

**实 验 报 告**

**课程名称：** **操作系统**

**姓 名：** **方桂安**

**学 号：** **20354027**

**专业班级：** **2020级智能科学与技术**

**任课教师：** **吴贺俊**

2022 **年** 9 **月** 16 **日**

**实验报告成绩评定表**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 评定项目 | 内 容 | 满 分 | 评 分 | 总 分 |
| 实验态度 | 态度端正、遵守纪律、出勤情况 | 10 |  |  |
| 实验过程 | 按要求完成算法设计、代码书写、注释清晰、运行结果正确 | 30 |  |
| 实验记录 | 展示讲解清楚、任务解决良好、实验结果准确 | 20 |  |
| 报告撰写 | 报告书写规范、内容条理清楚、表达准确规范、上交及时、无抄袭，抄袭记0分，提供报告供抄袭者扣分。 | 40 |  |
| **评语：**  **指导老师签字： 年 月 日** | | | | |

实验报告正文

**实验三 仿真存储系统**

1. **实验目的**
2. 掌握三级存储系统的思想；

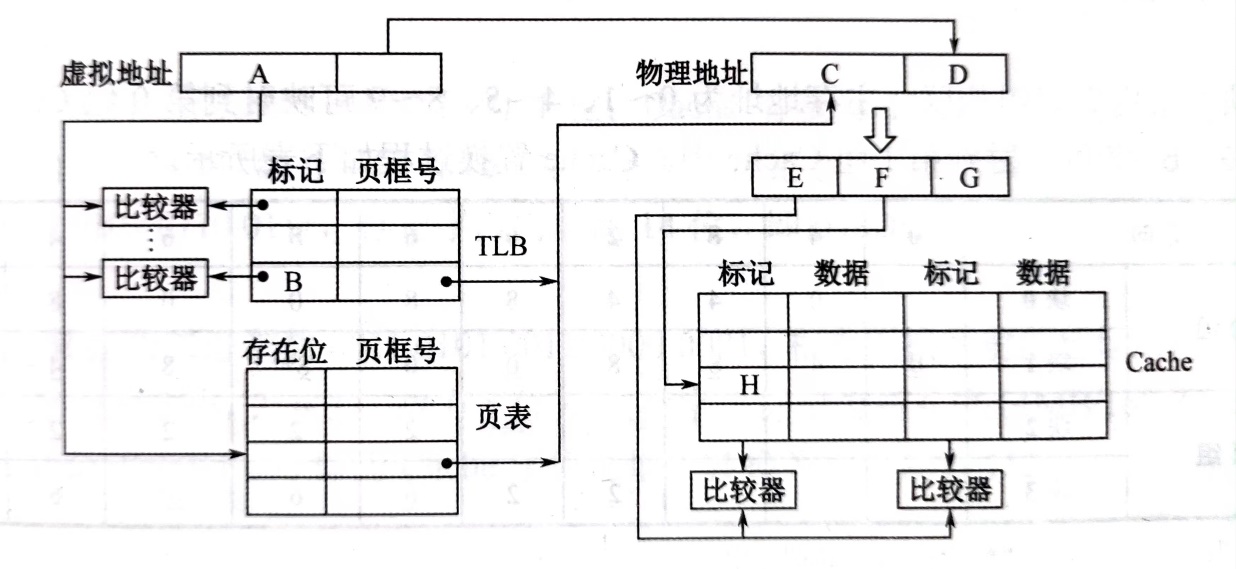
2. 掌握存储系统的工作流程。

1. **实验内容**
   1. **任务描述**

1）存储系统仿真简介

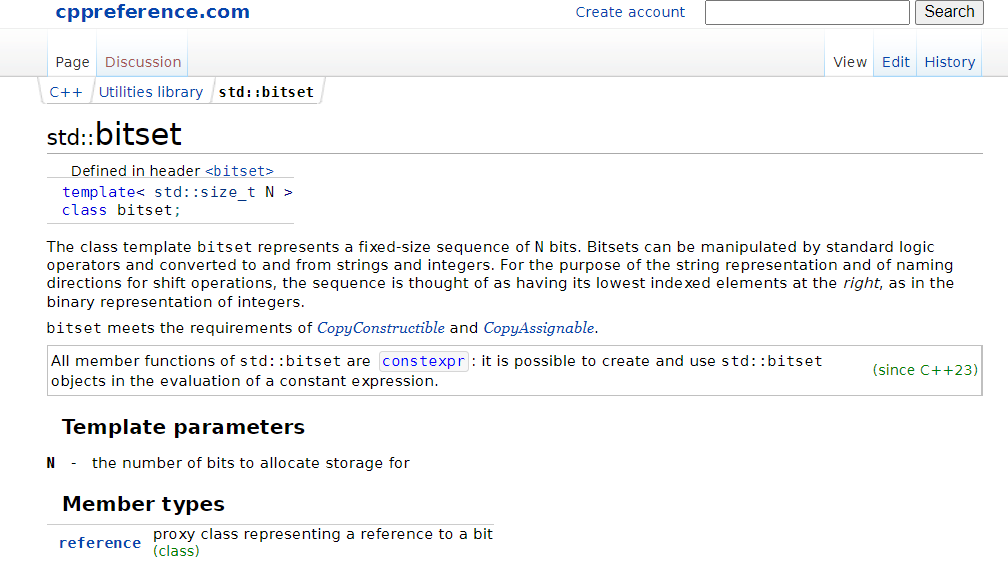
仿真采用页式虚拟存储管理方式的三级存储系统。该存储系统按字节编址，虛拟地址为 16位，物理地址为12 位，页大小为128B；TLB 采用全相联映射；Cache数据区大小为 1KB，按2路组相联方式组织，主存大小为4KB，主存和CACHE块大小为 32B。

2）存储访问过程示意图



* 1. **实验方案**

本设计采用了std::bitset作为主要的数据结构。



以cache\_line和memory\_data为例：

std::bitset<8> memory\_item[1024][MAX\_CACHE\_LINE];

