(); // 解锁。

cond.wait(); // 等待生产者的唤醒信号。

mutex.wait(); // 加锁。

}

// 数据元素出队。

ee = QQ->front(); QQ->pop();

mutex.post(); // 解锁。

// 处理出队的数据（把数据消费掉）。

cout << "no=" << ee.no << ",name=" << ee.name << endl;

usleep(100); // 假设处理数据需要时间，方便演示。

}

shmdt(QQ);

}

## 六、\_public.h

#ifndef \_\_PUBLIC\_HH

#define \_\_PUBLIC\_HH 1

#include <iostream>

#include <cstdio>

#include <cstdlib>

#include <cstring>

#include <unistd.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <sys/types.h>

#include <sys/sem.h>

using namespace std;

// 循环队列。

template <class TT, int MaxLength>

class squeue

{

private:

bool m\_inited; // 队列被初始化标志，true-已初始化；false-未初始化。

TT m\_data[MaxLength]; // 用数组存储循环队列中的元素。

int m\_head; // 队列的头指针。

int m\_tail; // 队列的尾指针，指向队尾元素。

int m\_length; // 队列的实际长度。

squeue(const squeue &) = delete; // 禁用拷贝构造函数。

squeue &operator=(const squeue &) = delete; // 禁用赋值函数。

public:

squeue() { init(); } // 构造函数。

// 循环队列的初始化操作。

// 注意：如果用于共享内存的队列，不会调用构造函数，必须调用此函数初始化。

void init()

{

if (m\_inited!=true) // 循环队列的初始化只能执行一次。

{

m\_head=0; // 头指针。

m\_tail=MaxLength-1; // 为了方便写代码，初始化时，尾指针指向队列的最后一个位置。

m\_length=0; // 队列的实际长度。

memset(m\_data,0,sizeof(m\_data)); // 数组元素清零。

m\_inited=true;

}

}

// 元素入队，返回值：false-失败；true-成功。

bool push(const TT &ee)

{

if (full() == true)

{

cout << "循环队列已满，入队失败。\n"; return false;

}

// 先移动队尾指针，然后再拷贝数据。

m\_tail=(m\_tail+1)%MaxLength; // 队尾指针后移。

m\_data[m\_tail]=ee;

m\_length++;

return true;

}

// 求循环队列的长度，返回值：>=0-队列中元素的个数。

int size()

{

return m\_length;

}

// 判断循环队列是否为空，返回值：true-空，false-非空。

bool empty()

{

if (m\_length == 0) return true;

return false;

}

// 判断循环队列是否已满，返回值：true-已满，false-未满。

bool full()

{

if (m\_length == MaxLength) return true;

return false;

}

// 查看队头元素的值，元素不出队。

TT& front()

{

return m\_data[m\_head];

}

// 元素出队，返回值：false-失败；true-成功。

bool pop()

{

if (empty() == true) return false;

m\_head=(m\_head+1)%MaxLength; // 队列头指针后移。

m\_length--;

return true;

}

// 显示循环队列中全部的元素。

// 这是一个临时的用于调试的函数，队列中元素的数据类型支持cout输出才可用。

void printqueue()

{

for (int ii = 0; ii < size(); ii++)

{

cout << "m\_data[" << (m\_head+ii)%MaxLength << "],value=" \

<< m\_data[(m\_head+ii)%MaxLength] << endl;

}

}

};

// 信号量。

class csemp

{

private:

union semun // 用于信号量操作的共同体。

{

int val;

struct semid\_ds \*buf;

unsigned short \*arry;

};

int m\_semid; // 信号量id（描述符）。

// 如果把sem\_flg设置为SEM\_UNDO，操作系统将跟踪进程对信号量的修改情况，

// 在全部修改过信号量的进程（正常或异常）终止后，操作系统将把信号量恢复为初始值。

// 如果信号量用于互斥锁，设置为SEM\_UNDO。

// 如果信号量用于生产消费者模型，设置为0。

short m\_sem\_flg;

csemp(const csemp &) = delete; // 禁用拷贝构造函数。

csemp &operator=(const csemp &) = delete; // 禁用赋值函数。

public:

csemp():m\_semid(-1){}

// 如果信号量已存在，获取信号量；如果信号量不存在，则创建它并初始化为value。

// 如果用于互斥锁，value填1，sem\_flg填SEM\_UNDO。

// 如果用于生产消费者模型，value填0，sem\_flg填0。

bool init(key\_t key,unsigned short value=1,short sem\_flg=SEM\_UNDO);

bool wait(short value=-1);// 信号量的P操作，如果信号量的值是0，将阻塞等待，直到信号量的值大于0。

bool post(short value=1); // 信号量的V操作。

int getvalue(); // 获取信号量的值，成功返回信号量的值，失败返回-1。

bool destroy(); // 销毁信号量。

~csemp();

