#### 

#### 操作系统课程设计报告-实验三

目录

[1 Cache和虚拟内存 3](#_Toc28189615)

[1.1 TLB 3](#_Toc28189616)

[1.1.1 题目要求 3](#_Toc28189617)

[1.1.2 解决方案 3](#_Toc28189618)

[1.1.3 代码实现 3](#_Toc28189619)

[1.2 Demand Paging 5](#_Toc28189620)

[1.2.1 题目要求 5](#_Toc28189621)

[1.2.2 解决方案 5](#_Toc28189622)

[1.2.3 代码实现 5](#_Toc28189623)

[1.3 测试截图 9](#_Toc28189624)

# Cache和虚拟内存

## TLB

### 题目要求

Implement software-management of the TLB, with software translation via an inverted page table.

### 解决方案

TLB—是一种小而专用的且快速的硬件缓冲转换表缓冲区（translation look-aside buffer）。反向页表-整个系统只有一个页表，对于每个物理内存的页只有一个相应的条目。

### 代码实现

主要的实现代码还是写在VMKernel和VMProcess两个类中。

/\*\* 通过物理页码索引的coremap \*/

**private** MemoryEntry[] coremap = **new** MemoryEntry[Machine.*processor*().getNumPhysPages()];

/\*\* 在时钟算法中使用的当前页面的持久索引 \*/

**private** **int** clockHand = 0;

/\*\* 从vaddr，PID到PPN的映射的反向页表 \*/

**private** Hashtable<TableKey,MemoryEntry> invertedPageTable = **new** Hashtable<TableKey,MemoryEntry>();

/\*\* A lock \*/

**private** Lock memoryLock;

/\*\* 固定的内存条目数。 \*/

**private** **int** pinnedCount;

/\*\* 如果没有未固定的页面，则所有进程都要等待的条件。\*/

**private** Condition allPinned;

在VMKernel中定义两个变量coremap和invertedPageTable，分别表示物理内存页号标号的内存条目和反转内存页表。coremap负责存放TLB条目和pid的对应关系，invertedPageTable负责存放页表键（虚拟内存地址和pid组成）与内存条目的映射关系。

/\*\*

\* 当进程需要访问一个内存地址时，

\* 通过retrievePage这个方法来获取内存页条目。

\* 先从内核中判断虚拟内存页号是否已经存在在页表中。

\* 如果存在，就标记占用并返回，若不存在则页错误，抛出异常

\*/

**public** TranslationEntry retrievePage(**int** vpn) {

TranslationEntry returnEntry = **null**;

**if** (thunkedSections.containsKey(vpn))

returnEntry = thunkedSections.get(vpn).execute();

**else** **if** ((returnEntry = *kernel*.pinIfExists(vpn, PID)) == **null**)

returnEntry = *kernel*.pageFault(vpn, PID);

Lib.*assertTrue*(returnEntry != **null**);

**return** returnEntry;

}

当进程需要访问一个内存地址时，通过retrievePage这个方法来获取内存页条目。先从内核中判断虚拟内存页号是否已经存在在页表中。如果存在，就标记占用并返回，若不存在则页错误，抛出异常，并在processor.regBadVAddr寄存器中读到发生页错误的虚拟内存页号，进入TLB未命中的方法处理。

/\*\*

\* 取消固定与物理页码相对应的页面。

\* **@param** ppn

\*/

**void** unpin(**int** ppn) {

memoryLock.acquire();

MemoryEntry me = coremap[ppn];

**if** (me.pinned)

pinnedCount--;

me.pinned = **false**;

allPinned.wake();

memoryLock.release();

}

/\*\*

\* 将页面固定（如果存在）。

\* **@param** vpn

\* **@param** pid

\* **@return** true if the page exists (and was pinned)

\*/

TranslationEntry pinIfExists(**int** vpn, **int** pid) {

MemoryEntry me = **null**;

memoryLock.acquire();

**if** ((me = invertedPageTable.get(**new** TableKey(vpn, pid))) != **null**) {

**if** (!me.pinned)

pinnedCount++;

me.pinned = **true**;

