

XJTU-ICS lab4: Cache Lab

XJTU-ICS Lab 4: Cache Lab

实验简介

芜湖~转眼来到第四个实验，不知道前三个实验大家玩的是否开心，是否都得到了自己满意的分数呢？是否已经习惯了一个又一个周末被XJTU-ICS夺走了呢（笑）？无论你是非常顺利快速地速通了前三个实验，还是刚刚好在Due堪堪提交你的答案。都请大家收拾心情，新实验是一个更新的挑战，将带来更多新的体验。

这个实验的目的是为了让大家在课堂理论知识的基础上，更好地理解Cache的运行过程以及Cache对于程序运行性能的影响。

本次实验由两部分构成：

第一部分（Part A），会要求大家使用C语言，实现一个**Cache模拟器**。实现模拟器并逐渐做到Bugfree的过程会让你加深对Cache结构的理解。

第二部分（Part B），会要求大家使用C语言，写一个矩阵转置的函数，这个部分的目标是使我们的代码跑的尽可能的快。不断减少Cache Miss次数的过程会帮助你理解Cache对程序运行性能的重要作用。

这次的实验比起之前的实验来说更难一点点，如果说前两个小实验分别对大家的计算机动手能力和特定的小的分支领域（bit表示和GDB debug、逆向程序能力）提出了一些要求的话，Cache Lab对同学们的代码能力提出了一定的要求。

但是相信在Coding的过程，在逐渐领会其中奥妙的过程中，你会加深对Cache以及Memory Hierarchy的理解。如果在这个过程中，同学的能力有了亿点点的增长，我们会很开心~

Enjoy and Have fun !

注意事项

1. 老生常谈但是第一重要：这是一个“个人”作业，请大家独立完成。
2. 上交的代码文件请注意一定不能有编译问题。**0 Warnings的程序**（那肯定更不能有error）才会帮助你顺利获得属于你的分数。
3. 本文中任何命令相关都运行在Linux环境中，并以如下形式进行书写：

```
1 | linux$ xxx
```

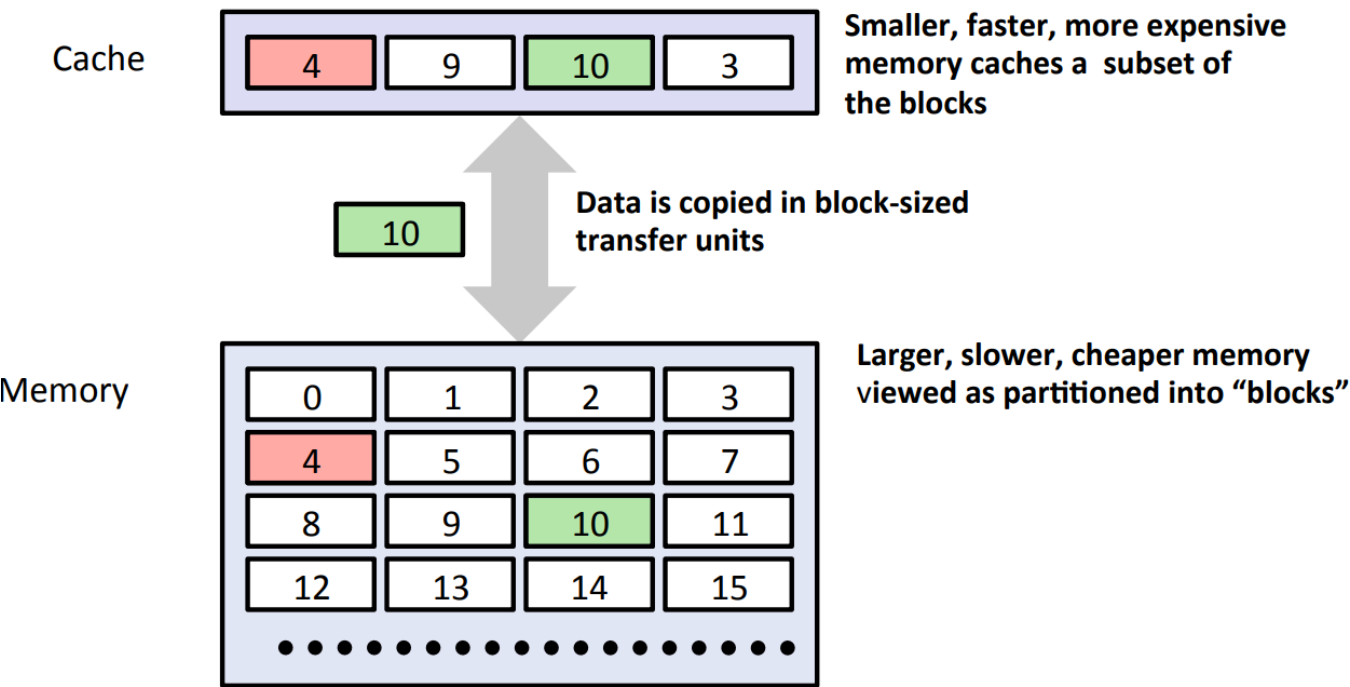
其中 `linux$` 为命令行标识符，`xxx` 为命令内容。请勿直接复制整行内容进行执行。

