计算机组成原理课后作业5

1. 目标

理解高速缓存对于程序性能的影响。

2. 内容

这个实验包括两部分内容。首先，你需要使用C语言编写一个小型程序（200-300行）用来模拟高速缓存；然后，对一个矩阵转置函数进行优化，以减少函数操作中的缓存未命中次数。

2.1 关于内存跟踪文件（trace files）

在traces文件夹下包含着一些trace文件，这些文件被用于验证你所实现的高速缓存模拟程序的正确性。这些跟踪文件由valgrind程序生成的。例如，在控制台中输入以下命令：

```
linux> valgrind --log-fd=1 --tool=lackey -v --trace-mem=yes ls -l
```

这时valgrind将会启动 ls -l 这个程序，并对这个程序的内存访问进行捕获和追踪，并将内存访问记录输出到控制台上。

valgrind的内存追踪记录类似于如下的格式

```
I 0400d7d4,8
 M 0421c7f0,4
 L 04f6b868,8
 S 7ff0005c8,8
```

每一行记录表示一次或两次内存的访问。格式说明如下

```
[空格]操作  地址，长度
```

其中【操作】有四种类型，分别为I、L、S和M。I表示进行了一次指令加载（读内存，I前无空格）；L表示进行了一次数据加载（读内存，L前有一个空格）；S表示进行了一次数据存储（写内存，S前有一个空格）；M表示进行了数据修改（先读内存，后写内存，M前有一个空格）。数据修改意味着，一条指令中对某个地址先进行读内存操作后进行写内存操作。例如：

```
addq %rax, (%rbx)    # 先将%rbx所指向的内存地址的数据取出与%rax的值相加，再存入%rbx所指向的地址中
```

【地址】是一个64位的整数，使用16进制数表示。【长度】表示当前数据访问操作的字节数。

2.2 任务A：编写一个高速缓存模拟程序

在这部分任务中，你将在csim.c文件中编写一个高速缓存仿真程序。这个程序使用valgrind的内存跟踪记录作为输入，模拟高速缓存的命中/未命中行为，然后输出总的命中次数，未命中次数和缓存块的替换次数。

在实验的资源中，我们提供了一个可执行程序 csim-ref 用于模拟高速缓存。这个程序以valgrind生成的跟踪文件作为输入，可以根据运行时的输入参数模拟任意结构（直接映射、全相连、组相连）和任意块大小的高速缓存。使用LRU（least-recently used,最近最少使用）算法作为缓存块的替换策略。

这个仿真器使用方法如下：

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

- -h：显示使用帮助
- -v：输出跟踪信息
- -s <s> 内存地址中组序号所占的位宽（高速缓存的组数 $S = 2^s$）
- -E <E> 相关性（高速缓存每组的缓存块个数）
- -b <b> 内存地址中偏移量所占的位宽（缓存块大小 $B = 2^b$）
- -t <tracefile> valgrind生成的跟踪文件的名称

例如，在控制台中输入以下命令，模拟一个包含16个组（2^4）,每组中1个块，每个块包含16个字节（2^4）的直接映射高速缓存。输入的内存跟踪文件为 traces/yi.trace （具体文件内容见本次实验所提供的的资源）。然后在控制台中会得到高速缓存的统计结果。

```
linux> ./csim-ref -s 4 -E 1 -b 4 -t traces/yi.trace
hits:4 misses:5 evictions:3
```

使用 -v 选项，会看到每次内存访问时高速缓存的工作情况。

```
linux> ./csim-ref -v -s 4 -E 1 -b 4 -t traces/yi.trace
L 10,1 miss
M 20,1 miss hit
L 22,1 hit
S 18,1 hit
L 110,1 miss eviction
L 210,1 miss eviction
M 12,1 miss eviction hit
hits:4 misses:5 evictions:3
```

你的任务

在这部分任务中，你需要填充csim.c文件，实现一个和./csim-ref具有相同功能的模拟器（功能相同，输入的参数格式相同，输出的格式也相同）。

要求

1. 你所实现的仿真器，必须能够处理任意的输入参数（s、E、b），所以仿真器中的数据结构应该是动态分配的空间，需要使用malloc函数在堆上申请空间，使用free函数释放空间。具体的函数使用方法，可以再控制台上输入“man malloc”和“man free”命令，查看函数的使用手册。

2. 在这个实验中，我们只关注高速缓存的对数据的加载和存储的性能影响，因此在你的仿真软件中应忽略所有指令加载的操作，即所有以I作为起始字母的行可以自动忽略。跟踪文件中记录的行起始字符为空格，然后是操作类型标识L/S/M。

3. 在程序的最后调用printSummary函数（已实现），输出最后的统计结果（命中总次数，未命中总次数，块替换总次数）。

   ```
   printSummary(hit_count, miss_count, eviction_count);
   ```

4. 在实验中，你可以假设所有的内存访问都满足64位系统下字节对齐的规范的，每次操作不会导致跨缓存块的内存访问。基于以上假设，在设计模拟器时，你可以忽略valgrind生成的跟踪文件的每条记录中的【长度】字段。

2.3 任务B：优化矩阵转置运算程序

在trans.c中编写一个矩阵转置函数，尽可能的减少程序对高速缓存访问的未命中次数。

A表示一个矩阵，$A_{i,j}$表示第i行第j列的元素。矩阵A与其转置矩阵 A^T 的之间的关系为 $A_{i,j} = A^T_{j,i}$。

在trans.c中，我们给出了一个示例代码，可以实现将$N \times M$的数组A转置为$M \times N$的数组B。

```
char trans_desc[] = "Simple row-wise scan transpose";
void trans(int M, int N, int A[N][M], int B[M][N])
```