}

memoryLock.release();

**if** (me == **null**)

**return** **null**;

**else**

**return** me.translationEntry;

}

如果TLB中存在物理地址和需求vaddr对应物理地址相同或者已经失效的页表项，则写入新的页表项。如果都不满足条件，就使用随机替换的方法。

/\*\*

\* 处理与虚拟地址关联的TLB未命中。

\* **@param** bad vaddr

\*/

**public** **void** handleTLBMiss(**int** vaddr) {

**if** (!validAddress(vaddr)) {

} **else** {

TranslationEntry retrievedTE = retrievePage(Processor.*pageFromAddress*(vaddr));

**boolean** unwritten = **true**;

//使所有指向新物理页面的条目无效

//最好查找一个无效（即为空）条目并将其替换。 否则，请使用随机替换。

Processor p = Machine.*processor*();

**for** (**int** i = 0; i < p.getTLBSize() && unwritten; i++) {

TranslationEntry tlbTranslationEntry = p.readTLBEntry(i);

//如果条目匹配则使该条目无效

**if** (tlbTranslationEntry.ppn == retrievedTE.ppn) {

**if** (unwritten) {

p.writeTLBEntry(i, retrievedTE);

unwritten = **false**;

} **else** **if** (tlbTranslationEntry.valid) {

tlbTranslationEntry.valid = **false**;

p.writeTLBEntry(i, tlbTranslationEntry);

}

} **else** **if** (unwritten && !tlbTranslationEntry.valid) {

p.writeTLBEntry(i, retrievedTE);

unwritten = **false**;

}

}

//如果我们尚未将其写入TLB，请使用随机替换政策

**if** (unwritten) {

**int** randomIndex = generator.nextInt(p.getTLBSize());

TranslationEntry oldEntry = p.readTLBEntry(randomIndex);

//将信息传播到内存中

**if** (oldEntry.dirty || oldEntry.used)

*kernel*.propagateEntry(oldEntry.ppn, oldEntry.used, oldEntry.dirty);

p.writeTLBEntry(randomIndex, retrievedTE);

}

//取消固定物理页面

*kernel*.unpin(retrievedTE.ppn);

}

}

## Demand Paging

### 题目要求

Implement demand paging of virtual memory. For this, you will need routines to move a page from disk to memory and from memory to disk. You should use the Nachos stub file system as backing store.

### 解决方案

第二部分是关于交换空间。交换空间允许物理内存页被交换到硬盘中，来实现理论上无限内存空间的效果。如果发生了页错误，内核会检查自己的页表来判断需求页是否在物理内存中。如果不在，会先从硬盘中调页，页表项指向新调的页，装入页表项并恢复程序运行。当然内核也要在物理内存中找到可以写入的区域，必要时会将修改过的内存页写回到硬盘中。这个机制的性能极大依赖了调页策略，经常使用的是最近访问的页不被调出的算法。并且如果内存页没有被修改过，就没必要写回硬盘，这样可以节省很多时间。

### 代码实现

我们在Kernel中定义一个Swap类，负责处理交换文件的创建、调入、调出等操作，交换文件通过StubFileSystem来创建名为”swapfile”的文件，存放在/test文件夹下。

/\*\*

\* **@return** 打开交换文件

\*/

**protected** OpenFile openSwapFile() {

**return** *fileSystem*.open("swapfile", **true**);

}

**private** **class** Swap {

Swap() {

swapFile = openSwapFile();

}

/\*\*

\* 将物理页面写入交换文件（如果尚未交换）或脏文件

\* 注意：为安全起见，应固定物理页

\*/

**void** swapOut(MemoryEntry me) {

**if** (me.translationEntry.valid) {

SwapEntry swapEntry = **null**;

TableKey tk = **new** TableKey(me.translationEntry.vpn, me.processID);

swapLock.acquire();

**if** (me.translationEntry.dirty || !swapTable.containsKey(tk)) {

// 使用空缺职位（如果有）

**if** (freeList.size() > 0) {

swapEntry = freeList.removeFirst();

swapEntry.readOnly = me.translationEntry.readOnly;

