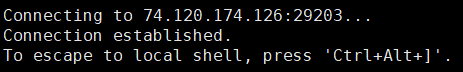
**实验环境为自己的服务器，Centos2.6.32**



**实验一 Shell编程**

实验任务：

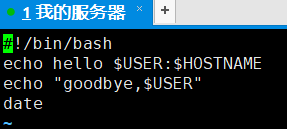
1. 编写shell脚本程序获取环境变量并输出
2. 编写包含控制语句的shell脚本程序

实验原理与方法

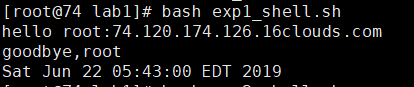
1. 简单的shell脚本程序类似批处理程序，只执行一些命令。

Greet程序，显示当前登录用户名以及域名，并显示当前时间。

程序清单：

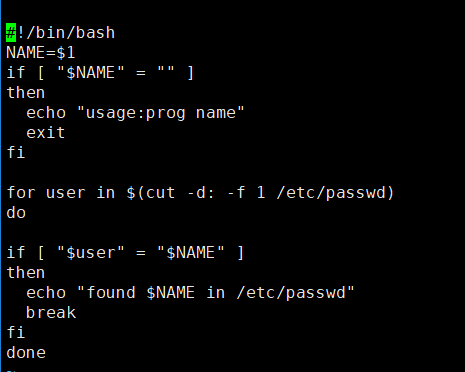


第一行#！表示运行/bin/bash ,并让/bin/bash执行程序内容。



1. Shell脚本也有三种控制结构，下面check程序是一个判断指定用户名是否在系统的passwd文件中的shell脚本，首先判断用户是否指定了一个要查找的用户名参数，然后循环在”/etc/passwd”文件中查找，最后提示是否找到。

程序清单



分析运行结果，



同学自己查一下，$1是什么意思，然后给每条语句加注释。

$1表示命令行参数的第二个参数

**实验二 gcc与make使用**

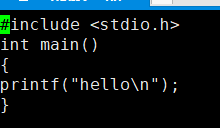
实验任务

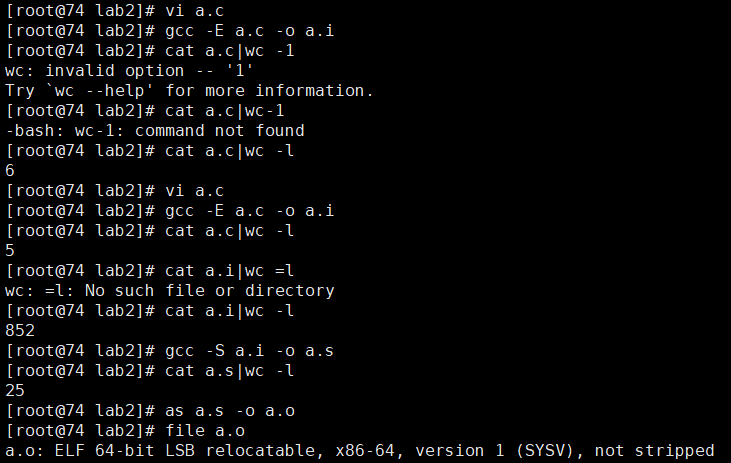
1. 会用gcc
2. 为自己的工程编写makefile文件

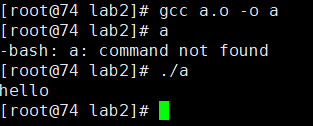
**实验内容：**

1、gcc执行过程示例

示例代码 a.c：







GCC编译简单例子

编写如下代码：

#include <stdio.h>

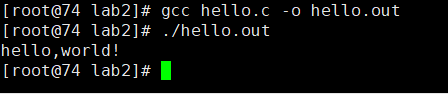
int main()

{

printf("hello,world!\n");

