Niels Joubert CS162-PC
Nadeem Laiwala CS162-PD
Akhil Dhar CS162-KK
Eugene Li CS162-KI
William Wu CS162-KJ
April 27, 2008
Computer Science 162

CS162 Design Doc Phase 3: Caching and Virtual Memory

Design Overview

This entire project about the interaction between virtual memory and physical memory is built around the MemoryMap class, which serves as a mapping from process ID and virtual page number to a physical page number and vice versa using MemoryMapEntry objects. This allows for constant-time lookups of both (PID,VPN) and PPN keys to MemoryMapEntries.

The MemoryMap class is the only place where we touch the TLB, Inverted Page Table, and the CoreMap. It follows the Consumer/Producer scheme, with the checking out and pinning of pages being equivalent to consuming, and checking in and unpinning pages being equivalent to producing. When a page is pinned, then the resource is "consumed" and no other process is able to interact with that resource. When the page is eventually unpinned, the resource is "produced" and the page is available for a different process to utilize.

We also used the design principal from version control when checking out physical pages to a process. When a process requests a physical page, rather than give a reference to the actual page, we merely mark the page as "checked out" and give a copy of the physical page. Then when the process is finished with the page, they "check in" the page. This performs an update of the physical page to reflect the copy, and then the page is no longer marked as "checked out". This is similar to the act of commiting a change to a repository.

Checking in and Checking out pages

Atomicity of page-handling operations is the hardest part of this project. We want to allow multiple threads concurrent access to growing its stack and loading its executable code dynamically. On the other hand, we do not want to allow a page that the kernel is currently modifying for a process (by, for example, loading executable data into) to be touched by any other process. To handle both these constraints, we designed the MemoryMap class to make page-level operations thread-safe while allowing concurrency in memory as a whole. This is accomplished through the process of checking in and checking out pages. Before you can do any access to a page that involves handling TLB misses or pagefaults, you have to check a page out of the MemoryMap. This makes that page unavailable during the time the page is checked out. Once the kernel has completed the operation on this page, it is checked back in, and any processes waiting for an available page is notified.

Operations on a page comes in three flavors:

- Loading the translation entry for a page in memory into the TLB so that the process can access it
- Getting a blank page - for writing arguments to or growing the stack
- Moving a page into or out of swap - either to make space for a different page or to load an already-accessed page from swap into memory.

All these operations can be made atomic on a page level through the above description of checking in and checking out. The consumer/producer design pattern allows any amount of pages and any amount of threads to work int his sceme, so that threads can sleep on the condition that they are waiting for an available page. Deadlock is prevented by allowing a process to check out only one page at a time.

Implementation:

