Part A

一、Part A概述：

（Ⅰ）任务：

编写一个cache模拟器，该模拟器可以模拟在一系列的数据访问中cache的命中、不命中与牺牲行的情况，其中，需要牺牲行时，用LRU替换策略进行替换。

cache模拟器需要能处理一系列如下的命令：

Usage: ./csim-ref [-hv] -s <s> -E <E> -b <b> -t <tracefile>

其中各参数意义如下：

①-h：输出帮助信息的选项；

②-v：输出详细运行过程信息的选项；

③-s：组索引的位数(意味着组数S=2^s)；

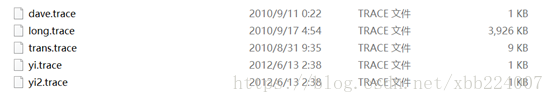
④-E：每一组包含的行数；

⑤-b：偏移位的宽度(意味着块的大小为B=2^b);

⑥-t：输入数据文件的路径(测试数据从该文件里面读取)。

二、测试数据说明：

traces文件夹里面包含五个文件：



这五个文件就是用于测试csim.c的输入文件，各个文件中包含了各种不同指令，用于测试hits、missses、evictons。

trace文件中的指令具有如下形式：

I 0400d7d4,8

M 0421c7f0,4

L 04f6b868,8

S 7ff0005c8,8

即每行代表一个或两个内存访问。每行的格式是

[空格]操作 地址，大小

操作字段表示存储器访问的类型，其中：

“I”表示指令加载，

“L”表示数据加载，

“S”表示数据存储，

“M”表示数据修改（即数据存储之后的数据加载）。

每个“I”前面都没有空格。每个“M”，“L”和“S”之前总是有空格。

地址字段指定一个32位的十六进制存储器地址。

大小字段指定操作访问的字节数；

通俗地解释一下各种操作：

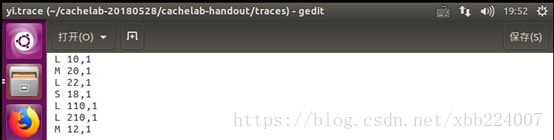
①对于‘I’指定地操作，实验说明中提到，我们不需要考虑：

意思就是valgrind运行地时候第一个指令总是为操作‘I’。

②对于‘L’以及‘S’指定的操作，我们简单地可以认为这两个操作都是对某一个地址寄存器进行访问（读取或者存入数据）；

③对于‘M’指定的操作，可以看作是对于同一地址连续进行‘L‘和’S‘操作。

以yi.trace中的数据为例：



其解释如下：

①对于地址0x10进行访问：

0x10=0000...00010000，偏移值为最低四位，故S=1;

访问结果为mis;

②连续对地址0x20进行连续两次访问：

0x20=000...00100000，S=2;

结果为第一次mis，第二次hit；

③对地址0x22进行访问：

0x22=000...00100100，S=2;

由于操作②以将该块存入高速缓存，故结果为hit;

④对地址0x18进行访问：

0x18=000...00011000，S=1;

由于操作①以将该块存入高速缓存，故结果为hit;

⑤对地址0x110进行访问：

0x110=0...000100010000，S=1;

虽然操作①使得第一组(只有一行有效)，但是这里的标志位的值Tag为1

故结果为先mis，后eviction;

⑥对地址0x210进行访问：

0x210=0...001000010000，S=1;

同操作⑤，但是这里的标志位的值为2，不匹配

故结果为先mis，后evicton;

⑦对地址0x12进行连续两次访问：

0x12=000...00000010010，S=1;