//main memory item

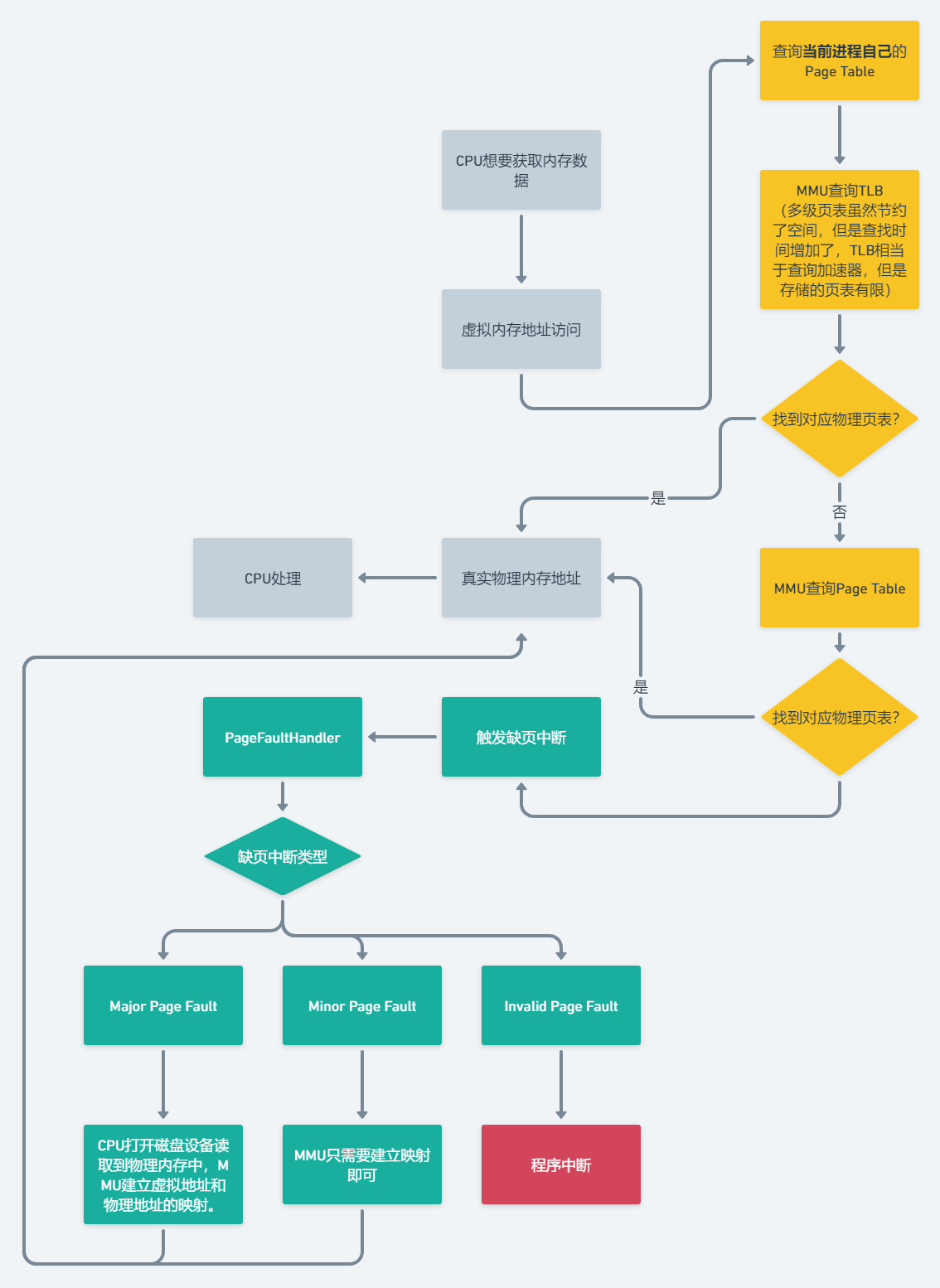