};

#endif

## 七、\_public.cpp

#include "\_public.h"

// 如果信号量已存在，获取信号量；如果信号量不存在，则创建它并初始化为value。

// 如果用于互斥锁，value填1，sem\_flg填SEM\_UNDO。

// 如果用于生产消费者模型，value填0，sem\_flg填0。

bool csemp::init(key\_t key,unsigned short value,short sem\_flg)

{

if (m\_semid!=-1) return false; // 如果已经初始化了，不必再次初始化。

m\_sem\_flg=sem\_flg;

// 信号量的初始化不能直接用semget(key,1,0666|IPC\_CREAT)

// 因为信号量创建后，初始值是0，如果用于互斥锁，需要把它的初始值设置为1，

// 而获取信号量则不需要设置初始值，所以，创建信号量和获取信号量的流程不同。

// 信号量的初始化分三个步骤：

// 1）获取信号量，如果成功，函数返回。

// 2）如果失败，则创建信号量。

// 3) 设置信号量的初始值。

// 获取信号量。

if ( (m\_semid=semget(key,1,0666)) == -1)

{

// 如果信号量不存在，创建它。

if (errno==ENOENT)

{

// 用IPC\_EXCL标志确保只有一个进程创建并初始化信号量，其它进程只能获取。

if ( (m\_semid=semget(key,1,0666|IPC\_CREAT|IPC\_EXCL)) == -1)

{

if (errno==EEXIST) // 如果错误代码是信号量已存在，则再次获取信号量。

{

if ( (m\_semid=semget(key,1,0666)) == -1)

{

perror("init 1 semget()"); return false;

}

return true;

}

else // 如果是其它错误，返回失败。

{

perror("init 2 semget()"); return false;

}

}

// 信号量创建成功后，还需要把它初始化成value。

union semun sem\_union;

sem\_union.val = value; // 设置信号量的初始值。

if (semctl(m\_semid,0,SETVAL,sem\_union) < 0)

{

perror("init semctl()"); return false;

}

}

else

{ perror("init 3 semget()"); return false; }

}

return true;

}

// 信号量的P操作（把信号量的值减value），如果信号量的值是0，将阻塞等待，直到信号量的值大于0。

bool csemp::wait(short value)

{

if (m\_semid==-1) return false;

struct sembuf sem\_b;

sem\_b.sem\_num = 0; // 信号量编号，0代表第一个信号量。

sem\_b.sem\_op = value; // P操作的value必须小于0。

sem\_b.sem\_flg = m\_sem\_flg;

if (semop(m\_semid,&sem\_b,1) == -1) { perror("p semop()"); return false; }

return true;

}

// 信号量的V操作（把信号量的值减value）。

bool csemp::post(short value)

{

if (m\_semid==-1) return false;

struct sembuf sem\_b;

sem\_b.sem\_num = 0; // 信号量编号，0代表第一个信号量。

sem\_b.sem\_op = value; // V操作的value必须大于0。

sem\_b.sem\_flg = m\_sem\_flg;

if (semop(m\_semid,&sem\_b,1) == -1) { perror("V semop()"); return false; }

return true;

}

// 获取信号量的值，成功返回信号量的值，失败返回-1。

int csemp::getvalue()

{

return semctl(m\_semid,0,GETVAL);

}

// 销毁信号量。

bool csemp::destroy()

{

if (m\_semid==-1) return false;

if (semctl(m\_semid,0,IPC\_RMID) == -1) { perror("destroy semctl()"); return false; }

return true;

}

csemp::~csemp()

{

}

## 八、makefile

all:demo1 demo2 demo3 incache outcache

demo1:demo1.cpp \_public.h \_public.cpp

g++ -g -o demo1 demo1.cpp \_public.cpp

demo2:demo2.cpp \_public.h \_public.cpp

g++ -g -o demo2 demo2.cpp \_public.cpp

demo3:demo3.cpp \_public.h \_public.cpp

g++ -g -o demo3 demo3.cpp \_public.cpp

incache:incache.cpp \_public.h \_public.cpp

g++ -g -o incache incache.cpp \_public.cpp

outcache:outcache.cpp \_public.h \_public.cpp

g++ -g -o outcache outcache.cpp \_public.cpp

clean:

rm -f demo1 demo2 demo3 incache outcache