**实验四 存储管理**

# 姓 名 郭宇哲 学 号 22013229 成绩

实验时间 2024/11/12 指导教师(签名)

**（诚信声明：本实验报告内容，均由本人亲自上机完成。 签名：\_\_\_\_\_\_\_\_\_\_\_\_\_）**

一．实验目的

1.通过请求页式存储管理中页面置换算法模拟设计，了解虚拟存储技术的特点

2.掌握请求页式存储管理的页面置换算法。

二．实验工具与设备

装有 Linux 操作系统的计算机。

三．实验内容

计算并输出下面各种算法在不同内存容量下的命中率：

1. FIFO（先进先出算法）
2. LRU （最近最少使用算法）
3. OPT （最优算法）
4. LFU （最少使用页面算法）
5. CLOCK （时钟算法）

**实验思路及核心代码：**

void FIFO(int total\_pf) {

int i;

struct pfc\_struct \*p;

initialize(total\_pf); /\* 初始化相关页面控制用数据结构 \*/

busypf\_head = busypf\_tail = NULL; /\* 忙页面队列，队列尾链接 \*/

for (i = 0; i < total\_instruction; i++) {

if (pl[page[i]].pfn == INVALID) { /\* 页面失效 \*/

diseffect += 1; /\* 失效次数 \*/

if (freepf\_head == NULL) { /\* 无空闲页面 \*/

p = busypf\_head->next;

pl[busypf\_head->pn].pfn = INVALID;

freepf\_head = busypf\_head; /\* 释放忙页面队列中的第一个页面 \*/

freepf\_head->next = NULL;

busypf\_head = p;

}

p = freepf\_head->next; /\* 按FIFO方式调新页面入内存页面 \*/

freepf\_head->next = NULL;

freepf\_head->pn = page[i];

pl[page[i]].pfn = freepf\_head->pfn;

if (busypf\_tail == NULL) {

busypf\_head = busypf\_tail = freepf\_head;

} else {

busypf\_tail->next = freepf\_head;

busypf\_tail = freepf\_head;

}

freepf\_head = p;

}

}

printf("FIFO: %6.4f ", 1 - (float)diseffect / 320);

}

void LRU(int total\_pf) {

int iMin, minIndex, i, j, present\_time;

initialize(total\_pf);

present\_time = 0;

for (i = 0; i < total\_instruction; i++) {

if (pl[page[i]].pfn == INVALID) { /\* 页面失效 \*/

diseffect++;

if (freepf\_head == NULL) { /\* 如果没有空闲页面 \*/

iMin = 32765;

for (j = 0; j < total\_vp; j++) {

/\* 每次都找到最近最久未使用的那个页面， 并且暂存起来 \*/

if (iMin > pl[j].time && pl[j].pfn != INVALID) {

iMin = pl[j].time;

minIndex = j;

}

}

freepf\_head = &pfc[pl[minIndex].pfn];

pl[minIndex].pfn = INVALID;

pl[minIndex].time = -1;

freepf\_head->next = NULL;

}

pl[page[i]].pfn = freepf\_head->pfn; /\* 有空闲页面则加入内存块 \*/

pl[page[i]].time = present\_time;

freepf\_head = freepf\_head->next; /\* 同时减少一个空闲页面 \*/

} else {

pl[page[i]].time = present\_time;

present\_time++;

}

}

printf("LRU: %6.4f ", 1 - (float)diseffect / 320);

}

void OPT(int total\_pf) {

int i, j, max, maxpage, dist[total\_vp];

initialize(total\_pf);

for (i = 0; i < total\_instruction; i++) {

if (pl[page[i]].pfn == INVALID) { /\* 页面失效 \*/

diseffect++;

if (freepf\_head == NULL) {

for (j = 0; j < total\_vp; j++) {

if (pl[j].pfn != INVALID) {

dist[j] = 32767;

} else {

dist[j] = 0;

}

}

max = -1;

for (j = 0; j < total\_vp; j++) {

if (max < dist[j]) {

max = dist[j];

maxpage = j;

}

}

freepf\_head = &pfc[pl[maxpage].pfn];

freepf\_head->next = NULL;

pl[maxpage].pfn = INVALID;

}

pl[page[i]].pfn = freepf\_head->pfn;

freepf\_head = freepf\_head->next;

}

}

printf("OPT: %6.4f ", 1 - (float)diseffect / 320);

}

void LFU(int total\_pf) {

int i, j, min, minpage;

initialize(total\_pf);

for (i = 0; i < total\_instruction; i++) {

if (pl[page[i]].pfn == INVALID) { /\* 页面失效 \*/

diseffect++;

if (freepf\_head == NULL) { /\* 如果没有空闲页面 \*/

min = 32767;

for (j = 0; j < total\_vp; j++) {

/\* 每次都找到最近最少使用的那个页面， 并且暂存起来 \*/

if (min > pl[j].counter && pl[j].pfn != INVALID) {

min = pl[j].counter;

minpage = j;

