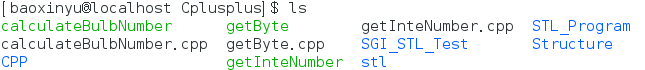
# 系统编程概念

## 标准C语言函数库（GNU C语言函数库，glibc）

### 确定系统的glibc版本

对于Linux不同版本的系统中，可以通过对某个与glibc进行动态链接的可执行文件，运行ldd（列出动态依赖性）程序，接下来，再检查输出的库依赖性列表，就能发现glibc的共享库的位置：



如上图，对一个已经经过编译链接的可执行文件，例如目录/CPlusPlus下的一个之前编译好的可执行文件calculateBulbNumber文件可以执行如下命令：

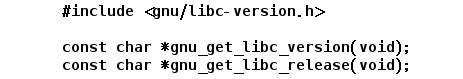
ldd calculateBulbNumber | grep libc

可以得到如图的结果：



也可以在应用程序中进行测试常量和调用库函数两种方式来确定glibc版本。

从版本2.0开始，glibc定义了两个常量：\_GLIBC\_和\_GLIBC\_MINOR\_，可以在程序里使用预编译命令#ifdef进行测试，确定版本。但是对于两个安装了不同版本的glibc的操作系统来说，这两个常量作用就有限。因此程序可以调用gnu\_get\_glibc\_version()函数来确定运行时的glibc的版本。在头文件中<gnu/libc-version.h>中，定义了该函数，如图：



使用man 3 gnu\_get\_glibc\_version即可查询到该函数所在的头文件。

具体程序如下：

#include <gnu/libc-version.h>

#include <iostream>

int main()

{

#ifdef \_GLIBC\_

std::cout << \_GLIBC\_ << std::endl;

#else

std::cout << "\_GLIBC\_ constant didn't defined in library" <<std::endl;

#endif

#ifdef \_GLIBC\_MINOR\_

std::cout << \_GLIBC\_MINOR\_ << std::endl;

#else

std::cout << "\_GLIBC\_MINOR\_ constant didn't defined in library" << std::endl;

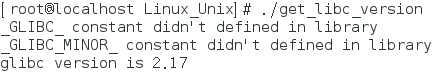
#endif

std::cout << "glibc version is "<< gnu\_get\_libc\_version() << std::endl;

return 0;

}

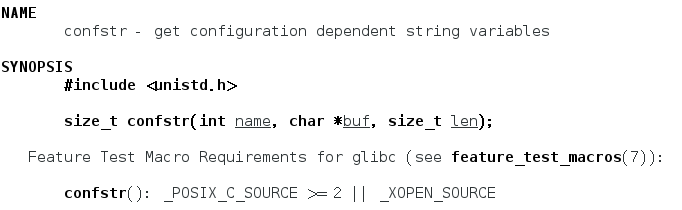
输出结果如图：



除了上述两种方式，还可以使用confstr()函数来获取\_GS\_GNU\_LIBC\_VERSION配置变量的值。调用系统的man命令：

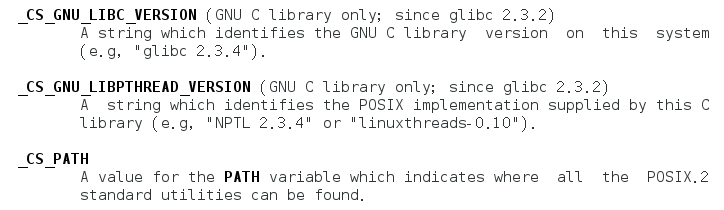


获取confstr的头文件和相关描述：



可以看到confstr函数有三个参数，整形变量name，字符串数组buf，无符号整型变量len

name参数是被查询的系统变量，系统提供了如下变量：



如果buf非空，而且长度非0，confstr函数复制了字符串的值并且删减到len-1的长度到buf中，以’\0’空字符结尾。这个过程会检查返回值的长度和len的数值，保证选择的是两者之间的较小值，因此buff的长度为min(return value,len)。如果len是0，buf是NULL，confstr只会返回下面定义的值。

如果name是一个合理的配置变量，confstr返回可以保存这个变量全部值的字节长度，这个长度也许会远大于len，也就意味着buf存着的字符串是删减过的。

如果name是一个合理的配置变量，但是这个变量没有值，confstr返回0,。

如果name和配置变量不匹配，confstr返回0，并且errno被设置为EINVAL。

程序如下：

#include <unistd.h>

#include <iostream>

int main()

{

char\* buff = NULL;//定义一个空字符串

size\_t n = 0;//定义一个变量记录返回值的长度

n = confstr(\_CS\_GNU\_LIBC\_VERSION,buff,(size\_t)0);

buff = new char[n];

n = confstr(\_CS\_GNU\_LIBC\_VERSION,buff,n);

std::cout << n << buff << std::endl;

return 0;

}

输出结果如图：



### 处理来自系统调用和库函数的错误

每个系统调用和库函数都会返回某类状态值，用来表明调用成功与否。要坚持对状态值进行检查，既能保证程序可以在出现情况时明确错误类型，也能减少程序调试时间。

少数几个系统函数在调用时从不失败，如getpid()总能成功返回进程的ID，而exit()总能终止进程。

**处理系统调用错误**

每个系统调用的手册页记录有调用的返回值说明，也会对错误情况有个说明，通常系统调用会-1表示出错，同时会对errno进行一个设置，来标识错误的类型。因此在程序中可以在调用之后做一个判断，确定是否出错。

程序代码可以如下类型：

fd = open(pathname,flags,mode); //系统调用open函数打开文件

if(fd == -1)

/\* 出错代码说明 \*/

在程序中应该包含<errno.h>，该文件提供了对errno的声明，以及一组针对各种错误编号定义的常量。所有的错误编号都以E开头。

程序代码如下：

cnt = open(pathname,flags,mode);

if(cnt == -1)

{

if(errno == EINTR)

fprint(stderr,”read was interrupted by a signal\n”);

else

/\* Some other error occurred \*/

}

若调用系统调用和库函数成功，errno不会被重置为0。因此在进行错误检测时，应该先对调用的返回值进行判断，在进行错误类型的匹配。在少数系统调用(如getpriority())在调用成功后，也会返回-1，因此要判断此类系统调用是否发生错误，要在调用前将errno置为0，再进行错误检测。

系统调用失败后，常见的做法之一是根据errno值打印错误消息。函数perror()和stderror()目的就是这样。

函数perror()函数会打印出其msg参数所指向的字符串，紧跟一条与当前errno值对应的消息。在