}

// 否则扩展交换文件

**else** {

System.***out***.println("发生页面置换");

swapEntry = **new** SwapEntry(maxTableEntry++, me.translationEntry.readOnly);

}

swapTable.put(tk, swapEntry);

}

swapLock.release();

**if** (swapEntry != **null**) {

// 写物理页面

Lib.*assertTrue*(swapFile.write(swapEntry.swapPageNumber \* Processor.***pageSize***,

Machine.*processor*().getMemory(),

me.translationEntry.ppn \* Processor.***pageSize***,

Processor.***pageSize***) == Processor.***pageSize***);

}

}

}

**private** **int** maxTableEntry = 0;

/\*\*

\* 从交换文件读取虚拟页面并将其写入物理内存

\* 注意：为安全起见，应固定物理页

\*/

**void** swapIn(**int** vpn, **int** pid, **int** ppn) {

swapLock.acquire();

SwapEntry swapEntry = swapTable.get(**new** TableKey(vpn, pid));

swapLock.release();

**if** (swapEntry != **null**) {

//读物理页

Lib.*assertTrue*(swapFile.read(swapEntry.swapPageNumber \* Processor.***pageSize***,

Machine.*processor*().getMemory(),

ppn \* Processor.***pageSize***,

Processor.***pageSize***) == Processor.***pageSize***);

coremap[ppn].translationEntry.readOnly = swapEntry.readOnly;

}

}

/\*\*

\* **@return** 如果给定页面在交换文件中，则为True

\*/

**boolean** pageInSwap(**int** vpn, **int** pid) {

swapLock.acquire();

**boolean** retBool = swapTable.containsKey(**new** TableKey(vpn, pid));

swapLock.release();

**return** retBool;

}

/\*\*

\* 与该过程关联的交换文件中的空闲页面条目，因此可以重复使用。

\*（免费列表上的“位置”页面）

\* **@param** maxVPN：进程中最高的VPN + 1

\*/

**void** freePages(**int** maxVPN, **int** pid) {

swapLock.acquire();

SwapEntry freeEntry;

**for** (**int** i = 0; i < maxVPN; i++)

**if** ((freeEntry = swapTable.get(**new** TableKey(i, pid))) != **null**)

freeList.add(freeEntry);

swapLock.release();

}

/\*\*

\* 关闭并删除交换文件

\*/

**void** cleanup() {

swapFile.close();

*fileSystem*.remove(swapFile.getName());

}

/\*\* 对包含换出页面的文件的引用

\* 该文件分为页面大小的块，这些块由SwapEntries索引

\*/

**private** OpenFile swapFile;

/\*\* 交换文件中当前未使用的列表 \*/

**private** LinkedList<SwapEntry> freeList = **new** LinkedList<SwapEntry>();

/\*\* 流程页面之间的映射以及它们在交换文件中的位置 \*/

**private** HashMap<TableKey, SwapEntry> swapTable = **new** HashMap<TableKey, SwapEntry>();

/\*\* A <tt>Lock</tt> \*/

**private** Lock swapLock = **new** Lock();

/\*\* 表示交换文件中交换页面位置的类 \*/

**private** **class** SwapEntry {

SwapEntry (**int** spn, **boolean** ro) {

swapPageNumber = spn;

readOnly = ro;

}

**int** swapPageNumber;

**boolean** readOnly;

}

}

在发生页错误时，内核会现在交换页表中查询进程所需的内存页是否存在，如果存在则从交换页表中读取交换页表项。

使用clockAlgorithm（二次机会算法）进行页选择调出时，会选择clockHand指向的下一个最近未使用或失效页面，将其使用swapOut()调出。

/\*\*

\* LRU置换算法-二次机会算法

\* **@return** ppn

\*/

**private** MemoryEntry clockAlgorithm() {

memoryLock.acquire();

**while** (pinnedCount == coremap.length) allPinned.sleep();

/\*

\* We only need to flush before we start the search.