}

pl[j].counter = 0;

}

freepf\_head = &pfc[pl[minpage].pfn];

pl[minpage].pfn = INVALID;

}

pl[page[i]].pfn = freepf\_head->pfn; /\* 有空闲页面则加入内存块 \*/

freepf\_head = freepf\_head->next; /\* 同时减少一个空闲页面 \*/

}

}

printf("LFU: %6.4f ", 1-( float) diseffect/320);

}

void CLOCK( int total\_pf)

{

int i, j, dp, cont\_flag, old\_dp;

pl\_type \*t;

initialize( total\_pf);

dp = 0;

for(i = 0; i < total\_instruction; i++)

{

if( pl[ page[i]]. pfn == INVALID) //如果页面失效

{

diseffect++;

if( freepf\_head == NULL) //如果没有空闲页面

{

cont\_flag = TRUE;

old\_dp = dp;

while( cont\_flag)

{

if( pl[ dp]. counter == 0 && pl[ dp]. pfn != INVALID)cont\_flag = FALSE; // 访问位为0时淘汰

else{

dp++;

if( dp == total\_vp)dp = 0;

if( dp == old\_dp) // 访问位为1时将访问位置为0

for(j=0; j< total\_vp; j++)

pl[j]. counter = 0;

}

}

//依照CLOCK算法置换掉访问位为0被淘汰的页面

freepf\_head = & pfc[ pl[ dp]. pfn];

pl[ dp]. pfn = INVALID;

freepf\_head-> next = NULL;

}

pl[ page[i]]. pfn = freepf\_head-> pfn;

freepf\_head = freepf\_head-> next;

}

else

pl[ page[i]]. counter = 1; //命中， 访问次数+1

if(i % clear\_period == 0)

for(j=0; j< total\_vp; j++)

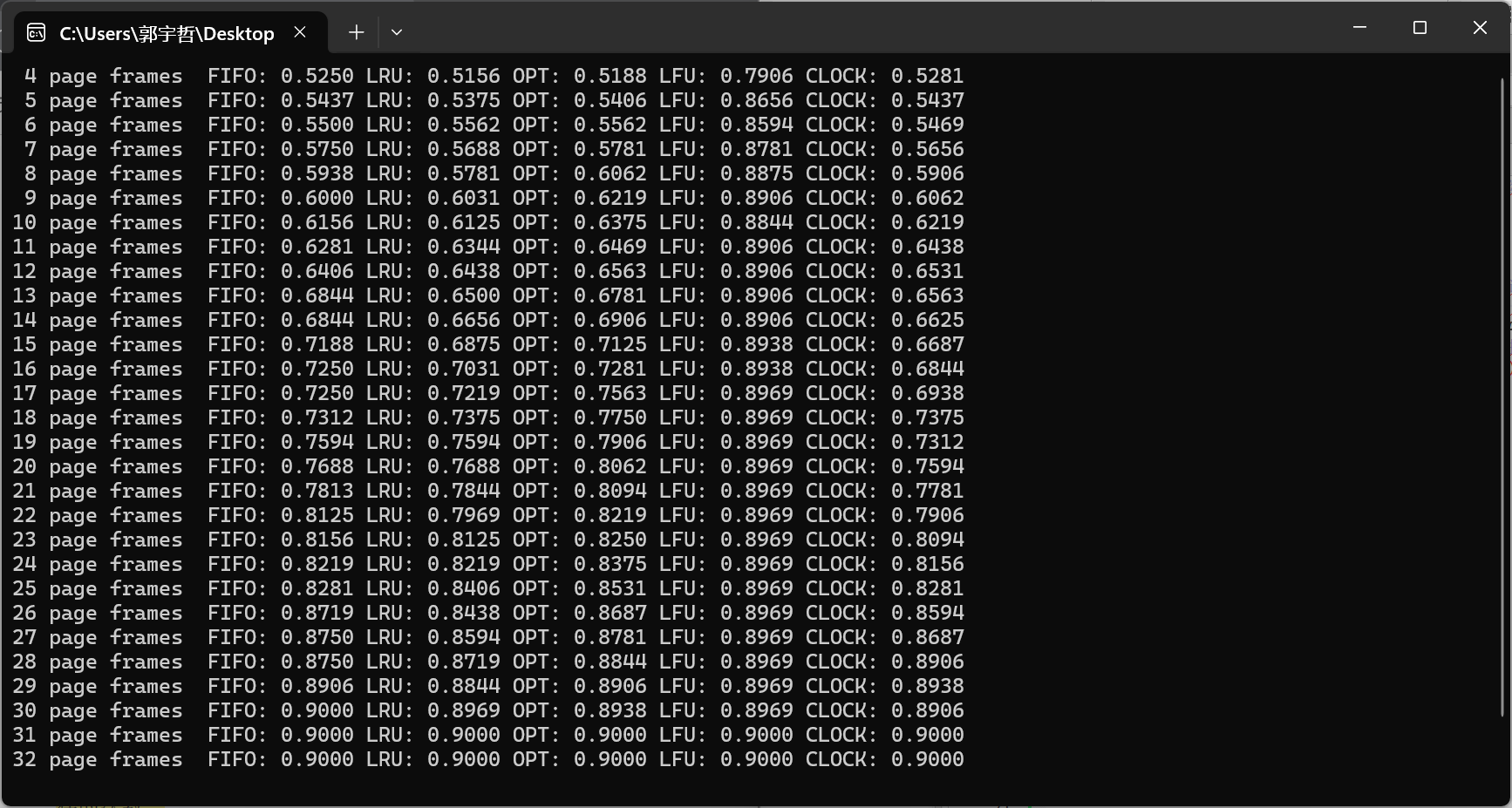
pl[j]. counter = 0;