MemoryMap.java

```
private Hashtable<Integer, Hashtable<Integer, Integer> > invertedPageTable;
private MemoryMapEntry[] coreMap;
private int clockHand;                    //for finding free or replaceable pages with
clockalgorithm

private int numUnpinnedPages;             //the number of unpinned pages.
private Lock mmLock;                      //provides synchronicity.
private Condition2 pagesAvailable;        //sleep on this if there are no pages
unpinned.

enum checkInEffect { WRITE_TO_TLB, INVALIDATE_IF_IN_TLB, IGNORE_TLB; }
//used by VMProcess when resolving page faults

private static MemoryMap theOnlyMemoryMap = null
```

```
/* Creates a memory map */
private MemoryMap() {
    Instantiate the various variables needed
    Fill in memory map with MemoryMapEntries that are invalid and unpinned
}

/* Check out the MME corresponding to the given pid and vpn.  Returns only a
valid pages.
 * This is used to return a page that is already in memory.
*/
public MemoryMapEntry checkOut(int pid, int vpn) {
    acquire mmLock
    obtain vpn,ppn processTable according to pid from invertedPageTable
    if this is null, release lock and return null

    obtain ppn acording to vpn from processTable
    if this is null, release lock and return null

    if ppn is pinned or is invalid, release lock and return null
    pin coreMap that has this ppn
    retrieve MemoryMapEntry according to this ppn

    release lock and return this MME
}

/* Check out any page to use. This is used when you need a free page that you
can load data into. */
public MemoryMapEntry checkOut(int pid) {
    write TLB back to coreMap
    acquire lock

    sleep if there are no unpinned pages

    now run clock algorithm to select a MemoryMapEntry to return
    if the selected entry is in the tlb, invalidate it

    create a new MemoryMapEntry according to the ppn of returned
MemoryMapEntry
    release lock and return this MemoryMapEntry
}

/* Checks in the given entry into the MemoryMap. This checks in invalid pages. */
public boolean checkIn(MemoryMapEntry entry, checkInEffect effect) {
    acquire lock
    set entry's used bit to true

    obtain processTable according to this pid
    if this entry is valid, then we know it's in the IPT
        if this processTable is null, return false
        obtain ppn from processTable
        if this ppn is null, return false
    else
        if processTable is null, create a new one and put this entry into IPT
        set this entry bit to true

    if effect is WRITE_TO_TLB, write this entry to tlb
    if effect is INVALIDATE_IF_IN_TLB, invalidate this tlb entry
    if effect is IGNORE_TLB, do nothing

    unpin this entry
    create a new MemoryMapEntry and set it to coreMap
    wake all threads that were sleeping because there were no unpinned pages
    release lock and return true
}

/* Moves a page from memory to swap */
public boolean removeFromMemory(MemoryMapEntry entry) {
    acquire lock
    if this entry is valid, we know we want to remove it
        invalidate this tlb entry
        remove this entry's vpn mapping from the processTable
        set this entry's readOnly bit to true
        invalidate this entry
        invalidate readOnly and valid bits for the corresponding coreMap entry
    release lock and return true
}

/* Cleans up all memory used by a process before killing the process */
public void invalidateProcess(int pid) {
    acquire the Lock
    write back tlb to coreMap

    obtain processTable associated with this pid
    if this returns null, release lock and return

    for every ppn in this processTable, set the coreMap entry to a new
MemoryMapEntry with this ppn
    release lock
}

public void TLBtoMemoryMap(int pid, boolean invalidateEntries) {
    obtain interrupt status of the Machine
    for every entry in our TLB, write-back this entry into TLB
    restore interrupt status
}

public void touchPhysicalPage(int ppn, boolean makeDirty) {
    set coreMap entry associated with this ppn to be used
    if makeDirty is true, then also set dirty bit of this entry
}

private void TLBEntryToMemoryMap(int pid, boolean invalidateEntriFes, int n,
boolean careAboutPins) {
    obtain interrupt status of the Machine

    obtain tlb entry associated with the n integer
    if this entry is dirty and valid
        if the coreMap[ppn] is pinned and careAboutPins is true
            make sure all the corresponding coreMap bits of this ppn is equal to
tlbEntry's bits
        else
            set the bits of the corresponding coreMap equal to the bits of the tlbEntry
    if invalidateEntries is true
        create new TranslationEntries and write into TLB

    restore interrupt status
}

/* Writes a TLB entry for the given MME.  The MME must be pinned. */
private void writeEntryToTLB(MemoryMapEntry entry) {
    obtain some tlb entry to replace
    write that tlb entry back to coremap
    create a new TranslationEntry object with the given MemoryMapEntry object
    write this new TranlsationEntry object to the TLB
}

private void invalidateTLBEntry(int ppn) {
    obtain interrupt status of the Machine
    for every tlbEntry in tlb, check to see if any correspond to given ppn
        if so, replace that tlb entry with a new TranslationEntry object
    restore the interrupt status
}

public void close() {
    set theOnlyMemoryMap entry to null
}

/* Singleton design pattern to ensure only one MemoryMap at any point in time */
public static MemoryMap getMemoryMap() {
    obtain interrupt status of machine
    if theOnlyMemoryMap is null, then create the object
    restore interrupt status
    return theOnlyMemoryMap
}

public void TLBwriteback(int pid) {
    for every entry in the TLB
        set corresponding coreMap entry information to the TLB entry
}
```