\*/

propagateAndFlushTLB(**false**);

// When we get here, there MUST be a non-pinned page to find

**while** (**true**) {

clockHand = (clockHand+1) % coremap.length;

MemoryEntry page = coremap[clockHand];

// 跳过固定页面

**if** (page.pinned)

**continue**;

// 优先选择无效页面

**if** (page.processID == -1 || page.translationEntry.valid == **false**)

**break**;

// 如果最近使用继续

**if** (page.translationEntry.used) {

page.translationEntry.used = **false**;

}

// Otherwise use this page!

**else** {

**if** (page.translationEntry.valid) {

}

**break**;

}

}

MemoryEntry me = coremap[clockHand];

pinnedCount++;//we don't have to check if it is pinned because it should only be unpinned at this point

me.pinned = **true**;

invalidateTLBEntry(clockHand);

//Remove the mapping from the page table

MemoryEntry me1 = **null**;

**if** (me.processID > -1)

me1 = invertedPageTable.remove(**new** TableKey(me.translationEntry.vpn, me.processID));

memoryLock.release();

//swap out the page here if the page had existed in memory

//**FIXME**: Will we always need to swap out?

**if** (me1 != **null**) swap.swapOut(me);

**return** me;

}

## 测试截图

进行测试时，写matmult3.c用户程序并编译生成coff文件：

/\* matmult.c

 \*    Test program to do matrix multiplication on large arrays.

 \*

 \*    Intended to stress virtual memory system. Should return 7220 if Dim==20

 \*/

#include "syscall.h"

#define Dim     57  /\* sum total of the arrays doesn't fit in

             \* physical memory

             \*/

int A[Dim][Dim];

int B[Dim][Dim];

int C[Dim][Dim];

int

main()

{

    int i, j, k;

    for (i = 0; i < Dim; i++)       /\* first initialize the matrices \*/

    for (j = 0; j < Dim; j++) {

         A[i][j] = i;

         B[i][j] = j;

         C[i][j] = 0;

    }

    for (i = 0; i < Dim; i++)       /\* then multiply them together \*/

    for (j = 0; j < Dim; j++)

            for (k = 0; k < Dim; k++)

         C[i][j] += A[i][k] \* B[k][j];

    printf("C[%d][%d] = %d\n", Dim-1, Dim-1, C[Dim-1][Dim-1]);

    return (C[Dim-1][Dim-1]);       /\* and then we're done \*/

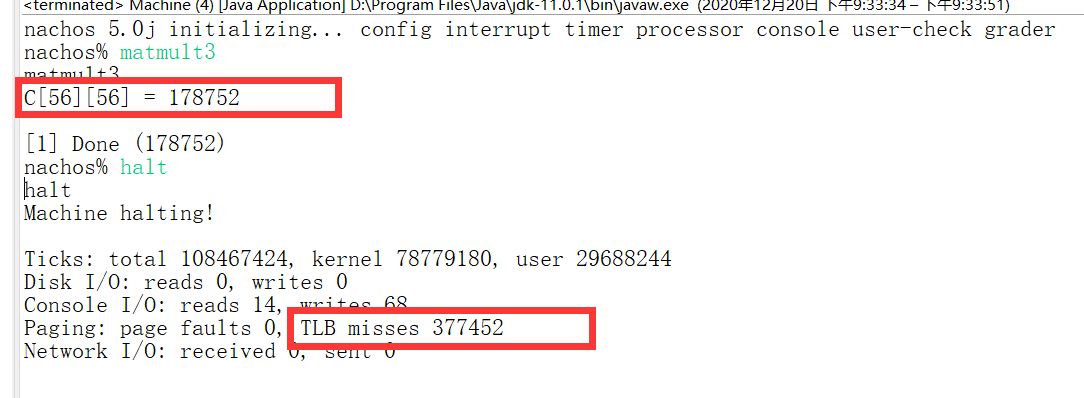
}

分析matmult3可知这是个矩阵乘法的程序，会用到大量内存，因此有频繁访存的过程。也会发生TLB不命中从而进行页置换操作。

实验二的基础上：由于不能进行换页，所以程序根本跑不起来。



实验三的基础上



能够正常运行并且发生了TLB未命中，进行调页，说明调用正确。