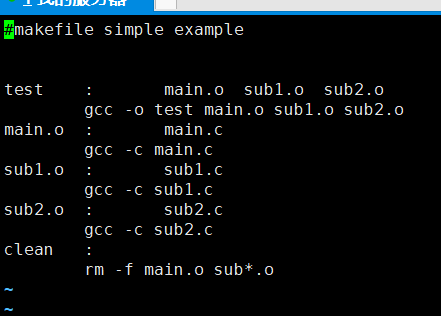
}

执行情况如下：

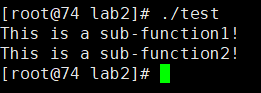


2、makefile例

makefile文件



执行make，默认在当前路径下查找makefile文件并执行。如下图所示

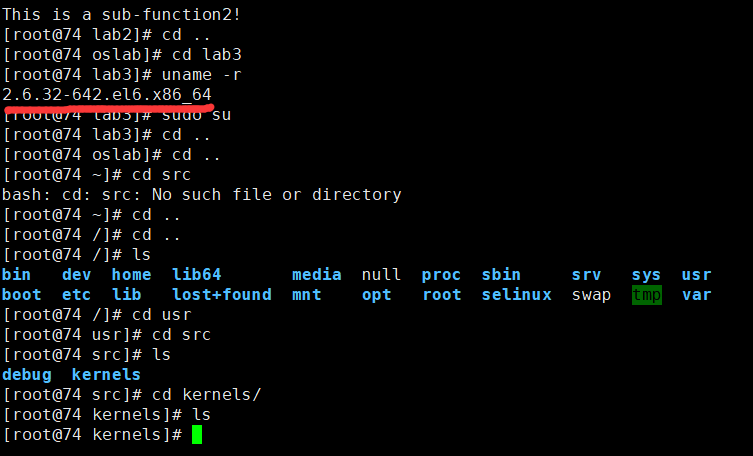


**实验三 安装、配置、编译Linux内核**

1、下载安装、配置

由于Ubuntu不带自己的内核源码，需要下载安装。

Step1：首先要获取当前系统的内核版本



Step 2：下载Linux内核3.19.3源码包并解压



感叹一下国外服务器的网速hhhh

解压：

没有…只好下载

但是这时候想起来，我是centos 啊！编译一个乌班图内核真的没问题吗！！！

而且我服务器上还有不少其他东西…所以不想作死了

老师这个实验我不做了，反正您也能看到这个实验就是照着敲命令，也就是呢么回事，我就不作死了，谢谢老师！

Step 5：重启，使用新内核

使用命令# uname -r 查看内核版本



不敢不敢……

**实验四 观察 Linux 行为**

**1、实验目的**

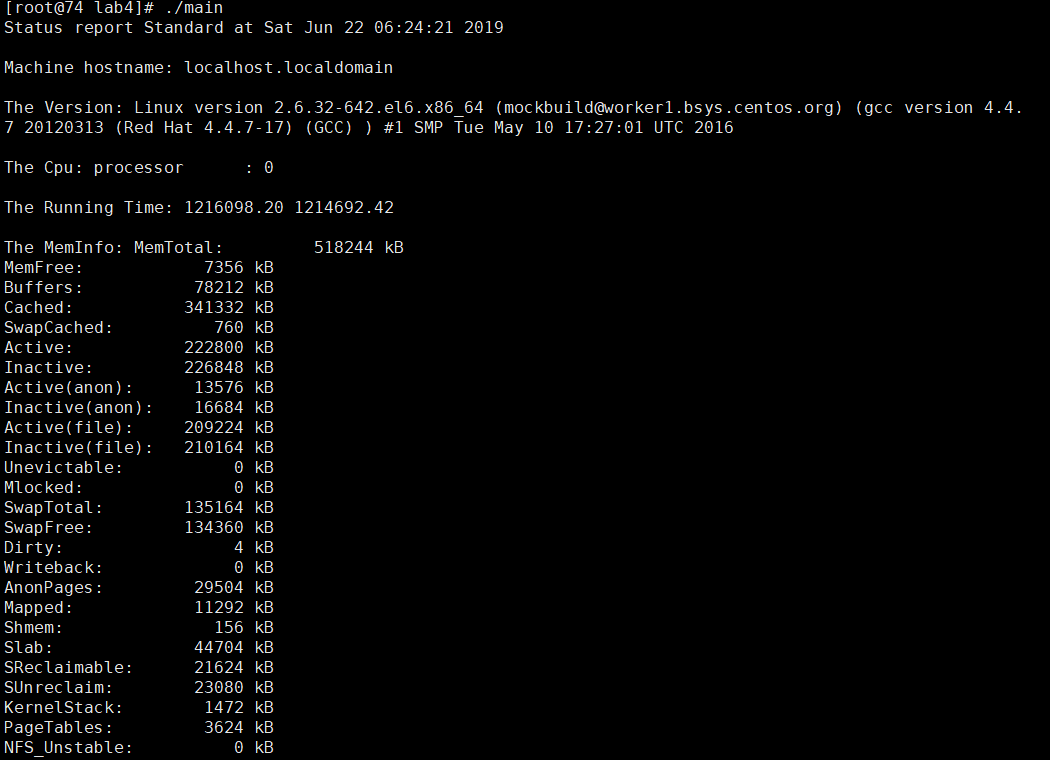
学习 linux 内核、进程、存储和其他资源的一些重要特性。通过使用/proc 文件系统接口,编写一个程序检查反映机器平衡负载、进程资源利用率方面的各种内核值, 学会使用/proc文件系统这种内核状态检查机制。

