atexit()函数使用说明

NAME

atexit - 用来注册执行 exit()函数前执行的终止处理程序.

SYNOPSIS

#include <stdlib.h>

int atexit(void (*function)(void))；

DESCRIPTION

atexit()用来注册终止处理程序，当程序通过调用 exit()或从 main 中返回时被调用.终止处理程序执行的顺序和注册时的顺序是相反的，终止处理程序没有参数传递.同一个函数若注册多次，那它也会被调用多次.按 POSIX.1-2001 规定，至少可以注册 32 个终止处理程序，若想查看实际可以注册多少个终止处理程序，可以通过调用 sysconf()函数获得.

当一个子进程是通过调用 fork()函数产生时，它将继承父进程的所有终止处理程序.在成功调用 exec 系列函数后，所有的终止处理程序都会被删除.

RETURN VALUE

成功返回 0，失败返回非 0 值.

CONFORMING TO

SVr4， 4.3BSD， C89， C99， POSIX.1-2001.

NOTES

如果一个进程被信号所中断，那由 atexit()函数注册的终止处理程序不会被调用.

如果在其中一个终止处理程序中调用了_exit()函数；那剩余的终止处理程序将不会得到调用，同时由 exit()函数调用的其他终止进程步骤也将不会执行.

在 POSIX.1-2001 标准中，在终止进程过程中，在终止处理程序中，不只一次的调用 exit()函数，这样导致的结果是未定义的.在某些系统中，这样的调用将会导致递归死循环.可移植的程序不应该在终止处理程序中调用 exit()函数.

函数 atexit()和 on_exit()在注册终止程序时，有一样的列表.在进程正常退出时执行终止处理程序，调用的顺序刚好与注册时相反.

按 POSIX.1-2001 的规定，如果在终止处理程序中调用 longjmp()函数，这样导致的结果是未定义的.

Linux notes

自 glibc2.2.3 版本后，atexit()和 on_exit()函数能够使用在共享库建立的函数，当共享库卸载时被调用.

EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
void bye(void)
{
   printf("That was all， folks\n")；
}
int   main(void)
{
   long a；
   int i；
   a = sysconf(_SC_ATEXIT_MAX)；
   printf("ATEXIT_MAX = %ld\n"， a)；