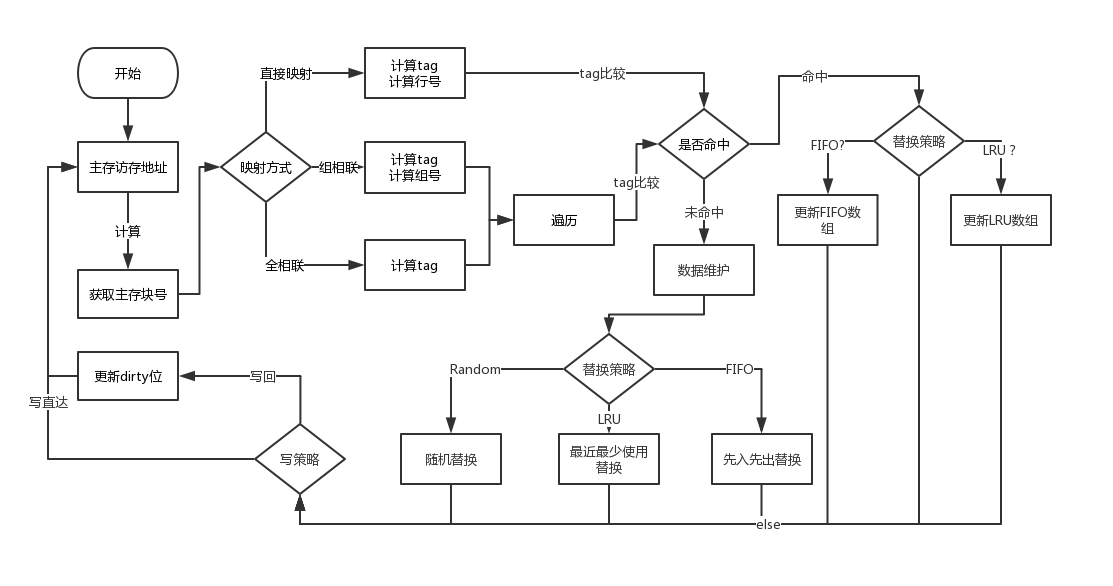
std::bitset<32> cache\_item[MAX\_CACHE\_LINE];

