清华大学 2020 年

全国优秀中学生信息学体验营

第三试

简单 Cache 系统实现

时间：2019 年 12 月 15 日 18:00 ~ 21:00

题目名称	Cache 一致性协议	Cache 替换算法	只读 Cache 实现	读写 Cache 实现	LIRS 替换算法实现	期待你的声音
题目类型	传统型	传统型	传统型	传统型	传统型	提交答案型
输入	标准输入	标准输入	标准输入	标准输入	标准输入	`*.in`
输出	标准输出	标准输出	标准输出	标准输出	标准输出	`*.out`
每个测试点时限	1.0 秒	1.0 秒	1.0 秒	1.0 秒	1.0 秒	N/A
内存限制	512 MB	512 MB	512 MB	512 MB	512 MB	N/A
子任务数目	4	7	3	5	5	1
测试点是否等分	否	否	否	否	否	否
预测试点数目	4	14	20	42	48	0

Cache 一致性协议（coherence）

【题目背景】

Cache 是计算机存储结构的重要组成部分。在多核 CPU 中，保证 Cache 的一致性（coherence）是非常重要的。

【题目描述】

在开始完成本题前，请确保你已经完整地阅读了下发的《学习手册》。

在本题中，你需要模拟多个属于不同 CPU 的 Cache，使得它们按着 MESI 协议的规定进行工作，处理来自 CPU 的读写请求。

为了简化起见，在本题中，我们只关注多个 Cache 中对应于同一个物理内存块的 cacheline（即一个 **MESI 组**）。它们不会被替换出 Cache，始终在四个状态间进行转换。

【输入格式】

从标准输入读入数据。

输入的第一行包含两个整数 M, N，前者代表 CPU 和对应 Cache 的数量（也是你需要处理的 MESI 组中 cacheline 的数量），后者代表收到的请求数量。各个 CPU 和 Cache 编号为 $1, \ldots, M$。

下面的 N 行，每行可能为下列情况之一：

1. `R m` 或者 `W m`：表示一个来自 CPU 的请求，其中 $1 \le m \le M$ 代表收到请求的 Cache 编号，`R` 表示读（即 `PrRd`），`W` 表示写（即 `PrWr`）；
2. `F`：表示一个来自外部的 Flush 请求到总线，即要求所有 Cache 将所有修改过的内容写回主存；
3. `Q m` 表示一个状态查询，询问 `m` 号 Cache 的这一 cacheline 当前的 MESI 状态。

【输出格式】

输出到标准输出。

你需要依次处理输入中的每一行。对于 `R/W/F` 类型的操作，你应该**顺次**输出 Cache 处理此请求时执行的每一个操作，格式为 `OP m`。其中 `m` 为 Cache 编号，`OP` 的取值可能为：

- `BusRd`
- `BusRdX`
- `BusUpgr`
- `FlushOpt`
- `MemoryRd`

- `MemoryWr`

其中前四个操作与 MESI 协议中定义的一致，后两个代表对主存的读写。请注意，对于这三种操作，每一行输入可能会产生零到多行输出。

对于 `Q`，你需要输出编号为 `m` 的 Cache 的对应 cacheline 当前的状态（同样单独成行），可能为以下之一：

- `Invalid`
- `Exclusive`
- `Shared`
- `Modified`

【样例 1 输入】

```
3 10
R 1
W 1
R 3
Q 3
W 3
R 1
R 3
W 2
W 1
Q 2
```

【样例 1 输出】

```
BusRd 1
MemoryRd 1
BusRd 3
FlushOpt 1
MemoryWr 1
Shared
BusUpgr 3
BusRd 1
FlushOpt 3
MemoryWr 3
BusRdX 2
FlushOpt 1
```

```
BusRdX 1
FlushOpt 2
MemoryWr 2
Invalid
```

【样例 1 解释】

本样例即为学习手册中给出的示例，请特别注意各个操作的顺序。

【样例 2 输入】

```
2 6
W 1
F
W 2
F
R 1
F
```

【样例 2 输出】

```
BusRdX 1
MemoryRd 1
MemoryWr 1
BusRdX 2
FlushOpt 1
MemoryWr 2
BusRd 1
FlushOpt 2
```

