******

**内存管理之项目**

姓 名：曹晓慈

学 号：2052844

所在院系：软件学院

学科专业：软件工程

指导教师：王冬青

二〇二二年六月

**调页存储管理实验报告**

1. **项目介绍**

假设初始态下，可用内存空间为640K，并有下列请求序列，请分别用首次适应算法和最佳适应算法进程内存块的分配和回收，并显示出每次分配和回收后的空闲分区链的情况来。

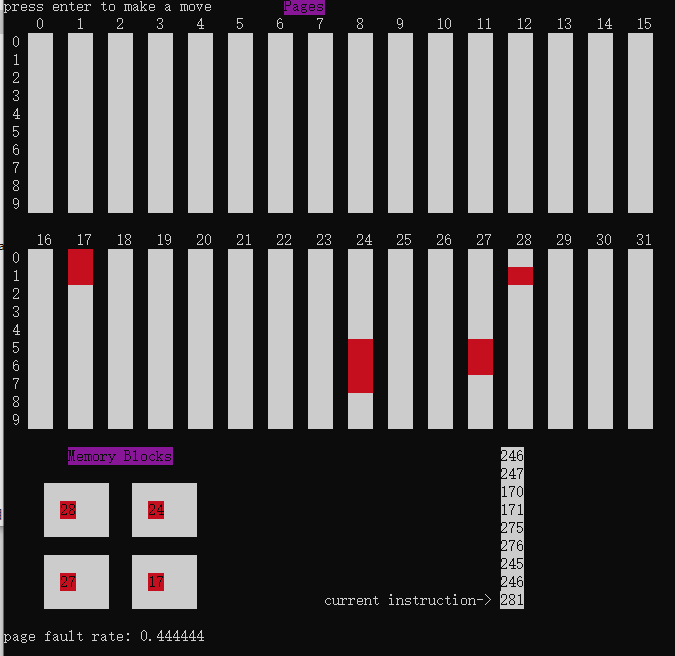
假设每个页面可存放10条指令，分配给一个作业的内存块为4。模拟一个作业的执行过程，该作业有320条指令，即它的地址空间为32页，目前所有页还没有调入内存。

在模拟过程中，如果所访问指令在内存中，将其执行，并转到下一条指令；如果没有在内存中，则发生缺页，此时需要记录缺页次数，并将其调入内存。如果4个内存块中已装入作业，则需进行页面置换。所有320条指令执行完成后，计算并显示作业执行过程中发生的缺页率。置换算法选用的是LRU算法。

1. **开发环境**

语言为c++，IDE为visual studio，Windows10

1. **GUI基本构成和实现思路**

****

上方的是32个页，每个页面有10条指令，开始时所有指令都用白色块表示，表明这些指令没有被访问，当指令被访问时，对应的地址块会变成红色；右下的是访问指令的地址，current instruction指向当前正在访问指令的地址，由于空间大小不足，因此只在窗口中显示了10条指令，当访问超过10条指令后，会从头开始显示当前的指令，覆盖之前的指令；左下的表示内存块，显示当前调入内存的页面，其中4个内存块按顺序排列，越往前的内存块表示其中的页面更近期的被使用，使用LRU算法时也遵循这个原则，page fault rate表示当前的缺页率。总的来说，current instruction指向新产生的指令，查看内存块中是否有该指令对应的页面，更新内存块，同时页面中对应指令的地址变为红色，表示被访问过。如此循环往复，直到320条指令全部被执行完毕。

1. **代码说明**

int page\_fault = 0, option = 0, index = 0;

//全局变量，page\_fault表示缺页次数，option中1表示单步执行0表示自动执行，index表示当前已执行的指令数目

bool one\_step(int cur, int block[MEMORY\_BLOCKS])

{

if (option)

press\_enter();

show\_inst(cur);//显示当前指向的指令地址

waver(cur);//当前指令闪烁

LRU(cur, block);//LRU算法

show\_block(block);//显示调入内存块的页面

cct\_gotoxy(0, 35);//光标调到（0,35）的位置

cct\_setcolor();//设置颜色为默认

cout << "page fault rate: " << (page\_fault + 0.0) / index << endl;