// [31]:valid,[30]:hit,[29]:dirty,[28]-[0]:data

memory\_item用来模拟主存中的数据，本次实验中主存块的大小为32B，每个字节有八位，故采用二维的长度为8的bitset来模拟，以本次实验为例，主存共有4KB，故主存块数为4KB/32B=128，主存数据则可以用bitset<8> memory\_item[128][32]来仿真。

cache\_item用来模拟cache块，其中第31位valid记录是否有效，30位hit记录是否命中，29位dirty标记是否需要写回主存；通过利用这几个标记，来实现FIFO(First In First Out)、LRU(Least Recently Used)、LFU(Least Frequently Used)等缓存算法。

* 1. **实验说明**

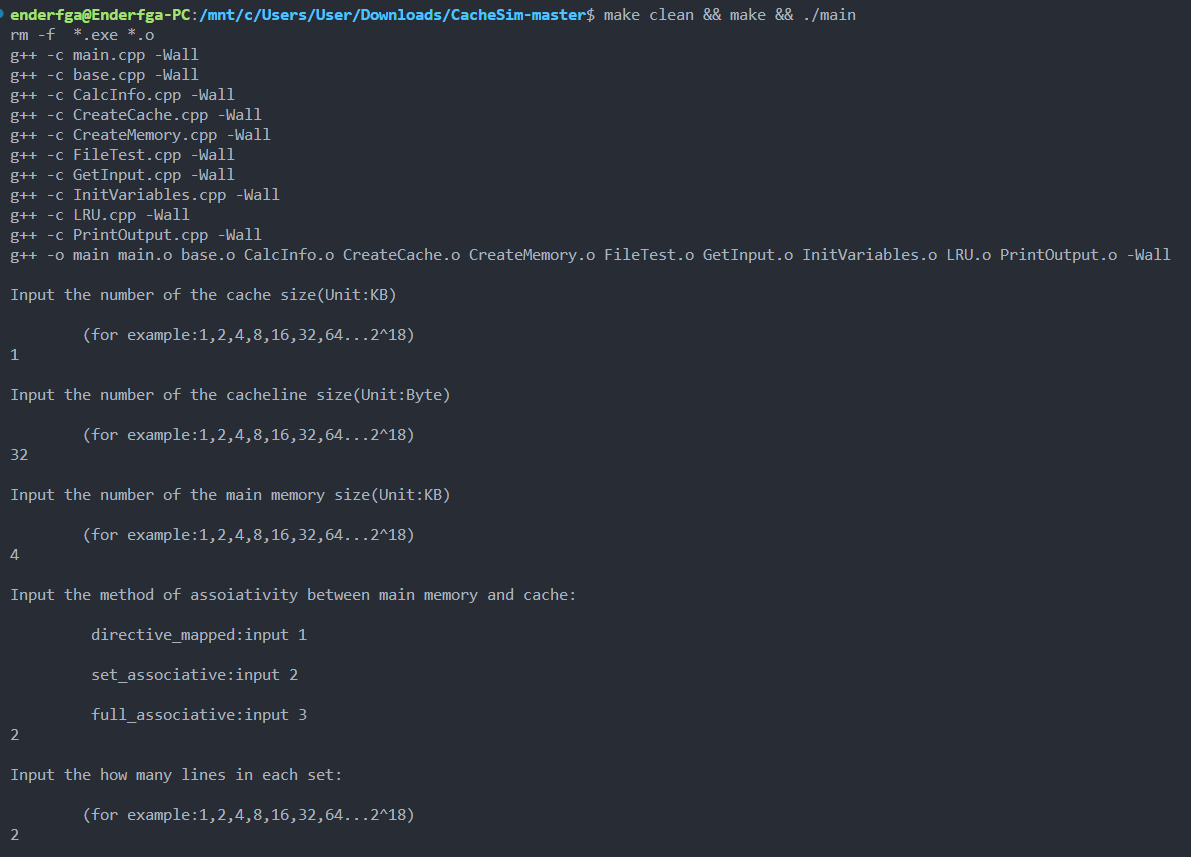


1. **实验记录**
   1. **实施步骤**

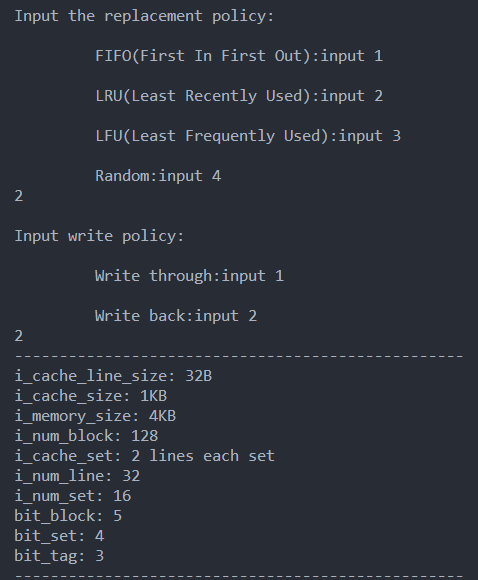
* 1）从1-50块，把奇数块号的模拟主存块每个字节都设为0x55，把偶数块号的模拟主存块每个字节都设为0xAA。打印块号为28和29中第一字节的内容。
* 2）将块号为29的模拟主存块装入模拟Cache，打印按函数映射的Cache组号，打印其对应的H字段内容。
* 3）模拟某个缺页处理的过程，并给出缺页处理的日志记录。
* 4）模拟修改某个页面内容，模拟回写过程，并给出回写过程的记录。
  1. **实验记录**

1. **准备工作**

首先根据实验步骤中的内容编写程序，运行效果如下：



按照题目要求设置cache大小为1KB,主存大小为4KB，主存和CACHE块大小为 32B，按照2路组相联方式组织。



然后选择replacement policy和write policy，题目没有要求具体采用哪一种，本次实验展示的是LRU以及Write bak。

通过以上输入可以得到共有128个主存块，16组2路共32个cache\_line。物理地址共有12位，通过主存/cache块大小32B可知，块内地址（offset）为5位，通过组数16可知，组号（index）为4位，剩下的3位为标记（tag）。