**2、实验内容**

编写一个默认版本的程序通过检查内核状态报告 Linux 内核行为。程序应该在 stdout上打印以下值：

1. CPU 类型和型号；
2. 所使用的 Linux 内核版本；
3. 从系统最后一次启动以来已经经历了多长时间（天，小时和分钟）；
4. 总共有多少 CPU 时间执行在用户态，系统态，空闲态；
5. 配置内存数量；当前可用内存数，磁盘读写请求数；
6. 内核上下文转换数；

结果：



**实验五 内核的定时机制实验**

**1 实验目的**

本实验是练习怎样编写调用内核的时间测量功能为应用程序测量和精确定时。通过这个实验我们可以进一步理解 Linux 内核的定时机制及其数据结构以及怎样从用户空间去访问内核空间的时间数据。

**2 实验内容**

A 在用户态编写一个程序，该程序设定一个定时器，在时间到期的时候做出某种可观察的响应（方法不限）。

B 用定时器 ITIMER\_REAL实现gettimeofday的功能。使其一秒钟产生一个信号，计算已经过的秒数。

设计思路：设置定时器 ITIMER\_REAL 间隔为一秒钟。并为计时到时设定信号处理程序，即singal(SIGALRM,…)，使其输出当前所记时间。

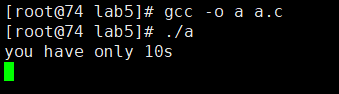
**C** 记录一个进程运行时所占用的real time, cpu time, user time, kernel time。

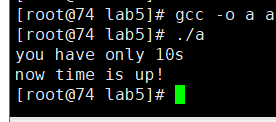
);

**实验程序：**

**代码与示例代码完全一致，以下为运行结果。**

A程序：

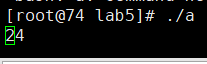




B程序：

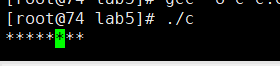






C程序参考了博客：





**实验六 内核模块编程**

**1实验目的**

模块是 Linux 系统的一种特有机制，可用以动态扩展操作系统内核功能。编写实现某些特定功能的模块，将其作为内核的一部分在管态下运行。本实验通过内核模块编程在/porc文件系统中实现系统时钟的读操作接口。

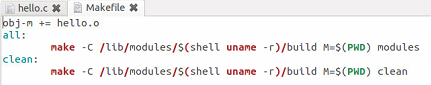
**4 实验内容**

4.1 一个简单的内核模块(实现)