   i = atexit(bye)；
   if (i != 0) {
```

```
        fprintf(stderr，  "cannot set exit function\n")；
        exit(EXIT_FAILURE)；
      }
      exit(EXIT_SUCCESS)；
    }
```

pthread_once()函数详解

在多线程环境中，有些事仅需要执行一次。通常当初始化应用程序时，可以比较容易地将其放在 main 函数中。但当你写一个库时，就不能在 main 里面初始化了，你可以用静态初始化，但使用一次初始化（pthread_once）会比较容易些。

int pthread_once(pthread_once_t *once_control, void (*init_routine) (void))；

功能：本函数使用初值为 PTHREAD_ONCE_INIT 的 once_control 变量保证 init_routine()函数在本进程执行序列中仅执行一次。

在多线程编程环境下，尽管 pthread_once()调用会出现在多个线程中，init_routine()函数仅执行一次，究竟在哪个线程中执行是不定的，是由内核调度来决定。

Linux Threads 使用互斥锁和条件变量保证由 pthread_once()指定的函数执行且仅执行一次，而 once_control 表示是否执行过。

如果 once_control 的初值不是 PTHREAD_ONCE_INIT（Linux Threads 定义为 0），pthread_once() 的行为就会不正常。

在 LinuxThreads 中，实际"一次性函数"的执行状态有三种：NEVER（0）、IN_PROGRESS（1）、DONE （2），如果 once 初值设为 1，则由于所有 pthread_once()都必须等待其中一个激发"已执行一次"信号，因此所有 pthread_once ()都会陷入永久的等待中；如果设为 2，则表示该函数已执行过一次，从而所有 pthread_once()都会立即返回 0。

适用范围

这种情况一般用于某个多线程调用的模块使用前的初始化，但是无法判定哪个线程先运行，从而不知道把初始化代码放在哪个线程合适的问题。

当然，我们一般的做法是把初始化函数放在 main 里，创建线程之前来完成，但是如果我们的程序最终不是做成可执行程序，而是编译成库的形式，那么 main 函数这种方式就没法做到了。

基本原理

Linux Threads 使用互斥锁和条件变量保证由 pthread_once()指定的函数执行且仅执行一次，而 once_control 则表征是否执行过。如果 once_control 的初值不是 PTHREAD_ONCE_INIT（Linux Threads 定义为 0），pthread_once()的行为就会不正常。在 Linux Threads 中，实际”一次性函数”的执行状态有三种：NEVER（0）、IN_PROGRESS（1）、DONE（2），如果 once 初值设为 1，则由于所有 pthread_once()都必须等待其中一个激发”已执行一次”信号，因此所有 pthread_once()都会陷入永久的等待中；如果设为 2，则表示该函数已执行过一次，从而所有 pthread_once()都会立即返回 0。

int pthread_once(pthread_once_t *once_control,void(*init_routine)(void));

参数：

once_control　　控制变量

init_routine　　初始化函数

返回值：

若成功返回 0，若失败返回错误编号。

类型为 pthread_once_t 的变量是一个控制变量。控制变量必须使用 PTHREAD_ONCE_INIT 宏静态地初始化。

pthread_once 函数首先检查控制变量，判断是否已经完成初始化，如果完成就简单地返回；否则，pthread_once 调用初始化函数，并且记录下初始化被完成。如果在一个线程初始时，另外的线程调用 pthread_once，则调用线程等待，直到那个现成完成初始话返回。

valgrind 的安装与使用

在线安装：sudo apt install valgrind

程序编译时候，添加的-g 选项：g++ -g -o memcheck memcheck.cpp

最常用的命令格式：valgrind [options] prog-and-args [options]

-tool=<name> 最常用的选项。运行 valgrind 中名为 toolname 的工具。默认 memcheck。

memcheck ------> 这是 valgrind 应用最广泛的工具，一个重量级的内存检查器，能够发现开发中绝大多数内存错误使用情况，比如：使用未初始化的内存，使用已经释放了的内存，内存访问越界等。

callgrind ------> 它主要用来检查程序中函数调用过程中出现的问题。
cachegrind ------> 它主要用来检查程序中缓存使用出现的问题。
helgrind ------> 它主要用来检查多线程程序中出现的竞争问题。
massif ------> 它主要用来检查程序中堆栈使用中出现的问题。
extension ------> 可以利用 core 提供的功能，自己编写特定的内存调试工具

-h –help 显示帮助信息。
-version 显示 valgrind 内核的版本，每个工具都有各自的版本。
-q –quiet 安静地运行，只打印错误信息。
-v –verbose 更详细的信息, 增加错误数统计。
-trace-children=no|yes 跟踪子线程? [no]
-track-fds=no|yes 跟踪打开的文件描述？[no]
-time-stamp=no|yes 增加时间戳到 LOG 信息? [no]
-log-fd=<number> 输出 LOG 到描述符文件 [2=stderr]
-log-file=<file> 将输出的信息写入到 filename.PID 的文件里，PID 是运行程序的进行 ID
-log-file-exactly=<file> 输出 LOG 信息到 file
-log-file-qualifier=<VAR> 取得环境变量的值来做为输出信息的文件名。 [none]
-log-socket=ipaddr:port 输出 LOG 到 socket ，ipaddr:port

LOG 信息输出
-xml=yes 将信息以 xml 格式输出，只有 memcheck 可用
-num-callers=<number> show <number> callers in stack traces [12]
-error-limit=no|yes 如果太多错误，则停止显示新错误? [yes]
-error-exitcode=<number> 如果发现错误则返回错误代码 [0=disable]
-db-attach=no|yes 当出现错误，valgrind 会自动启动调试器 gdb。[no]
-db-command=<command> 启动调试器的命令行选项[gdb -nw %f %p]

适用于 Memcheck 工具的相关选项：
-leak-check=no|summary|full 要求对 leak 给出详细信息? [summary]
-leak-resolution=low|med|high how much bt merging in leak check [low]
-show-reachable=no|yes show reachable blocks in leak check? [no]

最常用的命令格式举例：

valgrind --tool=memcheck --leak-check=full ./memcheck

“definitely lost” [ˈdefɪnətli]：确认丢失。程序中存在内存泄露，应尽快修复。当程序结束时如果一块动态分配的内存没有被释放且通过程序内的指针变量均无法访问这块内存则会报这个错误。

“indirectly lost”：间接丢失。当使用了含有指针成员的类或结构时可能会报这个错误。这类错误无需直接修复，他们总是与”definitely lost”一起出现，只要修复”definitely lost”即可。例子可参考我的例程。

“possibly lost”：可能丢失。大多数情况下应视为与”definitely lost”一样需要尽快修复，除非你的程序让一个指针指向一块动态分配的内存（但不是这块内存起始地址），然后通过运算得到这块内存起始地址，再释放它。例子可参考我的例程。当程序结束时如果一块动态分配的内存没有被释放且通过程序内的指针变量均无法访问这块内存的起始地址，但可以访问其中的某一部分数据，则会报这个错误。

“still reachable”：可以访问，未丢失但也未释放。如果程序是正常结束的，那么它可能不会造成程序崩溃，但长时间运行有可能耗尽系统资源，因此笔者建议修复它。如果程序是崩溃（如访问非法的地址而崩溃）而非正常结束的，则应当暂时忽略它，先修复导致程序崩溃的错误，然后重新检测。

“suppressed”[səˈprest]：已被解决。出现了内存泄露但系统自动处理了。可以无视这类错误。这类错误我没能用例程触发，看官方的解释也不太清楚是操作系统处理的还是 valgrind，也没有遇到过。所以无视他吧~