背景知识梳理

本实验与Cache强相关，所以我们首先从总体上带大家来梳理一下Cache相关的知识。

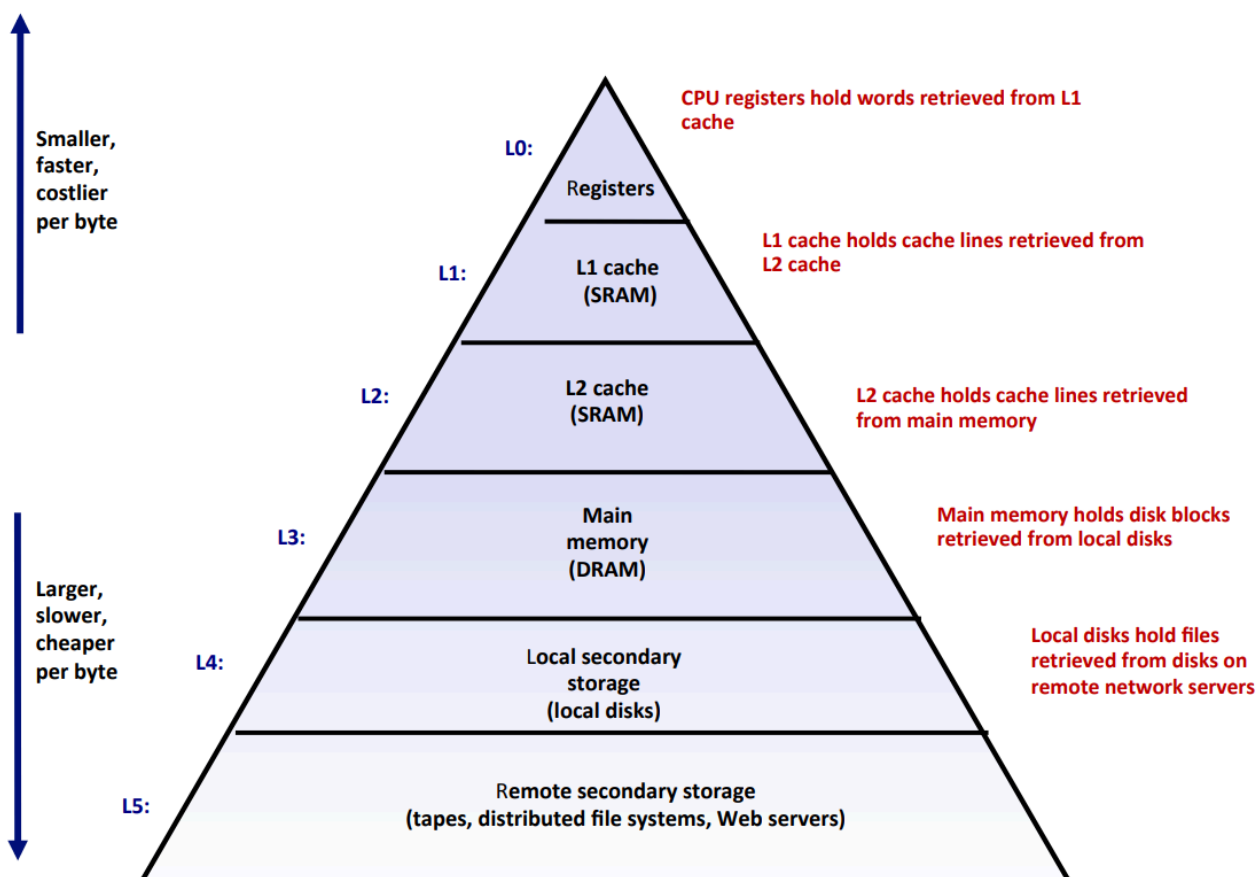
Cache

一般而言，Cache是一个小而快速的存储设备，它作为存储在更大、也更慢的设备中的数据对象的缓冲区域。使用cache的过程称为Caching。Cache的原理图如下：



Memory Hierarchy

根据我们课程中的学习，不同存储技术的访问时间差别很大，但速度较快的技术每字节的成本要比速度慢的技术高，而且容量小。现代计算机系统往往采用存储器层次结构的方法，下图展示了一个典型的存储器层次结构。

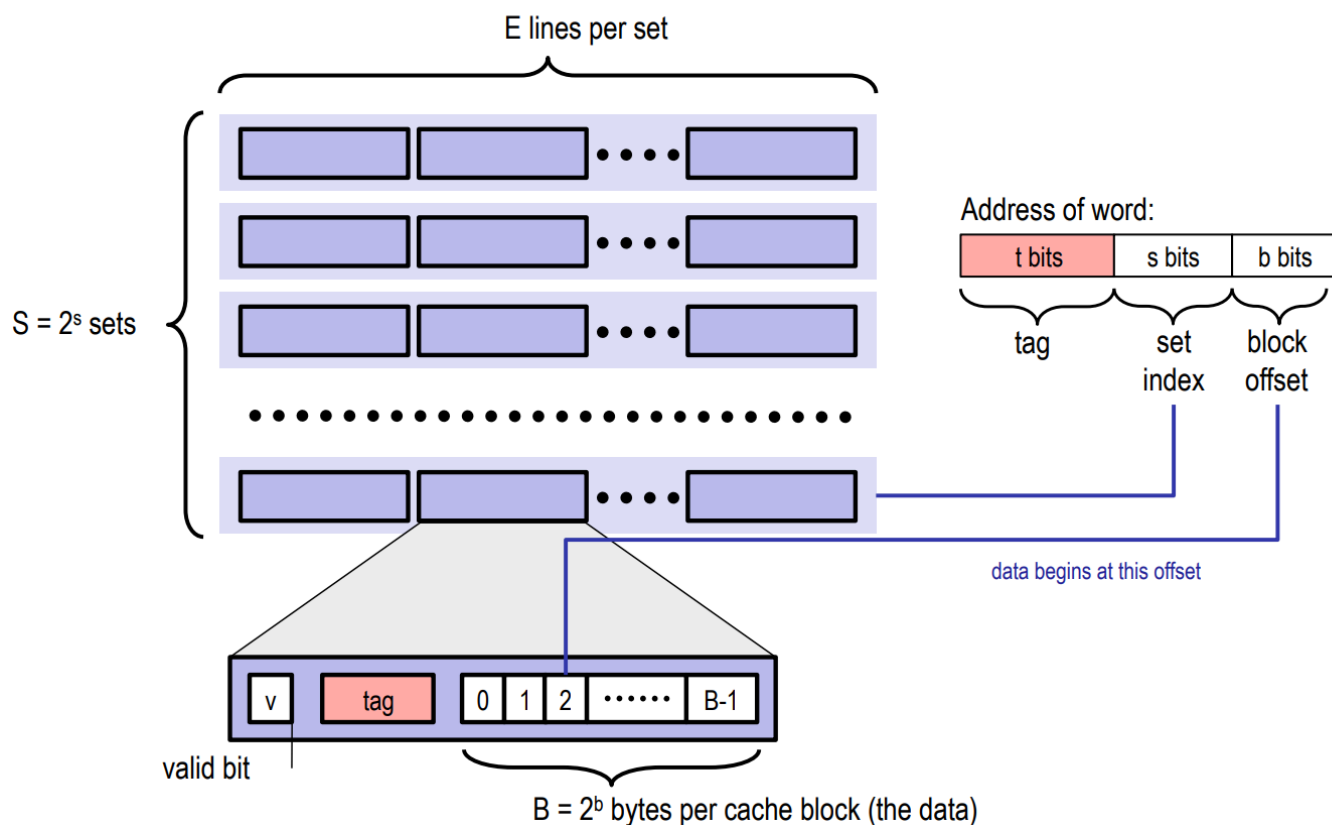


存储器层次结构的中心思想是：对于每个 k ，位于 k 层的更快更小的存储设备作为位于 $k+1$ 层的更大更慢的存储设备的cache。即每一层的cache的内容均来自于较低一层的数据对象。例如图中，L2 cache(SRAM)作为L3 主存的cache。