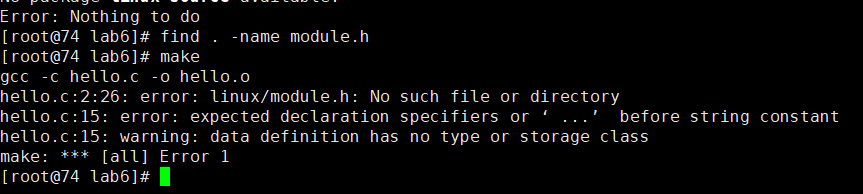
1) 程序清单：

3) 运行步骤

(1) 编写makefile

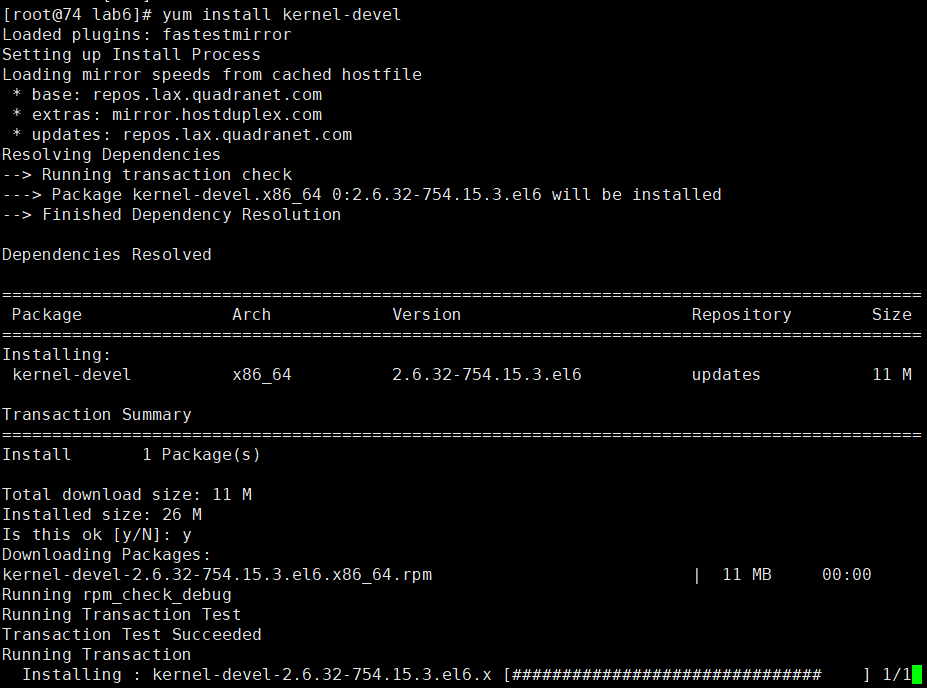


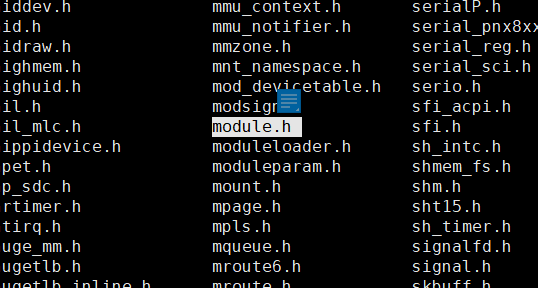
(2) makefile编译

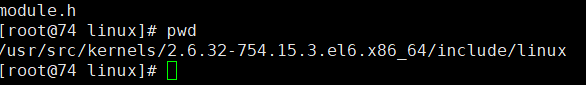


中奖了，发现没有linux/module.h

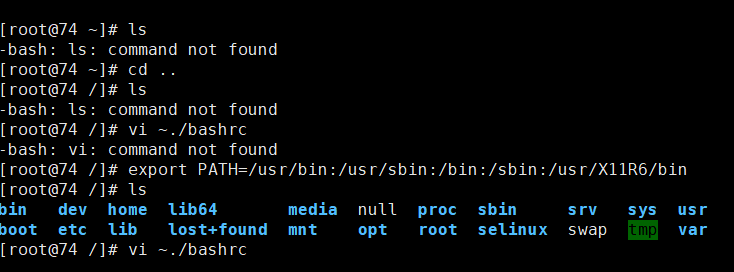
于是乎下载





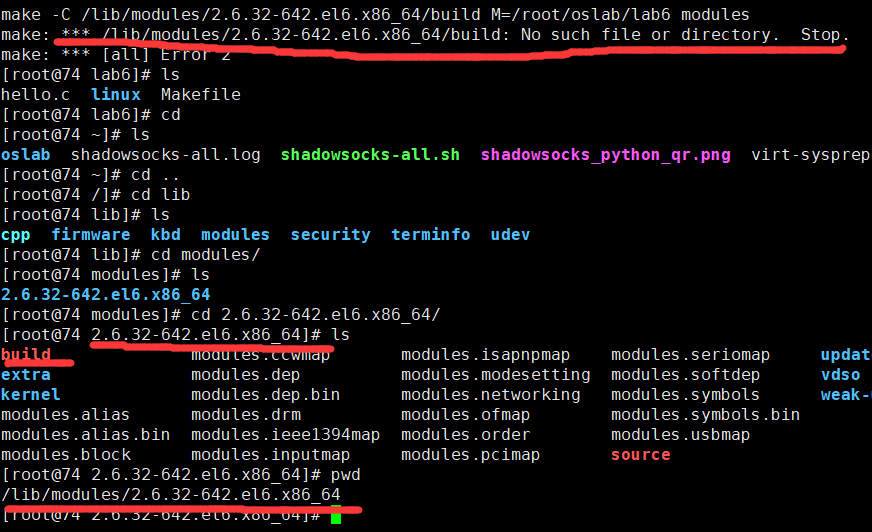


可以看到这个目录下已经有了，接下来修改环境变量，让所有目录都能访问。

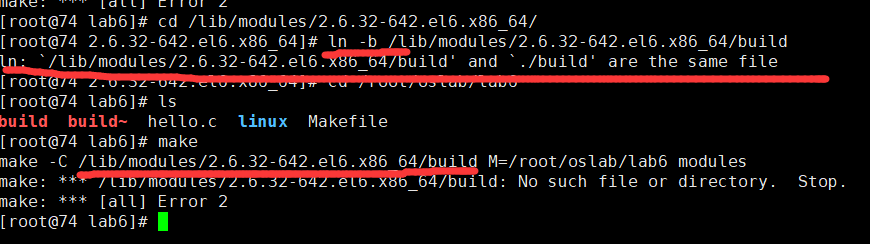


果然一不小心就把ls vi都给搞炸了，只好重新恢复一下。

恢复后还是没有…build也显示没有，但下图清清楚楚是有的



重新加软链



还是没有用…于是…放弃了

