Linux第三次作业

1. 在一个只有 128M 内存并且没有交换分区的计算机上，说说下面两个程序在编译及运行结果上的差别。

```
// 程序 1
#define MEMSIZE 1024*1024
int count = 0;
void *p = NULL;
while(1) {
  p = (void *)malloc(MEMSIZE);
  if (!p) break;
  printf("Current allocation %d MB\n",
++count);
}

// 程序 2
#define MEMSIZE 1024*1024
int count = 0;
void *p = NULL;
while(1) {
  p = (void *)malloc(MEMSIZE);
```

```
  if (!p) break;
  memset(p, 1, MEMSIZE);
  printf("Current allocation %d MB\n",
++count);
}
```

答：

区别在于程序二使用了memset函数做初始化。

程序一里，由于未使用malloc出来的内存空间，所以这块内存空间会被编译器优化掉，导致不会开辟空间，也不会爆堆，所以会一直死循环。

而在程序二中，是用来memset函数修改了malloc出来的内存空间中的值，所以这一部分不会被编译器优化掉，所以可以成功开辟这部分内存空间，并且会不断申请内存空间直到内存不足爆堆，最后退出。

2. 请使用程序设计语言(如 C 语言)，编程实现“ls”和“wc”命令。

   参见ls和wc文件夹中的代码。

ls实现思路：

首先，通过确定ls的选项，并将选项存储在一个整型数组中(选项默认顺序-l、-d、-R、-a、-i)。

接下来对选项进行处理。根据ls选项的特征，将整个处理过程分成三块：含有参数-d，没有-d且没有-R，没有-d但有-R。

1. 含有参数-d：-d选项表示对当前文件夹进行处理，所以只需要对`.`进行信息统计，并直接省去对-R和-a的处理；
2. 没有-d且没有-R：这里就不需要递归记录文件夹中所有的子文件夹，只需要统计当前文件夹中的所有内容；
3. 没有-d但有-R：记录下当前文件夹中子文件夹，递归调用函数，实现ls -R；
4. -l参数：使用stat函数获取到文件属性，文件类型、文件权限、所有者、所属组、大小、最后一次修改时间、文件名。-i参数：文件属性中的st_ino。-a参数：将当前文件夹`.`、父文件夹`..`、`.`开头的隐藏文件考虑进去。

wc实现思路：

1. 使用C库标准I/O函数，fopen打开文件，使用fgetc函数按字符读取，对字符进行判断，分别记录行数和单词数，打印文件大小。

2. \n 作为行分隔符，支持EOF 代表文件结尾。
3. 用空白符(\nt,\nv,\nn,\nr,\nf,空格)作为词分隔符
4. 使用全局变量来保存总用量

与源码进行比较：

ls源码

1. ls源码相当复杂，并且考虑十分全面，且代码结构很规整。
2. ubuntu ls源码处理过程大概是：初始化、设置权限、设置颜色、将当前命令行输入文件加到文件表中、文件表排序、打印当前文件表。
3. 在自己编写的ls中并没有实现文件类型所对应的颜色呈现；并且并没有对软链接进行处理，对于软链接的处理仍然是对于其指代源文件的处理。
4. 此外，支持在源码中，支持ls 指令通过 peding_dirs 结构实现了输出的正序，而我的实现当中，所有文件的输出顺序是杂乱的。源码使用 active_dir_set 结构避免软链接死循环，而我则是采用递归时忽略“.”和“..”来实现的。

wc源码：

1. wc源码很规整，层次清晰，大概的过程是：初始化

记录变量、处理选项、统计。

2. 源码打开一次文件，完成对所有信息的统计；而我是以函数的形式，打开两次文件；并且使用了文件属性stat。
3. 源码中的函数职责清晰，可拓展性强。

3. 请用中文描述一下字符型 Linux 驱动模块的编写以及编译过程?

驱动编写

总体上，可以用module_init与module_exit来编写驱动加载与退出时的行为。

具体来说：首先要申请设备号；然后生命并实现驱动提供的可被调用的函数；接下来需要定义一个file_operations结构体，它保存了驱动以及设备的信息，把它注册到系统，并和设备号绑定起来；最后创建挂在的文件节点。

驱动编译

写Makefile或者手动编译成动态链接的.so文件，把它放到对应的目录下，使用insmod加载。