这样的体系结构，使得整体计算机的整体存储系统呈现出高速且大容量的整体特性。

Cache 结构

考虑一个计算机系统，其中每个存储器地址有 m 位，形成 $M = 2^m$ 个不同的地址，如下图所示。在这样一个计算机中，Cache会被分成 $S = 2^s$ 个Cache Set，每个Cache Set包含 E 个Cache Line。

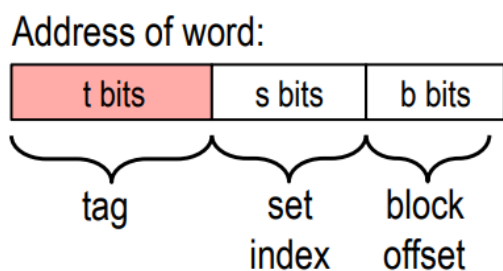


Cache Line

每个Cache Line由三部分构成：

- ▶ 有效位(valid bit)：指示Cache Line是否有效。Cache Line有效意味着它存储着来自主存的数据块；否则Cache Line的其他信息都是无效的，应该被忽略。
- ▶ 标记位(tag bit)：帮助确定Cache Line中存储的数据地址。
- ▶ 数据块(block)：主存中某个数据块的副本。

Cache工作时，在Cache看来，主存的地址被分成三部分，如下图所示：



其中， s 位的Set Index字段（组索引）是 S 个Cache Set的索引。例如，第一个组的索引是0，第二个组的索引是1，以此类推。Set Index字段告诉我们这个数据必须存在哪个Cache Set中。

t 位的Tag字段是用来确定该Cache Line是否存储着目标地址的数据。

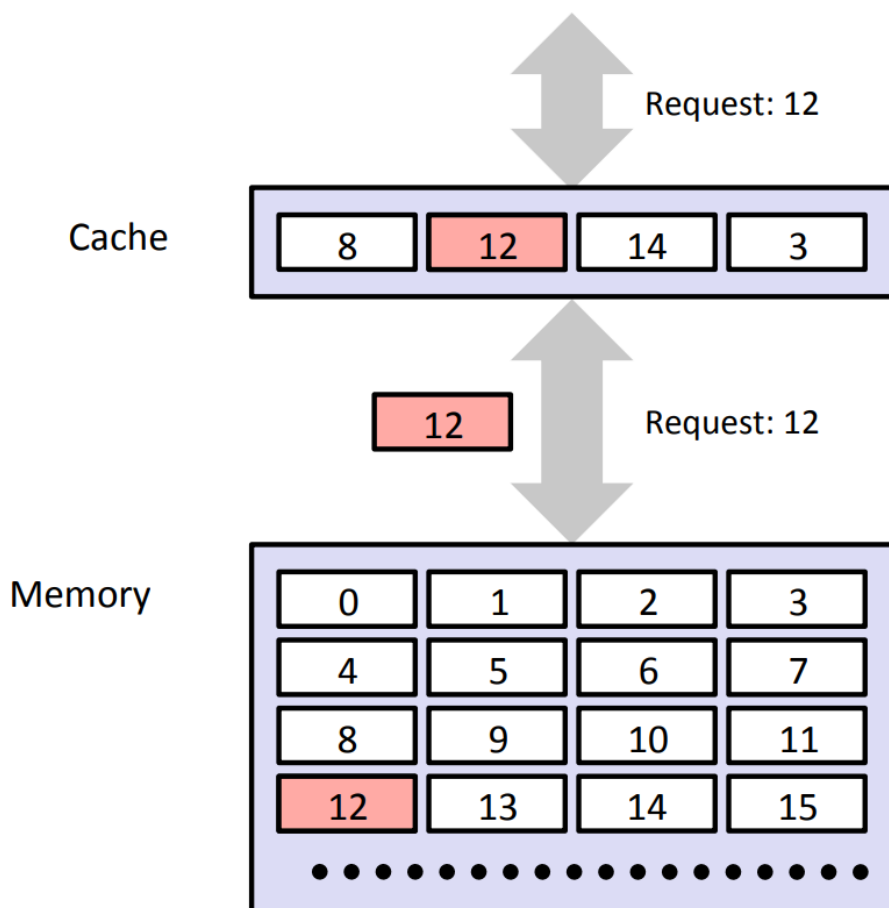
b 位Block Offset是可以在确定目标数据在块中偏移量。

Cache Set

Cache Set，是一个包含 E 个Cache Line的Set。根据每个Cache Set所包含Cache Line的数目（即 E ），Cache地址映射方式可分为3种：

- ▶ 直接映射: $E = 1$, 主存中的一个块只能映射到Cache的某一特定Cache Line中去
- ▶ 组相联: $E > 1$, 主存和Cache都分组, 主存中一个组内的块数与Cache中的分组数相同, 组间采用直接映射, 组内采用全相联映射。
- ▶ 全相联: Cache比较灵活, 主存中任何一块都可以映射到Cache中的任何一个Cache Line中

访问 Cache 产生的事件



Cache Hit

对于上面的图, 如果程序需要Memory层的数据对象d, 它会首先访问Cache, 如发现Cache中恰好存在数据对象d (例如访问8), 此时就叫Cache Hit (命中)。