MemoryMapEntry inner class:

```
public class MemoryMapEntry {
    private int pid, vpn, ppn
    private boolean dirty, used, valid, readOnly, pinned

    MemoryMapEntry(MemoryMapEntry entry) {
        copy given entry to this.entry
    }

    MemoryMapEntry(int ppn) {
        Creates the default entry for the given ppn
        set this.ppn = ppn
        set pid and vpn to -1
        set dirty, used, valid, readOnly, pinned to false
    }

    private void pinThis() {
        set this.pinned to true
        decrement numUnpinnedPages
    }

    private void unPinThis() {
        increment numUnpinnedPages
        set this.pinned to false
    }

    public boolean setPid(int pid) {
        if this page is valid and is pinned
            set this.pid to given pid and return true
        else return false
    }

    public boolean setVpn(int vpn) {
        if this page is valid and is pinned
            set this.pid to given pid and return true
        else return false
    }
}
```

The MemoryMap class creates encapsulation for the virtual memory elements that would be prone to corruption. With this class, we can now implement the three tasks defined for this project.

Task One - (30%) Software Management of TLB

This task of the project is concerned with the interactions between the Translation Lookaside Buffer (TLB) and the inverted page table. The inverted page table is a global data structure that maps a process ID and a virtual page number to a physical page number. It resides inside of the MemoryMap. This is a departure from a scheme where each process has its own personal page table. We must ensure that the translations inside of the TLB, which is managed by the processor, are consistent with the translations that exist in the inverted page table.

Correctness Constraints

- On context switch, TLB entries must be invalidated.
- Over a large sample of memory accesses, all TLB entries are utilized.
- On replacement, the kernel must write back to disk before removing the TLB entry.
- Must utilize all invalid TLB entries before adopting random replacement policy.
- Must keep track of all the pages in the system that are currently in use.
- The inverted page table must only contain entries for pages that are actually in physical memory.

Implementation

The foremost goal of this task is to ensure consistency between the inverted page table (IPT) and the TLB. Since we store our IPT inside of MemoryMap, we implemented a number of methods to serve as the interface for maintaining this consistency. TLBtoMemoryMap() will perform write back from your TLB to your MemoryMap. This is called at every checkOut, at every invocation of invalidateProcess, and at every context switch to ensure that all information in your TLB is reflected in your MemoryMap. The touchPhysicalPage method gets called on readVM and writeVM to ensure that the correct bits are set for an accessed page.

TLB Misses and Page Faults

If an address translation does not reside in the TLB, we call that a TLB miss. The processor throws an exception where the cause is Processor.exceptionTLBMiss. When we catch this exception, we must look in the inverted page table for the translation. If the IPT does not yield a translation, then the page does not reside in physical memory, so we must bring that page in from disk. This is the job of Task 2.

Managing the TLB

We provide a number of methods earlier mentioned in MemoryMap to facilitate the managing of the TLB. When a context switch occurs, we first move the contents of the TLB to the MemoryMap, where "MemoryMap" is both the inverted page table and the coreMap. Then we invalidate all the entries in the TLB to make sure that the next running process does not have access to any of the previous process's address translations.

Our TLB entry replacement policy will be random, since the TLB is small and its contents change rapidly. It gets emptied fairly frequently (on every context switch), so there isn't much sense in putting any effort into making its entry replacement policy perform better. The cost of the overhead to maintain used bits or timestamps is too great given the low potential for benefit.

Inverted Page Table

To create an inverted page table, we will create a hashtable that stores hashtables. We first index it by process ID to obtain a process-specific hash table. Only the currently running process is allowed to access its own processTable. This second table can be indexed by vpn to retrieve the ppn of the physical page that contains the desired virtual page. Every entry in the IPT must be valid, and it is the job of the page swapping mechanisms to ensure that the validity of IPT entries maintains true. This is outlined inside of MemoryMap. We use the inverted page table when we want to perform address translation, which occurs when we want to checkIn and checkOut pages. We determine a virtual to physical page mapping and then perform the desired operation on that physical page, either a read or a write.