【样例 2 解释】

第一次 `Flush` 的时候，编号为 1 的 Cache 处于 `Modified` 状态，所以进行一次 `MemoryWr 1` 写回。第二次 `Flush` 的时候，编号为 2 的 Cache 处于 `Modified` 状态，所以进行一次 `MemoryWr 2` 写回。第三次 `Flush` 的时候，没有 Cache 处于 `Modified` 状态，所以没有任何操作。

【子任务】

子任务	子任务分数	M	N
1	10	2	$\leq 10^2$
2		4	$\leq 10^4$
3		16	
4		128	

Cache 替换算法（algo）

【题目背景】

替换算法是决定 Cache 效率的关键因素。在 Cache 的设计过程中，通过软件对于替换算法进行模拟是有效的手段。

【题目描述】

在本题中，你需要实现多种指定的 Cache 替换算法，包括伪随机（Pseudo Random）、先入先出（FIFO）、最不经常使用（LFU）、最近最少使用（LRU）、最近最多使用（MRU）以及伪最近最少使用（PLRU）。各个算法的实现**以学习手册中的说明为准**。

由于替换算法与具体的 Cache 结构无关，因此我们在本题中仅关注 Cache 中的一组 cacheline，并且仅给出地址的标签（tag）部分。

【输入格式】

从标准输入读入数据。

输入的第一行包括三个整数 W, R, N，其中 $1 \le W \le 16$ 且一定为 2 的幂，表示组内块数（相连度），R 表示替换算法的编号，N 表示访问序列的长度。

需要实现的替换算法包括以下几种：

1. N/A 无替换算法（保证当 $W = 1$ 时，R 只能取为 1，当 $R = 1$ 时，W 只能取为 1）
2. 伪随机（Pseudo Random）算法
3. 先入先出（FIFO）算法
4. 最不经常使用（LFU）算法
5. 最近最少使用（LRU）算法
6. 最近最多使用（MRU）算法
7. 伪最近最少使用（PLRU）算法

接下来 N 行每行包括一个整数 $n < 2^{31}$，表明访问地址对应的 tag。

【输出格式】

输出到标准输出。

输出需要包含 N 行，每行一个整数，表示对于这次访问算法选出的替换块（victim）编号。如果某次访问不引发替换（如命中了 Cache，或者被替换的块处于 Invalid 状态），则输出 -1。

【样例 1】

见题目目录下的 ***1.in*** 与 ***1.ans***。

【样例 1 解释】

第一个样例中 $W = 1, R = 1, N = 10$ ，只有一个块，所以每次替换都只能替换第 0 块，下面的样例都用 I 表示 Invalid。

访问	访问前第 0 块状态	操作
0x0000000	I	填入第 0 块
0x0000000	Tag=0x0000000	命中
0x0000000	Tag=0x0000000	命中
0x0000001	Tag=0x0000000	替换第 0 块
0x1000000	Tag=0x0000001	替换第 0 块
0x0000100	Tag=0x1000000	替换第 0 块
0x0000000	Tag=0x0000100	替换第 0 块
0x0000000	Tag=0x0000000	命中
0x0000100	Tag=0x0000000	替换第 0 块
0x0000100	Tag=0x0000100	命中

【样例 2】

见题目目录下的 ***2.in*** 与 ***2.ans***。

【样例 2 解释】

第二个样例中 $W = 2, R = 2, N = 10$ ，有两个块，按照伪随机数进行替换。

访问	访问前第 0 块状态	访问前第 1 块状态	操作
0x0000000	I	I	填入第 0 块
0x1000000	Tag=0x0000000	I	填入第 1 块
0x0000000	Tag=0x0000000	Tag=0x1000000	命中
0x1000000	Tag=0x0000000	Tag=0x1000000	命中
0x2000000	Tag=0x0000000	Tag=0x1000000	伪随机数 16838，替换第 0 块
0x1000000	Tag=0x2000000	Tag=0x1000000	命中
0x2000000	Tag=0x2000000	Tag=0x1000000	命中
0x3000000	Tag=0x2000000	Tag=0x1000000	伪随机数 5758，替换第 0 块
0x2000000	Tag=0x3000000	Tag=0x1000000	伪随机数 17515，替换第 1 块
0x4000000	Tag=0x3000000	Tag=0x2000000	伪随机数 31051，替换第 1 块