总结一下，这个真的不知道为什么有问题，明明是有但就是说找不到文件夹。真的不知道怎么做了。希望老师理解（我感觉也可能是我用的是centos的缘故吗？）

(3) insmod命令加载模块到内核，并用dmesg命令显示结果





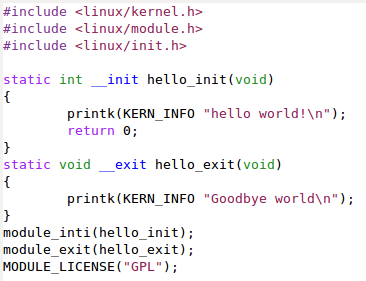
(4) rmmod命令卸载模块，并用dmesg命令显示结果





4.2 一个简单的标准的内核模块

1) 程序清单



2) 说明：

(1) 例2为了和例1对比，用了自己定义的入口和出口，并用宏module\_init()与module\_exit做了指定。同样实现了hello world输出。

(2) static int \_\_init hello\_init(void)

static void \_\_exit hello\_exit(void)

static声明，因为这种函数在特定文件之外没有其它意义

\_\_init标记, 该函数只在初始化期间使用。模块装载后，将该函数占用的内存空间释放

\_\_exit标记 该代码仅用于模块卸载。

注意：\_\_init和\_\_exit前面是两个\_，而不是一个\_.

