**Linux性能评测工具之一：gprof**

**1**简介

改进应用程序的性能是一项非常耗时耗力的工作，但是究竟程序中是哪些函数消耗掉了大部分执行时间，这通常都不是非常明显的。GNU 编译器工具包所提供了一种剖析工具 GNU profiler（gprof）。gprof 可以为 Linux平台上的程序精确分析性能瓶颈。gprof精确地给出函数被调用的时间和次数，给出函数调用关系。gprof 用户手册网站 <http://sourceware.org/binutils/docs-2.17/gprof/index.html>

**2**功能

Gprof 是GNU gnu binutils工具之一，默认情况下linux系统当中都带有这个工具。gprof实际上只是一个用于读取profile结果文件的工具。gprof采用混合方法来收集程序的统计信息，他使用检测方法，在编译过程中在函数入口处插入计数器用于收集每个函数的被调用情况和被调用次数；也使用采样方法，在运行时按一定间隔去检查程序计数器并在分析时找出程序计数器对应的函数来统计函数占用的时间。

Gprof具有以下优缺点：

1） 优点：

a) GNU工具，人手一个；

b) 混合方法采集信息。

2） 缺点：

a) 需要编译选项支持：

i. 使用gcc/cc编译和链接时需要加入-pg选项

ii. 使用ld链接时需要用/lib/gcrt0.o代替crt0.o作为第一个input文件

iii.如果要调试libc库需要使用-lc\_p代替-lc参数

b)调试多线程程序只能统计主线程的信息（所以不能用于kingbase）。

1. 可以显示“flat profile”，包括每个函数的调用次数，每个函数消耗的处理器时间，

2. 可以显示“Call graph”，包括函数的调用关系，每个函数调用花费了多少时间。

3. 可以显示“注释的源代码”－－是程序源代码的一个复本，标记有程序中每行代码的执行次数。

**3**原理

通过在编译和链接程序的时候（使用 -pg 编译和链接选项），gcc 在你应用程序的每个函数中都加入了一个名为mcount ( or  "\_mcount"  , or  "\_\_mcount" , 依赖于编译器或操作系统)的函数，也就是说你的应用程序里的每一个函数都会调用mcount, 而mcount 会在内存中保存一张函数调用图，并通过函数调用堆栈的形式查找子函数和父函数的地址。这张调用图也保存了所有与函数相关的调用时间，调用次数等等的所有信息。

**4**使用流程

1. 编译程序

在编译和链接时 加上-pg选项。一般我们可以加在 makefile 中。

使用gcc/cc编译和链接时需要加入-pg选项

使用ld链接时需要用/lib/gcrt0.o代替crt0.o作为第一个input文件

如果要调试libc库需要使用-lc\_p代替-lc参数

2. 运行程序生成统计信息

正常运行编译好的程序，程序正常结束后会在当前目录生成统计信息文件gmon.out。

程序必须正常退出（调用exit或从main中返回）才能生成统计信息。

当前目录下如果有gmon.out文件，内容将被本次运行生成的统计信息覆盖，多次运行统一程序请将前一次的gmon.out改名。执行编译的二进制程序。执行参数和方式同以前。

3. 结束进程。这时 gmon.out 会再次被刷新。

5. 使用gprof查看统计结果gmon.out 文件。

**5 命令格式**

gprof options [executable-file [profile-data-files...]] [> outfile]

常用参数介绍：

symspec表示需要加入或排除的函数名，和gdb指定断点时的格式相同。

参数说明

-b 不再输出统计图表中每个字段的详细描述。

-p 只输出函数的调用图（Call graph的那部分信息）。

-q 只输出函数的时间消耗列表。

-e Name 不再输出函数Name 及其子函数的调用图（除非它们有未被限制的其它父函数）。可以给定多个 -e 标志。一个 -e 标志只能指定一个函数。

-E Name 不再输出函数Name 及其子函数的调用图，此标志类似于 -e 标志，但它在总时间和百分比时间的计算中排除了由函数Name 及其子函数所用的时间。

-f Name 输出函数Name 及其子函数的调用图。可以指定多个 -f 标志。一个 -f 标志只能指定一个函数。

-F Name 输出函数Name 及其子函数的调用图，它类似于 -f 标志，但它在总时间和百分比时间计算中仅使用所打印的例程的时间。可以指定多个 -F 标志。一个 -F 标志只能指定一个函数。-F 标志覆盖 -E 标志。