【样例 3】

见题目目录下的 ***3.in*** 与 ***3.ans***。

【样例 3 解释】

第三个样例中 $W = 2, R = 3, N = 10$ ，有两个块，按照 FIFO 算法进行替换。下列表格中 T 代表的是进入缓存的时间。

访问	访问前第 0 块状态	访问前第 1 块状态	操作
0x0000000	I	I	填入第 0 块
0x0000001	Tag=0x0000000, T=0	I	填入第 1 块
0x0000001	Tag=0x0000000, T=0	Tag=0x0000001, T=1	命中
0x0000002	Tag=0x0000000, T=0	Tag=0x0000001, T=1	替换第 0 块
0x0000003	Tag=0x0000002, T=3	Tag=0x0000001, T=1	替换第 1 块
0x0000000	Tag=0x0000002, T=3	Tag=0x0000003, T=4	替换第 0 块
0x0000002	Tag=0x0000000, T=5	Tag=0x0000003, T=4	替换第 1 块
0x0000001	Tag=0x0000000, T=5	Tag=0x0000002, T=6	替换第 0 块
0x0000002	Tag=0x0000001, T=7	Tag=0x0000002, T=6	命中
0x0000004	Tag=0x0000001, T=7	Tag=0x0000002, T=6	替换第 1 块

【样例 4】

见题目目录下的 ***4.in*** 与 ***4.ans***。

【样例 4 解释】

第四个样例中 $W = 2, R = 4, N = 10$ ，有两个块，按照 LFU 算法进行替换。下列表格中 T 代表的是被访问的次数。

访问	访问前第 0 块状态	访问前第 1 块状态	操作
0x0000000	I	I	填入第 0 块
0x0000001	Tag=0x0000000, T=1	I	填入第 1 块
0x0000000	Tag=0x0000000, T=1	Tag=0x0000001, T=1	命中
0x0000002	Tag=0x0000000, T=2	Tag=0x0000001, T=1	替换第 1 块
0x0000003	Tag=0x0000000, T=2	Tag=0x0000002, T=1	替换第 1 块
0x0000003	Tag=0x0000000, T=2	Tag=0x0000003, T=1	命中
0x0000002	Tag=0x0000000, T=2	Tag=0x0000003, T=2	替换第 0 块

访问	访问前第 0 块状态	访问前第 1 块状态	操作
0x0000001	Tag=0x0000002, T=1	Tag=0x0000003, T=2	替换第 0 块
0x0000002	Tag=0x0000001, T=1	Tag=0x0000003, T=2	替换第 0 块
0x0000004	Tag=0x0000002, T=1	Tag=0x0000003, T=2	替换第 0 块

【样例 5】

见题目目录下的 ***5.in*** 与 ***5.ans***。

【样例 5 解释】

第五个样例中 $W = 2, R = 5, N = 10$ ，有两个块，按照 LRU 算法进行替换。下列表格中 T 代表的是最后一次被访问的时间。

访问	访问前第 0 块状态	访问前第 1 块状态	操作
0x0000000	I	I	填入第 0 块
0x0000001	Tag=0x0000000, T=0	I	填入第 1 块
0x0000000	Tag=0x0000000, T=0	Tag=0x0000001, T=1	命中
0x0000001	Tag=0x0000000, T=2	Tag=0x0000001, T=1	命中
0x0000002	Tag=0x0000000, T=2	Tag=0x0000001, T=3	替换第 0 块
0x0000001	Tag=0x0000002, T=4	Tag=0x0000001, T=3	命中
0x0000002	Tag=0x0000002, T=4	Tag=0x0000001, T=5	命中
0x0000000	Tag=0x0000002, T=6	Tag=0x0000001, T=5	替换第 1 块
0x0000000	Tag=0x0000002, T=6	Tag=0x0000000, T=7	命中
0x0000001	Tag=0x0000002, T=6	Tag=0x0000000, T=8	替换第 0 块