//显示缺页率

if (index == PAGES \* INST\_OF\_PAGE)//320条指令执行完毕

return true;

return false;

}

One\_step函数是每执行一条指令需要进行的操作，首先显示指令当前位置，然后使用LRU算法判断是否需要调入当前页面，更新内存块中的页面显示，然后在其下显示缺页率

void LRU(int cur, int block[MEMORY\_BLOCKS])

{

int i = 0;

for (; i < MEMORY\_BLOCKS; i++)

{

if (block[i] == cur/INST\_OF\_PAGE)

{

//将block[i]删除

for (int j = i; j < MEMORY\_BLOCKS - 1; j++)

block[j] = block[j + 1];

break;

}

}

if (i == 4)//当前内存块中的页面中没有当前指向的指令

page\_fault++;

//整体后移

for (int i = MEMORY\_BLOCKS - 1; i > 0; i--)

block[i] = block[i - 1];

block[0] = cur / INST\_OF\_PAGE;//更新第一个内存块调入当前页面

}

越往前的内存块表示其中的页面更近期的被使用，因此如果需要调入页面则将当前页面调入第一个内存块，其余内存块中的页面往后顺移，多出来的页面则被置换；如果不需要调入页面，则将当前指令对应的页面调入到第一个内存块，其余内存块中的页面往后顺移。

int main()

{

cout << "Please enter the display speed: (0 for auto ,1 for step by step) ";

while (1)//错误检查

{

cin >> option;

if (!cin.good() || option > 1 || option < 0)

{

cin.clear();

cin.ignore(1024, '\n');

}

else

break;

}

cct\_setcursor(CURSOR\_INVISIBLE);//设置光标不可见

display();//显示伪图形化界面

if (option)

cct\_showstr(0, 0, "press enter to make a move", COLOR\_BLACK, COLOR\_WHITE, 0);

cct\_setcolor();

int block[MEMORY\_BLOCKS] = { -1,-1,-1,-1 };

srand((unsigned int)time(NULL));

int cur = rand() % (PAGES \* INST\_OF\_PAGE);

one\_step(cur, block);

cur++;

one\_step(cur, block);

/\* 按照50%的指令是顺序执行的，25%是均匀分布在前地址部分，25％是均匀分布在后地址部分的原则产生指令 \*/

while (1)

{

cur = rand() % (cur - 1);

if (one\_step(cur, block))

break;

cur++;

if (one\_step(cur, block))

break;

cur = rand() % (319 - cur) + cur + 1;

if (one\_step(cur, block))

break;

cur++;

if (one\_step(cur, block))

break;

}

return 0;

}

main函数的实现也比较清楚，首先选择自动执行还是单步执行，输入option，在显示过程中Sleep(10\*option)就能达到这一效果。如果输入的是0，则不休眠，表现为自动执行，如果输入的是1，则每执行一步都有按下回车键，并通过Sleep使展示放慢速度。之后在main函数中进行错误检查，显示初始化界面，进行一些变量的初始化，随机产生第一条指令等等。然后是按照50%的指令是顺序执行的，25%是均匀分布在前地址部分，25％是均匀分布在后地址部分的原则产生指令，具体实现为：

