Nachos Phase 2

这次说一个总的流程，细节上不赘述了。首先，CPU在TLB找不到page了，调用这个：

private void handleTLBMiss(int vaddr)

{

int vpn = vaddr / pageSize;

int ppn = loadPage(vpn);

TranslationEntry page = getEntryPPN(ppn);

Lib.assert(page.valid);

int toRemove = -1;

for (int i = 0; i < Machine.processor().getTLBSize(); ++i)

{

TranslationEntry entry = Machine.processor().readTLBEntry(i);

if (!entry.valid)

{

toRemove = i;

break;

}

else if (!entry.used)

{

toRemove = i;

}

}

if (toRemove == -1)

{ // all TLB are valid and used

toRemove = Lib.random(Machine.processor().getTLBSize());

}

Machine.processor().writeTLBEntry(toRemove, page);

unlockPage(ppn);

}

这里最关键的是loadPage，它负责把即便不再内存里的page也调度近来：

private int loadPage(int vpn)

{

if (vpn >= numPages)

doExit(true, 0);

int ppn;

if (VMKernel.virtualMemoryManager.contains(getProcessID(), vpn))

{

ppn = lockPage(vpn, false);

}

else

{

int s = sectionTable[vpn];

boolean readonly = false;

if (s == -1)

{

ppn = lockPage(vpn, true);

clearPage(ppn);

}

else

{

CoffSection section = coff.getSection(s);

int i = vpn - section.getFirstVPN();

ppn = lockPage(vpn, true);

section.loadPage(i, ppn);

readonly = section.isReadOnly();

}

TranslationEntry entry = getEntryPPN(ppn);

entry.dirty = true;

entry.readOnly = readonly;

}

if (ppn == -1) doExit(true, 0);

return ppn;

}

这里分2种情况：在VM中和不在VM中。这里VM已经包括硬盘swap, 因此不在VM中只有lazy loading或者错误2种情况。在那个else后处理。

如果在VM中，则调用：ppn = lockPage(vpn, false)来将其载入并锁定（设置一个锁定位，如果被锁定则不会被scheduler调整出去。在handle TLB miss的最后会unlock.）

public int lockPage(int pid, int vpn, boolean newpage)

{

lock.acquire();

TableEntry vaddr = getVaddr(pid, vpn);

if (vaddr == null)

{

int ppn = loadPage(pid, vpn, newpage);

if (ppn == -1)

{

lock.release();

return -1;

}

vaddr = invertedPageTable[ppn];

}

vaddr.inuse = true;

lock.release();

return vaddr.entry.ppn;

}

这里的首先用getVaddr查找反转页表，如果找到了，那么page在内存中，直接返回即可，否则就要调用loadPage函数，和上面那个不一样，它handle那些在硬盘里面的页:

public int loadPage(int pid, int vpn, boolean newpage)

{

Lib.assert(lock.isHeldByCurrentThread());

Lib.assert(getVaddr(pid, vpn) == null);

TranslationEntry freepage = getFreePage();

if (freepage == null)

{

return -1;

}

freepage.vpn = vpn;

freepage.valid = true;

freepage.dirty = newpage ? true : false;

freepage.used = true;

freepage.readOnly = false;

invertedPageTable[freepage.ppn].pid = pid;

Lib.assert(invertedPageTable[freepage.ppn].entry == freepage);

if (!newpage && !pagefile.read(pid, vpn, freepage.ppn))

{

TranslationEntry[] pages = new TranslationEntry[1];

pages[0] = freepage;

VMKernel.memoryManager.deallocPages(pages);

releasePageFromPhysicalMemory(freepage.ppn);

return -1;

}

VirtualAddress key = new VirtualAddress(pid, vpn);

Lib.assert(!hashtable.containsKey(key));

hashtable.put(key, invertedPageTable[freepage.ppn]);

return freepage.ppn;

}

这里主要是getFreePage 这个函数将返回一个空闲的页。当系统有空余内存的时候，类似phase 2直接查找free page表然后返回即可，否则需要使用时钟替换算法来替换一个页面。在找到freepage后，如果不是请求新的页面，就需要从pageFile里面掉。PageFile类的实现在最后给出。这里看一下getFreePage：

private TranslationEntry getFreePage()

{

Lib.assert(lock.isHeldByCurrentThread());

TranslationEntry[] entries

= VMKernel.memoryManager.allocPages(1);

TranslationEntry entry = null;

if (entries != null)

{

int ppn = entries[0].ppn;

entry = invertedPageTable[ppn].entry;

entry.ppn = ppn;

entry.dirty = false;

entry.valid = false; // set to true later

}

else

{

int steps = 0;

int pid = -1;

while (true)