当发生Cache Hit时, 程序就不用访问主存了, 直接从Cache读取目标数据即可。

Cache Miss

如果程序需要主存的数据对象d, 它首先访问Cache, 发现Cache中并没有存储数据对象d对应的数据块(block), 此时就叫Cache Miss (不命中)。

当发生Cache Miss时, Cache层会从主存中取出包含d的数据块, 然后根据地址中的Set Index找到Cache中的对应Cache Set。如果这个Cache Set中还有空的Cache Line (即Valid Bit = 0), 直接存入, 并更新Tag字段和Valid位; 否则, 需要进行Cache Line替换, Cache Line替换需要覆盖一个块, 所以也可以称为Cache Eviction。

Cache Eviction

上述Cache Line替换的过程是需要覆盖原有的一个块的, 所以可以称为Cache Eviction。一个Cache Set中有多个Cache Line, 替换时需要决定替换哪个Cache Line, 这是由Cache替换策略决定的, 例如常见的最近最少被使用策略(LRU策略)会替换掉一个访问时间距离现在最长的块。

开始实验

为了减轻大家的负担，我们将实验所需的分发包放在了同学的主目录下。

登录自己ICS账号可以在主目录下看到一个文件 `cachelab-handout.tar`，在你的目录下运行下面命令：

```
1 | linux$ tar xvf cachelab-handout.tar
```

此时可以看到一个文件夹 `cachelab-handout`，进入之后是这样的：

```
1 | linux$ cd cachelab-handout/  
2 | linux$ ls  
3 | cachelab.c  cachelab.h  csim.c  csim-ref  driver.py  Makefile  README  test-cs:
```

你可以执行 `make` 来编译这些文件。

```
1 | linux$ make clean  
2 | linux$ make  
3 | gcc -g -Wall -Werror -std=c99 -m64 -o csim csim.c cachelab.c -lm  
4 | gcc -g -Wall -Werror -std=c99 -m64 -O0 -c trans.c  
5 | gcc -g -Wall -Werror -std=c99 -m64 -o test-trans test-trans.c cachelab.c trans  
6 | gcc -g -Wall -Werror -std=c99 -m64 -O0 -o tracegen tracegen.c trans.o cachelab  
7 | # Generate a handin tar file each time you compile  
8 | tar -cvf xxx-lab4-handin.zip csim.c trans.c  
9 | csim.c  
10 | trans.c
```

此时运行 `ls` 会发现目录下会多出几个文件：

```
1 | linux$ ls  
2 | cachelab.c  csim  csim-ref  Makefile  test-csim  test-trans.c  tracegen.c  
3 | cachelab.h  csim.c  driver.py  README  test-trans  tracegen  traces
```

其中 `csim` 文件是 `csim.c` 生成的可执行文件，`trans.o` 是 `trans.c` 生成的文件，`xxx-ics-lab4-handin.zip` 是你未来将要提交的 `zip` 文件。

文件介绍：

- `csim.c`：Part A 中你需要修改的代码文件。

- `trans.c` : Part B 中你需要修改的代码文件。
- `test-csim` : 用于测试Part A得分的可执行文件。
- `test-trans` : 用于测试Part B得分的可执行文件。
- `trace` : 包含一些 `trace` 文件, 可以测试 `csim` 文件的正确性 (至于什么是 `trace` 文件, 后文中会有介绍)。
- `driver.py` : 一键检测总得分的脚本。
- `csim-ref` : 参考Cache模拟器, 可以与自己所实现的csim程序进行比对。
- `tracegen` : `test-trans` 需要用到的可执行文件。
- `tracegen.c` : 生成 `tracegen` 的源文件。
- `cachelab.c cachelab.h` : 包含一些实验用到的函数的定义, 大家可忽略。

Part A: Cache模拟器

本次实验一共两个部分, 在Part A中, 你需要实现一个Cache模拟器。

Part A 前置知识: Trace文件简介

在 `cachelab-handout` 目录下的 `traces` 目录中有许多以 `.trace` 结尾的文件, 我们称它们为trace文件。trace文件中是一系列访存日志, 它作为Part A中Cache模拟器的输入, 用来判断程序的正确性。trace文件是通过 `Valgrind` 工具生成的。



`Valgrind` 是一款用于内存调试、内存泄漏检测以及性能分析的软件开发工具。

例如:

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

以上命令可以输出执行 `ls -l` 命令时实际产生的所有内存访问日志。

trace文件的格式如下:

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

每一行代表一个内存访问指令, 每个指令可能会有一次或两次的内存访问, 每一行的格式如下:

```
1 | [space]operation address,size
```

- `operation` 表示内存访问指令的类型，分为4种：
 - `I` 表示一条指令的加载 (Instruct)
 - `L` 表示数据读取 (Load)
 - `S` 表示数据存储 (Store)
 - `M` 表示数据修改 (Modify) (**实际上是一次Load再加一次Store**)
 - **注意**：在trace文件中，`I` 前面没有空格，但是 `M`，`L`，`S` 前面一定有一个空格。
- `address` 表示一个64位**十六进制**内存地址。
- `size` 表示本次内存访问的字节数。

Part A 实验内容

在Part A中，你需要在 `csim.c` 中写一个Cache模拟器，这个模拟器以 `valgrind` 生成的trace文件作为输入，基于Cache的运行原理，模拟在这个trace文件中的每一次访存操作，判断是否发生Cache hit或Cache Miss，以及有无Cache Eviction，最后输出所有内存访问后**Cache Hit**、**Cache Miss**和**Cache Eviction**的总次数。

我们已经提供了一个**参考Cache模拟器**：`csim-ref`，该模拟器是一个二进制可执行文件，可以模拟Cache在运行指定的trace文件的访存指令时的行为，并且使用LRU策略来进行Cache Eviction。

`csim-ref` 的用法如下：

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

- `-h`: [可选] 打印帮助信息
- `-v`: [可选] 打印详细的trace日志
- `-s <s>`: Set Index位数 (总Cache Set数 $S = 2^s$)
- `-E <E>`: 每个Cache Set中包含的Cache Line的数目
- `-b `: Block Offset位数 (Block总大小 $B = 2^b$ 字节)
- `-t <tracefile>`: 指定一个trace文件作为程序输入