Test

We want to test our implementation for conformance to the correctness constraints we laid down with the following test cases:
After each memory access, ensure validity of accessed TLB entry.
On each memory access, print utilized TLB index to ensure complete TLB utilization

Load a user process P1
- Read from memory location M1
- Write from memory location M2
- Read from memory location M2
- Read from memory location M3, M4, M5, M6 ... check that write-back occurs for data in M2

Load another process P2 (context switch)
- Check that all N TLB entries are invalid, where N is the number of TLB entries
- Read from N+2 different memory locations
- Ensure that the first N memory accesses choose invalid entries
- Print notification to screen within each random number generator call to check that it's

being used on the N+1st and N+2nd accesses

Task Two - (40%) Demand Paging of Virtual Memory

This task addresses the kernel's actions when presented with a page fault. A page fault is where a process tries to access a virtual memory address whose page does not currently reside in physical memory. The goal of this task is to implement a system where pages can move between memory and disk to give the impression of a much larger physical memory. When a page is requested by the processor, it should be brought from disk into memory to satisfy the request. Also, pages in memory that have not been recently accessed should spill out to disk to make room for new pages.

All of these actions should be invisible to the processor. When a page fault occurs, the processor only needs to know that it can redo the instruction that caused the page fault exception and the correct address translation will conveniently reside in its TLB.

Correctness Constraints

- The inverted page table must only contain entries for pages that are actually in physical memory.
- Pages that are not dirty, including read-only pages, must not be written to the swap file.
- A process must not move a page while another is doing another operation on that same page.
- The swap file may grow to an arbitrary large size
- The swap file should be closed and deleted when VMKernel.terminate() is called.
- The process must be killed if the process experiences an I/O error while accessing the swap file.
- The dirty bit for a page should be set when you write to it
- The used bit for a page should be set when you read from it or write to it

Implementation

To keep track of all pages in the system, we use the MemoryMap that was previously mentioned. It internally maintains a CoreMap, which is an array of MemoryMapEntrys, where each physical page in memory is represented by a MemoryMapEntry.

To give the impression that we have an almost unlimited amount of physical memory, we use the disk as a temporary storage of pages that don't fit in physical memory. To accomplish this, we create a class called Swap.java that follows the Singleton design pattern. It has a reference to an OpenFile called "swapFile.dat", which is simply a file on disk that we can write to. It also has a number of book keeping items, such as the size of the swapfile and the "pages" in the swapfile that are available to write to. The word "pages" is in quotes because we enforce the constraint that you must read or write exactly one page worth of bytes to the swapfile for a single operation.

Swap.java

```
private static final String swapFileName = "swapFile.dat";
private static int swapInCount=0;
private static int swapOutCount=0;
private static int lastSwapFileSize;

private Lock swapLock;
private OpenFile swapFile;
private LinkedList<Integer> freeList;
private int swapFileSize;
private Hashtable<Integer, Hashtable<Integer, Integer>> swapFileTable; //Maps
pid to a hashtable that maps vpn to ppn

private static Swap theOnlySwap;  // Gets initialized inside of VMKernel.initialize()
```

The interface for interacting with the swapfile is the getSwap method, which returns the Singleton object. Once you hold a reference to the Singleton object, the only operations that may be called externally are contains, moveToSwap, moveFromSwap. Contains() can query the swapfile to see if a matching (pid,vpn) exists in the swapfile, while moveToSwap and moveFromSwap will ensure an atomic read from or write to the swapfile.

```
public static Swap getSwap() {
    obtain interrupt status from Machine
    if theOnlySwap points to null, create a new one
    restore status
    return theOnlySwap
}

public boolean contains(int pid, int vpn) {
    acquire the lock
    check if (pid,vpn) has a valid offset
    release lock
    return boolean
}

public boolean moveToSwap(MemoryMapEntry entry) {
    if the entry is not pinned, return false
    acquire lock
    if entry is not readOnly and is dirty and is valid
        obtain offset within the swapfile.  if it doesn't have one, get a new one
        write entry information into swapFile at offset
        check if all information was correctly written into swapFile, return false if not
        record this new offset with pid and vpn
    increase swapOutCount
    remove this entry from the MemoryMap
    release lock and return true
}

public boolean moveFromSwap(MemoryMapEntry entry) {
    if the entry is not pinned, return false
    acquire lock
    obtain offset in the swapFile associated with this entry
    read the swapfile from this offset
    check if all information was correctly read, return false if not
    increase swapInCount
    set entry to readOnly
    check in this entry into the MemoryMap
    release lock and return true
}
```