1. **设置模拟主存**

根据题目要求，从1-50块,把奇数块号的模拟主存块每个字节都设为0x55，把偶数块号的模拟主存块每个字节都设为0xAA，代码如下：

string x55 = "01010101";

string xAA = "10101010";

std::bitset<8> tmp1(x55);

std::bitset<8> tmp2(xAA);

for(i=0; i<50; i++)

{

if(i%2==0)

{

for(j=0; j<i\_cache\_line\_size; j++)

{

memory\_item[i][j] = tmp1;

}

}

else

{

for(j=0; j<i\_cache\_line\_size; j++)

{

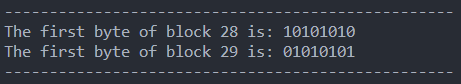
memory\_item[i][j] = tmp2;

}

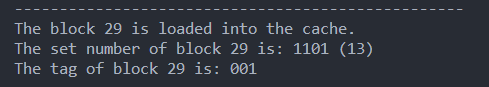
}

}

将字符串转换为二进制序列即可满足我们的需求，打印输出块号为28,29中第一字节的内容：



结合前面的分析可知，块号+块内地址=主存物理地址=标记+组号+块内地址。所以块号“29”的二进制表示为"11101"，补0至与标记和组号长度之和（7位），故H字段标记为前3位——“001”。利用程序分析，可以得到相同结果：



根据计算得知，组相联映射的组号为13，所以存入set\_13中的2个cache\_line中的任意一个，仿真代码如下：

cout << "The block 29 is loaded into the cache." << endl;

//将29转换为二进制，块号+块内地址=主存物理地址=标记+组号+块内地址

string x29 = "11101";

//std::size\_t len =bit\_tag+bit\_set; //=7

std::bitset<7> tmp3(x29);

//打印tmp3的后bit\_set位，bit\_set=4以及10进制形式

//转为10进制

int n = 0;

for (int i = 0; i < bit\_set; i++)

{

n += tmp3[i] \* pow(2, i);