程序执行示例：

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

你可以通过 `-v` 参数打印更加详细的信息：

```
1 | linux$ ./csim-ref -v -s 4 -E 1 -b 4 -t traces/yi.trace
2 | L 10,1 miss
3 | M 20,1 miss hit
4 | L 22,1 hit
5 |
```



```

5 | S 18,1 hit
6 | L 110,1 miss eviction
7 | L 210,1 miss eviction
8 | M 12,1 miss eviction hit
9 | hits:4 misses:5 evictions:3

```

总之，你需要完成的任务是将 `csim.c` 文件补全，在编译后实现一个Cache模拟器的功能，功能上要与我们给出的参考Cache模拟器(`csim-ref`)基本一致，即在相同访存日志下与参考Cache模拟器(`csim-ref`)输出相同结果（我们主要关注hit, miss, 和eviction的次数是否一致）。



提示：在coding前请务必读完下面的规则部分，本次实验的规则（尤其是规则3~6）可能决定了你如何写代码。

Part A 代码规则：

1. `csim.c` 文件可以正常编译。
2. 你写的模拟器可以在任意的 `s`，`E` 和 `b` 参数下正常工作。因此你可能需要使用 `malloc` 函数来根据不同的参数对Cache数据结构进行动态内存分配。
3. **忽略指令读取操作：**对于本实验，我们只对数据访问的性能感兴趣，因此你的Cache模拟器应该忽略所有指令读取（以 `I` 开头的行）。注意一下 `valgrind` 总是把 `I` 放在行首（即前面没有空格），而 `M`、`L` 和 `S` 前面有空格。这个特征可帮助你分析 `trace` 文件。
4. 要获得分数，必须要调用我们提供的 `printSummary` 函数，该函数有三个参数，分别是你的 `cache` 模拟器所记录的hit总数、miss总数以及eviction总数。

```

1 | printSummary(hit_count, miss_count, eviction_count);

```

5. **对于本次实验，你应该假设每次内存访问都是对齐且不越界的，因此在这个实验中我们可以直接忽略掉 `trace` 文件中的 `size`。**
6. 我们并没有对地址位数等参数做严格的限制，理论上能过 `TestCase` 的结果都是允许的，换句话说，如果所有 `TestCase` 里的最高地址位数是24位的，我们允许你在模拟器代码中使用32或者24位地址。但是为了避免歧义，以及产生一些不必要的麻烦，将地址设成64位一定不会存在问题。本实验Cache模拟器中我们将**64位地址**作为**参考地址位长**，可以采用如下的方式进行地址定义：

```

1 | /* Type: Memory address */
2 | typedef unsigned long long int mem_addr_t;

```

7. 本实验场景为单核无外设的内存场景，**无需考虑**Cache的内存写入策略（Write back/Write through），有兴趣的同学可以尝试思考为什么，以及什么时候Cache需要考虑其他因素对Cache内数据有效性的影响。

Part A 提示

1. Cache? Cache Simulator! Not Cache!

注意我们写的是模拟器，只需要模拟出miss、hit以及eviction的总次数即可。

所以：

- ▶ 不需要存储实际内容（数据块内容）
- ▶ 不需要使用最低的 `b` 位的偏移量(`block offset`)
- ▶ 只需要记录 `miss`、`hit`、`eviction` 的次数即可

Cache应该如何模拟呢？我们可以定义一个Cache Line作为最小单位，单独写一个结构体：

```
1 | struct cache_line {  
2 |  
3 | }
```

那根据我们学的知识，一个 `cache_set` 是 `E` 个 `cache_line`，一个 `cache` 是 2^s 个 `cache_set`。

“

如果你的Cache Line中存有时间戳，那建议你不要使用系统时间，因为程序运行太快可能更新不上，可以自己设置计数器来模拟时间戳。

2. 使用组相联映射以及LRU策略

在需要覆盖时，覆盖一个最近最少使用的块。

“

提示：使用队列或链表存储 `Cache Line`？或者记录每个 `Cache Line` 的时间戳？

3. 处理输入

在处理输入时，我们可以使用 `getopt()` 函数，`getopt()` 的一个示例如下：

```
1 | #include <getopt.h>  
2 | #include <unistd.h>  
3 | #include <stdio.h>  
4 |  
5 | int main(int argc, char** argv){  
6 |     int opt,x,y;  
7 |     /* looping over arguments */  
8 |     while(-1 != (opt = getopt(argc, argv, "x:y:"))){  
9 |         /* determine which argument it's processing */  
10 |         switch(opt) {  
11 |             case 'x':  
12 |                 x = atoi(optarg);  
13 |                 break;  
14 |  
15 |             case 'y':  
16 |
```

```

17         y = atoi(optarg);
18         break;
19     default:
20         printf("wrong argument\n");
21         break;
22     }
23 }

```

如果这个程序的可执行文件为 `foo`。那么运行 `./foo -x 1 -y 2` 就可以把1和2传递给 `main` 函数中定义的变量 `x,y`。

4. 读取trace文件

例如，我们有一个 `yi.trace` 文件内容如下：

```

1 | L 10,1
2 | M 20,1
3 | L 22,1
4 | S 18,1
5 | L 110,1
6 | L 210,1
7 | M 12,1

```

使用 `fscanf`，类似 `scanf`，可以按照以下方法按行读取：

```

1 | FILE * f; //pointer to FILE object
2 | f = fopen("/traces/yi.trace","r"); //open file for reading
3 | char identifier;
4 | unsigned address;
5 | int size;
6 |
7 | // Reading lines like " L 10,1" or " M 20,1"
8 | while(fscanf(f," %c %x,%d", &identifier, &address, &size)>0) {
9 |     // Do stuff
10 | }

```

注意每一行前面的空格~

5. 其他提示

- 可以以一些简单的 `trace` 文件作为输入进行debug（例如 `traces/dave.trace`），确定没问题之后再去尝试运行一些较长的 `trace` 文件。
- 运行 `csim-ref` 时，可以加上 `-v` 参数来输出每次内存访问的的hit、miss、eviction次数，你不需要在你的 `csim.c` 中实现此功能，但我们建议你实现这个功能，因为你可以根据它的输出与 `csim-ref` 进行比对，以便调试。