There are also a few housekeeping methods for the swapfile. Once a process exits, all entries in the swap file corresponding to the process must be invalidated. Also, when the Kernel terminates, the swapfile must be close and deleted. These requirements are addressed by the methods invalidateProcess and close.

```
public void invalidateProcess(int pid) {
    acquire lock
    obtain the processTable according to this pid
        add all offsets associated with this pid to the freelist
    release lock
}

public void close() {
    close the openFile swapFile
    remove from the fileSystem
    set theOnlySwap to point to null
}
```

We decided to segment out a few of the common subroutines required by the previous swap methods. This makes the code cleaner, but the existence of these methods is invisible to everyone outside of the Swap class. When these methods are called, we can assert that the current thread owns the lock on the swapfile, so all of these methods are thread safe.

```
private int getFreeOffset() {
    if freeList is not empty, then return the first first
    else increment swapFileSize and return swapFileSize-1
}

private int getOffsetOfPage(int pid, int vpn) {
    obtain processTable associated with pid
    obtain offset from this processTable
    return offset
}

private void addOffsetOfPage(int pid, int vpn, int offset) {
    obtain processTable associated with pid
    if the processTable is null, create one andadd it to swapFileTable
    obtain current offset from processTable with vpn
    if current offset is null, then put given offset with vpn
    else assert true that offset is equal to current offset
}
```

Page Pinning

Pins are used to provide a locking mechanism on a per-page basis. The methodology is to pin as much as possible while ensuring multiprocessing. A page is pinned the moment it is selected as the free page we can work with, or when a page is checked out by a process for reading or writing. When a page is pinned, no other process is able to check out the page. Also, the clock algorithm will skip over the page when considering a page to evict from memory. Also, when a page is pinned, the possible actions on the page are: moving the page to swap, loading from swap into the page, loading from an executable into the page, or returning the page as a free page. It is unpinned after this whole procedure is completed. The interfaces for pinning are the methods MemoryMapEntry.pinThis and MemoryMapEntry.unpinThis.

Page Replacement

When a page fault occurs, we must bring a page into physical memory to satisfy the fault. However, it could be that every page in physical memory is valid, so we must choose a page to evict from memory and send to the swap file. We use a clock algorithm to choose the page that we want to remove from memory. The benefit of this comes from the assumption that if we have not used a page for a while, then it most likely will not be used in the near future. So by sweeping through memory and scanning for an old or invalid page, we can find a good candidate page for replacement. We use the moveToSwap method provided by the Swap class to move a page from memory to the swapfile. This method interacts with the memory map by using the moveFromMemory method inside of MemoryMap.

Clock algorithm inside MemoryMap

```
boolean done starts as false
MemoryMapEntry entry = coreMap[clockHand]
while not done:
    entry = coreMap[clockHand]
    if entry is pinned
        move on, because we don't touch pinned pages
    else if entry is not valid
        done = true             // This one is free for the picking
        pin this entry
    else if entry is used
        set entry's used bit to false
    else                        // All your pages are valid
        done = true
        pin this entry          //Have to swap this one out
    increment clockHand and mod by the number of physical pages
return entry
```

Tests:

Let multiple processes obtain many physical pages, and make sure that one page isn't given to two processes.
Assert that every physical page can be allocated to some process.
Assert that all physical pages are valid before ejecting a valid page from memory.
Assert that read-only pages and non-dirty pages aren't being written to the swap.
Assert that all available space in a swap file is used before growing the size of the swapfile.
For a very large number of calls to moveToSwap and moveFromSwap, assert that the size of the swapfile doesn't grow too large.
When pages start to get placed in the swap file, assert that all pages in physical memory are

valid.
When a page is moved out of physical memory, assert that the TLB does not think that page still resides in memory.

Performance of page replacement policy:

Write a C program that creates three large matrices and multiplies them together, which should create lots of paging. Find the number of page faults for a random page replacement scheme. Compare to the number of page faults for page replacement scheme that uses the Clock algorithm.