}

cout << "The set number of block 29 is: " << tmp3.to\_string().substr(bit\_tag,bit\_set) << " (" << n << ")" << endl;

//将第29块memory\_item随机放到第n块cache\_data的前8位或者后8位

int x = rand()%2;

for(i=0; i<i\_cache\_line\_size; i++)

{

//cache\_data[n-1][i]有16位，memory\_item[28][i]有8位，随机放到前8位或者后8位

if(x==0)

{

for(j=0; j<8; j++)

{

cache\_data[n-1][i][j] = memory\_item[28][i][j];

}

}

else

{

for(j=8; j<16; j++)

{

cache\_data[n-1][i][j] = memory\_item[28][i][j-8];

}

}

}

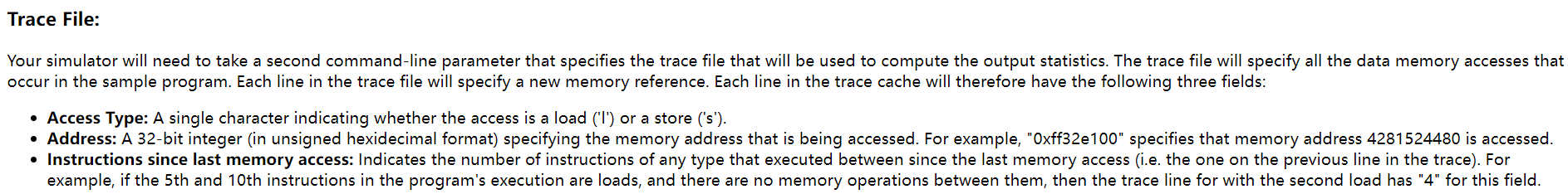
//打印tmp3的前bit\_tag位，bit\_tag=3

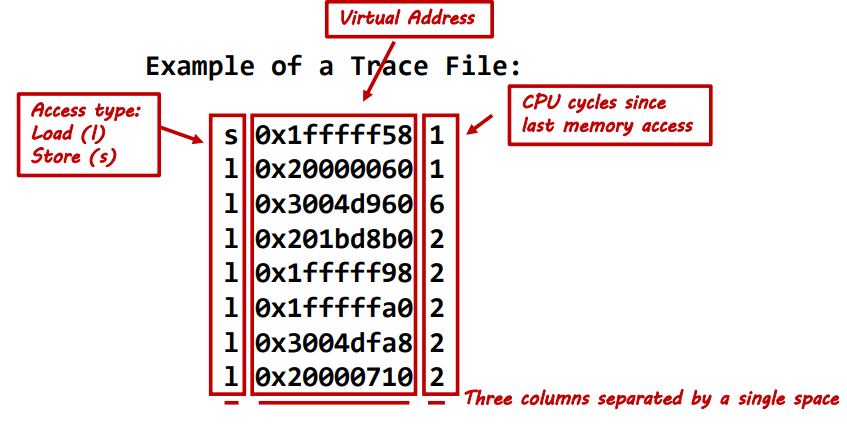
cout<<"The tag of block 29 is: "<<tmp3.to\_string().substr(0,bit\_tag)<<endl;