-z 显示使用次数为零的例程（按照调用计数和累积时间计算）。

例如： gprof –b 二进制程序 gmon.out >report.txt

**6**报告说明

Gprof 产生的信息解释：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| %time | Cumulative  seconds | Self  Seconds | Calls | Self  TS/call | Total  TS/call | name |
| 该函数消耗时间占程序所有时间百分比 | 程序的累积执行时间  （只是包括gprof能够监控到的函数） | 该函数本身执行时间  （所有被调用次数的合共时间） | 函数被调用次数 | 函数平均执行时间  （不包括被调用时间）  （函数的单次执行时间） | 函数平均执行时间  （包括被调用时间）  （函数的单次执行时间） | 函数名 |

Call Graph 的字段含义：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Index | %time | Self | Children | Called | Name |
| 索引值 | 函数消耗时间占所有时间百分比 | 函数本身执行时间 | 执行子函数所用时间 | 被调用次数 | 函数名 |

注意：

程序的累积执行时间只是包括gprof能够监控到的函数。工作在内核态的函数和没有加-pg编译的第三方库函数是无法被gprof能够监控到的，（如sleep（）等）

Gprof 的具体参数可以 通过 man gprof 查询。

**7**共享库的支持

对于代码剖析的支持是由编译器增加的，因此如果希望从共享库中获得剖析信息，就需要使用 -pg 来编译这些库。提供已经启用代码剖析支持而编译的 C 库版本（libc\_p.a）。

如果需要分析系统函数（如libc库），可以用 –lc\_p替换-lc。这样程序会链接libc\_p.so或libc\_p.a。这非常重要，因为只有这样才能监控到底层的c库函数的执行时间，（例如memcpy()，memset()，sprintf()等）。

gcc example1.c –pg -lc\_p -o example1

注意要用ldd ./example | grep libc来查看程序链接的是libc.so还是libc\_p.so

**8**用户时间与内核时间

gprof 的最大缺陷：它只能分析应用程序在运行过程中所消耗掉的用户时间，无法得到程序内核空间的运行时间。通常来说，应用程序在运行时既要花费一些时间来运行用户代码，也要花费一些时间来运行 “系统代码”，例如内核系统调用sleep()。

有一个方法可以查看应用程序的运行时间组成，在 time 命令下面执行程序。这个命令会显示一个应用程序的实际运行时间、用户空间运行时间、内核空间运行时间。

如 time ./program

输出：

real    2m30.295s

user    0m0.000s

sys     0m0.004s

**9**注意事项

1. g++在编译和链接两个过程，都要使用-pg选项。

2. 只能使用静态连接libc库，否则在初始化\*.so之前就调用profile代码会引起“segmentation fault”，解决办法是编译时加上-static-libgcc或-static。

3. 如果不用g++而使用ld直接链接程序，要加上链接文件/lib/gcrt0.o，如ld -o myprog /lib/gcrt0.o myprog.o utils.o -lc\_p。也可能是gcrt1.o

4. 要监控到第三方库函数的执行时间，第三方库也必须是添加 –pg 选项编译的。

5. gprof只能分析应用程序所消耗掉的用户时间.

6. 程序不能以demon方式运行。否则采集不到时间。（可采集到调用次数）

7. 首先使用 time 来运行程序从而判断 gprof 是否能产生有用信息是个好方法。

8. 如果 gprof 不适合您的剖析需要，那么还有其他一些工具可以克服 gprof 部分缺陷，包括 OProfile 和 Sysprof。

9. gprof对于代码大部分是用户空间的CPU密集型的程序用处明显。对于大部分时间运行在内核空间或者由于外部因素（例如操作系统的 I/O 子系统过载）而运行得非常慢的程序难以进行优化。

10. gprof 不支持多线程应用，多线程下只能采集主线程性能数据。原因是gprof采用ITIMER\_PROF信号，在多线程内只有主线程才能响应该信号。但是有一个简单的方法可以解决这一问题：<http://sam.zoy.org/writings/programming/gprof.html>

11. gprof只能在程序正常结束退出之后才能生成报告（gmon.out）。

a) 原因： gprof通过在atexit()里注册了一个函数来产生结果信息，任何非正常退出都不会执行atexit()的动作，所以不会产生gmon.out文件。

b) 程序可从main函数中正常退出，或者通过系统调用exit()函数退出。

**10**多线程应用

gprof 不支持多线程应用，多线程下只能采集主线程性能数据。原因是gprof采用ITIMER\_PROF信号，在多线程内只有主线程才能响应该信号。

采用什么方法才能够分析所有线程呢？关键是能够让各个线程都响应ITIMER\_PROF信号。可以通过桩子函数来实现，重写pthread\_create函数。

//////////////////// gprof-helper.c////////////////////////////

#define \_GNU\_SOURCE

#include <sys/time.h>

#include <stdio.h>

#include <stdlib.h>

#include <dlfcn.h>

#include <pthread.h>

static void \* wrapper\_routine(void \*);