Task Three - (30%) Lazy Loading

Now that we have demand paging implemented, we can use lazy loading to utilize this demand paging. The motivation for lazy loading is that we only load pages into memory on demand, rather than reading all of the code and data from a user program at startup time. This task is mainly concerned with figuring out which section and which page of the Coff file a certain VPN corresponds to. Once we have this data, we can load the corresponding page. Rather than doing all of this on process initialization, we defer it to when a VPN for a Coff page we have not loaded is requested.

Code pages are marked readOnly, thus we never need to worry about swapping them to disk or about accidentally writing back to the executable file. Whenever a request is made for a code page, it is either already in memory, or we translate the VPN to the correct section and page, and load this into memory, by-passing the swap code completely.

Data pages can be modified, and if they are swapped in and out, they need to be written to the swap space so that a process does not lose its global variables as other processes gets swapped in. There is no place in our code where we write back to the executable file, thus we do not need to consider the case of accidentally writing modified data pages back into the original executable.

Stack pages can be modified and swapped exactly like heap pages, we just need to limit them to be no more than the allotted 8 stack pages in number. Stack pages can possibly contain data from the previous process' page that occupied the same physical memory.

Correctness Constraints

- Must ensure changes to memory image of executable are not written back.
- Must allocate stack pages on demand as well. Therefore, do not allocate the full set of stack pages to the process when it starts. Only allocate a new stack page when a page fault occurs.
- Maximum stack pages needed by process is 8.

Implementation

Much of the implementation for lazy loading resides in VMProcess.java. The goal is to load in a section of the coff that contains our desired vpn. There are two general cases. If the given entry's vpn is within numCoffPages, then we'll try to load in a Coff section. Else, a stack page has grown, so we make that entry's readOnly bit to false. The lazy-loading method goes as follows:

```
private void lazilyLoad(MemoryMapEntry entry) {
    obtain number of coff pages.  set boolean done to false

    if entry's vpn is less than number of coff pages
        for each section in the coff
            for each spn value
                determine if difference between vpn and spn is same as the firstVpn of
the section
                    if so, load the coff section into the ppn
                    set the entry to readOnly, and set done to true.  break
            if done is true, then break
    else
        set entry's readOnly bit to false
}
```

This lazilyLoad method is called by resolveMemory, which handles some memory consistency and also the case when a desired coff section is not currently in memory.

```
public MemoryMapEntry resolveMemory(VMProcess proc, int vpn, checkInEffect
effect) {
    set boolean suicide to false
    obtain memory map from MemoryMap.getMemoryMap()
    obtain swap from Swap.getSwap
    initialize MemoryMapEntry entry pointer

    if the entry checked out from MemoryMap associated with the pid and vpn is not
null
        check in this entry into Memorymap with the given effect argument
    else if swapFile contains an entry corresponding to the pid and vpn
        increment page fault count
        check out any page from MemoryMap for this process to use
        if this return entry is valid
            move this entry to swap, and set suicide to be opposite of the success of
this call
        if suicide is still false
            set pid and vpn of this entry to given arguments
            move entry to memory, and set suicide to be opposite of the success of
this call
    else if vpn is in the coff range
        increment page fault count
        check out entry from memoryMap
        if entry is valid, move the entry to swap
        set pid and vpn of this entry according to the arguments
        call lazilyLoad for this entry
        check in this process into memoryMap
    else
        increment page fault count
        potential error: not in memory, not in swap, and not in coff range

    if suicide is true, make machine exit
    return entry
}
```

The resolveMemory method is called upon in three locations. Two locations are read and write virtual memory, and another is in handleException to support the case of a TLB miss. Therefore, handleException must be overridden to support this case.

```
public void handleException(int cause) {
    obtain the processor from Machine.processor
    switch(cause)
    if the case is TLB miss exception
        read the bad virtual address from the register
        obtain the vpn
        call resolveMemory with this current VMProcess, vpn, and enum
WRITE_TO_TLB
        break
    default, call super.handleException
    break
}
```