- ▶ 每个数据读取 (L) 或数据存储 (S) 操作最多可能引起一次miss。数据修改操作 (M) 被视为先进行一次数据读取，然后对同一地址进行一次数据存储。因此，一个M操作可能导致两次Cache hit，或者导致一次miss 加上一次hit以及可能存在的eviction。

Part A 评测

“

注意：最终打分所采用的trace文件和你自己评测所使用的trace文件不同，所以请保证你的程序的正确性。

我们会依次测试 `traces` 目录下的6个文件，如果输出与我们给出的 `csim-ref` 的输出相同，就可以拿到满分。

```
1 | linux$ ./csim -s 1 -E 1 -b 1 -t traces/yi2.trace
2 | linux$ ./csim -s 4 -E 2 -b 4 -t traces/yi.trace
3 | linux$ ./csim -s 2 -E 1 -b 4 -t traces/dave.trace
4 | linux$ ./csim -s 2 -E 1 -b 3 -t traces/trans.trace
5 | linux$ ./csim -s 2 -E 2 -b 3 -t traces/trans.trace
6 | linux$ ./csim -s 2 -E 4 -b 3 -t traces/trans.trace
7 | linux$ ./csim -s 5 -E 1 -b 5 -t traces/trans.trace
8 | linux$ ./csim -s 5 -E 1 -b 5 -t traces/long.trace
```

你可以使用 `csim-ref` 获得正确的结果，在调试时，可以使用 `-v` 参数来获得命中或不命中的具体记录。每个测试文件的命中数量、不命中数量、删除条目数量各占1/3的分值，也就是说，如果一个测试文件占9分，你的hit数量正确，miss数量也正确，但是eviction数量有误，只能得到6分。

当然你如果只想判断拿了多少分不需要这么复杂，我们提供了一个自动测试分数的程序，叫做 `test-csim`，可以测试你的代码在以 `traces` 目录下各个 `trace` 文件作为输入的正确性。注意运行 `test-csim` 之前要先编译你的代码。

典型的输出如下（这是一个满分输出）：

```
linux$ make
linux$ ./test-csim
```

	Your simulator			Reference simulator			
Points (s,E,b)	Hits	Misses	Evicts	Hits	Misses	Evicts	
9 (1,1,1)	9	8	6	9	8	6	traces/yi2.trace
9 (4,2,4)	4	5	2	4	5	2	traces/yi.trace
9 (2,1,4)	2	3	1	2	3	1	traces/dave.trace
9 (2,1,3)	167	71	67	167	71	67	traces/trans.trace
9 (2,2,3)	201	37	29	201	37	29	traces/trans.trace
9 (2,4,3)	212	26	10	212	26	10	traces/trans.trace
9 (5,1,5)	231	7	0	231	7	0	traces/trans.trace
9 (5,1,5)	265189	21775	21743	265189	21775	21743	traces/long.trace

72

Part B: 优化矩阵转置函数

Part B 实验内容

Part B部分，你需要要在 `trans.c` 中写一个矩阵转置函数，使其在运行过程中尽可能少地引起Cache Miss。

A 表示一个矩阵， $A_{i,j}$ 表示 A 矩阵的第 i 行、第 j 列，矩阵 A 的转置用 A^T 表示，满足 $A_{i,j} = A_{j,i}^T$ 。

为了让你更好地开始优化转置函数，在 `trans.c` 中我们给出了一个示例的转置函数，这个函数可以计算 $N \times M$ 矩阵 A 的转置并存储到 $M \times N$ 矩阵 B 中：

```

1 | char trans_desc[] = "Simple row-wise scan transpose";
2 | void trans(int M, int N, int A[N][M], int B[M][N])
3 | {
4 |     int i, j, tmp;
5 |
6 |     for (i = 0; i < N; i++) {
7 |         for (j = 0; j < M; j++) {
8 |             tmp = A[i][j];
9 |             B[j][i] = tmp;
10 |        }
11 |    }
12 | }
```

显然这个函数是正确的，但是对于Cache而言执行效率很低，会导致大量的Cache Miss，在Part B中你的工作是在 `trans.c` 中写一个功能相同的函数 `transpose_submit`，尽可能地减小Cache Miss次数。

```

1 | char transpose_submit_desc[] = "Transpose submission";
2 | void transpose_submit(int M, int N, int A[N][M], int B[M][N])
3 | {
4 |     // your code here
5 | }
```

注意不要修改上面的描述字符串 `"Transpose submission"`，打分时会寻找这个字符串来确定你提交的函数。

Part B 评测



你可能会奇怪为什么Part B的实验评测部分会在这时候说明，实际上Part B是允许你进行“**面向测试用例编程**”，所以在这里先给出PartB的所有的测试用例，以及每个测试用例对于Cache Miss次数的要求。

可以通过运行 `test-trans` 文件来测试你的函数在运行时Cache Miss的数量，例如你想测试你的转置函数对于一个 61×67 的矩阵进行转置时Cache Miss的数量，可以使用如下命令进行测试：

```
1 | linux$ ./test-trans -M 61 -N 67
```

命令行中的 `-M` 以及 `-N` 后跟的是矩阵的行数和列数。

在Part B测试时，**Cache结构是固定的**，参数为(`s = 5`, `E = 1`, `b = 5`)，并且你不需要对于所有的大小的矩阵都完成转置的优化，我们在进行评分的时候**只会**对如下三个测试用例进行测试：

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

这三个测试用例以及每个测试用例的Cache Miss限制如下（假设某个测试用例中Cache Miss的次数为 m ）：

- 32×32 : 10 points if $m < 300$, 0 points if $m > 600$
- 64×64 : 8 points if $m < 1300$, 0 points if $m > 2000$
- 61×67 : 10 points if $m < 2000$, 0 points if $m > 3000$