【样例 6】

见题目目录下的 ***6.in*** 与 ***6.ans***。

【样例 6 解释】

第六个样例中 $W = 2, R = 6, N = 10$ ，有两个块，按照 MRU 算法进行替换。下列表格中 T 代表的是最后一次被访问的时间。

访问	访问前第 0 块状态	访问前第 1 块状态	操作
0x0000000	I	I	填入第 0 块

访问	访问前第 0 块状态	访问前第 1 块状态	操作
0x0000001	Tag=0x0000000, T=0	I	填入第 1 块
0x0000000	Tag=0x0000000, T=0	Tag=0x0000001, T=1	命中
0x0000001	Tag=0x0000000, T=2	Tag=0x0000001, T=1	命中
0x0000002	Tag=0x0000000, T=2	Tag=0x0000001, T=3	替换第 1 块
0x0000001	Tag=0x0000000, T=2	Tag=0x0000002, T=4	替换第 1 块
0x0000002	Tag=0x0000000, T=2	Tag=0x0000001, T=5	替换第 1 块
0x0000000	Tag=0x0000000, T=2	Tag=0x0000002, T=6	命中
0x0000000	Tag=0x0000000, T=7	Tag=0x0000002, T=6	命中
0x0000001	Tag=0x0000000, T=8	Tag=0x0000002, T=6	替换第 0 块

【样例 7】

见题目目录下的 ***7.in*** 与 ***7.ans***。

【样例 7 解释】

第七个样例中 $W = 4, R = 7, N = 10$ ，有四个块，按照 PLRU 算法进行替换。

访问	访问前树根	访问前树根的左孩子	访问前树根的右孩子	操作
0x0000000	0	0	0	填入第 0 块
0x0000001	1	1	0	填入第 2 块
0x0000002	0	1	1	填入第 1 块
0x0000003	1	0	1	填入第 3 块
0x0000004	0	0	0	替换第 0 块
0x0000001	1	1	0	命中
0x0000000	0	1	1	替换第 1 块
0x0000000	1	0	1	命中
0x0000003	1	0	1	命中
0x0000002	0	0	0	替换第 0 块

【子任务】

子任务	子任务分数	W	R	N
1	4	≤ 1	1	$\leq 10^3$
2		≤ 8	2	
3	8		3	
4			4	
5			5	
6			6	
7	16		7	

【提示】

1. 部分算法的实现比较困难，因此你可以先实现容易上手的部分，也能获得大部分的分数；
2. 在每一个 cacheline 中，可能需要存储额外的信息（如是否有效、访问次数等），以便于替换算法的实现。

只读 Cache 实现（readonly）

【题目描述】

在本题中，你需要实现一个仅支持读操作的 Cache。

规定计算机的物理地址空间大小为 4GB，Cache 总大小为 64KB，Cache 和主存的块大小为 4 字节。即有 $N = 32, M = 16, K = 2$。如果你不了解这些参数的具体含义，请认真阅读学习手册的“基本假设”一节。

为了方便实现，我们提供了一套简单的框架。你的程序需要包含 `cache_ro.h` 头文件，并实现以下的函数：

```
void cache_init(uint8_t associativity, uint8_t algorithm);
uint32_t cache_read(uint32_t address);
```

同时，我们提供了以下的辅助函数：

```
uint32_t memory_read(uint32_t address);
```

测试框架首先会调用 `cache_init` 函数，同时告知相连度 W 和替换算法 R（定义与上一题相同）。而后，框架会调用若干次 `cache_read`，向你的 Cache 发出读取请求。当且仅当你需要访问内存获取数据时，只需调用 `memory_read` 函数即可。

为了方便测试，我们提供了框架的一个实现，你只需要将 `cache_ro.cpp` 与你编写的代码一起编译，即可获得一个可执行文件，能够接受标准输入并向标准输出打印信息。它使用的输出/输出格式可见下面的叙述。在做题时，你必须**精确地**调用 `memory_read` 从内存中读取数据并且完整地保存下来，而**不能**对数据进行任何假设，否则将无法通过评测。

框架会比对你的函数**每一次**返回的读取结果和访问内存的地址，确保与标准实现的**顺序与内容**完全相同。只要你的实现的输出存在差异（如返回结果不一致、多或者少调用了内存读取），就会立刻退出并判定为错误。