这个示例代码是正确的，但是缓存不友好。

你的任务

在这部分任务中，你需要写一个相同功能的缓存友好的矩阵转置函数transpose_submit。函数原型如下：

```
char transpose_submit_desc[] = "Transpose submission";
void transpose_submit(int M, int N, int A[N][M], int B[M][N]);
```

注意：不要修改用于进行描述的字符串（“Transpose submission”）

要求

1. 编译时不允许出现任何的warning。
2. 转置函数中定义的int型局部变量总数不能超过12个。
3. 不允许使用long等数据类型，在一个变量中存储多个数组元素以减少内存访问。
4. 不允许使用递归。
5. 在程序中不能修改矩阵A中的内容，但是，你可以任意使用矩阵B中的空间，只要保证最终的结果正确即可。
6. 在函数中不能定义任何的数组，不能使用malloc分配额外的空间。

3. 关于程序的构建

编译整个程序，需要输入以下命令：

```
linux> make clean    # 清除旧的程序文件
linux> make          # 重新构建新的程序
```

4. 评价

整个实验一共60分，其中：

- 任务A：27分
- 任务B：26分
- 编码风格：7分

4.1 任务A的评价方法（27分）

我们使用不同的高速缓存参数和内存跟踪文件进行8组测试。前7组，每组测试通过得3分；最后一组测试通过得6分。

```
linux> ./csim -s 1 -E 1 -b 1 -t traces/yi2.trace
linux> ./csim -s 4 -E 2 -b 4 -t traces/yi.trace
linux> ./csim -s 2 -E 1 -b 4 -t traces/dave.trace
linux> ./csim -s 2 -E 1 -b 3 -t traces/trans.trace
```

```
linux> ./csim -s 2 -E 2 -b 3 -t traces/trans.trace
linux> ./csim -s 2 -E 4 -b 3 -t traces/trans.trace
linux> ./csim -s 5 -E 1 -b 5 -t traces/trans.trace
linux> ./csim -s 5 -E 1 -b 5 -t traces/long.trace
```

你可以对比csim-ref程序的输出结果验证你所编写的程序的正确性。每组实验输出三个结果，命中次数，未命中次数和页替换次数，每个结果占该组总分数的$\frac{1}{3}$。

4.2 任务B的评价方法（26分）

我们对矩阵转置函数使用不同大小的矩阵进行测试

- $32 \times 32 (M = 32, N = 32)$
- $64 \times 64 (M = 64, N = 64)$
- $61 \times 67 (M = 61, N = 67)$

使用valgrind生成程序的内存跟踪文件，并使用高速缓存模拟器进行分析（参数：s=5，E=1，b=5）。定义m为未命中次数，给分标准如下：

当程序结果正确时：

- 32×32：m<300得8分，m>600得0分
- 64×64：m<1300得8分，m>2000得0分
- 61×67：m<2000得10分，m>3000得0分

如果程序结果不正确：得0分。

4.3 编码风格（7分）

根据编码风格、可读性、注释的清晰程度等条件综合给出。如果程序设计中违反了任务A或任务B中的编码要求，得0分。

5. 提示

5.1 任务A

在实验资源中提供了一个自动测试工具 test-csim，用于比对你所实现的模拟器与标准程序之间的差异。在控制台中输入以下命令进行比对：

```
linux> make          # 构建程序
linux> ./test-csim   # 结果比对
```

一些建议

- 完成代码设计后，可以先从最小的数据集开始进行测试 traces/dave.trace。
- 使用-v选项输出高速缓存工作细节，并与标准程序进行比对。
- 可以使用getopt函数处理传入的参数，以简化参数的处理环节。使用该函数需要引入以下头文件

```
#include <getopt.h>
#include <stdlib.h>
#include <unistd.h>
```

更多关于opt函数的用法请使用“man 3 getopt”命令查阅相关手册。

- 每个加载操作L或存储操作S至多会产生一次缓存未命中；数据修改操作M可以被认为是对相同的地址先进行加载操作然后进行存储操作，所以一个M操作可能会产生两种情况：a. 两次内存命中, b. 一次内存未命中和一次内存命中以及相应的页替换。

5.2 任务B

在实验资源中我们提供了一个自动测试程序test-trans.c用于进行批量测试和比对，并自动计算结果。你可以在trans.c中同时实现多个版本（<100）的转置函数，并进行比对。每个函数都要遵循以下的原型进行实现：

```
char trans_simple_desc[] = "A simple transpose";
void trans_simple(int M, int N, int A[N][M], int B[M][N])
{
/* your transpose code here */
}
```

然后使用下面的函数将你的转置函数在registerFunctions函数中进行注册，以实现自动化测试和比对（这里面会使用到valgrind和csim-ref，在运行csim-ref时，会自动将参数设置为：s=5,E=1,b=5）：

```
registerTransFunction(trans_simple, trans_simple_desc);
```

在控制台上输入以下命令完成程序构建和测试：

```
linux> make                          # 构建程序
linux> ./test-trans -M 32 -N 32      # 测试 32 * 32的矩阵转置
```

一些建议

- test-trans 测试程序运行后，会在控制台输出结果，同时会为每一个函数生成相应的内存跟踪文件（trace.f[i]）。更多的信息你可以通过阅读内存跟踪文件，并使用csim-ref测试跟踪文件来获得
- 在该任务中，高速缓存的组织方式是直接映射高速缓存，因此减少未命中次数时需要考虑解决冲突未命中（conflict misses）的情况，尤其是在处理对角线附近的数据时。
- 数据分块是一种有效的减少未命中次数的技术

5.3 driver.py

在控制台中运行下面的命令，可以直接将任务A和任务B的全部测试结果同时输出。

```
linux> ./driver.py
```

6. 作业提交

只上传 csim.c 和 trans.c文件。