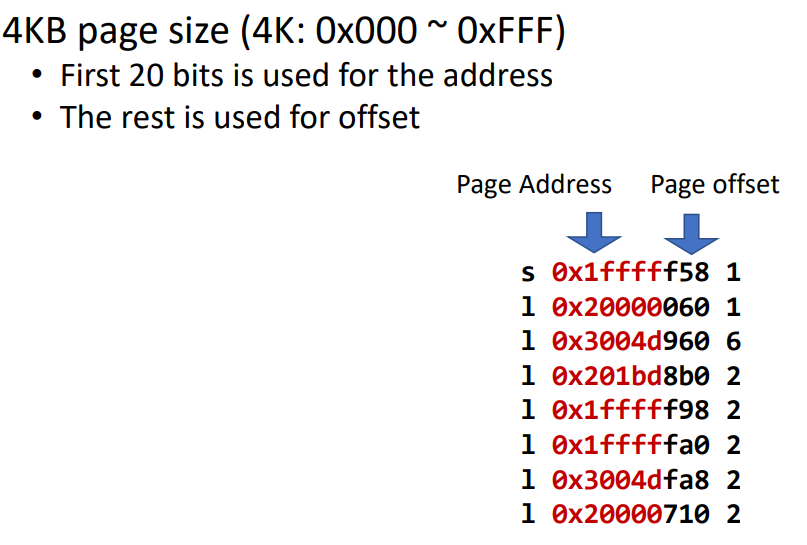
}

* 1. **实验结果**

本次实验中的测试数据参考自UCSD的cse240a Project1，以下是介绍原文：

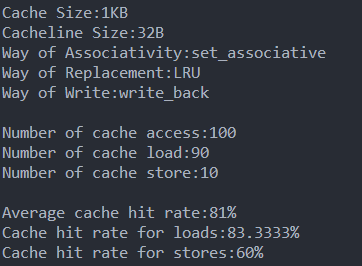






每行的第一个字符是表示该条指令的类型，s为写（store），r为读（read）。中间的十六进制数为内存地址，最后一个整数是指存储器访问指令之间的间隔指令数。例如第5条指令和第10条指令为存储器访问指令，且中间没有其他存储器访问指令，则间隔指令数为4。 本次实验的模拟中暂不考虑最后这个参数的使用。

借此模拟缺页处理以及回写过程的结果如下：



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Way of Replacement | FIFO | LRU | LFU | Random |
| Number of cache access | 727230 | 727230 | 727230 | 727230 |
| Number of cache load | 5972 | 5972 | 5972 | 5972 |
| Number of cache store | 721258 | 721258 | 721258 | 721258 |
| Average cache hit rate | 55.994% | 99.9945% | 55.994% | 99.9941% |
| Cache hit rate for loads | 69.2565% | 99.5981% | 69.2565% | 99.5814% |
| Cache hit rate for stores | 55.8842% | 99.9978% | 55.8842% | 99.9975% |

每当程序所要访问的页面未在内存时， 便向CPU发出缺页中断，中断处理程序首先保留CPU环境，分析中断原因后，转入缺页中断处理程序。该程序通过查找页表，得到该页在外在原物理块后，如果此时内存能容纳新页，则启动磁盘I/O将所缺之页调入内存，然后修改页表。如果内存已满，则须先按照某种**置换算法**从内存中选出一页准备换出；如果该页未被修改过，可不必将该页写回磁盘；但如果此页已被修改，则必须将它**写回**磁盘，然后再把所缺的页调入内存，并修改页表中的相应表项，置其存在位“1”，并将此页表项写入快表中。在缺页调入内存后，利用修改后的页表，去形成所要访问数据的物理地址，再去访问内存数据。整个页面的调入过程对用户是透明的。

代码中分析trace文件时会产生大量的日志记录，为了使输出美观，大部分输出都使用NDEBUG宏来避免拖慢程序运行效率。

#ifndef NDEBUG

{

}

#endif // NDEBUG

缺页处理代码示例：

else if((!hit) && is\_load) // 没命中，读操作

{

i\_num\_access++;

i\_num\_load++;

#ifndef NDEBUG

cout << "Loading" << endl;

cout << "Not Hit" << endl;

#endif // NDEBUG

GetRead(flags); // read data from memory

#ifndef NDEBUG

cout << "Read from Cache!" << endl;

#endif // NDEBUG

if(t\_replace == LRU)

{

LruUnhitSpace();

}

}

else if((!hit) && is\_store) // 没命中，写操作

{

i\_num\_access++;

i\_num\_store++;

#ifndef NDEBUG

cout << "Storing" << endl;

cout << "Not Hit" << endl;

#endif // NDEBUG

GetRead(flags); // read data from memory

#ifndef NDEBUG

cout << "Write to Cache" << endl;

#endif // NDEBUG

cache\_item[current\_line][29] = true; //设置dirty为true

if(t\_replace == LRU)

{

LruUnhitSpace();

}

}

回写主存代码示例：

void GetWrite() //写入内存

{

#ifndef NDEBUG

cout << "Writing to the Main Memory!" <<endl;