【输入格式】

从标准输入读入数据。

文件的第一行包括三个整数 W, R, N，其意义与上一题中完全相同。

从第二行开始有 N 组操作，每组操作的构成为（`0xFFFFFFFF` 表示用 `0x%08X` 格式化输出的十六进制 32 位整数）：

1. 首先是一条 `R 0xFFFFFFFF`，表示对 Cache 的读操作，交互库会调用 `cache_read` 函数

2. 出现零到一条 `MR 0xFFFFFFFF 0xFFFFFFFF`，表示 Cache 对内存进行了一次读操作，前后两个十六进制数分别对应地址和数据，即 `memory_read` 的参数和返回值
3. 最后是一条 `CR 0xFFFFFFFF` ，表示 Cache 读操作的结果，即 `cache_read` 的返回值。

【输出格式】

输出到标准输出。

当交互库判定你实现的 Cache 通过了测试时，会输出 `PASS`；否则，将会输出 `FAIL` 和错误发生时读入的一行数据。

【样例 1】

见题目目录下的 ***1.in*** 与 ***1.ans***。

【样例 1 解释】

第一个样例中 $W = 1, R = 1, N = 10$ ，一共有 16384 个组，每组一个块，地址的最高 16 位为 tag，接着 14 位为 index，最后 2 位为 offset。

访问	Tag	Index	操作
0x00000000	0x0000	0x0000	填入
0x00000000	0x0000	0x0000	命中
0x00000004	0x0000	0x0001	填入
0x00000008	0x0000	0x0002	填入
0x10000008	0x1000	0x0002	替换
0x00000108	0x0000	0x0042	填入
0x00000004	0x0000	0x0001	命中
0x00000118	0x0000	0x0046	填入
0x00001018	0x0000	0x0406	填入
0x00001008	0x0000	0x0402	填入

【样例 2】

见题目目录下的 ***2.in*** 与 ***2.ans***。

【样例 2 解释】

第二个样例中 $W = 2, R = 2, N = 10$ ，一共有 8192 个组，每组 2 个块，地址的最高 17 位为 tag，接着 13 位为 index，最后 2 位为 offset。

访问	Tag	Index	操作
0x00000000	0x0000	0x0000	填入
0x00008000	0x0001	0x0000	填入
0x00000000	0x0000	0x0000	命中
0x00008000	0x0001	0x0000	命中
0x00010000	0x0002	0x0000	伪随机得到 16838，替换第 0 块
0x00010004	0x0002	0x0001	填入
0x00008008	0x0001	0x0002	填入
0x00018000	0x0003	0x0000	伪随机得到 5758，替换第 0 块
0x00010000	0x0002	0x0000	伪随机得到 17515，替换第 1 块
0x00020000	0x0004	0x0000	伪随机得到 31051，替换第 1 块

【子任务】

子任务	子任务分数	W	R	N
1	16	≤ 1	≤ 1	$\leq 10^4$
2		≤ 4	≤ 6	
3	24	≤ 8	≤ 7	

读写 Cache 实现（readwrite）

【题目描述】

在本题中，你需要实现一个支持读写操作的 Cache。各个假设与上一题中相同。

本题中，你的程序需要包含 `rw_cache.h` 头文件，并实现以下的函数：

```
void cache_init(uint8_t associativity, uint8_t algorithm,
                uint8_t write_hit_policy, uint8_t write_miss_policy);
uint32_t cache_read(uint32_t address);
void cache_write(uint32_t address, uint32_t data);
```

我们提供了以下的辅助函数：

```
uint32_t memory_read(uint32_t address);
void memory_write(uint32_t address, uint32_t data);
```

同样地，测试框架首先会调用 `cache_init` 函数。相比上题中，这一函数多传入了两个参数指定 Cache 的写策略：

- `write_hit_policy`：0 表示 Write Through，1 表示 Write Back
- `write_miss_policy`：0 表示 No-write Allocate，1 表示 Write Allocate

而后，框架会调用若干次 `cache_read` 和 `cache_write`，向你的 Cache 发出读取请求。当你需要访问内存读写数据时，调用提供的两个辅助函数即可。

我们提供了框架的一个实现 `cache_rw.cpp`，使用方法与上一题中也相同。同样地，框架会对你的 Cache 返回的读取结果和所有的内存访问（包括读和写）与标准实现进行比对，只有完全相同时才认为实现正确。