由于标志位不匹配，故第一次访问时mis，并evicton

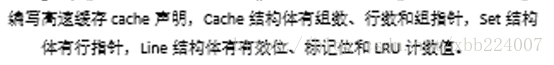
第二次访问时当然就是hit。

上述分析也就是解释了实验说明中的示例：

三、Cache结构体的声明：

（Ⅰ）数据成员声明：

实验指导中说到：



但是考虑到Cache的结构特别像多维数组，故这里简单地用一个动态开辟地三维数组表示，结构体中还有数据成员s-组数、E-每组的行数、b-块的字节数，如下：



其中数组的第一维表示组数，第二维表示每组的行数，第三维就是每行的三个值有效位，标志位以及LRU值。

（Ⅱ）各种操作的实现：

定义了数据成员以后，需要对数据进行各种操作(设值、申请内存、释放内存，查询数据等等)，如下：



上图中实现的操作有：

Set\_Cache()：s、E、b的设置；

Get\_Valid()：返回第\_s组第\_E行的有效位；

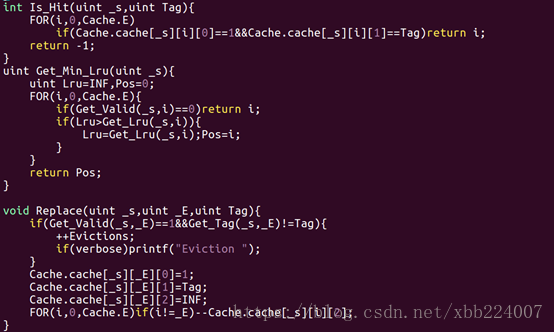
Get\_Tag()：返回第\_s组第\_E行的标志位；

Get\_Lru()：返回第\_s组第\_E行的LRU计数值；

Get\_Cache()：动态为数组分配内存；

Free\_Cache()：释放内存。

接下来是：



上图中实现的操作有：

Is\_Hit()：判断是否命中——命中返回命中的行号，否则返回-1；

Get\_Min\_Lru()：查询LRU值最小的行号；

Replace()：替换第\_s组中的第\_E行，用于替换的标志位为Tag；

写完各种操作以后，我们还能想到什么——查看Cache中的内容，这样便可以实时地查看Cache中的情况，如果有错误通过查看Cache内容也能比较容易地找出错误出现的位置，如下：



第一个是输出每一组的信息，第二个函数是输出每一个组（整个Cache的信息）;

四、数据的输入：

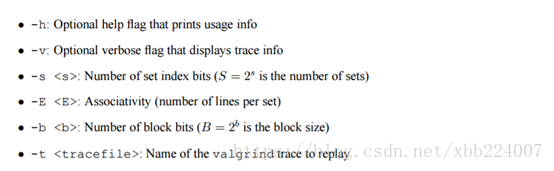
前面还只是对Cache进行声明以及操作的实现，还没有具体的应用，要具体地用起来，当然要用户输入请求，然后再去用各种操作实现请求，所以接下来写输入的处理。输入的处理可以分为两个阶段来写：一个是命令行输入的解析，另一个个是测试数据的输入。下面分开来分析：

（Ⅰ）首先是命令行输入的解析：

命令行的输入就是再终端输入的一系列命令和选项，我们就是根据这些选项来设置各种值以及测试数据的文件位置的路径，比如：

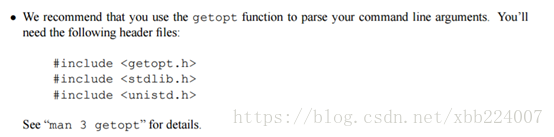
./scim -s 4 -E 1 -b 4 -t traces/yi.trace，就是将s的值设为4，E的值设为1，b的值设为4，文件的路径为traces/yi.trace。

具体的选项实验说明中也有解释：



我们要做的就是将输入的这样的一串字符分析，得到具体的信息并利用。

把输入当做字符串来储存，然后再慢慢地分析未尝不是一个解决方法，但是这里有更加便捷的方法，即调用库函数getopt()函数来解析。实际上作者也建议我们这么做：



该函数的具体细节这里不做过多赘述，这里做一个简单地说明：

①首先调用这个函数需要上面提到的三个头文件；

②函数需要一个选项字符串作为参数，选项字符串即由各种选项构成的一个字符串，而选项就是上面提到的-s -E -b之类的了，所以考虑所有的选项，这里的选项字符串为“-hvs:E:b:t:”。

③然后函数会根据选项字符串去和你的输入进行匹配，当某个选项匹配时，函数会返回该选项对应的字符，而该选项对应的参数会存入变量optarg中，该变量是包含在头文件中的，这样一来的话我们就可以快速而简洁地分析命令行中地字符串了。