在0－319条指令之间，随机选取一个起始执行指令，如序号为m

顺序执行下一条指令，即序号为m+1的指令

通过随机数，跳转到前地址部分0－m-1中的某个指令处，其序号为m1

顺序执行下一条指令，即序号为m1+1的指令

通过随机数，跳转到后地址部分m1+2~319中的某条指令处，其序号为m2

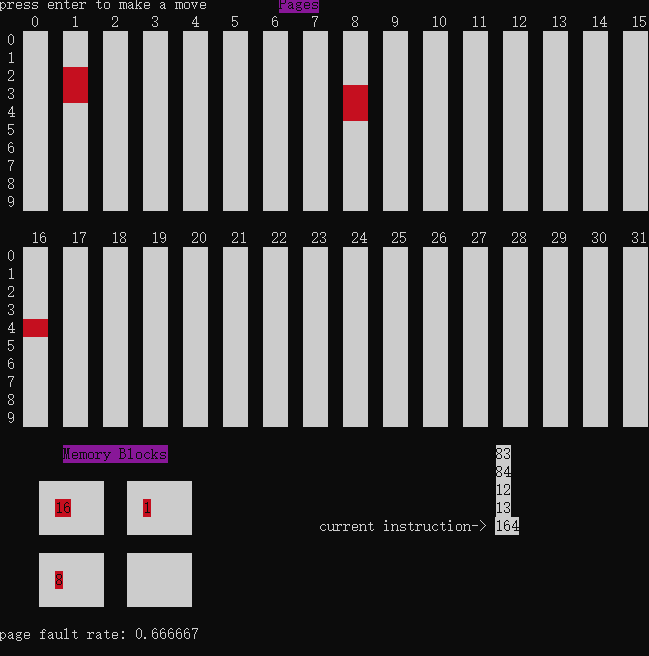
顺序执行下一条指令，即m2+1处的指令。

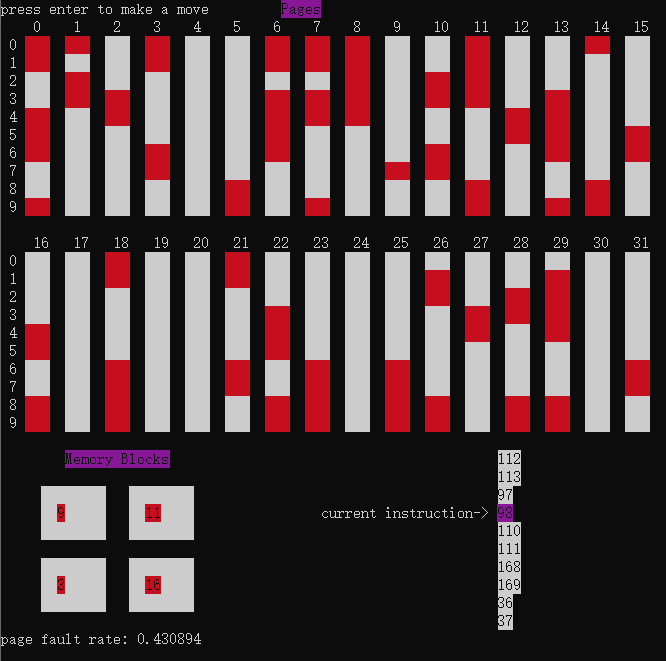
重复跳转到前地址部分、顺序执行、跳转到后地址部分、顺序执行的过程，直到执行完320条指令。

1. **总体架构**

项目的总体架构大致与main函数中所显示的相同，首先选择执行方式，然后对各种变量进行初始化，接着按照指令产生的顺序一步一步执行，指令产生的原则之前已经说明，这里不再赘述。需要强调的是，本次项目的重点是LRU算法，即最近最少使用的页面被置换掉，每执行一步，都给内存块中的页面进行重新调整，排在最后的页面就会满足最近最少使用的条件，需要置换页面时则将其置换掉，这样就能实现LRU算法。

**六、项目截图**

****

****

**七、心得体会**

这次请求调页存储管理方式实验模拟了操作系统中的指令执行过程，主要学习的重点还是LRU算法，页面置换算法的思想可以说就是将数据块中，每次使用过的数据放在数据块的最前端，然后将存在的时间最长的，也就是数据块的末端的数据剔除掉，但是在实现的时候我却遇到不少的问题，比如，一开始想实现更简单一点的FIFO算法，但是考虑到这只需要实现一个队列就好了，感觉并没有什么挑战性，但是要实现LRU算法要怎么做呢，之前想给每条指令标记访问次数，但是太麻烦了，就想到给内存块中的指令进行排序，排在最后的页面就被置换，这样既能达到目标，实现起来也不会那么麻烦。通过研究和代码实现，这次的请求调页存储管理方式实验还是让我学到了不少东西的，对LRU算法也有了更深刻的理解。