/\* Original pthread function \*/

static int (\*pthread\_create\_orig)(pthread\_t \*\_\_restrict,

                                  \_\_const pthread\_attr\_t \*\_\_restrict,

                                  void \*(\*)(void \*),

                                  void \*\_\_restrict) = NULL;

/\* Library initialization function \*/

void wooinit(void) \_\_attribute\_\_((constructor));

void wooinit(void)

{

    pthread\_create\_orig = dlsym(RTLD\_NEXT, "pthread\_create");

    fprintf(stderr, "pthreads: using profiling hooks for gprof/n");

    if(pthread\_create\_orig == NULL)

    {

        char \*error = dlerror();

        if(error == NULL)

        {

            error = "pthread\_create is NULL";

        }

        fprintf(stderr, "%s/n", error);

        exit(EXIT\_FAILURE);

    }

}

/\* Our data structure passed to the wrapper \*/

typedef struct wrapper\_s

{

    void \* (\*start\_routine)(void \*);

    void \* arg;

    pthread\_mutex\_t lock;

    pthread\_cond\_t  wait;

    struct itimerval itimer;

} wrapper\_t;

/\* The wrapper function in charge for setting the itimer value \*/

static void \* wrapper\_routine(void \* data)

{

    /\* Put user data in thread-local variables \*/

    void \* (\*start\_routine)(void \*) = ((wrapper\_t\*)data)->;start\_routine;

    void \* arg = ((wrapper\_t\*)data)->;arg;

    /\* Set the profile timer value \*/

    setitimer(ITIMER\_PROF, &((wrapper\_t\*)data)->;itimer, NULL);

    /\* Tell the calling thread that we don't need its data anymore \*/

    pthread\_mutex\_lock(&((wrapper\_t\*)data)->;lock);

    pthread\_cond\_signal(&((wrapper\_t\*)data)->;wait);

    pthread\_mutex\_unlock(&((wrapper\_t\*)data)->;lock);

    /\* Call the real function \*/

    return start\_routine(arg);

}

/\* Our wrapper function for the real pthread\_create() \*/

int pthread\_create(pthread\_t \*\_\_restrict thread,

                   \_\_const pthread\_attr\_t \*\_\_restrict attr,

                   void \* (\*start\_routine)(void \*),

                   void \*\_\_restrict arg)

{

    wrapper\_t wrapper\_data;

    int i\_return;

    /\* Initialize the wrapper structure \*/

    wrapper\_data.start\_routine = start\_routine;

    wrapper\_data.arg = arg;

    getitimer(ITIMER\_PROF, &wrapper\_data.itimer);

    pthread\_cond\_init(&wrapper\_data.wait, NULL);

    pthread\_mutex\_init(&wrapper\_data.lock, NULL);

    pthread\_mutex\_lock(&wrapper\_data.lock);

    /\* The real pthread\_create call \*/

    i\_return = pthread\_create\_orig(thread,

                                   attr,

                                   &wrapper\_routine,

                                   &wrapper\_data);

    /\* If the thread was successfully spawned, wait for the data

     \* to be released \*/

    if(i\_return == 0)

    {

        pthread\_cond\_wait(&wrapper\_data.wait, &wrapper\_data.lock);

    }

    pthread\_mutex\_unlock(&wrapper\_data.lock);

    pthread\_mutex\_destroy(&wrapper\_data.lock);

    pthread\_cond\_destroy(&wrapper\_data.wait);

    return i\_return;

}

///////////////////

然后编译成动态库 gcc -shared -fPIC gprof-helper.c -o gprof-helper.so -lpthread -ldl

使用例子：

/////////////////////a.c/////////////////////////////

#include <stdio.h>;

#include <stdlib.h>;

#include <unistd.h>;

#include <pthread.h>;

#include <string.h>;

void fun1();

void fun2();

void\* fun(void \* argv);

int main()

{

        int i =0;

        int id;

        pthread\_t    thread[100];

        for(i =0 ;i< 100; i++)

        {

                id = pthread\_create(&thread[i], NULL, fun, NULL);

                printf("thread =%d/n",i);

        }

        printf("dsfsd/n");

        return 0;

}

void\*  fun(void \* argv)

{

        fun1();

        fun2();

        return NULL;

}

void fun1()

{

        int i = 0;

        while(i<100)

        {

                i++;

                printf("fun1/n");

        }

}

void fun2()

{

        int i = 0;

        int b;

        while(i<50)

        {

                i++;

                printf("fun2/n");

                //b+=i;

        }

}

///////////////

gcc -pg a.c  gprof-helper.so

运行程序:

./a.out

分析gmon.out:

gprof -b a.out gmon.out