<https://blog.csdn.net/hnlyyk/article/details/51444617>

慢系统调用(slow system call)：此术语适用于那些可能永远阻塞的系统调用。永远阻塞的系统调用是指调用有可能永远无法返回，多数网络支持函数都属于这一类。如：若没有客户连接到服务器上，那么服务器的accept调用就没有返回的保证。  
  
EINTR错误的产生：当阻塞于某个慢系统调用的一个进程捕获某个信号且相应信号处理函数返回时，该系统调用可能返回一个EINTR错误。例如：在socket服务器端，设置了信号捕获机制，有子进程，当在父进程阻塞于慢系统调用时由父进程捕获到了一个有效信号时，内核会致使accept返回一个EINTR错误(被中断的系统调用)。  
  
当碰到EINTR错误的时候，可以采取有一些可以重启的系统调用要进行重启，而对于有一些系统调用是不能够重启的。例如：accept、read、write、select、和open之类的函数来说，是可以进行重启的。不过对于套接字编程中的connect函数我们是不能重启的，若connect函数返回一个EINTR错误的时候，我们不能再次调用它，否则将立即返回一个错误。针对connect不能重启的处理方法是，必须调用select来等待连接完成。  
   
在 linux 或者 unix 环境中， errno 是一个十分重要的部分。在调用的函 数出现问题的时候，我们可以通过 errno 的值来确定出错的原因，这就会 涉及到一个问题，那就是如何保证 errno 在多线程或者进程中安全？我们希望在多线程或者进程中，每个线程或者进程都拥有自己独立和唯一的一个 errno ，这样就能够保证不会有竞争条 件的出现。一般而言，编译器会自动保证 errno 的安全性，但是为了妥善期间，我们希望在写 makefile 的时 候把 \_LIBC\_REENTRANT 宏定义，比 如我们在检查 <bits/errno.h> 文件中发现如下的定义：   
  
C代码 

**[cpp]** [view plain](https://blog.csdn.net/hnlyyk/article/details/51444617) [copy](https://blog.csdn.net/hnlyyk/article/details/51444617)

1. # ifndef \_\_ASSEMBLER\_\_
2. /\* Function to get address of global `errno' variable. \*/
3. **extern** **int** \*\_\_errno\_location (**void**) \_\_THROW \_\_attribute\_\_ ((\_\_const\_\_));

6. # if !defined \_LIBC || defined \_LIBC\_REENTRANT
7. /\* When using threads, errno is a per-thread value. \*/
8. # define errno (\*\_\_errno\_location ())
9. # endif
10. # endif /\* !\_\_ASSEMBLER\_\_ \*/
11. #endif /\* \_ERRNO\_H \*/

也就是说，在没有定义 \_\_LIBC 或者定义 \_LIBC\_REENTRANT 的时候， errno 是多线程 / 进程安全的。   
一般而言， \_\_ASSEMBLER\_\_, \_LIBC 和 \_LIBC\_REENTRANT 都不会被编译器定义，但是如果我们定义 \_LIBC\_REENTRANT 一次又何妨那？   
为了检测一下你编译器是否定义上述变量，不妨使用下面一个简单程序。   
C代码

**[cpp]** [view plain](https://blog.csdn.net/hnlyyk/article/details/51444617) [copy](https://blog.csdn.net/hnlyyk/article/details/51444617)

1. #include <stdio.h>
2. #include <errno.h>
4. **int** main( **void** )
5. {
6. #ifndef \_\_ASSEMBLER\_\_
7. printf( "Undefine \_\_ASSEMBLER\_\_\n" );
8. #else
9. printf( "define \_\_ASSEMBLER\_\_\n" );
10. #endif
12. #ifndef \_\_LIBC
13. printf( "Undefine \_\_LIBC\n" );
14. #else
15. printf( "define \_\_LIBC\n" );
16. #endif
18. #ifndef \_LIBC\_REENTRANT
19. printf( "Undefine \_LIBC\_REENTRANT\n" );
20. #else
21. printf( "define \_LIBC\_REENTRANT\n" );
22. #endif
24. **return** 0;
25. }

希望读者在进行移植的时候，读一下相关的 unix 版本的 <bits/errno.h> 文 件，来确定应该定义什么宏。不同的 unix 版本可能存在着一些小的差别！  
   
有时候，在调用系统调用时，可能会接收到某个信号而导致调用退出。譬如使用system调用某个命令之后该进程会接收到SIGCHILD信号，然后如果这个进程的线程中有慢系统调用，那么接收到该信号的时候可能就会退出，返回EINTR错误码。  
EINTR  
　　linux中函数的返回状态，在不同的函数中意义不同：  
1）write  
　　表示：由于信号中断，没写成功任何数据。  
　　The call was interrupted by a signal before any data was written.  
2）read  
　　表示：由于信号中断，没读到任何数据。  
　　The call was interrupted by a signal before any data was read.  
3）sem\_wait  
　　函数调用被信号处理函数中断  
　　The call was interrupted by a signal handler.  
4）recv  
　　由于信号中断返回，没有任何数据可用。  
　　function was interrupted by a signal that was caught, before any data was available.  
   
调用系统调用的时候,有时系统调用会被中断.此时,系统调用会返回-1,并且错误码被置为EINTR.但是,有时并不将这样的情况作为错误.有两种处理方法:  
  
  
1.如果错误码为EINTR则重新调用系统调用,例如Postgresql中有一段代码:

**[cpp]** [view plain](https://blog.csdn.net/hnlyyk/article/details/51444617) [copy](https://blog.csdn.net/hnlyyk/article/details/51444617)

1. retry1:
2. **if** (send(port->sock, &SSLok, 1, 0) != 1)
3. {
4. **if** (errno == EINTR)
5. **goto** retry1; /\* if interrupted, just retry \*/
6. }

2.重新定义系统调用,忽略错误码为EINTR的情况.例如,Cherokee中的一段代码:

**[cpp]** [view plain](https://blog.csdn.net/hnlyyk/article/details/51444617) [copy](https://blog.csdn.net/hnlyyk/article/details/51444617)

1. **int** cherokee\_stat (**const** **char** \*restrict path, **struct** stat \*buf)
2. {
3. **int** re;
4. **do** {
5. re = stat (path, buf);
6. } **while** ((re == -1) && (errno == EINTR));
7. **return** re;
8. }

今天使用select调用的时候总是出错，返回EINTR错误->Interrupted system call，主要是由于代码中调用了signal捕获子进程退出信号SIGCHLD的处理，故我采用忽略EINTR的策略，代码改为如下解决

**[cpp]** [view plain](https://blog.csdn.net/hnlyyk/article/details/51444617) [copy](https://blog.csdn.net/hnlyyk/article/details/51444617)

1. signal(SIGCHLD,sig\_wait);
2. **while**(1){
3. rdset=ctlset;
4. /\* 如果可用处理子进程等于0个，那么select就暂时不再监听连接描述符，由listen函数的backlog控制连接数目队列
5. if(iavailable\_child <= 0)
6. FD\_CLR(ifdlisten, &rdset);  // turn off if no available children
7. \*/
8. iselectret=select(ifdmax+1, &rdset,NULL,NULL,NULL);
9. **if**(iselectret<0){
10. **if**(errno==EINTR){
11. //              printf("Receives the interrupt signal\n");
12. **continue**;
13. }
14. **else**{
15. printf("select error,will be exit,error msg:%s \n",strerror(errno));
16. exit(-1);
17. }
18. }

版权声明：【博主微信公众号：不忘初心的行者】【本文为博主原创,未经博主允许不得转载】 https://blog.csdn.net/u010193457/article/details/51444617