(3) init/exit

宏：module\_init/module\_exit

声明模块初始化及清除函数所在的位置

装载和卸载模块时，内核可以自动找到相应的函数

module\_init(hello\_init);

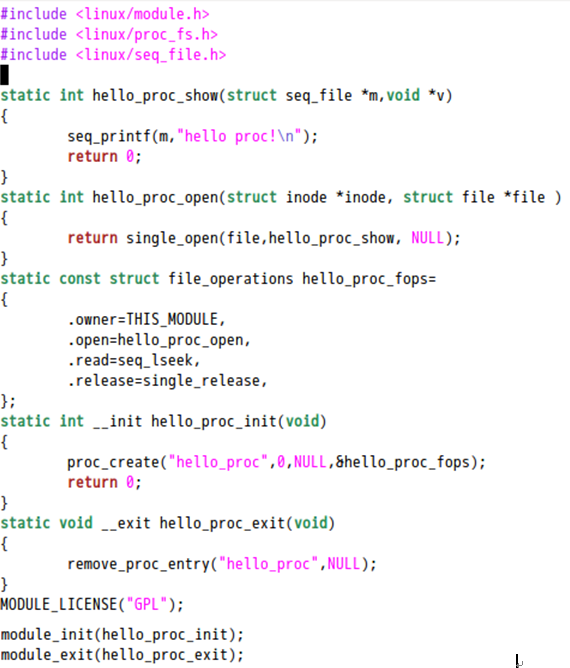
module\_exit(hello\_exit);

3) 执行步骤

同上略

4.3 用[proc\_create和](http://blog.csdn.net/a_ran/article/details/37629899" \t "_blank)[seq\_file创建proc文件](http://blog.csdn.net/a_ran/article/details/37629899" \t "_blank)

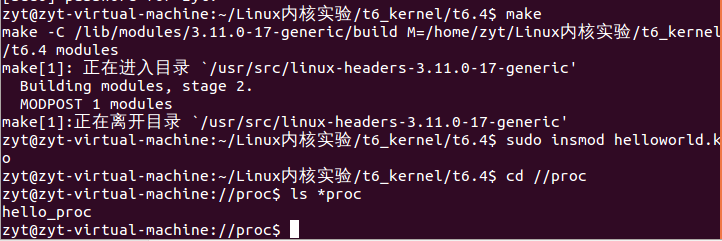
1 程序清单



2 说明

* 用proc\_create与seq\_file采用内核模块创建/proc文件，seq\_file是系列文件库，创建大文件所用。
* 在linux 3.9以下，用create\_proc\_read\_entry()创建；3.10之后用，proc\_create()创建。见<http://stackoverflow.com/questions/25746461/error-implicit-declaration-of-function-create-proc-read-entry-werror-implic>

3 运行结果



在/proc下有一个hello\_proc文件。

4.4 一个实现系统时钟的读操作内核模块

设计内容：构建一个在/proc文件系统中的内核模块clock，支持read()操作，read()返回值为一字符串，其中包块一个空各分开的两个子串，为别代表xtime.tv\_sec 和 xtime.tv\_usec。

设计思路：文件中主要包含 init\_module()， cleanup\_module()， proc\_read\_clock()三个函数。其中init\_module()， cleanup\_module()负责将模块从系统中加载或卸载，以及增加或删除模块在/proc 中的入口。 read\_func()负责产生/proc/clock 被读时的动作。