【输入格式】

从标准输入读入数据。

文件的第一行包括五个整数 $W, R, P_{hit}, P_{miss}, N$，分别为相连度、替换算法、写命中策略、写缺失策略、请求数量。

从第二行开始的 N 组操作，格式有以下两类（地址与数据均为 16 进制整数）：

第一类是读操作，首先是一条 `R 0xFFFFFFFF`，表示对 Cache 的读操作，交互库会调用 `cache_read` 函数；接着出现零到一条 `MR 0xFFFFFFFF 0xFFFFFFFF`，表示 Cache 向内存进行了一次读操作，前后两个十六进制数分别对应地址和数据，即 `memory_read` 的参数和返回值；接着出现零到一条 `MW 0xFFFFFFFF 0xFFFFFFFF`，表示 Cache 向内存进行了一次写操作，前后两个十六进制数分别对应地址和数据，是 `memory_write` 函数的两个参数；最后是一条 `CR 0xFFFFFFFF`，表示对 Cache 的读操作的结果，即 `cache_read` 的返回值。

第二类是写操作，首先是一条 `W 0xFFFFFFFF 0xFFFFFFFF`，表示对 Cache 的写操作，交互库会调用 `cache_write` 函数，前后两个十六进制数分别对应两个参数；接着出现零到一条 `MR 0xFFFFFFFF 0xFFFFFFFF`，表示 Cache 向内存进行了一次读操作，前后两个十六进制数分别对应地址和数据，即 `memory_read` 的参数和返回值；最后出现零到一条 `MW 0xFFFFFFFF 0xFFFFFFFF`，表示 Cache 向内存进行了一次写操作，前后两个十六进制数分别对应地址和数据，是 `memory_write` 函数的两个参数；

【输出格式】

输出到标准输出。

当交互库判定你实现的 Cache 通过了测试时，会输出 `PASS`；否则，将会输出 `FAIL` 和错误发生时读入的一行数据。

【样例 1】

见题目目录下的 ***1.in*** 与 ***1.ans***。

【样例 1 解释】

第一个样例中 $W = 1, R = 1, Phit = 0, Pmiss = 0, N = 10$，策略是 Write Through + No-write Allocate，一共有 16384 个组，每组一个块，地址的最高 16 位为 tag，接着 14 位为 index，最后 2 位为 offset。

访问	Tag	Index	操作
W 0x00000000 0x12341234	0x0000	0x0000	写缺失，因为 No-write Allocate，所以 MW
R 0x00000000	0x0000	0x0000	读缺失，MR
W 0x00000000 0x43214321	0x0000	0x0000	写命中，因为 Write Through，所以 MW
R 0x00000000	0x0000	0x0000	读命中
R 0x00000000	0x0000	0x0000	读命中
R 0x00000004	0x0000	0x0001	读缺失，MR
W 0x10000000 0x11111111	0x1000	0x0000	写缺失，因为 No-write Allocate，所以 MW
R 0x00000000	0x0000	0x0000	读命中
W 0x00000004 0x22222222	0x0000	0x0001	写命中，因为 Write Through，所以 MW
R 0x10000000	0x1000	0x0000	读缺失，MR

【样例 2】

见题目目录下的 ***2.in*** 与 ***2.ans***。

【样例 2 解释】

第二个样例中 $W = 1, R = 1, Phit = 0, Pmiss = 1, N = 10$ ，策略是 Write Through + Write Allocate，一共有 16384 个组，每组一个块，地址的最高 16 位为 tag，接着 14 位为 index，最后 2 位为 offset。

访问	Tag	Index	操作
W 0x00000000 0x12341234	0x0000	0x0000	写缺失，因为 Write Allocate 所以 MR，因为 Write Through 所以 MW
R 0x00000000	0x0000	0x0000	读命中
W 0x00000000 0x43214321	0x0000	0x0000	写命中，因为 Write Through 所以 MW
R 0x00000000	0x0000	0x0000	读命中
R 0x00000000	0x0000	0x0000	读命中
R 0x00000004	0x0000	0x0001	读缺失，MR
W 0x10000000 0x11111111	0x1000	0x0000	写缺失，因为 Write Allocate 所以 MR，因为 Write Through 所以 MW
R 0x00000000	0x0000	0x0000	读缺失，MR
W 0x00000004 0x22222222	0x0000	0x0001	写命中，MW
R 0x10000000	0x1000	0x0000	读缺失，MR