#endif

cache\_item[current\_line][29] = false; //设置dirty为false

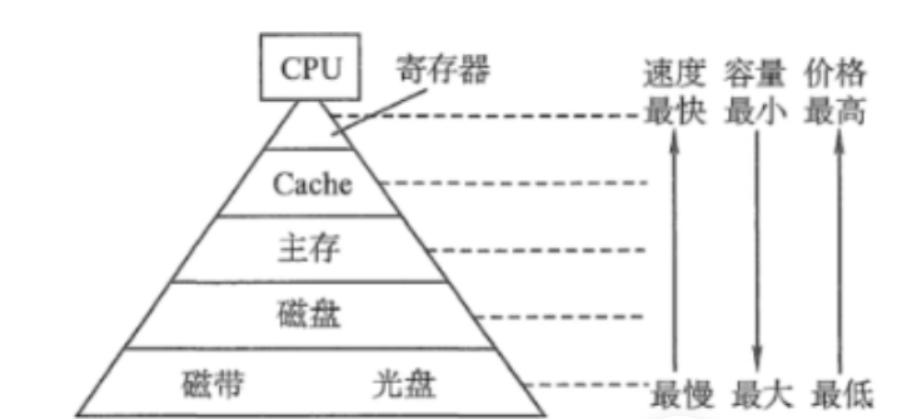
cache\_item[current\_line][30] = false; //设置hit为false

}

1. **总结与讨论**

1 、存储系统为什么要设置为三级结构的？

为了解决存储系统大容量、高速度和低成本3个相互制约的矛盾，在计算机系统中，通常采用多级存储器结构，在图中由上至下，位价越来越低，速度越来越慢，容量越来越大，CPU访问的频度也越来越低。



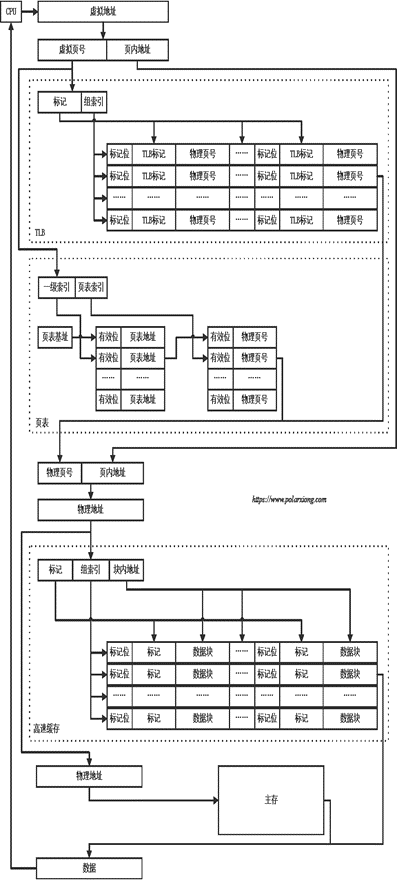
实际上，存储系统层次结构主要体现在"Cache-主存”层次和“主存－辅存”层次。前者主要解决CPU和主存速度不匹配的问题，后者主要解决存储系统的容量问题。在存储体系中，Cache、主存能与CPU直接交换信息，辅存则要通过主存与CPU交换信息；主存与CPU、Cache、辅存都能交换信息。存储器层次结构的主要思想是上一层的存储器作为低一层存储器的高速缓存。从CPU的角度看，"Cache—主存”层次速度接近于Cache,容量和位价却接近千主存。从“主存—辅存”层次分析，其速度按近于主存，容址和位价动接近于辅存。这就解决了速度、容量、成本这三者之间的矛盾。

2、高速缓存命中和未命中如何仿真？

本次实验中我采取的方法是在cache\_line中设置hit位来记录是否命中的信息，bitset存储的是二进制序列，而hit位的true or false可以方便地读取并分析。

3、虚拟存储的地址怎么转化为实际地址呢？

流程图如下：



**五、附：程序模块的源代码**

源代码已打包，附有Makefile，运行时只需要在终端中输入

make && ./main