Much of the read and write virtual memory methods remain the same from UserProcess.java. However, due to the MemoryMap data structure we have, we must make sure we correctly update the necessary values as we read or write from virtual memory.

```
public int readVirtualMemory(int vaddr, byte[] data, int offset, int length) {
    ...
    while (length > 0)
        obtain MemoryMapEntry from resolveMemory
        obtain the ppn from the entry
        call touchPhysicalPage with this ppn without making it dirty
    ...
}

public int writeVirtualMemory(int vaddr, byte[] data, int offset, int length) {
    ...
    while (length > 0)
        obtain MemoryMapEntry from resolveMemory
        obtain the ppn from the entry
        call touchPhysicalPage with this ppn with making it dirty
    ...
}
```

VMProcess.java must also override the loadSections() method, to make sure our lazy-loading structures are used. We simply return true for this method. The method unloadSections() method must also be overridden, to make sure that we clean up any MemoryMap and Swapfile references to a particular pid. We call MemoryMap.invalidateProcess(pid) and Swap.invalidateProcess(pid) to do this.

Test

Testing isCoffOrPageSection(vpn)
- instantiate several different VMProcesses, each with a different coff file
- call isCoffOrPageSection with different vpn values, making sure the border cases work:
    - 0 must be a coff page
    - numCoffPages must be a coff page
    - numCoffPages+1 must be a stack page
    - numCoffPages+7 must be a stack page

Test loadCoffStackPage
- instantiate several different VMProcesses, each with a different coff file
- call loadCoffStackPage for each process
    - loop, call for each coff and stack page
        - check IPT for valid entries after each call
- call isCoffOrStackPage
    - if its a stack page, should NOT be loaded again
    - if its a coff page, should be loaded again (we simulate page faults by calling handlePageFault directly