同学们自己做

**实验七 系统调用实验**

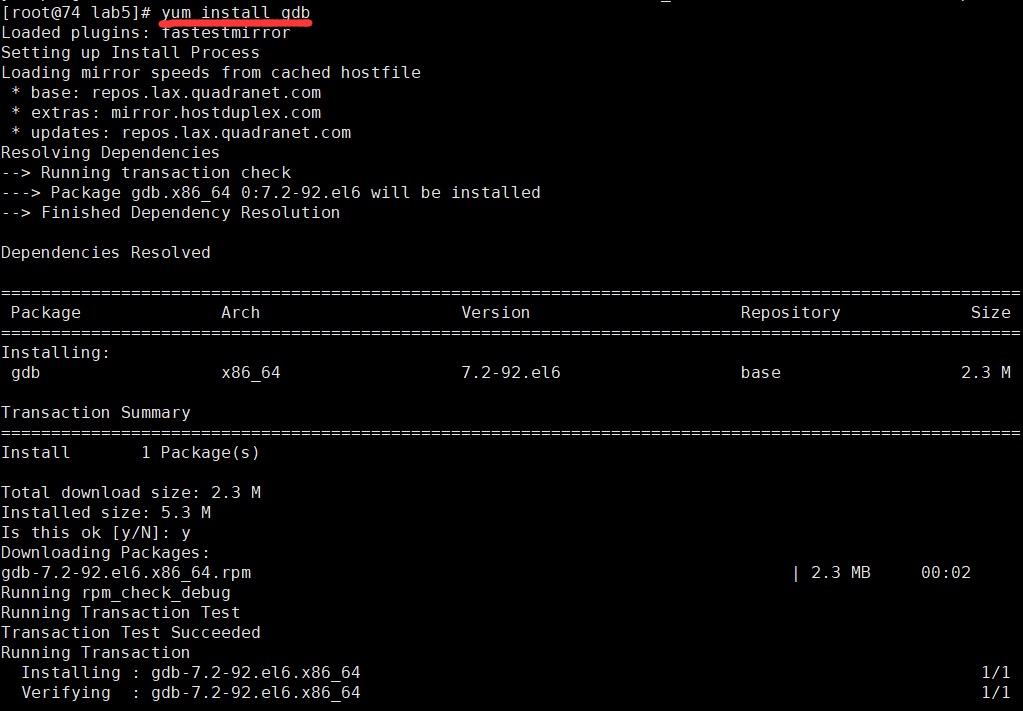
**Hit做了**

**实验4.1 观察实验(存储管理实验)**

**1 实验步骤**

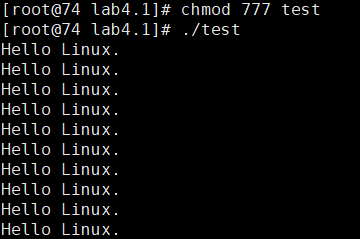
(1) 安装GDB

记得先 yum-update



1. 编写观测程序

按下方程序进行编写，可以运行



(3) 按照指令手册进行观察操作

**2 观测程序源代码**

#include<stdio.h>

#include<stdlib.h>

char str[50] = "Hello Linux.";

int main()

{

int num = 10;

while(num--){

printf("%s\n",str);