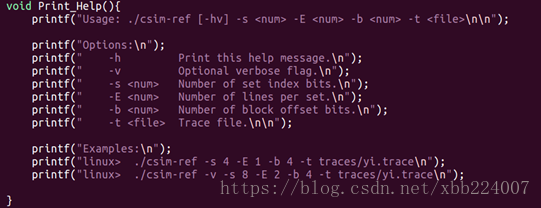
实现如下：



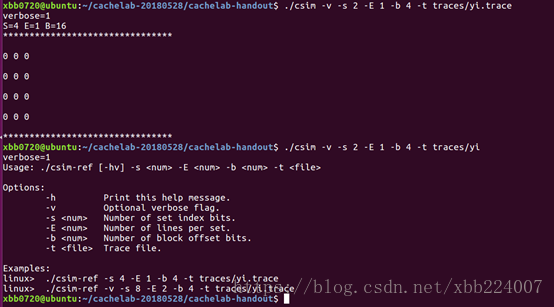
说明几点：

①这里用到了库函数atol()，即把一个字符串转化为一个整数；

②当解析选项字符串失败或者参数不合法时，应该要打印提示信息，即Print\_Help():



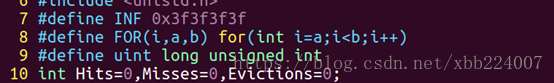
有了这一部分输入以后，我们便可以验证我们对于Cache的创建及初始化的正确性了，输入如下：



至此Cache的创建于初始已结束。

（Ⅱ）测试数据的输入：

这里用于测试的数据已经由文件给定，我们要做的就是从文件中读取。从文件读取用到了库函数fscanf()，该函数与普通的scanf的区别就是该函数多一个参数，就是指向目标文件的文件指针，也就是从指向的文件读取数据。写法如下：





说明：这里的操作类型“I、L、S、M”虽然只是一个字符，但是这里用的是字符串输入，字符串输入比字符输入不容易出错。至此数据的输入已经完成。

五、Cache的模拟

定义了Cache以及有了测试数据以后我们就可以开始用Cache来模拟了，我们可以先理一下Cache工作的流程：

1、输入数据指定需要访问的地址寄存器。

2、分析输入的地址，并判断是否命中。

3、如果命中，则hits++，并更新LRU值。

4、如果不命中，则misses++，然后判断是否需要eviction，并更新LRU值。

下面进行逐条分析：

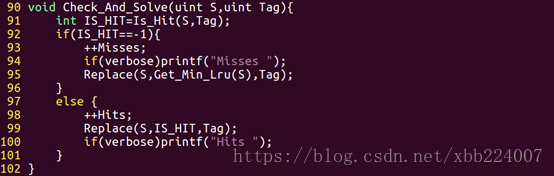
（Ⅰ）数据的输入由fscanf()处理。

（Ⅱ）地址的分析目的就是要得到指定的组数S以及对应的标志值Tag。计算如下：



因为Tag就是地址的高（m-s-b）位，同理可以分析S的产生。

（Ⅲ）接下来就是3和4，这里用一个函数来处理，并调用了对Cache的各种操作来完成：



解释如下：

第91行：判断是否命中，命中则返回命中第S组中的行数，否则返回-1；

第92~96行：如果没有命中，先++miss，然后按照LRU替换策略进行替换；

第97~101行：如果命中，则替换命中的行，其实这里没有必要替换，主要是要更新LRU值。

主函数中调用如下：

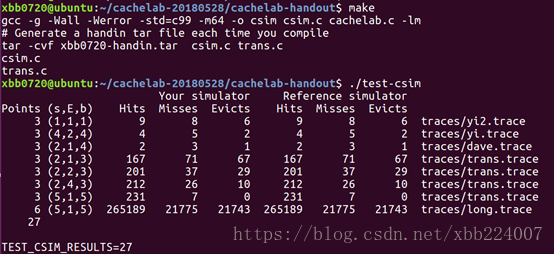


最终调用printSummary()进行结果的输出，并释放内存：



六、Cache的测试

测试运行结果如下：



结果正确，至此所有的输入数据均已通过，Part A完成！