}

printf("CLOCK: %6.4f", 1-( float) diseffect/320);

}

**运行结果：**



# 四．思考题

1. 为什么要进行内存管理，虚拟存储器的特点是什么？

**（1）进行内存管理的原因：**

内存管理是操作系统中的一个核心任务，其目的是高效地利用计算机系统的内存资源，以保证系统运行的稳定性、效率和安全性。主要原因包括：

①有限的内存资源：计算机的内存资源是有限的，而现代操作系统需要同时运行多个程序（进程）。内存管理确保各个进程能够合理地分配和使用内存，以避免程序冲突或内存资源浪费。

②隔离进程：每个进程需要有独立的内存空间，以避免相互干扰。内存管理通过提供进程间的隔离性，防止一个进程的错误或崩溃影响到其他进程。

③内存共享：对于某些共享资源（例如共享库、内存映射文件等），操作系统需要确保不同进程能够安全、高效地共享内存中的一部分数据。

④内存的动态分配和回收：程序的内存需求不是固定的，内存管理需要支持动态的内存分配和回收机制。这样可以让操作系统根据进程的实际需求调整内存的分配，避免浪费。

⑤提高系统效率：有效的内存管理可以提高系统的总体性能，避免由于内存碎片、过度的内存交换等问题导致的性能瓶颈。

⑥虚拟内存支持：通过虚拟内存管理，操作系统可以提供比实际物理内存更大的“虚拟内存”空间，从而让应用程序认为它们有足够的内存，而不受物理内存的限制。

**（2）虚拟存储器的特点：**

虚拟存储器是操作系统管理内存的一种技术，它通过硬件和软件的协作，提供给程序一个比物理内存更大的虚拟地址空间。虚拟存储器的主要特点包括：

①虚拟地址空间：每个进程都有一个独立的虚拟地址空间，虚拟地址被映射到实际的物理内存地址上。这样，每个进程认为它有独占的内存空间，实际上可能共享物理内存的一部分。虚拟地址空间通常比实际的物理内存要大，操作系统通过虚拟内存技术使得程序可以使用比物理内存更多的空间。

②内存分页：虚拟存储器将内存划分为固定大小的页，并将物理内存划分为块（或称为页框，）。虚拟页与物理页框之间的映射由页表来管理。当程序访问某个虚拟地址时，操作系统通过页表查找对应的物理地址。如果该虚拟页没有被加载到物理内存中，则会发生“页缺失”，操作系统会将对应的页面从磁盘（通常是交换空间或文件）加载到物理内存中。

③内存交换：当物理内存不足时，操作系统会将不活跃的内存页交换到硬盘上的交换空间，以便腾出空间给活跃的进程。交换机制使得操作系统能够在物理内存不足时继续运行多个程序，但频繁的交换会导致性能下降，称为“交换抖动”。

④内存保护：虚拟内存使得操作系统能够对不同进程的内存进行保护。每个进程只能访问它自己的虚拟地址空间，操作系统可以防止进程访问其他进程的内存空间，从而提高系统的安全性和稳定性。如果程序尝试非法访问内存（如访问未分配或已释放的内存），操作系统会触发异常，通常会导致程序崩溃。

⑤地址映射：虚拟内存通过“地址映射”机制将虚拟地址映射到物理内存地址。这个映射通常由硬件中的内存管理单元（MMU）和操作系统的管理软件协作完成。操作系统在内存管理单元中维护着页表，它记录了虚拟地址到物理地址的映射关系。