}

}

//gcc -g -o testing testing.c

**3 实验结果及分析**

(1).Gdb程序观察一个程序文件的内容和结构

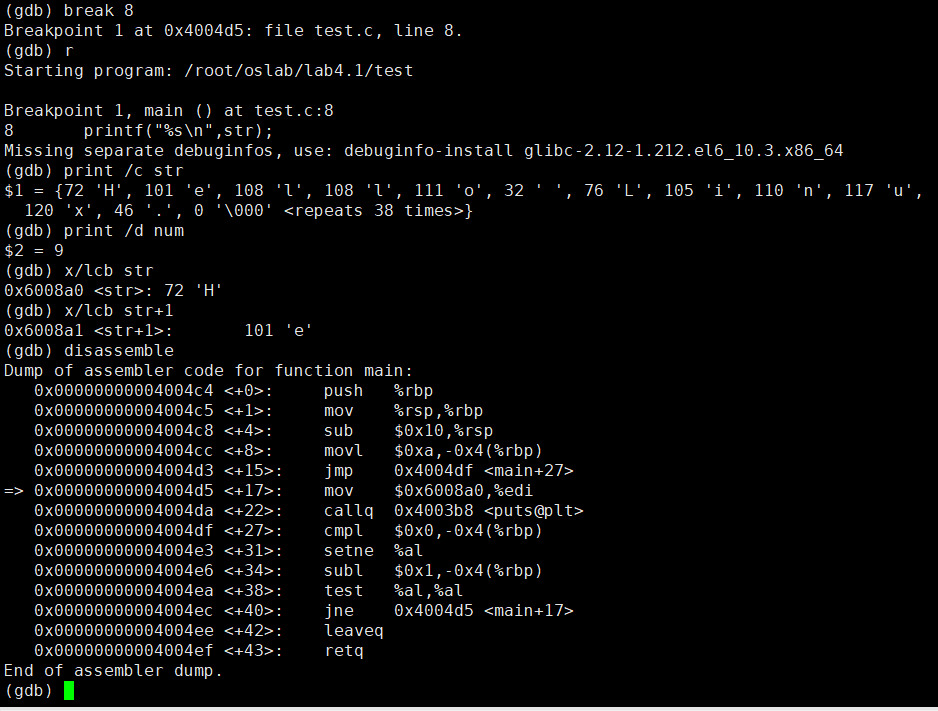
结果截图：



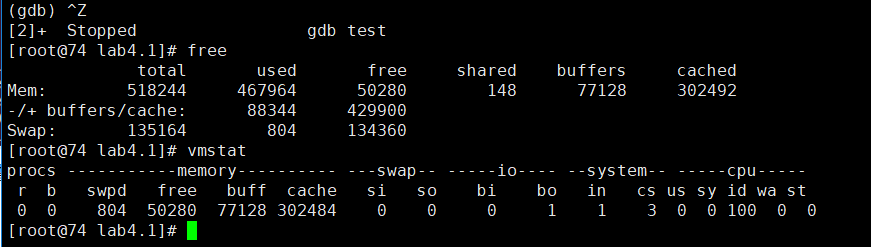


注意： 用GDB调试程序，编译时，一定要用-g参数。否则提示“没有符号表被读取..”

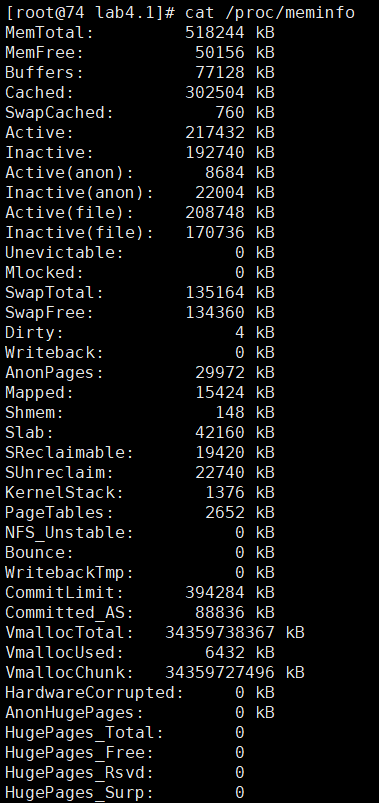
(2).GDB观察程序内存映象的内容和结构



1. .在Linux下，用free 和vmstat命令观察内存使用情况



(4).在Linux下，查看/proc与内存管理相关的文件，并解释显示结果



MemTotal: 所有可用RAM大小 （即物理内存减去一些预留位和内核的二进制代码大小）

MemFree: LowFree与HighFree的总和，被系统留着未使用的内存

Buffers: 用来给文件做缓冲大小

Cached: 被高速缓冲存储器（cache memory）用的内存的大小（等于diskcache minus SwapCache ）.

SwapCached:被高速缓冲存储器（cache memory）用的交换空间的大小

已经被交换出来的内存，但仍然被存放在swapfile中。用来在需要的时候很快的被替换而不需要再次打开I/O端口。

Active: 在活跃使用中的缓冲或高速缓冲存储器页面文件的大小，除非非常必要否则不会被移作他用.

Inactive:在不经常使用中的缓冲或高速缓冲存储器页面文件的大小，可能被用于其他途径.

HighTotal:

HighFree: 该区域不是直接映射到内核空间。内核必须使用不同的手法使用该段内存。

LowFree: 低位可以到高位内存一样的作用，而且它还能够被内核用来记录一些自己的数据结构。

SwapTotal: 交换空间的总大小

SwapFree: 未被使用交换空间的大小

Dirty: 等待被写回到磁盘的内存大小。

Writeback: 正在被写回到磁盘的内存大小。

AnonPages：未映射页的内存大小

Mapped: 设备和文件等映射的大小。

Slab: 内核数据结构缓存的大小，可以减少申请和释放内存带来的消耗。

SReclaimable:可收回Slab的大小

SUnreclaim：不可收回Slab的大小（SUnreclaim+SReclaimable＝Slab）

PageTables：管理内存分页页面的索引表的大小。

NFS\_Unstable:不稳定页表的大小

VmallocTotal: 可以vmalloc虚拟内存大小

VmallocUsed: 已经被使用的虚拟内存大小。

**实验5.1 观察实验（进程通信）**

在Linux下，用ipcs()命令观察进程通信情况，了解Linux基本通信机制

实验结果（截图）：