也就是说，你可以在你的代码中显式地判断输入参数中矩阵的行数和列数，然后对每个测试用例进行单独编程。

“

在开始写PartB之前请先看一下下面PartB的代码规则部分，明确要求后，你可以省去一些多余的工作。

Part B 代码规则

1. `trans.c` 文件一定要可以正常编译（0 warning 0 error）。
2. 在每个转置函数中，你只允许定义最多12个 `int` 类型的局部变量。
3. 不允许使用任何 `long` 类型的变量或者尝试将多个值存储到一个变量中。
4. 不允许使用递归。
5. 如果你选择使用辅助函数，则从转置函数开始的栈中同时存在局部变量数目不能超过12个，例如你的转置函数定义了8个局部变量，转置函数中调用了一个函数定义了2个局部变量，这个函数又定义了4个局部变量，那栈中就会存在14个局部变量，你就违反规则了。
6. 你的转置函数不能修改A矩阵的值，但可以随意修改B矩阵的值。
7. 代码中不允许定义任何数组或者调用类似 `malloc` 的函数。

在最后评分时，都会按照如上的代码规则进行严苛的检查，请大家注意不要出现超出代码规则的操作。

Part B 提示

1. Cache Miss的三种情况：

- ▶ **Complusory Miss**: 在刚开始冷启动的时候, 这时候Cache内没有包含任何局部的任何信息, 此时Cache Miss是无法忽略的无法避免的, 但是准确估计这部分的Cache Miss数量对后续的实现有很多好处。
 - ▶ **Capacity Misses**: 当你的工作集 (Working Set) 太大的时候, 往往Cache没法保存下所有的信息, 此时会出现一些由于容量不足而导致的Cache Miss, 比如我们实验中的Cache, 通常只能保存下 $32 * 8$ 个 `int`。为了可以减少这种**Capacity Misses**, 你可以尝试缩小你的求解问题的规模, 大问题转化为小问题, 大矩阵转化为几个小矩阵 (啊呀我好像说漏嘴了一些关键信息 (笑))。
 - ▶ **Conflict Misses**: 有些时候, 不同地址的内容会映射到同一个Cache Line, 这会导致简单的操作却反复的出现Cache Miss, 一个好的解决方法是, 另开空间或者采用临时变量 (在本实验中允许12个以内的临时变量), 来减轻这些问题的产生。
2. 64×64 矩阵转置的优化可能比较复杂, 可以先完成 32×32 和 61×67 矩阵转置的优化, 从中寻找灵感, 再去完成 64×64 矩阵转置的优化。

Cache性能分析

如果大家看完上面的提示之后还是没有什么思路, 不要着急, 这一小节对一个简单的矩阵转置函数的Cache Miss次数进行粗略计算, 让你了解Cache访问过程, 从中你可能会找到减少Cache Miss的办法。

首先我们不考虑什么Cache Miss, 仅关注矩阵转置的正确性, 你可能会写出如下三行非常精简的代码:

```
1 | char transpose_submit_desc[] = "Transpose submission";
2 | void transpose_submit(int M, int N, int A[N][M], int B[M][N])
3 | {
4 |     for(int i = 0; i < N; i++)
5 |         for(int j = 0; j < M; j++)
6 |             B[j][i] = A[i][j];
7 | }
```

OK, 大功告成! 这一段代码肯定是功能正确的, 于是我们信心满满地开始尝试使用评分软件来测试咱们可以得到的分数。我们把这个代码直接写入 `trans.c`, 编译后执行测评程序得到的典型运行结果如下

我们直接使用 `test-trans`, 进行第一个测试用例 (32×32) 的测试。

```
1 | linux$ ./test-trans -M 32 -N 32
```

得到的典型输出结果:

```
1 | linux$ ./test-trans -M 32 -N 32
2 |
3 | Function 0 (2 total)
4 | Step 1: Validating and generating memory traces
5 | File deleted successfully
6 | Step 2: Evaluating performance (s=5, E=1, b=5)
7 | func 0 (Transpose submission): hits:869, misses:1184, evictions:1152
8 |
9 | Function 1 (2 total)
```

```

10 | Step 1: Validating and generating memory traces
11 | File deleted successfully
12 | Step 2: Evaluating performance (s=5, E=1, b=5)
13 | func 1 (Simple row-wise scan transpose): hits:869, misses:1184, evictions:1152
14 |
15 | Summary for official submission (func 0): correctness=1 misses=1184
16 |
17 | TEST_TRANS_RESULTS=1:1184

```

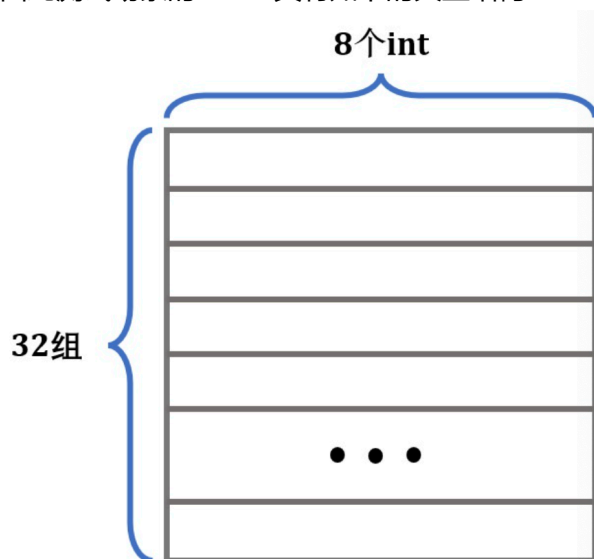
运行结果的输出含义是，在 (s=5, E=1, b=5) 的Cache下，你的函数运行时，Cache Hit的次数是 hits:869，Cache Miss的次数是 misses:1184。在Part B评测部分我们提到，如果你想在 32×32 的 testcase 下获得满分，你需要把你的程序控制在300次Cache Miss以下。