【样例 3】

见题目目录下的 ***3.in*** 与 ***3.ans***。

【样例 3 解释】

第三个样例中 $W = 1, R = 1, Phit = 1, Pmiss = 0, N = 10$ ，策略是 Write Back + No-write Allocate，一共有 16384 个组，每组一个块，地址的最高 16 位为 tag，接着 14 位为 index，最后 2 位为 offset。

访问	Tag	Index	操作
W 0x00000000 0x12341234	0x0000	0x0000	写缺失，因为 No-write Allocate 所以 MW
R 0x00000000	0x0000	0x0000	读缺失，MR
W 0x00000000 0x43214321	0x0000	0x0000	写命中，因为 Write Back 所以没有 MW
R 0x00000000	0x0000	0x0000	读命中
R 0x00000000	0x0000	0x0000	读命中

访问	Tag	Index	操作
R 0x00000004	0x0000	0x0001	读缺失，MR
W 0x10000000 0x11111111	0x1000	0x0000	写缺失，因为 No-write Allocate 所以 MW
R 0x00000000	0x0000	0x0000	读命中
W 0x00000004 0x22222222	0x0000	0x0001	写命中
R 0x10000000	0x1000	0x0000	读缺失，MR，因为 Write Back 所以 MW Dirty 块

【样例 4】

见题目目录下的 ***4.in*** 与 ***4.ans***。

【样例 4 解释】

第四个样例中 $W = 1, R = 1, Phit = 1, Pmiss = 1, N = 10$ ，策略是 Write Back + Write Allocate，一共有 16384 个组，每组一个块，地址的最高 16 位为 tag，接着 14 位为 index，最后 2 位为 offset。

访问	Tag	Index	操作
W 0x00000000 0x12341234	0x0000	0x0000	写缺失，因为 Write Allocate 所以 MR，因为 Write Back 所以没有 MW
R 0x00000000	0x0000	0x0000	读命中
W 0x00000000 0x43214321	0x0000	0x0000	写命中，因为 Write Back 所以没有 MW
R 0x00000000	0x0000	0x0000	读命中
R 0x00000000	0x0000	0x0000	读命中
R 0x00000004	0x0000	0x0001	读缺失，MR
W 0x10000000 0x11111111	0x1000	0x0000	写缺失，因为 Write Allocate 所以 MR，因为 Write Back 所以 MW Dirty 块
R 0x00000000	0x0000	0x0000	读缺失，MR，因为 Write Back 所以 MW Dirty 块
W 0x00000004 0x22222222	0x0000	0x0001	写命中，因为 Write Back 所以没有 MW
R 0x10000000	0x1000	0x0000	读缺失，MR

【子任务】

子任务	子任务分数	W	P_{hit}	P_{miss}	R	N
1	10	≤ 8	0	0	≤ 7	$\leq 10^4$
2				1		
3			1	0		
4				1		
5	30		0, 1	0, 1		

LIRS 替换算法实现（lirs）

【题目描述】

本题的要求与上一题相同，但 Cache 使用的替换算法将增加 LIRS 算法。你仍然需要在代码里实现上一题的所有算法。

关于此算法的具体实现，请参考《学习手册》与文献的相应部分。

【输入格式】

从标准输入读入数据。

与上一题相同。

【输出格式】

输出到标准输出。

与上一题相同。

【样例 1】

见题目目录下的 ***1.in*** 与 ***1.ans***。

【样例 1 解释】

样例中 $W = 4, R = 8, N = 20$ ，有四个块，按照 LIRS 算法进行替换，此时 LIRS 算法的参数中 $L = 4, L_{lirs} = 2, L_{hirs} = 2$，在这个样例中，只有读操作，没有写操作，每个地址的 Index 都为 0，只有 Tag 是不同的。下面的 L(n) 表示一个 Tag = n 的 LIRS 块，H(n) 表示一个 Tag = n 的 HIRS 块。

