Contents

[1. 检查Linux所使用的资源： 1](#_Toc6403820)

[2. 文件系统测试工具： 1](#_Toc6403821)

[2.1 Stress\_test\_for\_filesystem 1](#_Toc6403822)

[2.2 iozone 1](#_Toc6403823)

[2.2.1 测试说明： 2](#_Toc6403824)

[2.2.2 在X86-Linux上测试 2](#_Toc6403825)

[2.2.3 在ARM-Linux上测试 2](#_Toc6403826)

[3. CPU测试 3](#_Toc6403827)

[3.1 nbench 3](#_Toc6403828)

[3.1.1 X86上测试： 3](#_Toc6403829)

[3.1.2 在arm上测试： 4](#_Toc6403830)

# 检查Linux所使用的资源：

## top命令

## free命令：

# free

total used free shared buffers cached

Mem: 1027284 342516 684768 0 8960 103456

-/+ buffers/cache: 230100 797184

Swap: 0 0 0

另外如果以Mbyte为单位，显示Memory信息：（UI24）

# free -m

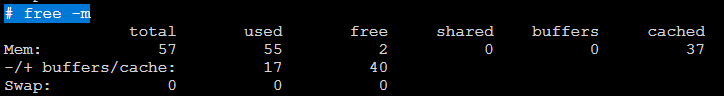
total used free shared buffers cached

Mem: 1003 334 668 0 8 101

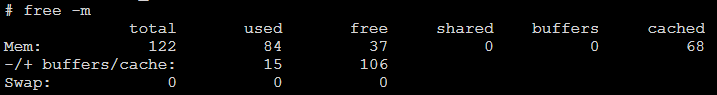
-/+ buffers/cache: 224 778

Swap: 0 0 0

(dvi)



（cpi2000）



解释一下Linux上free命令的输出。

　　下面是free的运行结果，一共有4行。为了方便说明，我加上了列号。这样可以把free的输出看成一个二维数组FO(Free Output)。例如：

* FO[2][1] = 24677460
* FO[3][2] = 10321516

                   1          2          3          4          5          6  
1              total       used       free     shared    buffers     cached  
2 Mem:      24677460   23276064    1401396          0     870540   12084008  
3 -/+ buffers/cache:   10321516   14355944  
4 Swap:     25151484     224188   24927296

free的输出一共有四行，第四行为交换区的信息，分别是交换的总量（total），使用量（used）和有多少空闲的交换区（free），这个比较清楚，不说太多。

free输出地第二行和第三行是比较让人迷惑的。这两行都是说明内存使用情况的。第一列是总量（total），第二列是使用量（used），第三列是可用量（free）

1. **第一行解释：**

　　第一行的输出时从操作系统（OS）来看的。也就是说，从OS的角度来看，计算机上一共有:

* 24677460KB（缺省时free的单位为KB）物理内存，即FO[2][1]；
* 在这些物理内存中有23276064KB（即FO[2][2]）被使用了；
* 还用1401396KB（即FO[2][3]）是可用的；

这里得到第一个等式：

* FO[2][1] = FO[2][2] + FO[2][3]

FO[2][4]表示被几个进程共享的内存的，现在已经deprecated，其值总是0（当然在一些系统上也可能不是0，主要取决于free命令是怎么实现的）。

FO[2][5]表示被OS buffer住的内存。FO[2][6]表示被OS cache的内存。在有些时候buffer和cache这两个词经常混用。不过在一些比较低层的软件里是要区分这两个词的，看老外的洋文:

* A buffer is something that has yet to be "written" to disk.
* A cache is something that has been "read" from the disk and stored for later use.

也就是说buffer是用于存放要输出到disk（块设备）的数据的，而cache是存放从disk上读出的数据。这二者是为了提高IO性能的，并由OS管理。

Linux和其他成熟的操作系统（例如windows），为了提高IO read的性能，总是要多cache一些数据，这也就是为什么FO[2][6]（cached memory）比较大，而FO[2][3]比较小的原因。我们可以做一个简单的测试:

* 释放掉被系统cache占用的数据；

echo 3>/proc/sys/vm/drop\_caches

* 读一个大文件，并记录时间；
* 关闭该文件；
* 重读这个大文件，并记录时间；

第二次读应该比第一次快很多。原来我做过一个BerkeleyDB的读操作，大概要读5G的文件，几千万条记录。在我的环境上，第二次读比第一次大概可以快9倍左右。

1. **第二行解释：**

free输出的第二行是从一个应用程序的角度看系统内存的使用情况。

* 对于FO[3][2]，即-buffers/cache，表示一个应用程序认为系统被用掉多少内存；
* 对于FO[3][3]，即+buffers/cache，表示一个应用程序认为系统还有多少内存；

因为被系统cache和buffer占用的内存可以被快速回收，所以通常FO[3][3]比FO[2][3]会大很多。

这里还用两个等式：

* FO[3][2] = FO[2][2] - FO[2][5] - FO[2][6]
* FO[3][3] = FO[2][3] + FO[2][5] + FO[2][6]