⑥支持大容量内存：通过虚拟内存，操作系统能够支持比物理内存更大的应用程序内存空间。例如，64位操作系统理论上可以支持高达18亿GB的虚拟内存，而实际物理内存可能远远小于这个数字。

⑦性能提升：虚拟内存机制使得操作系统能够高效地管理内存，减少了程序运行中的内存碎片问题。通过分页、分段等技术，虚拟存储器使得内存管理更加灵活和高效。

1. 几种内存管理算法有何区别与联系？

内存管理算法是操作系统中用来分配和管理内存资源的策略，它们的目标是合理分配内存，以避免内存浪费、提高内存利用率，并且防止不同进程之间的干扰。常见的内存管理算法有：连续分配算法、分页算法、分段算法、以及分页与分段结合算法等。区别与联系如下：

**（1）连续分配算法**

连续分配算法是最简单的一种内存管理方法。它要求所有的程序或进程在内存中占据连续的空间。内存管理的核心就是将可用的内存块按顺序分配给需要的进程。常见的连续分配算法有固定分配和动态分配。

连续分配的特点有：

①简单易实现。

②内存分配和释放速度较快。

③存在内存碎片问题（外部碎片）。

④适用于一些内存需求稳定、较为简单的应用场景。

区别与联系：

①连续分配和其他算法相比，它最大的缺点就是外部碎片的问题，也就是内存虽然没有用完，但由于不能提供连续的空间，导致分配失败。

②它与分页和分段算法的区别是，后者可以避免外部碎片，通过不同的方式管理内存资源。

**（2）分页算法**

分页算法通过将内存和程序都划分为固定大小的页面和页框，进程的虚拟地址空间和物理内存空间都被划分成固定大小的单元，进程的页面映射到物理内存中的页框，不需要连续空间。

特点：

①避免外部碎片：分页算法能有效避免外部碎片问题，因为页面大小固定，并且可以在任何地方分配物理内存页框。

②内部碎片：每个页面的大小是固定的，若进程实际需要的内存小于页面大小，就会产生一定的内部碎片。

③支持虚拟内存：分页可以将进程的地址空间映射到物理内存中，在内存不足时可以使用磁盘上的交换空间来存储不活跃的页面。

区别与联系：

①与连续分配算法相比，分页算法减少了外部碎片，但可能产生内部碎片。

②分页算法更适合多任务环境，能够支持虚拟内存和较大的程序地址空间。

③分页比连续分配更复杂，需要维护页表来管理虚拟地址到物理地址的映射。

**（3）分段算法**

分段算法将内存划分为不同大小的逻辑段（如代码段、数据段、栈段等），每个段都有一个段名和段的大小。每个段在内存中都有一个基地址和长度，操作系统通过段表来进行管理。

特点：

①更符合程序结构：分段算法将内存分配与程序的逻辑结构相结合（如代码段、数据段），使得内存管理更加符合程序的实际需求。

②无内部碎片：由于每个段的大小由程序指定，因此不会产生内部碎片。

③可能有外部碎片：如果分配的段之间没有足够的连续空间，就可能导致外部碎片的问题。

区别与联系：

①分段算法比分页更加灵活，因为它允许按需分配不同大小的段，但它仍然可能会导致外部碎片。

②分段与连续分配算法相比，具有更好的内存利用率和更灵活的内存管理，但仍然可能存在外部碎片。

**（4）分页与分段结合算法**

分页与分段结合算法是一种将分页和分段结合起来的内存管理方法。它首先对进程的地址空间进行分段，每个段内部再进行分页。也就是说，首先按照逻辑划分段，再对每个段内的内容进行分页。特点如下：

①避免外部碎片：通过分页来避免外部碎片问题，同时使用分段来保持程序的逻辑结构。

②灵活性高：结合了分页和分段的优点，既能避免外部碎片，又能更好地符合程序的结构。

③管理复杂：由于需要同时维护段表和页表，系统的管理更加复杂。

区别与联系：

①分页与分段结合算法综合了分页和分段的优点，避免了它们各自的缺点。

②由于它结合了两者的特点，能够支持较为灵活的内存管理，适合大型复杂的系统，尤其是现代操作系统常用的内存管理方式。

**（5）段页式管理**

段页式管理是对分段和分页结合的进一步优化。它将进程的地址空间分为多个段，每个段内部又进行分页。具体来说，每个段由多个页组成，而每个页存放一部分程序数据或代码。特点如下：

①提供了更高的灵活性：段提供了程序的逻辑结构，分页则提供了高效的内存管理。

②避免碎片问题：通过分段解决了外部碎片，通过分页解决了内部碎片。

③较复杂：需要维护段表和页表，管理上比单一的分页或分段复杂。