那我们这里的程序性能到底糟糕在哪呢？这就需要分析一下程序运行的访存过程。

程序运行的Cache访问过程分析

我们的测试场景是一个直接映射的、Block大小是32字节的、一共有32个Cache Set的Cache模拟器(s=5, E=1, b=5)。

因此测试场景的Cache具有如下的典型结构：



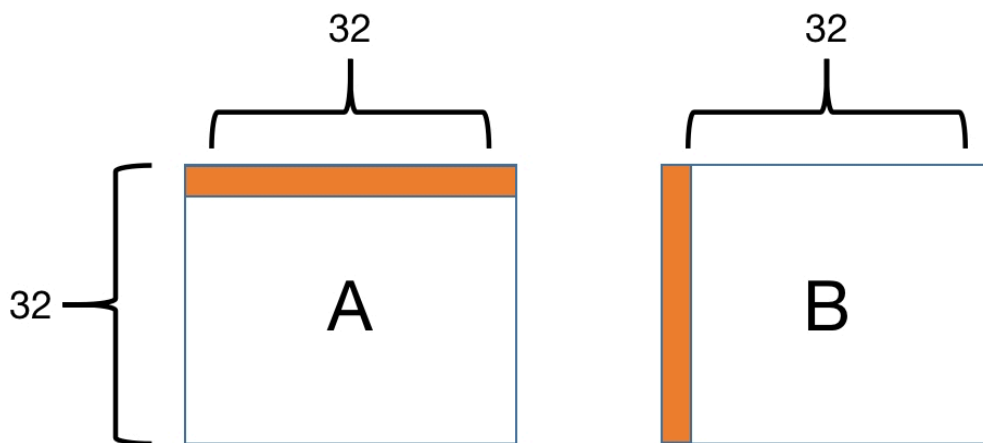
一个Cache Line可以保存8个 int，我们以这个Cache结构为例，考虑我们刚才的暴力做法：

```

1 | void transpose_submit(int M, int N, int A[N][M], int B[M][N])
2 | {
3 |     for(int i = 0; i < N; i++)
4 |         for(int j = 0; j < M; j++)
5 |             B[j][i] = A[i][j];
6 | }

```

这里我们会按行优先读取 A 矩阵，然后一列一列地写入 B 矩阵。



我们知道，Cache是以Cache Line形式读取内存的。以第1行为例，在从内存读 `A[0][0]` 的时候，除了 `A[0][0]` 被加载到Cache中，它之后的 `A[0][1]---A[0][7]` 也会被加载进Cache。

但是内容写入 `B` 矩阵的时候是一列一列地写入，在列上相邻的元素不在一个内存块上，这样每次写入都不命中Cache。并且一列写完之后再返回，原来写入Cache的内容可能被覆盖了，这样就又会不命中。

接下来我们来定量地分析Cache Miss的次数。Cache只够存储一个矩阵的四分之一，`A` 中的元素对应的Cache Line每隔8行就会重复。`A` 和 `B` 的地址由于取余关系，每个元素对应的地址是相同的，各个元素对应Cache Line如下：

8个int			
0	1	2	3
	●	●	●
28	29	30	31
0	1	2	3
	●	●	●
28	29	30	31
0	1	2	3
	●	●	●
28	29	30	31
0	1	2	3
	●	●	●
28	29	30	31

对于 `A`，每8个 `int` 就会占满Cache的一组，所以每一行会有 $32/8=4$ 次不命中；而对于 `B`，考虑最坏情况，每一列都有32次不命中，由此，算出总不命中次数为 $4\times32+32\times32=1152$ 。

但是为什么出现了1184次的Cache Miss，这是由于对角线上的元素通常存在一些特殊情况，对角线上的情况请大家自行分析哦~

评分方法

- Part A 72分
- Part B 28分

和往常一样，我们将最终用于评分的 `driver.py` 脚本也分发给了大家，大家可以用于快速自测分数，使用方法如下：

```
1 | linux$ make
2 | linux$ ./driver.py
```

先 `make` 保证当前你的可执行文件已编译为你最新的提交版本，确认无误后，执行 `./driver.py` 指令。

通常来说你会获得类似如下的输出：（这是一个满分输出）

```
1 | linux$ ./driver.py
2 | Part A: Testing cache simulator
3 | Running ./test-csim
4 |
5 |           Your simulator      Reference simulator
6 | Points (s,E,b) Hits Misses Evicts Hits Misses Evicts
7 | 9 (1,1,1)      9      8      6      9      8      6 traces/yi2.trace
8 | 9 (4,2,4)      4      5      2      4      5      2 traces/yi.trace
9 | 9 (2,1,4)      2      3      1      2      3      1 traces/dave.trace
10 | 9 (2,1,3)     167     71     67    167     71     67 traces/trans.trace
11 | 9 (2,2,3)     201     37     29    201     37     29 traces/trans.trace
12 | 9 (2,4,3)     212     26     10    212     26     10 traces/trans.trace
13 | 9 (5,1,5)     231      7      0    231      7      0 traces/trans.trace
14 | 9 (5,1,5)  265189  21775  21743  265189  21775  21743 traces/long.trace
15 | 72
16 |
17 | Part B: Testing transpose function
18 | Running ./test-trans -M 32 -N 32
19 | Running ./test-trans -M 64 -N 64
20 | Running ./test-trans -M 61 -N 67
21 |
22 | Cache Lab summary:
23 |
24 |           Points      Max pts      Misses
25 | Csim correctness    72.0         72
26 | Trans perf 32x32    10.0         10      288
27 | Trans perf 64x64     8.0          8     1108
28 | Trans perf 61x67    10.0         10     1914
29 | Total points    100.0        100
```

如上分别是各个部分的组成，最后几行会输出你的总成绩。你可以通过如上方法快速得知你最终会得到多少的分数。

迟交

在超过原定的截止时间后，我们仍然接受同学的提交。此时，在lab中能获得的最高分数将随着迟交天数的增加而减少，具体服从以下给分策略：

超时7天（含7天）以内时，每天扣除3%的分数

超时7~14天（含14天）时，每天扣除4%的分数

超时14天以上时，每天扣除7%的分数，直至扣完


以上策略中超时不足一天的，均按一天计，自ddl时间开始计算。届时在线学习平台将开放迟交通道。

评分样例：如某同学小H在lab中取得95分，但晚交3天，那么他的最终分数就为 $95 * (1 - 3 * 3\%) = 86.45$ 分。同样的分数在晚交8天时，最终分数则为 $95 * (1 - 7 * 3\% - 1 * 4\%) = 71.25$ 分。

代码提交

每次在 `cachelab-handout` 目录下执行 `make` 命令时，都会生成一个名为 `<userid>-lab4-handin.zip` 的压缩文件。

如果不是请重命名 `<userid>-lab4-handin.zip`（其中 `<userid>` 为 `你的学号-ics`，也就是ICSServer中你的用户名）。

在[在线学习平台](#)  上的作业模块中，将该文件作为附件提交即可。