这二者都不难理解。

# 文件系统测试工具：

## Stress\_test\_for\_filesystem

这是一个很简单的脚本测试程序，其实质就是通过tar进行反复得打包和解包操作，并进行一定的校验，确保打包和解包能正常进行。

由于打包和解包中的文件非常多，因此比较能考验文件系统的稳定性。

使用很简单：

./test.sh /mnt/disk2/test\_dir 200

第一个参数是测试的目录，可以选择当前目录，如： ./test.sh . 200

也可以选择其他目录，如: ./test.sh /mnt/share/git/. 200

第二个参数是测试时长，单位为秒。

## iozone

iozone也是一个文件系统的性能测试程序。

下载源代码：

wget http://www.iozone.org/src/current/iozone3\_487.tar

tar -xvf iozone3\_487.tar

### 测试说明：

参数：-i用于指明测试某些功能。

说明：iozone有很多稀奇古怪的文件读写测试，比如倒序读文件，乱序读文件，其实很多我们都用不到。具体参考下面解释：

Used to specify which tests to run. (0=write/rewrite, 1=read/re-read, 2=random-read/write, 3=Read-backwards, 4=Re-write-record, 5=stride-read, 6=fwrite/re-fwrite, 7=fread/Re-fread, 8=mixed workload, 9=pwrite/Re-pwrite, 10=pread/Re-pread, 11=pwritev/Re-pwritev, 12=preadv/Re-preadv). One will always need to specify 0 so that any of the following tests will have a file to measure. -i # -i # -i # is also supported so that one may select more than one test.

参数：-a 自动测试，测试解释如下：

Used to select full automatic mode. Produces output that covers all tested file operations for record sizes of 4k to 16M for file sizes of 64k to 512M.

参数：-f 用于指明测试文件（不是测试目录），iozone将对该文件进行创建/读/写/删除等操作。

参数：-n 用于指明测试文件的最小size。比如： 100k / 32m / 1g

参数：-g 用于指明测试文件的最大size，比如: 100k / 32m / 1g

参数: -Rb 用于指明参数一个测试报告文件。

### 在X86-Linux上测试

* **编译：**

cd iozone3\_487/src/current

make linux-ia64

编译结果： iozone

* **测试：**

测试临时文件：/mnt/temp.dat, 测试结果：./iozone.xls,测试大小：512M – 4G Bytes， 测试内容：read (-i 0),write (-i 1),和Strided Read(-i 5)

#./iozone -a -n 512m -g 4g -i 0 -i 1 -i 5 -f /mnt/temp.dat -Rb ./iozone.xls

### 在ARM-Linux上测试

* **编译：**

修改makefile，将编译器改成：

CC = $(CROSS\_COMPILE)gcc

C89 = $(CROSS\_COMPILE)gcc

GCC = $(CROSS\_COMPILE)gcc

然后直接用命令行进行编译：

make linux-arm ARCH=arm CROSS\_COMPILE=/mnt/disk2/git/dvi/software/Atmel/buildroot-2017.02.5/output/host/usr/bin/arm-linux- LD\_LIBRARY\_PATH=/mnt/disk2/git/dvi/software/Atmel/buildroot-2017.02.5/output/host/usr/lib

* **测试：**

测试临时文件：/root/temp.dat, 测试结果：./iozone.xls,测试大小：10M – 100M Bytes， 测试内容：read (-i 0),write (-i 1),和Strided Read(-i 5)

./iozone -a -n 10m -g 100m -i 0 -i 1 -i 5 -f /root/temp.dat -Rb ./iozone.xls

# CPU测试

## nbench

在<http://slackbuilds.org/repository/13.37/system/nbench/>中可以下载nbench，或者：

下载源代码：<https://fossies.org/linux/misc/old/nbench-byte-2.2.3.tar.gz/>

### X86上测试：

* **编译：**

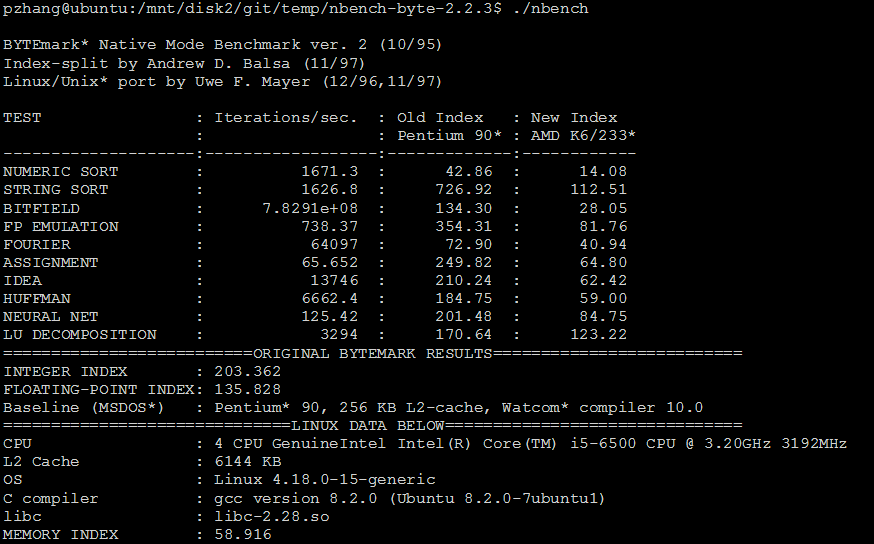
我们什么都不用改，直接在X86上运行make就可以编译了：

cd nbench-byte-2.2.3

make

* **运行：**

**./nbench**



### 在arm上测试：

* **编译：**

**我们需要修改makefile中的编译器，如下：**

CC = $(CROSS\_COMPILE)gcc

然后运行如下脚本就可以开始编译：

make ARCH=arm CROSS\_COMPILE=/mnt/disk2/git/dvi/software/Atmel/buildroot-2017.02.5/output/host/usr/bin/arm-linux- LD\_LIBRARY\_PATH=/mnt/disk2/git/dvi/software/Atmel/buildroot-2017.02.5/output/host/usr/lib

* **运行：**

./nbench

测试结果：