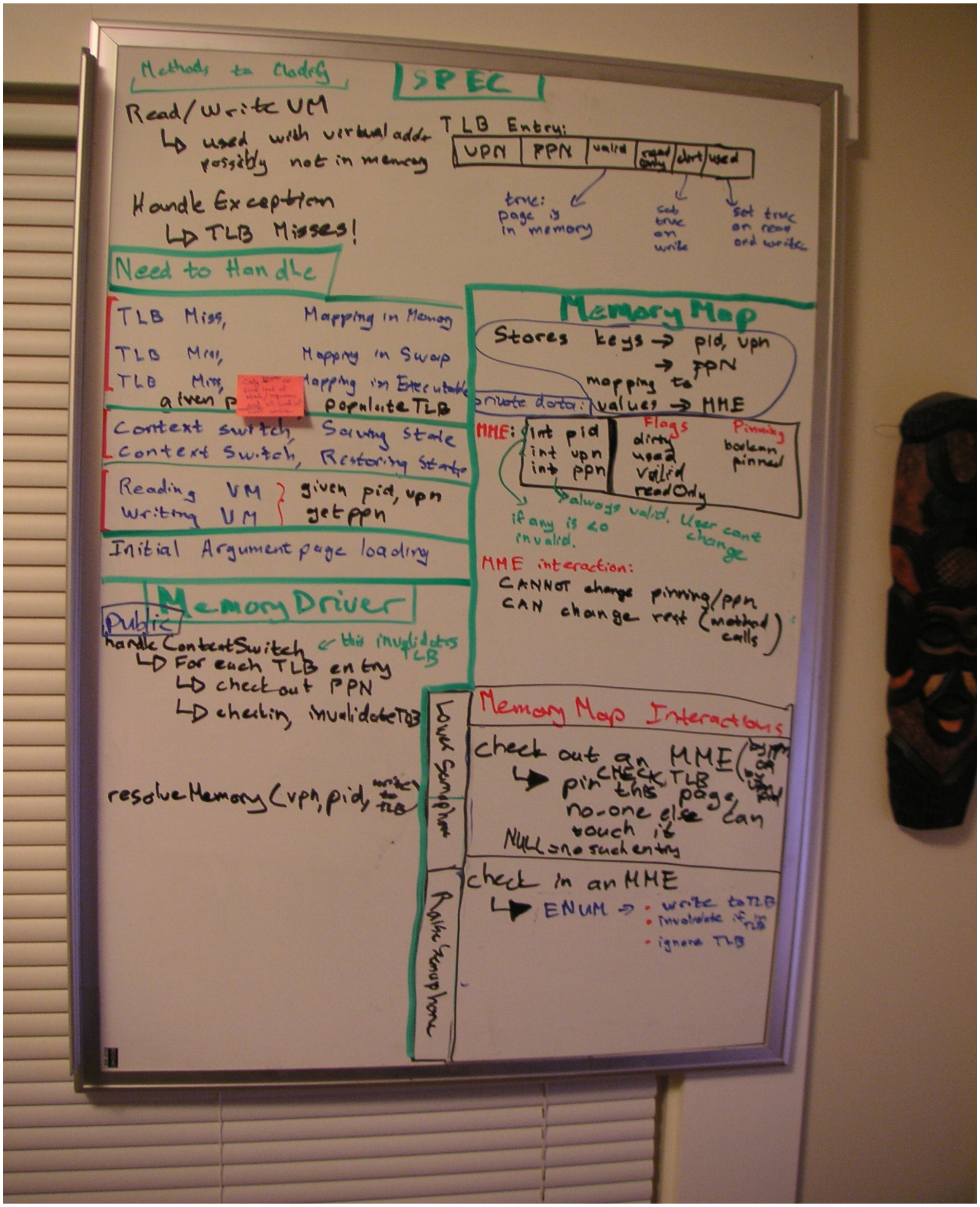
Global Testing

Since much of this project is concerned with the virtual memory system as a whole, we will consider the VM system as a black box, and feed it different files to execute. We can make a test framework whose inputs are an executable name, the arguments to pass the executable, and the expected outputs. Then we will create a new UserProcess and pass in the executable and arguments for the process to execute. Then we can also create a CaptureListener class that listens for standard output and logs those outputs, which we can compare against the expected outputs. We can run this suite of tests for various total numbers of physical pages, which we must manually set inside of nachos.conf if we want it to vary. The tests we will run are:

1) Simple test of echo.coff, checking for echoed outputs
2) Reading and writing a file various times, ranging from 0 to 15
3) Scanning through a file, especially with only two pages
4) Writing to an illegal virtual address or to a read-only page
5) Tests heavy page swapping with matmult and other executables with large arrays to iterate through
6) Joining on various processes that have various page requirements
7) Running one process at a time, running multiple processes concurrently

Reference Image

This is an image of the different requirements of the TLB and processor, and the possible states of the TLB that we must address. It also shows the data structures that we need.



Methods to Clarify

SPEC

Read/Write VM
↳ used with virtual addr
possibly not in memory

Handle Exception
↳ TLB Misses!

TLB Entry:

VPN	PPN	valid	read Only	dirt	used

valid → true: page is in memory

dirt → set true on write

used → set true on read and write

Need to Handle

TLB Miss, Mapping in Memory
TLB Miss, Mapping in Swap
TLB Miss, Mapping in Executable
given p[illegible] populate TLB

[illegible]

Context switch, Saving state
Context Switch, Restoring State

Reading VM } given pid, vpn
Writing VM } get ppn

Initial Argument page loading

Memory Driver

Public

handleContextSwitch ← this invalidates TLB
↳ For each TLB entry
↳ check out PPN
↳ checkin, invalidateTLB

resolveMemory(vpn, pid, write to TLB)

Lock Semaphore

Release Semaphore

Memory Map

Stores keys → pid, vpn
→ PPN
mapping to
values → MME

private data:

MME:

int pid	Flags	Pinning
int vpn	dirty	boolean
int ppn	used	pinned
	valid	
	readOnly	

if any is ≤0 invalid.

always valid. User can't change

MME interaction:
CANNOT change pinning/ppn
CAN change rest (method calls)

Memory Map Interactions

check out an MME (by ppn or by vpn/pid)
↳ CHECK TLB
pin this page, no-one else can touch it
NULL = no such entry

check in an MME
↳ ENUM →
- write to TLB
- invalidate if in TLB
- ignore TLB