{

TableEntry vaddr = invertedPageTable[clockNeedle];

TranslationEntry cur = vaddr.entry;

if (!vaddr.inuse)

{

if (cur.used)

cur.used = false;

else

{

entry = cur;

pid = vaddr.pid;

break;

}

}

clockNeedle

= (clockNeedle + 1) % invertedPageTable.length;

if (steps == invertedPageTable.length + 2)

{

// already turned one round + 2 page, still nothing found

// this may happen only many pages are in use

steps = 0;

lock.release();

KThread.yield();

lock.acquire();

}

++steps;

}

if (entry.valid)

{

if (entry.dirty)

{

if (!pagefile.write(pid, entry))

{

return null;

}

releasePageFromPhysicalMemory(entry.ppn);

}

else

{

Lib.assert(entryInSwap(pid, entry.vpn));

}

}

}

Lib.assert(invertedPageTable[entry.ppn].entry == entry);

return entry;

至此整个流程结束。

PageFile: 很类似于readVirtualMemory这类函数。不过这里一次只需要读一块。

package nachos.vm;

import java.util.Hashtable;

import nachos.machine.Lib;

import nachos.machine.Machine;

import nachos.machine.OpenFile;

import nachos.machine.Processor;

import nachos.machine.TranslationEntry;

import nachos.threads.Lock;

import nachos.threads.ThreadedKernel;

import nachos.userprog.MemoryManager;

public class PageFile {

public PageFile(String filename) {

lock = new Lock();

lock.acquire();

freepages = new MemoryManager(0);

swapfile = ThreadedKernel.fileSystem.open(filename, true);

hashtable = new Hashtable();

this.filename = filename;

lock.release();

}

public void close() {

swapfile.close();

swapfile = null;

ThreadedKernel.fileSystem.remove(filename);

}

public boolean containsEntry(int pid, int vpn) {

lock.acquire();

VirtualAddress key = new VirtualAddress(pid, vpn);

boolean retval = hashtable.containsKey(key);

lock.release();

return retval;

}

public boolean write(int pid, TranslationEntry entry) {

lock.acquire();

if (freepages.freepageCount() < 1)

expand(DLT\_EXP);

VirtualAddress key = new VirtualAddress(pid, entry.vpn);

boolean newpage = false;

TranslationEntry page =

(TranslationEntry)hashtable.get(key);

if (page == null) {

newpage = true;

page = freepages.allocPages(1)[0];

hashtable.put(key, page);

}

page.vpn = entry.vpn;

byte[] memory = Machine.processor().getMemory();

int paddr = entry.ppn \* pageSize;

int faddr = page.ppn \* pageSize;

int bytesWritten = swapfile.write(faddr, memory, paddr, pageSize);

if (bytesWritten != pageSize) {

if (newpage) free(pid, page.vpn);

lock.release();

return false;

}

lock.release();

return true;

}

public boolean read(int pid, int vpn, int ppn) {

lock.acquire();

VirtualAddress key = new VirtualAddress(pid, vpn);

TranslationEntry page =

(TranslationEntry)hashtable.get(key);

byte[] memory = Machine.processor().getMemory();

int paddr = ppn \* pageSize;

int faddr = page.ppn \* pageSize;

int bytesRead = swapfile.read(faddr, memory, paddr, pageSize);

if (bytesRead != pageSize) {

lock.release();

return false;

}

lock.release();

return true;

}

public void free(int pid, int vpn) {

lock.acquire();

VirtualAddress key = new VirtualAddress(pid, vpn);

hashtable.remove(key);

TranslationEntry [] entries = new TranslationEntry[1];

freepages.deallocPages(entries);

lock.release();

}

private void expand(int n) {

TranslationEntry []entry = new TranslationEntry[n];

for (int i = 0; i < n; ++i) {

entry[i] = new TranslationEntry();

entry[i].ppn = pageNum++;

entry[i].valid = true;

}

freepages.deallocPages(entry);

}

private MemoryManager freepages = null;

private OpenFile swapfile = null;

private Hashtable hashtable = null;

private int pageNum = 0;

private Lock lock = null;

private String filename;

private static final int DLT\_EXP = 10;

private static final int pageSize = Processor.pageSize;

}

至于测试，只需要重复phase 2的测试即可，只要全部pass即可。这次的内存默认配置比上次少得多，只有16个page, 可以递归调用sh来消耗内存。

note: 发现一个nachos的bug导致matmult默认无法运行

CoffSection.java

line 40:

if (vaddr%Processor.pageSize != 0 || size < 0 ||

contentOffset < 0 /\*|| contentOffset+size > file.length()\*/)