Tag	访问前 S	访问前 Q	访问前各块	操作
0x0		I I	I I I I	填入第 0 块
0x0	0x0	I I	L(0) I I I	命中
0x1	0x0	I I	L(0) I I I	填入第 1 块
0x2	0x0 0x1	I I	L(0) L(1) I I	填入第 2 块
0x3	0x0 0x1 0x2	0x2 I	L(0) L(1) H(2) I	填入第 3 块
0x4	0x0 0x1 0x2 0x3	0x3 0x2	L(0) L(1) H(2) H(3)	替换第 2 块
0x1	0x0 0x1 0x2 0x3 0x4	0x4 0x3	L(0) L(1) H(4) H(3)	命中
0x0	0x0 0x2 0x3 0x4 0x1	0x4 0x3	L(0) L(1) H(4) H(3)	命中
0x0	0x1 0x0	0x4 0x3	L(0) L(1) H(4) H(3)	命中
0x3	0x1 0x0	0x4 0x3	L(0) L(1) H(4) H(3)	命中

Tag	访问前 S	访问前 Q	访问前各块	操作
0x4	0x1 0x0 0x3	0x3 0x4	L(0) L(1) H(4) H(3)	命中
0x4	0x1 0x0 0x3 0x4	0x4 0x3	L(0) L(1) H(4) H(3)	命中
0x2	0x0 0x3 0x4	0x1 0x3	L(0) H(1) L(4) H(3)	替换第 3 块
0x5	0x0 0x3 0x4 0x2	0x2 0x1	L(0) H(1) L(4) H(2)	替换第 1 块
0x3	0x0 0x3 0x4 0x2 0x5	0x5 0x2	L(0) H(5) L(4) H(2)	替换第 3 块
0x1	0x4 0x2 0x5 0x3	0x0 0x5	H(0) H(5) L(4) L(3)	替换第 1 块
0x1	0x4 0x2 0x5 0x3 0x1	0x1 0x0	H(0) H(1) L(4) L(3)	命中
0x2	0x3 0x1	0x4 0x0	H(0) L(1) H(4) L(3)	替换第 0 块
0x0	0x3 0x1 0x2	0x2 0x4	H(2) L(1) H(4) L(3)	替换第 2 块
0x0	0x3 0x1 0x2 0x0	0x0 0x2	H(2) L(1) H(0) L(3)	命中

【子任务】

<table>
<tr><th>子任务</th><th>子任务分数</th><th>W</th><th>P_{hit}</th><th>P_{miss}</th><th>R</th><th>N</th></tr>
<tr><td>1</td><td rowspan="4">12</td><td rowspan="4">≤ 8</td><td rowspan="2">0</td><td>0</td><td rowspan="5">≤ 8</td><td rowspan="5">$\leq 10^4$</td></tr>
<tr><td>2</td><td>1</td></tr>
<tr><td>3</td><td rowspan="2">1</td><td>0</td></tr>
<tr><td>4</td><td>1</td></tr>
<tr><td>5</td><td>30</td><td>≤ 16</td><td>0, 1</td><td>0, 1</td></tr>
</table>

期待你的声音（feedback）

这是一道提交答案题。

【题目描述】

再一次地，我们在冬令营中引入了这样一道“工程题”。与前几次不同的是，这一题涉及的知识更为底层，也与程序设计竞赛密切相关。掌握了 Cache 的相关知识，你就能以更底层的视角审视程序的性能表现，做出更有效的优化，而非在“卡常数”的玄学路上越走越偏。

与之前相同，我们也希望你能够留下你对于这道题的想法（使用纯文本格式书写并作为答案文件提交即可）。无论是建议还是批评、赞同或者吐槽，我们都会认真对待，并用来改进我们今后的工作。当然，不管你提交与否，也不管你提交什么内容，都**不会以任何形式**影响你的最终成绩。

我们也提供了一些可选的问题供你回答：

- 招生考试是否应该引入此类工程题目？
- 目前工程题与传统算法题的分数占比是否合适？
- 如果还将参与，你想看到来自哪些领域的工程题目？

无论这题做得是否顺利，无论你之后去向何方，我们都诚挚地希望你能够保持对计算机科学的热情。祝你成功，我们在清华等你来！